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

Chapter 6

Design Your Own Universe

6.1 Project Brief

6.2 Planning the Universe

6.3 Mathematical Analysis

6.4 Python Component

6.5 Presentations and Reflection

Podcast: Episode 5 – Universe Builders