Counting Worlds:
From Usernames to Universes

Jeremy Evert

November 11, 2025

Contents

Chapter 1

Welcome to *Counting Worlds*

1.1 A Story About Too Many Possibilities

Imagine this:

- You open a new game.
- You have to choose a character: race, class, hairstyle, outfit, special skill.
- Five minutes later, you are still on the character creation screen.

It feels like there are *infinitely many* options. In reality, there is a large but finite *universe of possibilities*. This chapter—and this whole mini-unit—is about learning how to *count* those universes.

We are going to ask questions like:

- How many ways can we seat six friends in a row of chairs?
- How many usernames can a website support under a given naming rule?
- How many sundaes can we build from a limited supply of scoops and flavors?
- How often should a particular dice pattern show up if our math is right?
- How big is the universe of characters, decks, or skill builds in a game?

Instead of guessing, we will use discrete mathematics and a bit of Python to get real answers. Along the way, we will see that:

> *Counting is how we tame spaces that feel infinite.*

1.2 What You Will Be Able to Do

By the end of this counting unit, you should be able to:

Mathematical skills

- Use the product principle ("AND" means multiply) and the sum principle ("either/or" means add, when choices are disjoint) in real problems.
- Work with *permutations* (where order matters) using factorials.
- Work with *combinations* (where order does not matter) using binomial coefficients $\binom{n}{k}$.
- Use the *stars and bars* technique to count ways of distributing identical items (like scoops, points, or coins) into bins.
- Connect counting to *probability* in simple finite situations.

Computational skills

- Use Python as an "experimental math lab":
  - generate all outcomes for small cases;
  - check whether a formula seems to be right;
  - estimate probabilities via Monte Carlo simulation.
- Translate informal stories (about games, food, or chaos) into:
  - precise combinatorial models, and
  - short, readable Python scripts.

Design and creativity

- Design your own small "universe" (a game, system, or scenario).
- Identify where permutations, combinations, and stars-and-bars appear inside that universe.
- Justify your counting formulas in words, not just symbols.

1.3 How This Mini-Book Works

Each chapter follows the same basic rhythm:

1. **Start with a story.** We begin with something that feels familiar: seating friends, building sundaes, creating usernames, rolling dice, or designing a game world.
2. **Extract the structure.** We strip away the flavor and look at the underlying combinatorial skeleton: choices, constraints, order vs. no order, repetition vs. no repetition.
3. **Do the math.** We use the tools of discrete mathematics to turn the story into symbols and formulas and compute exact counts.
4. **Check or explore with Python.** For small versions of the problem, we let a Python script:
   - generate the whole universe explicitly,
   - count how many outcomes have a certain property,

- approximate probabilities with random trials.

5. **Reflect and remix.** At the end of the chapter, we check understanding with short exercises and a brief "podcast" conversation that ties the ideas back to the story.

You do *not* need to be a Python expert. The code will be short and focused, and you can treat it as executable pseudocode if you are still learning the language.

1.4 Python as Our Counting Laboratory

Here is roughly how Python shows up throughout this unit:

- We use `itertools` to generate permutations, combinations, and simple products (like all possible usernames of a given form).
- We use the `math` module for things like `math.factorial` and `math.comb`.
- We use the `random` module to simulate coin flips, dice rolls, and other random experiments.

Most scripts will follow this template:

1. Set up a small, concrete version of the problem.
2. Compute a theoretical count or probability using a formula.
3. Use Python to:
   - either enumerate all possibilities, or
   - run a large number of random trials.
4. Compare the result from Python with the formula.

You will be encouraged to modify the scripts: change parameters, add constraints, or invent your own stories that use the same combinatorial ideas.

1.5 Roadmap for the Five Chapters

Here is the plan for the rest of this counting unit:

Chapter 1: Seating the Party at the Round Table

We start with *factorials* and *permutations*. You will:

- Count the number of ways to seat people in a row.
- Handle restrictions (must sit together, must not sit together, fixed seats).
- Use Python to brute-force small seating problems and check your formulas.

Chapter 2: Usernames, License Plates, and Other Noisy Strings

We move to *strings with repetition*. You will:

- Model usernames, license plates, and PINs as strings from an alphabet.
- Use the product principle to count how many such strings exist.
- Compare different username rules and discuss which ones create larger (or smaller) universes of names.

Chapter 3: How Many Sundaes Can We Build?

We introduce *stars and bars*. You will:

- Count ways to distribute scoops among flavors or points among skills.
- Visualize stars-and-bars as a picture and connect it to a formula.
- Use Python to generate all small distributions and verify your counts.

Chapter 4: Roll the Dice, Check the Math

We tie counting to *probability* and *simulation*. You will:

- Compute exact probabilities using combinatorial reasoning.
- Run Monte Carlo simulations in Python to estimate those probabilities.
- Watch simulated frequencies drift toward theoretical values as the number of trials increases.

Chapter 5: Design Your Own Universe

Finally, you become the world-builder. You will:

- Design a small universe of your choice (game, system, or scenario).
- Identify at least one permutation, one combination, and one stars-and-bars situation inside it.
- Analyze your universe with formulas *and* Python, then present your findings.

Podcast: Episode 0 — Trailer for the Counting Worlds

At the end of this intro, there is a short podcast episode that you can listen to while walking, commuting, or setting up your Python environment. The episode can follow this rough script:

- Introduce the "cast" of the unit (for example, recurring student characters or narrators).
- Share one real-life story where counting matters (picking teams, building a schedule, designing a game).
- Tease each chapter in one sentence:

  - "First, we figure out how many ways to seat a chaotic friend group."
  - "Then, we see how many usernames and license plates fit in the system."
  - "Next, we over-engineer sundae bars and skill trees."
  - "After that, we roll dice to see if our math holds up."
  - "Finally, *you* design your own universe and count it."

- Close with a simple challenge:

  Before Chapter 1, try to guess: *How many different outfits can you build from your own closet?*

This podcast is not graded; it is here to set the tone, tell stories, and give you a human voice walking you into the math.

Chapter 2

Seating the Party at the Round Table

2.1 Story Hook: Dinner Disaster

You are in charge of a big group dinner.

There are six friends and six chairs in a row. Everyone has Opinions:

- Amina *must* sit on an end.
- Jake refuses to sit next to Zahra.
- Lin and Zahra are best friends and want to sit together.

The host asks you a simple question:

"How many different seating charts are possible?"

At first, this seems like a small thing. But as the number of people grows, the number of possible seatings explodes. By the end of this chapter you will know:

- how to count all possible seatings when everyone is distinct;
- how to handle simple constraints (must sit together / apart / on an end);
- how to use Python to *check* your counting on small examples.

2.2 Warm-Up: Listing All Seatings for Three Friends

Let us start very small. Suppose we only have three people: Amina (A), Lin (L), and Jake (J), and three chairs in a row.

Try it yourself

1. Draw three boxes to represent the chairs.
2. List every possible way to place A, L, and J into those chairs.

If you do this carefully, you should find these six arrangements:

$$\text{ALJ},\ \text{AJL},\ \text{LAJ},\ \text{LJA},\ \text{JAL},\ \text{JLA}.$$

There are 6 different seatings. This is already a lot for just three people! Soon we will see how this connects to the factorial function and why three people give six seatings, four people give twenty-four seatings, and so on.

2.3 Core Ideas: Factorials and Linear Permutations

2.3.1 The factorial function

The *factorial* of a positive integer n is written $n!$ and defined as

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1,$$

with the special convention that $0! = 1$.

Some small values:

$$1! = 1, \quad 2! = 2, \quad 3! = 6, \quad 4! = 24, \quad 5! = 120.$$

2.3.2 Factorials as seatings

We can interpret $n!$ as the number of ways to seat n distinct people in n distinct chairs in a row.

Why?

- For the first chair, we can choose any of the n people.
- For the second chair, we have $n-1$ people left.
- For the third chair, we have $n-2$ people left.
- ...
- For the last chair, we have 1 person left.

By the *product principle*, the total number of seatings is

$$n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 = n!.$$

This explains why:

- $3! = 6$ seatings for A, L, J (which we listed),
- $4! = 24$ seatings for four distinct people,
- $6! = 720$ seatings for six distinct people.

Already at six people, 720 is a lot of possible seating charts. No human host is going to check all of those by hand.

2.3.3 Permutations of a set

If we have a set S of n distinct elements, any ordering of the elements of S in a row is called a *permutation* of S. The discussion above says:

> The number of permutations of a set of size n is $n!$.

This is one of the most basic counting facts in discrete mathematics: whenever you are arranging n distinct things in a line and every order is allowed, the answer is $n!$.

2.4 Restricted Seatings: Adding Drama

Real dinners have rules. Let us see how a few common restrictions change the counting. Throughout this section, think of six seats in a row and six friends: Amina (A), Lin (L), Zahra (Z), Jake (J), plus two more friends, M and N.

2.4.1 Example 1: One person in a fixed seat

Suppose Amina *must* sit in the first chair.

- We no longer have a choice for the first chair: it is always A.
- For the remaining five chairs, we can seat the other five friends in any order.

So the total number of seatings is simply

$$5!$$

because we are only permuting the remaining five people.

In general, if one person has a fixed location and the others are free, the number of seatings is $(n-1)!$.

2.4.2 Example 2: Two people sit together as a block

Now suppose Lin and Zahra insist on sitting next to each other.

A standard trick is to treat the pair (LZ) as a single "block."

- First, create a new list of "objects": the block (LZ) plus the other four people A, J, M, N. That gives 5 objects to arrange.
- The number of ways to arrange these 5 objects is 5!.
- Inside the block, Lin and Zahra can sit in two orders: LZ or ZL.

By the product principle:

$$\text{number of seatings with L,Z together} = 5! \cdot 2.$$

This idea appears over and over: when some group must stay together, we often treat that group as a single unit, count arrangements of the units, then multiply by the internal arrangements of each unit.

2.4.3 Example 3: Two people must not sit together

Now suppose Jake and Zahra *refuse* to sit next to each other. How many seatings avoid J and Z being adjacent?

A common strategy is:

1. Count all possible seatings with no restriction: 6!.
2. Count how many seatings have J and Z together (as a block).
3. Subtract: (all seatings) − (bad seatings).

We already know from the previous example that the number of seatings with J and Z together is $5! \cdot 2$ (treat JZ as a block, and swap inside the block). Therefore:

$$\text{seatings with J and Z not adjacent} = 6! - 5! \cdot 2.$$

This pattern—*count everything, then subtract the bad cases*—is a very powerful idea in counting. We will see it again later in the unit with more complicated sets of "bad" outcomes.

2.5 Python Lab: Brute-Forcing the Seating

Mathematics gives us formulas, but Python lets us *test* those formulas on small examples and explore different constraints without doing all the bookkeeping by hand.

In this lab you will write or modify a script (for example, `round_table.py`) with the following goals.

Step 1: Generate all seatings

1. Create a list of names, such as `["Amina", "Lin", "Zahra", "Jake"]`.
2. Use `itertools.permutations` to generate all possible orderings of the list.
3. Confirm that the number of permutations matches $n!$ for your n.

A short code sketch might look like this:

```
import itertools
import math

friends = ["A", "L", "Z", "J"]
perms = list(itertools.permutations(friends))

print("Number of permutations:", len(perms))
print("Should be:", math.factorial(len(friends)))
```

Step 2: Count seatings with a fixed person at one end

Modify your code so that it counts only those permutations where Amina sits in the first chair.

- Compare the Python count to the formula $(n-1)!$.
- Try both ends (first or last) and see what happens.

Step 3: Count seatings where two friends sit together

Choose two friends (for example, Lin and Zahra) and:

- count how many permutations have them sitting in adjacent seats;
- compare to the "block" formula $5! \cdot 2$ (for $n = 6$);
- experiment with different values of n and positions.

Step 4: Count seatings where two friends do *not* sit together

Finally, let Python estimate how many permutations avoid J and Z being neighbors.

- Compute the number directly by checking every permutation.
- Compare with the subtraction formula $6! - 5! \cdot 2$.
- Try other pairs of friends and see if the pattern holds.

The goal is not to write huge programs, but to use short scripts as mirrors that reflect the combinatorial structure we discovered on paper.

2.6 Practice and Extensions

Here are some practice problems you might see after reading this chapter.

Basic practice

1. Explain in words what 5! means in the context of seating five distinct friends in five chairs.
2. Compute 6! and describe a real-life scenario where 6! is the correct answer.

Medium spice

1. Six friends are to be seated in a row. Two of them insist on sitting together. How many seatings are possible?
2. Six friends are to be seated in a row. Two of them refuse to sit next to each other. How many seatings are possible?

Extra spicy (optional)

1. Circular table variation: six friends sit around a round table. Two seatings that can be rotated into each other are considered the same. How many distinct seatings are possible?
2. Modify your Python script to treat rotations as identical and test your answer on small values of n.

Podcast: Episode 1 — The Seating Chart

At the end of this chapter, there is a short podcast episode. A possible outline for the episode:

- The characters are trying to organize a group dinner.
- They first try to list out seatings for a small group and quickly get overwhelmed as the group grows.
- Someone introduces the idea of $n!$ and the "product of choices" story: first seat, second seat, and so on.

- They play with the idea of treating two friends as a "block" and of subtracting the bad arrangements.
- They mention that Python helped them check their answers by generating all seatings for small examples.
- They end with a teaser:

> "If we can count seatings, can we also count usernames? Next time: gamer tags, license plates, and the size of the internet."

Chapter 3

Usernames, License Plates, and Other Noisy Strings

3.1 Story Hook: Launch Day Username Panic

It is launch day.

The new game *DragonMath Online* goes live at midnight. Amina, Lin, Zahra, and Jake all smash the "Create Account" button at the same time.

Amina types `Amina`.

System: *Sorry, that username is already taken.*

No big deal. She tries `Amina1`. Taken. `Amina01`. Taken. `RealAmina`. Taken.

Meanwhile, Jake proudly types `xXJakeXx`.

System: *Sorry, that username is already taken.*

After five minutes, they are not battling dragons. They are battling the username system.

The developers are nervous for a different reason: they picked a very simple username rule.

> Usernames must be exactly four characters long, and each character must be a capital letter A–Z.

Someone finally asks the real question:

"Are there actually enough usernames for all our players?"

This chapter is about turning that kind of panic into calm arithmetic.

We will:

- Treat usernames, license plates, and PIN codes as *strings* built from an *alphabet*.
- Use the **product principle**: when you make a sequence of independent choices, you *multiply* the number of options.
- See how tiny changes to the rules can explode (or shrink) the size of the username universe.
- Use Python to explore small universes and sanity check our formulas.

By the end, you should feel comfortable looking at a string format and thinking, almost automatically,

"Okay, that is k^n possibilities."

3.2 Counting Strings with Repetition

We will be very systematic.

Strings, alphabets, and formats

A **string** is just an ordered list of characters. The characters come from some **alphabet**:

- 26 uppercase letters: $A, B, \ldots, Z$,
- 10 digits: $0, 1, \ldots, 9$,
- or maybe letters + digits + a few symbols.

A **format** is a pattern for strings. For example:

- **LLLL**: four letters (like `ABCD` or `GAME`).
- **LLDD**: two letters followed by two digits (like `CS19`).
- **DDD**: three-digit PINs (like `042`).

The key combinatorial question:

Given a format, how many possible strings follow that format?

The product principle in disguise

The **product principle** says:

> If you make a sequence of independent choices, and
>
> - the first choice can be made in a_1 ways,
> - the second choice can be made in a_2 ways,
> - ...
> - the n-th choice can be made in a_n ways,
>
> then the total number of possible outcome sequences is
>
> $$a_1 \cdot a_2 \cdot \cdots \cdot a_n.$$

A string of length n is just n choices in a row: one character for each position.

Example: four-letter usernames

Suppose the rule is:

> Usernames must be exactly four uppercase letters.

How many usernames are possible?
Each position can be any of 26 letters. The four choices are independent, so

$$\underbrace{26}_{\text{first letter}} \cdot \underbrace{26}_{\text{second}} \cdot \underbrace{26}_{\text{third}} \cdot \underbrace{26}_{\text{fourth}} = 26^4.$$

So there are 26^4 possible usernames. That is 456,976 names. Not infinite — but definitely enough to keep the first few thousand players happy.

Example: license plates (letters then digits)

Now consider a classic license-plate style format:

Three letters followed by three digits: LLLDDD.

For each plate:

- First character: 26 choices (any letter).
- Second character: 26 choices.
- Third character: 26 choices.
- Fourth character: 10 choices (any digit).
- Fifth character: 10 choices.
- Sixth character: 10 choices.

By the product principle,

$$26 \cdot 26 \cdot 26 \cdot 10 \cdot 10 \cdot 10 = 26^3 \cdot 10^3.$$

If we actually computed this number, it would be big, but the structure is what matters: *one factor for each position.*:contentReference[oaicite:0]index=0

Example: strings from a mixed alphabet

Suppose a website allows usernames that are six characters long, and each character can be

- an uppercase letter (26 options), or
- a digit (10 options).

That is $26 + 10 = 36$ choices for each of the six positions.
The total number of usernames is

$$36^6.$$

We do not even need to expand it. The point is that the universe of names scales like "alphabet size to the power of length."

Adding mild drama: constraints on strings

Real systems often add rules:

- "Must contain at least one digit."
- "Cannot start with a digit."
- "No banned substrings like `BAD` or `XXX`."

These constraints are where the product principle teams up with other ideas, like the **sum principle** and **complements** ("count everything, subtract the bad").
We will practice a few of these designs in the exercises, but for now the main idea is:

As long as each position is a choice, and we understand which choices are allowed, the product principle is our go-to tool for counting strings.

3.3 Python Lab: Exploring the Username Space

We will now turn Python into our username telescope. For tiny alphabets, we can literally list every possible name. For realistic alphabets, we let Python compute the counts and sample a few candidates.

Step 1: A toy universe

We start with a small alphabet so that we can enumerate everything.

Listing 3.1: Toy username universe with tiny alphabets

```
import itertools

# A tiny alphabet so we can actually list everything
letters = ["A", "B", "C"]
digits = ["0", "1"]

def all_usernames_LLLL():
    """All usernames of format LLLL over {A,B,C}."""
    return ["".join(p) for p in itertools.product(letters, repeat=4)]

def all_usernames_LLDD():
    """All usernames of format LLDD over {A,B,C} and {0,1}."""
    return [
        "".join(p)
        for p in itertools.product(letters, letters, digits, digits)
    ]

names_LLLL = all_usernames_LLLL()
names_LLDD = all_usernames_LLDD()

print("LLLL count (Python):", len(names_LLLL))
print("LLLL count (formula):", len(letters) ** 4)

print("LLDD count (Python):", len(names_LLDD))
print("LLDD count (formula):", len(letters) ** 2 * len(digits) ** 2)
```

For this toy world,

- there should be $3^4 = 81$ four-letter usernames, and
- $3^2 \cdot 2^2 = 36$ usernames of the form LLDD.

Your script should confirm these counts exactly.

Step 2: Scaling up to realistic alphabets

Once we trust the pattern in the toy world, we can move to full alphabets without listing millions of strings.

Listing 3.2: Counting real-world username formats

```
import string
```

```
letters = string.ascii_uppercase      # 26 letters
digits = string.digits                # 10 digits

def count_LLLL():
    return len(letters) ** 4          # 26^4

def count_LLDD():
    return len(letters) ** 2 * len(digits) ** 2 # 26^2 * 10^2

def count_mixed(length):
    """Usernames of given length using letters+digits."""
    alphabet_size = len(letters) + len(digits)
    return alphabet_size ** length

print("LLLL usernames:", count_LLLL())
print("LLDD usernames:", count_LLDD())
print("8-char mixed usernames:", count_mixed(8))
```

Here, Python is acting as a calculator with really good vibes:

- You write down the combinatorial expression (36^8, etc.).
- Python evaluates it cleanly and correctly.
- You can change the length or alphabet and see how fast the universe explodes.

Step 3: Checking simple constraints

We can also ask Python to explore basic constraints, like "must contain at least one digit." On full alphabets we use formulas, but on toy alphabets we can inspect every string.

Listing 3.3: Toy constraint: at least one digit

```
import itertools

letters = ["A", "B", "C"]
digits = ["0", "1"]

alphabet = letters + digits

def all_strings_of_length(n):
    return ["".join(p) for p in itertools.product(alphabet, repeat=n)]

def has_digit(s):
    return any(ch in digits for ch in s)

def count_with_at_least_one_digit(n):
    strings = all_strings_of_length(n)
    good = [s for s in strings if has_digit(s)]
    return len(good)

n = 3
print("Total strings:", len(alphabet) ** n)
```

```
print("With at least one digit (Python):", count_with_at_least_one_digit(n))
```

This is a great place to connect code back to math:

- Total number of strings: $|\text{alphabet}|^n$.
- Strings with no digit: $|\text{letters}|^n$.
- Strings with at least one digit:

$$|\text{alphabet}|^n - |\text{letters}|^n.$$

Python lets you *see* the difference on small n before trusting the formula for large n.

3.4 Practice and Design Questions

Here are some practice problems and design prompts you might use after this chapter.

Basic practice

1. A username rule says: "exactly five uppercase letters." How many usernames are possible?
2. A door lock uses a 4-digit PIN from 0000 to 9999.
   (a) How many PINs are there in total?
   (b) How many PINs start with a 0?
3. In your own words, explain how the product principle appears in the format LLDD.

Medium spice

1. A website uses the rule: "usernames are of the form LLLDD, where L is a letter and D is a digit."
   (a) How many usernames are possible?
   (b) How many usernames start with the letter `A`?
   (c) How many usernames start with either `A` or `B`?
2. A game uses 6-character usernames made from uppercase letters and digits.
   (a) How many usernames are possible in total?
   (b) How many have no digits at all (letters only)?
   (c) Use your answer from (a) and (b) to count how many have *at least one* digit.
3. A state is considering a license plate format: two letters, three digits, then one letter (LLDDDL). How many plates are possible?

Extra spicy (optional)

1. A chat app allows usernames of length 8 where each character can be

   - an uppercase letter,
   - a lowercase letter,
   - or a digit.

   Write a product-principle expression (you do not have to multiply it out) for the total number of usernames.

2. The rule now says: "Usernames must be length 6 and must contain *at least one* digit and *at least one* letter." Describe two different strategies to count such usernames:

   - Using complements (subtracting the all-letter and all-digit cases).
   - Using a case split ("exactly one digit," "exactly two digits," etc.) and the sum principle.

   You do not need to carry out all the algebra; focus on the *structure* of the counting.

Podcast: Episode 2 – Name Yourself Wisely

At the end of this chapter, you might record a short podcast episode. A possible outline:

- The characters are on voice chat trying to claim their dream usernames before the servers fill up.
- They keep seeing "that name is taken" and start wondering how many names the system actually allows.
- One character explains the product principle using toy examples (two-letter words, simple PINs).
- They compare different username rules:
  - all letters vs. letters+digits,
  - fixed length vs. variable length.
- They joke about terrible password choices like `PASSWORD` and `123456`, and connect the size of the search space to security.
- They end with a teaser:

  > "If we can count usernames, can we count ice cream sundaes? Next time: stars, bars, and way too many toppings."

Chapter 4

How Many Sundaes Can We Build?

4.1 Story Hook: Infinite Ice Cream Bar

It is Friday. The math department has made a terrible mistake: they gave you a *build-your-own-sundae* bar.

There are k flavors of ice cream lined up: vanilla, chocolate, strawberry, mint, cookie dough, and whatever experimental flavor Lin convinced the shop to try.

You are given n scoops to distribute among these flavors. The rules:

- Scoops of the *same* flavor are indistinguishable.
- The only thing that matters is how many scoops of each flavor you take.

Someone asks, with a suspicious smile:

> "How many different sundaes can you build?"

If you try to list them all by hand, your brain melts faster than the ice cream. But with the right counting idea, you can answer the question cleanly.

This chapter is about **stars and bars** — a way to count how many ways we can distribute identical objects (scoops) into labeled boxes (flavors).

4.2 Stars and Bars: Distributing Identical Objects

From sundaes to equations

Suppose we have k flavors and n scoops total.

Let:

$x_1 =$ number of scoops of flavor 1, $\quad x_2 =$ number of scoops of flavor 2, $\quad \ldots, \quad x_k =$ number of scoops of flavor k

Every possible sundae corresponds to a solution of the equation

$$x_1 + x_2 + \cdots + x_k = n,$$

where each $x_i \geq 0$ is an integer.

So our counting problem becomes:

How many solutions in nonnegative integers $(x_1, \ldots, x_k)$ satisfy

$$x_1 + x_2 + \cdots + x_k = n?$$

Each solution is one way to build a sundae.

Drawing the picture: stars and bars

We imagine each scoop as a *star* ($*$), and we use *bars* ($|$) to separate flavors.

For example, suppose $n = 4$ scoops and $k = 3$ flavors. One possible scoop distribution is:

$$x_1 = 1, \quad x_2 = 2, \quad x_3 = 1.$$

In stars-and-bars form, that looks like:

$$* \mid ** \mid *$$

- The first block of stars (before the first bar) is flavor 1.
- The second block (between bars) is flavor 2.
- The last block (after the second bar) is flavor 3.

Another distribution, say $x_1 = 0$, $x_2 = 3$, $x_3 = 1$, would look like:

$$\mid *** \mid *$$

Here, flavor 1 gets zero stars (no scoops), which is fine.

In general:

- We have n stars total (for n scoops).
- We need $k - 1$ bars to carve the line into k blocks.

So every solution corresponds to a line of n stars and $k - 1$ bars:

$$\underbrace{* * \cdots *}_{n \text{ stars}} \quad \text{and} \quad \underbrace{|| \ldots |}_{k-1 \text{ bars}},$$

arranged in some order.

Counting the arrangements

We now just have to count how many strings of length $n + (k - 1)$ can be made from n identical stars and $k - 1$ identical bars.

One way to think about it:

- There are $n + k - 1$ total positions.
- We choose which $(k - 1)$ of those positions will be bars.
- The remaining positions automatically become stars.

So the number of different star-bar arrangements is

$$\binom{n+k-1}{k-1}.$$

Each such arrangement corresponds to exactly one solution $(x_1, \ldots, x_k)$, so we arrive at the classic stars-and-bars formula:

The number of nonnegative integer solutions to

$$x_1 + x_2 + \cdots + x_k = n$$

is

$$\binom{n+k-1}{k-1}.$$

In sundae language:

The number of ways to build a sundae with n scoops and k flavors (allowing some flavors to get zero scoops) is

$$\binom{n+k-1}{k-1}.$$

A small example: 4 scoops, 3 flavors

Take $n = 4$ and $k = 3$.

Our formula says there should be

$$\binom{4+3-1}{3-1} = \binom{6}{2} = 15$$

different sundaes.

If we list all integer solutions to $x_1 + x_2 + x_3 = 4$ with $x_i \geq 0$, we do indeed get 15 possibilities, such as

$$(4,0,0), (3,1,0), (3,0,1), \ldots, (0,0,4).$$

We will let Python do the actual listing later. For now, the important thing is to trust that the *picture* (stars and bars) really matches the *equation* and the *formula*.

4.3 Variants: Minimums and Caps

Real ice cream shops — and real combinatorics problems — often come with extra rules.

Everyone gets at least one scoop

Suppose you must use all k flavors at least once. In sundae form: every flavor gets at least one scoop. In math form:

$$x_i \geq 1 \quad \text{for all } i, \qquad x_1 + \cdots + x_k = n.$$

We can convert this into a nonnegative problem by "giving everyone one scoop up front." Let

$$y_i = x_i - 1.$$

Then $y_i \geq 0$, and

$$(x_1 + \cdots + x_k = n) \quad \Longleftrightarrow \quad (y_1 + \cdots + y_k = n - k).$$

So the number of solutions is

$$\binom{(n-k)+k-1}{k-1} = \binom{n-1}{k-1},$$

as long as $n \geq k$ (otherwise it's impossible to give everyone at least one scoop).

In sundae language:

> The number of sundaes with n scoops and k flavors, where every flavor appears at least once, is
>
> $$\binom{n-1}{k-1}.$$

Caps: at most some number per flavor

A trickier variation is when each flavor has a maximum number of scoops:

> Each flavor can have at most c scoops.

So we want integer solutions to

$$x_1 + \cdots + x_k = n, \quad 0 \leq x_i \leq c.$$

There is no single magic formula as simple as the basic stars-and-bars case, but there are strategies:

- For small n, k, c, we can list all possibilities with Python and count.
- On paper, we can sometimes:
  - use symmetry,
  - break into cases,
  - or use "count everything, subtract the violations" if only a few solutions violate $x_i \leq c$.

In this chapter, we mostly treat caps as optional challenge problems and lean on Python to double-check our reasoning for small numbers.

4.4 Python Lab: Enumerating Sundaes

As before, we use Python as a friendly lab assistant. For sundaes, Python is great for:

- generating all small integer solutions to $x_1 + \cdots + x_k = n$;
- verifying that the total matches $\binom{n+k-1}{k-1}$;
- experimenting with minimums and caps.

Step 1: Enumerate all $(x_1, \ldots, x_k)$ with $x_1 + \cdots + x_k = n$

For small n and k, we can loop over all possibilities and collect those that sum to n. Conceptually, the code will:

1. Fix values for n and k.
2. Loop over all k-tuples of nonnegative integers whose sum is n.
3. Store them in a list (each tuple is one sundae).
4. Count them and compare with $\binom{n+k-1}{k-1}$ using `math.comb`.

We can then label the flavors and translate each tuple into a human-readable description, like:

`(2,1,1)` $\rightarrow$ 2 scoops vanilla, 1 scoop chocolate, 1 scoop strawberry.

Step 2: Enforcing minimums

We can modify the code so that each flavor must get at least one scoop ($x_i \geq 1$). In code, this might be as simple as:

- generate all nonnegative solutions to $y_1 + \cdots + y_k = n - k$,
- then set $x_i = y_i + 1$.

Our Python count should match $\binom{n-1}{k-1}$.

Step 3: Playing with caps

For caps, we can use brute-force checking on small examples:

1. Generate all $(x_1, \ldots, x_k)$ with sum n.
2. Filter for those where $x_i \leq c$ for all i.
3. Compare the number to what you get from case-by-case reasoning (if possible).

The main point: the code does not have to be long or fancy. It just needs to mirror the combinatorial rules.

4.5 Practice: Scoops, Spells, and Skill Trees

Here are some practice directions that fit naturally after this chapter.

Scoops and sundaes

1. You have 5 scoops and 3 flavors.
   (a) How many different sundaes can you build if some flavors may get zero scoops?
   (b) How many different sundaes if every flavor must get at least one scoop?
2. A shop offers 4 toppings for your ice cream, and you may add up to 6 scoops of toppings total (multiple scoops of the same topping are allowed). How many different topping combinations are possible, counting only how many scoops of each topping you take?

Spells and mana points

1. In a game, a wizard has 8 mana points to distribute among 3 spells: Fire, Ice, and Lightning. Each point assigned to a spell increases its power.

   (a) How many ways can the wizard assign the 8 points if some spells may get zero?

   (b) How many ways if each spell must get at least 1 point?

2. A character has 10 skill points to distribute among 4 stats: Strength, Dexterity, Intelligence, and Charisma. Each stat can receive any nonnegative number of points.

   (a) How many builds are possible?

   (b) How many builds if every stat must have at least 1 point?

Extra spicy (optional)

1. A student council has 7 identical scholarships to distribute among 4 clubs. Each club may receive any number of scholarships (including 0).

   (a) How many distributions are possible?

   (b) How many distributions if no club may receive more than 3 scholarships? (Hint: this is a good place to combine stars-and-bars ideas with either casework or a small Python brute-force check.)

Podcast: Episode 3 – Sundae Architect

At the end of this chapter, you might record a short podcast episode where the characters wildly over-engineer their sundaes. A possible outline:

- Cold open:
  - The group is standing in front of an absurd ice cream bar.
  - Someone claims: “There are, like, a million sundaes you could build.”
- Turn to math:
  - One character models the situation with variables $x_1, \ldots, x_k$ and the equation $x_1 + \cdots + x_k = n$.
  - They draw a stars-and-bars picture and count the arrangements.
  - They translate the formula $\binom{n+k-1}{k-1}$ back into plain language.
- Variants:
  - Someone insists on using *every* flavor at least once.
  - Someone else wants a cap like “no more than three scoops of any one flavor.”
  - They discuss how the counts change, and when it is easier to let Python handle the messy details.
- Closing:

- The group realizes this same math applies to:
  - distributing skill points in games,
  - splitting resources in planning problems,
  - and even some probability questions.
- Teaser for the next chapter:

  > "We've counted sundaes, stats, and scholarships. Next up: dice, randomness, and whether the universe is actually trolling you."

Chapter 5

Roll the Dice, Check the Math

5.1 Story Hook: Can We Trust Our Formulas?

The group is back at the game table.

Amina has done the math and is absolutely convinced:

> "The chance of seeing at least one 6 in four rolls of a fair die is about $\frac{2}{3}$. I computed it with counting, and math never lies."

Jake, of course, does not believe in anything that isn't shiny and empirical:

> "Cool story. Hand me the dice. If your formula is right, we should *see* it in the data, right?"

Lin is halfway between them:

> "I trust the math, but I also know we can make mistakes. Let's use the computer as a referee."

Zahra is just here for the drama:

> "So if the dice don't match the formula, do we get to flame Amina's notebook?"

This chapter is about that tension:

- We already know how to count outcomes using permutations, combinations, and stars & bars.
- Now we tie that counting to *probability*.
- Then we let Python run huge numbers of random experiments to see whether reality agrees with our formulas.

The slogan for the chapter:

Counting gives us exact answers. Simulation lets us test and explore them.

5.2 From Counting to Probability

We start with a very simple but powerful idea.

Definition: Equally Likely Outcomes

Suppose an experiment (like rolling dice, flipping coins, or drawing a random username) has a finite set Ω of outcomes, and we assume each outcome is equally likely. If $E \subseteq \Omega$ is an event (the set of "good" outcomes), then

$$P(E) = \frac{|E|}{|\Omega|}.$$

So every time we say "What is the probability that…," we are really asking:

How many outcomes make this thing happen, divided by how many outcomes are possible at all?

Example: Exactly 3 Heads in 5 Flips

Flip a fair coin 5 times.

- Each outcome is a string like HHTHT.
- There are 2 choices (H or T) for each of the 5 positions.
- By the product principle,

$$|\Omega| = 2^5 = 32.$$

Let E be the event "exactly 3 of the 5 flips are heads." To choose such an outcome:

- Choose which 3 of the 5 positions are H.
- The remaining 2 positions are automatically T.

So the number of favorable outcomes is

$$|E| = \binom{5}{3} = 10,$$

and

$$P(E) = \frac{|E|}{|\Omega|} = \frac{10}{32} = \frac{5}{16} \approx 0.3125.$$

Amina would write this down calmly. Jake would start flipping coins.

Example: At Least One Six in Four Rolls

Roll a fair six-sided die 4 times.

- Each outcome is a sequence like (2, 6, 3, 6).
- Each roll has 6 options, so

$$|\Omega| = 6^4.$$

We want the event E: "at least one 6 appears."
It is often easier to count the complement:

- Let E^c be "no sixes at all."

- A roll with no six has only 5 possible values: 1, 2, 3, 4, 5.
- So

$$|E^c| = 5^4.$$

Thus

$$P(E) = 1 - P(E^c) = 1 - \frac{|E^c|}{|\Omega|} = 1 - \frac{5^4}{6^4}.$$

If you like numbers,

$$\frac{5^4}{6^4} = \frac{625}{1296} \approx 0.482 \quad \text{so} \quad P(E) \approx 1 - 0.482 = 0.518.$$

So in four rolls, it is a little more likely than not that we see at least one 6, but far from guaranteed. Perfect fodder for an argument at the table.

Connecting Back to Counting Worlds

Notice how this uses everything we've been doing:

- Counting all possible strings of coin flips or dice rolls (product principle).
- Counting subsets of positions (combinations).
- Sometimes using complements to avoid messy direct counting.

Once we can count, the jump to probability is literally one division step:

$$\text{probability} = \frac{\text{good outcomes}}{\text{all outcomes}}.$$

The formulas are beautiful. Now we ask: *does reality agree?*

5.3 Python Lab: Monte Carlo vs Exact

In this lab, Python plays the role of a moody universe simulator.

We will:

1. Compute an exact probability using counting, as above.
2. Use Python to run the experiment many times at random.
3. Compare the theoretical probability to the simulated frequency.

Simulating Coin Flips

Let's revisit "exactly 3 heads in 5 flips." The exact probability is $\frac{5}{16}$.

Here is a simple simulation:

Listing 5.1: Monte Carlo for exactly 3 heads in 5 flips

```
import random

def flip_coin():
    """Return 'H' or 'T' with equal probability."""
    return 'H' if random.random() < 0.5 else 'T'

def run_trial():
    """Simulate 5 flips and return True if we see exactly 3 heads."""
    flips = [flip_coin() for _ in range(5)]
    num_heads = flips.count('H')
    return (num_heads == 3)

def estimate_probability(num_trials):
    """Run many trials and estimate the probability."""
    successes = 0
    for _ in range(num_trials):
        if run_trial():
            successes += 1
    return successes / num_trials

for N in [10, 100, 1000, 10000]:
    print(N, estimate_probability(N))
```

What you should see:

- For $N = 10$: the estimate might be something wild like 0.2 or 0.4.
- For $N = 100$: usually closer, maybe around 0.31.
- For $N = 10{,}000$: often very close to 0.3125.

The code does not know any math. It just flips coins over and over and keeps score.

Simulating Dice: At Least One Six

Now let's simulate "at least one 6 in four rolls" and compare to

$$P(\text{at least one 6}) = 1 - \frac{5^4}{6^4} \approx 0.518.$$

Listing 5.2: Monte Carlo for at least one 6 in 4 rolls

```
import random

def roll_die():
    """Return an integer from 1 to 6."""
    return random.randint(1, 6)

def at_least_one_six():
    """Roll a die 4 times and check if at least one roll is 6."""
    rolls = [roll_die() for _ in range(4)]
    return 6 in rolls
```

```
def estimate_probability(num_trials):
    successes = 0
    for _ in range(num_trials):
        if at_least_one_six():
            successes += 1
    return successes / num_trials

for N in [10, 100, 1000, 10000, 100000]:
    print(N, estimate_probability(N))
```

Again, for large N, the printed estimates should hover close to the theoretical value.

Your Turn: Custom Experiments

Here are some variations students can try:

- Exactly two 6's in 8 rolls.
- No repeated value in 4 rolls (all distinct results).
- A simple username rule: randomly generate usernames and estimate the probability that a random username contains at least one digit.

For each variation, the workflow is:

1. Write down the combinatorial model: what is the sample space? what counts as a success?
2. Compute the theoretical probability using counting techniques.
3. Write or adapt a Python simulation to estimate that probability.
4. Compare and discuss.

5.4 Interpreting the Results

Simulation outputs *numbers*. We want to turn those numbers into *understanding*.

Why Does the Estimate Bounce Around?

When you run a simulation with a small number of trials (say $N = 10$), the estimate can be pretty far off. This is not the universe "breaking" math; it is just randomness.

- Each run of the experiment produces random results.
- With only a few trials, the random noise dominates the signal.
- As N grows, the average smooths out the randomness.

This is an informal peek at the *law of large numbers*: with many independent trials, the simulated frequency tends to drift toward the true probability.

When is Simulation Easier Than Counting?

Counting can get hard quickly:

- Complicated constraints ("no two sixes next to each other and total sum at least 15").
- Huge sample spaces where it's painful to reason about every type of outcome.

Simulation, on the other hand:

- Only needs us to know how to generate random outcomes and check a condition.
- Can handle weird, messy, real-world rules.
- Gives approximate answers that improve as we increase the number of trials.

However, simulation *never* proves that a formula is correct. It only gives evidence. Exact counting is still the gold standard when it is feasible.

Math, Code, and Reality

In this chapter, students should come away with three overlapping perspectives:

1. **Theoretical:** counting-based formulas for probabilities.
2. **Computational:** Monte Carlo simulations that test those formulas.
3. **Conceptual:** understanding why large samples give better estimates.

The best questions to ask in class:

- "What did we assume when we did the counting?"
- "Does our simulation actually match those assumptions?"
- "If they disagree, is it the formula, the code, or our interpretation that is wrong?"

Podcast: Episode 4 – The Dice Don't Lie (Much)

Cast: Amina (math brain), Jake (simulation gremlin), Lin (mediator), Zahra (agent of chaos).

Possible beats for the episode:

- Cold open: Jake rolling dice loudly while Amina complains that the sample size is too small.
- A quick recap that probability is "good outcomes over all outcomes" when things are equally likely.
- Amina presents a careful calculation; Jake counters with some early simulation results that look very different.
- Lin forces them to increase the number of trials; the estimates start drifting toward the theoretical value.

- Zahra suggests a wild, messy scenario (weird house rules, custom dice, or cursed usernames) where counting is hard but simulation is easy.
- Closing reflection: math gives us a map; simulation lets us walk around and see if the map fits the territory.
- Teaser for the capstone: “If we can simulate dice, what about whole universes?”

Chapter 6

Design Your Own Universe

6.1 Project Brief

By this point in the course, we have seen permutations, combinations, stars-and-bars distributions, and probabilities built from counting. We have also touched Python scripts that help us explore and verify our work. In this capstone you will flip the script:

Your mission: design a small "universe of possibilities" that you actually care about, and then analyze it with the tools from this chapter.

Your universe might be

- a card game or deck-building game,
- an RPG character builder or loadout system,
- a resource distribution system (skill points, credits, crafting mats),
- or anything else where there are many possible outcomes and choices.

To keep the project focused, your universe must include at least:

- **One permutation-based situation** where order matters (for example, initiative order, turn order, or a queue).
- **One combination-based situation** where order does not matter (such as choosing a team, a hand of cards, or a set of items).
- **One stars-and-bars style distribution** of identical things into distinct buckets (for example, skill points into stats, scoops into flavors, or coins into piggy banks).
- **At least one probability** that you can compute exactly from counts (for example, the probability of drawing a certain hand, or getting a certain type of build).

Your final deliverable will be a blend of story and math: a short write-up explaining your universe, a set of clearly stated counting questions, their solutions, and a small Python component that checks at least one of your answers by brute force or simulation.

6.2 Planning the Universe

Before you rush into formulas, spend time making your universe fun and coherent. Use these questions as planning prompts; sketch answers in words or pictures before turning anything into symbols.

Who and what lives in your universe?

Describe the basic ingredients:

- **Characters or agents:** players, heroes, monsters, robots, students, NPCs, ...
- **Items or resources:** cards, weapons, spells, skill points, coins, tokens, ...
- **Actions or choices:** drawing a hand, choosing a team, allocating points, rolling dice, selecting a username, ...

Write this part so that someone who has *not* taken the class could still understand the story.

What are the important decisions?

Think about where combinatorics naturally appears. For each core decision in your universe, ask:

- What is being chosen or arranged?
- Are the objects *distinct* (like named players or unique cards) or *identical* (like generic coins or hit points)?
- Does the *order* of the choice matter?
- Are there any constraints? (At least one wizard, no more than two legendaries, exactly three scoops, ...)

Jot each decision down in a little table such as:

Decision	Distinct/Identical?	Order?	Constraints?
Team of heroes	distinct	order doesn't matter	at least one healer
Turn order	distinct	order matters	all heroes used once
Mana points	identical	N/A	total of 10 points

Matching decisions to counting tools

Now connect each decision to a counting method:

- **Permutations** for order-sensitive arrangements of distinct objects.
- **Combinations** for order-insensitive selections of distinct objects.
- **Stars and bars** for distributing identical objects into labeled boxes.
- **Product rule** and **sum rule** to glue simpler counts together.

You do not need to cram *every* topic into *every* decision. It is better to have a few well-chosen situations that clearly fit one or two methods than a giant tangle of formulas that no one (including you) enjoys.

6.3 Mathematical Analysis

Once your universe is sketched, it is time to zoom in on the math. Choose a small number of core questions (three to five is typical) and analyze them in detail. For each question:

1. State the problem clearly

Write a short, precise statement that connects story language to math. For example:

> "In my card game there are 12 different spell cards and 8 different item cards. At the start of the game, each player gets a hand of 5 cards drawn without replacement from the 20-card deck. How many different 5-card hands are possible?"

Then translate this into pure math language, for example:

> "We are choosing a 5-element subset from a 20-element set, so we want $\binom{20}{5}$."

2. Choose and justify a method

For each problem, name the method you are using and justify briefly:

- "We use *permutations* because the order of the players in the initiative track matters."
- "We use *combinations* because only which heroes you pick matters, not the order we list them."
- "We use *stars and bars* because we are distributing identical points into labeled stats."

This does not need to be a full formal proof, but it should show that you know why your formula is appropriate.

3. Compute concrete answers

Give actual numbers, not just formulas. If the numbers are enormous, it is fine to use Python or a calculator, but:

- show the combinatorial expression (like $\binom{20}{5}$ or $6!/2!$),
- then give the numerical value (like 15,504).

If two different methods give the same count, say so, and briefly explain why they must agree.

4. Build at least one probability

Pick at least one event in your universe and compute its probability using the formula

$$P(E) = \frac{|E|}{|\Omega|},$$

where Ω is the set of all equally likely outcomes and E is the event you care about.

Examples:

- Probability a random hand has exactly two healers,
- Probability a randomly generated character has total strength at least 10,
- Probability a randomly formed team includes a specific character.

Whenever possible, say in words what your answer means: "So there is roughly an 18% chance that a random hand has at least one legendary card."

6.4 Python Component

The Python part of the project lets you check that your formulas match reality (or at least simulated reality). It does not need to be long or fancy, but it does need to be *connected* to your universe.

Minimum requirements

Your Python code should:

- **Compute at least one count or probability using formulas**, making use of tools like `math.factorial` and `math.comb` (or your own helper functions).
- **Verify at least one result** by:
  - brute-force listing of all possibilities (for very small cases), or
  - Monte Carlo simulation (randomly sample many times and estimate a probability).
- **Print a short, human-readable summary** such as

```
Exact probability: 0.1826
Simulated probability after 100000 trials: 0.1819
```

You are encouraged to recycle patterns from earlier labs:

- Using `itertools.permutations` and `itertools.combinations` for exact counts,
- Using loops and the `random` module for Monte Carlo experiments,
- Structuring your code into small functions so your main script reads like a story of what you are doing.

Nice-to-have features

If you have time and interest, you might:

- allow the number of players, cards, or points to be parameters,
- add a function that generates a random valid team, loadout, or hand,
- print a few example outcomes so readers can see what a typical random result looks like.

Keep in mind that the goal is not to build a video game—it is to *show off the math* with a bit of computational support.

6.5 Presentations and Reflection

At the end of the project, you will share your universe with others. The format may be a short written report, a brief in-class talk, a poster, or some combination, depending on your instructor.

What to share

Your presentation should include:

- **A story summary** of your universe in plain language.
- **Your key counting challenges** and how you solved them: what was being counted, what method you used, and what you found.
- **At least one probability** and a brief explanation of how you got it.
- **One surprising or interesting discovery** from coding or simulating. For example:
  - the simulated probability took a long time to stabilize,
  - a built-in intuition ("this almost never happens") turned out to be wrong,
  - two different-looking formulas gave exactly the same number.

Reflection prompts

To wrap up, write a short reflection (a few paragraphs) addressing questions like:

- Where did combinatorics make your life easier?
- Where did it get messy? How did you manage that complexity?
- If you were to extend your universe, what new counting or probability questions would appear?
- How might someone in game design, security, data science, or another field use similar counting ideas in the real world?

Honest reflections (including "this part was surprisingly hard") are more valuable than pretending that every step was smooth and obvious.

Podcast: Episode 5 – Universe Builders

This final episode of the podcast closes the arc of the course.

Script notes (suggested):

- The cast interviews each other about their favorite student-style universes: card games, RPG worlds, security systems, classroom seating chaos, and more.
- Each character describes one counting challenge they faced and how they solved it, in informal language: "At first I thought I had to list everything by hand, but then I realized it was just a combination problem."
- They play with the idea that behind every "simple" game rule there might be a huge invisible combinatorial space.

- They revisit earlier themes:
  - factorial explosions from seating at the round table,
  - username spaces and noisy strings,
  - sundaes and resource distributions,
  - dice, probabilities, and simulations.
- The episode closes with the big-picture message: counting is how we tame spaces of possibilities that feel infinite, and how we design systems—games, passwords, experiments, universes—on purpose rather than by accident.

You have now built and analyzed a universe of your own. The same tools will follow you into algorithms, data structures, probability, statistics, and any field where we care about what can happen, how often, and why.

Chapter 7

Tiny Tactics Arena: A Fully Counted Universe

7.1 Story Hook: Balancing a Mini-Game

You have been hired as the *official mathematician* for a tiny tactics game called *Tiny Tactics Arena*.

The game designers are worried:

- "Do we have enough variety, or will players see the same teams over and over?"
- "Are some builds secretly way more common than others?"
- "If we tweak a rule, how much does it shrink or grow the universe of possibilities?"

Your mission in this chapter is to *fully count* a small, self-contained game universe:

- who the heroes are,
- how they act (turn order),
- what artifacts they bring,
- how they distribute skill points,
- and a few simple probabilities.

Along the way, you will use:

- permutations (order matters),
- combinations (order does not),
- stars-and-bars (distributing identical points),
- and Python to sanity-check your math.

7.2 The Rules of Tiny Tactics Arena

We begin with a deliberately small rule set.

Basic ingredients

- There are exactly 3 hero classes:

  $$\text{Warrior (W)}, \quad \text{Mage (M)}, \quad \text{Rogue (R)}.$$

- Each team brings *one hero of each class*. So every team has three heroes: W, M, and R.
- There are 4 artifacts available:

  $$\text{Sword (S)}, \quad \text{Staff (T)}, \quad \text{Dagger (D)}, \quad \text{Shield (H)}.$$

  Artifacts are all distinct.
- At the start of a match, the team chooses *exactly 2* different artifacts to bring. For now, we do not care who holds which artifact.
- Each hero has 5 skill points to distribute among three stats: STR (Strength), DEX (Dexterity), and INT (Intellect). Skill points are indistinguishable: they are just little pips.

Game state we will count

To keep things focused, define a *configuration* to consist of:

1. An *initiative order* of the three heroes (who goes first, second, third).
2. A *set of 2 artifacts* chosen from the 4 available.
3. A *skill build* for each hero (how each hero distributes 5 points among STR, DEX, INT).

In this chapter we will:

- Count each piece separately,
- Assemble them into a total count for all configurations,
- Write Python code to confirm that our counting matches reality for the tiny universe.

7.3 Permutations: Initiative Order

The three heroes W, M, and R will act in some order each round.

Counting the orders

An *initiative order* is simply a permutation of the set $\{W, M, R\}$.

- There are 3 choices for who goes first.
- Then 2 choices remain for who goes second.
- Then 1 choice remains for who goes third.

By the product rule:

$$3 \cdot 2 \cdot 1 = 3! = 6$$

possible initiative orders.

Example orders: (W, M, R), (M, W, R), (R, M, W), and so on.

Here, *order matters*: (W, M, R) is not the same as (M, W, R), because different heroes act first.

7.4 Combinations: Choosing Artifacts

The team must choose exactly 2 distinct artifacts from the 4 available: $\{S, T, D, H\}$.

Order does not matter

If we only care which two artifacts are brought, and we do not care which one we list first (or who holds them), this is a combinations problem.

We are choosing 2 objects out of 4, without repetition, and without order:

$$\binom{4}{2} = \frac{4!}{2! \cdot 2!} = 6.$$

We can even list them all:

$$\{S, T\},\ \{S, D\},\ \{S, H\},\ \{T, D\},\ \{T, H\},\ \{D, H\}.$$

Each of these sets is one possible artifact loadout for the squad.

7.5 Stars and Bars: Skill Trees for a Single Hero

Now consider a single hero, for example the Mage.

Distributing skill points

The Mage has 5 identical skill points to distribute among three stats: STR, DEX, INT.

Let

$$x_{\text{STR}}, \quad x_{\text{DEX}}, \quad x_{\text{INT}}$$

be the number of points in each stat.

We are looking at integer solutions to

$$x_{\text{STR}} + x_{\text{DEX}} + x_{\text{INT}} = 5,$$

with

$$x_{\text{STR}}, x_{\text{DEX}}, x_{\text{INT}} \geq 0.$$

This is a classic *stars-and-bars* situation: we place 5 stars into 3 bins.

The number of solutions is

$$\binom{5+3-1}{3-1} = \binom{7}{2} = 21.$$

Each solution corresponds to one skill build. For example:

- $(5, 0, 0)$ means full Strength,
- $(3, 2, 0)$ means 3 STR, 2 DEX, 0 INT,
- $(1, 1, 3)$ means 1 STR, 1 DEX, 3 INT.

All three heroes

In our universe, *each* hero independently distributes 5 points among the 3 stats.

So, if each hero has 21 possible skill builds, and order of heroes is fixed as (W, M, R) *for the purpose of counting builds*, then:

$$\text{number of skill configurations for all 3 heroes} = 21 \cdot 21 \cdot 21 = 21^3.$$

We will leave the number as 21^3 for now, and multiply everything together in the next section.

7.6 Putting It All Together

We are ready to count full configurations:

- Initiative order of the three heroes,
- Set of 2 artifacts,
- Skill builds for all three heroes.

Step-by-step product

From previous sections:

- Initiative orders: $3! = 6$.
- Artifact sets: $\binom{4}{2} = 6$.
- Hero skill configurations: each hero has 21 possibilities, so 21^3 for the trio.

Assuming all of these choices are made independently, the product rule tells us the total number of configurations is:

$$\underbrace{3!}_{\text{initiative}} \times \underbrace{\binom{4}{2}}_{\text{artifacts}} \times \underbrace{21^3}_{\text{skills}} = 6 \times 6 \times 21^3.$$

If you really want the raw integer:

$$21^2 = 441, \quad 21^3 = 21 \cdot 441 = 9261,$$

so

$$6 \times 6 \times 9261 = 36 \times 9261.$$

One way:

$$9261 \cdot 30 = 277{,}830, \quad 9261 \cdot 6 = 55{,}566,$$

so

$$277{,}830 + 55{,}566 = 333{,}396.$$

Thus our tiny universe already has

$$333{,}396$$

different configurations!

Even with only three heroes, four artifacts, and 5 skill points each, we are way past the point where "just listing them all" is pleasant.

7.7 Python Lab: Brute-Forcing the Arena

Now we will write a short Python script to:

1. Generate all initiative orders,
2. Generate all artifact sets,
3. Generate all skill builds for a single hero,
4. Combine them and count how many full configurations we get,
5. Check that the count matches our formula $3! \cdot \binom{4}{2} \cdot 21^3$.

Generating skill builds

We can generate the $(x_{\text{STR}}, x_{\text{DEX}}, x_{\text{INT}})$ triples for a single hero using simple loops.

Listing 7.1: Generating all skill builds for one hero

```
def skill_builds(points=5):
    """Return all (STR, DEX, INT) triples that sum to ‘points‘."""
    builds = []
    for str_pts in range(points + 1):
        for dex_pts in range(points + 1 - str_pts):
            int_pts = points - str_pts - dex_pts
            builds.append((str_pts, dex_pts, int_pts))
    return builds
```

You can quickly check:

```
>>> len(skill_builds(5))
21
```

which matches our stars-and-bars count.

Putting the whole universe together

Here is a complete script for Tiny Tactics Arena.

Listing 7.2: Brute-force Tiny Tactics Arena

```
import itertools
from math import comb, factorial

CLASSES = ["W", "M", "R"]
ARTIFACTS = ["S", "T", "D", "H"]

def skill_builds(points=5):
    builds = []
    for str_pts in range(points + 1):
        for dex_pts in range(points + 1 - str_pts):
            int_pts = points - str_pts - dex_pts
            builds.append((str_pts, dex_pts, int_pts))
```

```
    return builds

def main():
    # 1. Initiative orders (permutations of W, M, R)
    initiatives = list(itertools.permutations(CLASSES))
    print("Number of initiative orders:", len(initiatives))

    # 2. Artifact sets: choose 2 out of 4 (combinations)
    artifact_sets = list(itertools.combinations(ARTIFACTS, 2))
    print("Number of artifact sets:", len(artifact_sets))

    # 3. Skill builds for a single hero
    hero_builds = skill_builds(points=5)
    print("Number of builds for one hero:", len(hero_builds))

    # 4. Skill configurations for all three heroes
    triple_builds = list(itertools.product(hero_builds, repeat=3))
    print("Number of builds for three heroes:", len(triple_builds))

    # 5. Combine everything into full configurations
    count = 0
    for init in initiatives:
        for artifacts in artifact_sets:
            for builds in triple_builds:
                # builds is a triple:
                # (build_for_W, build_for_M, build_for_R)
                count += 1

    print("Total configurations (brute force):", count)

    # 6. Compare with the formula:
    formula = factorial(3) * comb(4, 2) * (len(hero_builds) ** 3)
    print("Total from formula:", formula)


    if count == formula:
        print("Counts match! Our combinatorics is consistent.")
    else:
        print("Mismatch -- something is off somewhere.")


if __name__ == "__main__":
    main()
```

This script is intentionally straightforward (and not optimized). On a modern machine, it should happily crunch through the 333,396 configurations.

7.8 Probabilities in the Arena

Once we can count configurations, we can also talk about probabilities.

To keep things simple, suppose:

- Each team configuration is chosen uniformly at random from the 333,396 possibilities.

We can ask questions like:

- What is the probability that the Mage acts first?
- What is the probability that the team brings a Shield?
- What is the probability that the Rogue is "glass cannon" (0 points in STR and 0 points in DEX)?

We will do just one example here.

Example: probability the Mage acts first

How many configurations have the Mage acting first?

- Fix Mage first. Then the other two heroes (W and R) can be arranged in 2 ways: WR or RW. So there are 2 initiative orders with Mage first.
- The rest of the choices (artifact sets, skill builds) do not depend on this detail. For each initiative order, there are $\binom{4}{2} \cdot 21^3$ ways to choose artifacts and skills.

So the number of configurations with Mage first is

$$2 \times \binom{4}{2} \times 21^3.$$

The probability is

$$\frac{2 \times \binom{4}{2} \times 21^3}{3! \times \binom{4}{2} \times 21^3} = \frac{2}{6} = \frac{1}{3}.$$

This matches our intuition: all three heroes are symmetric in the rules, so each should be first one-third of the time.

7.9 Design Variations and Chaos Modes

Once the basic universe is built, it is easy (and fun) to mutate it.

Here are some design prompts:

- **Add more heroes:** What happens if you add a fourth class (Healer)? How does that change the initiative permutations and total configuration count?
- **Artifact limits:** What if the team can bring 3 artifacts instead of 2? Or what if certain classes are not allowed to use some artifacts?
- **Skill caps:** You could require that no stat may exceed 3 points. How many skill builds are left for a hero? (This becomes a stars-and-bars problem with upper bounds.)
- **Random generation:** Write Python code that generates random valid configurations, and then estimates the probability of some condition (for example, "Mage acts first and has at least 3 INT"). Compare the simulated probability to the exact one.

Your challenge: take this tiny tactics universe, tweak one rule, and then *track exactly how the counts change*. That is exactly what game designers, security engineers, and combinatorics fans do in the real world.

Podcast: Episode 6 – Live from the Arena