Counting Worlds

Example Solution: Universe Design Project

Instructor Example

November 12, 2025

Contents

Chapter 00: Universe Project Assignment

Discrete Structures: Counting Worlds Capstone

Chapter 5 Project – Design Your Own Universe

Overview

In Chapter 5 you are asked to *design a small universe* (game, system, or scenario) and show that you can *count* what lives inside it.

Your universe could be:

- a small card game or draft system,
- a character builder with races, classes, and loadouts,
- a squad / team builder with roles and items,
- a resource or skill point distribution system,
- or any other world where you make structured choices.

The core idea: you tell a story, then you expose the combinatorics behind it.

Pair programming: You will work in pairs. Both partners are responsible for understanding the math and the code, and both names go on all deliverables.

Project Requirements (Mathematics)

Your universe must include at least:

1. **One permutation situation** (order matters).
   Example: ordering players in a queue, arranging cards, turn order, seat order.
2. **One combination situation** (order does not matter).
   Example: choosing a team of heroes, selecting a hand of cards, picking a loadout.
3. **One stars-and-bars style distribution**.
   Example: distributing points among stats, skill points among abilities, resources among locations, scoops among flavors.
4. **At least one probability question** that you compute from counts.
   Example: probability of drawing a certain type of hand, probability that a random build satisfies a constraint, etc.

For each of these, you should:

- State the problem clearly in words.
- Translate it into symbols (for example, "choose k from n", or "number of integer solutions to $x_1 + \ldots + x_k = n$").
- Name the relevant tool (product rule, sum rule, permutation, combination, stars-and-bars).
- Show the formula and at least one worked example with numbers.

Project Requirements (Python)

Write at least one short Python script (more is fine) that interacts with your universe.

Minimum expectations:

- Use formulas (`math.factorial`, `math.comb`, or your own functions) to compute at least one important count in your universe.
- For a small version of your universe:
  - either *enumerate* all possibilities (for example, all teams of size 3),
  - or run a *Monte Carlo simulation* to estimate a probability.
- Compare the Python result to your theoretical count or probability and comment on whether they agree (and why they might differ for small simulations).

Optional AI / ML extension (extra credit / enrichment):

- Define a simple score for builds, teams, or states in your universe.
- Use search or sampling to find "good" builds (for example, random search, hill-climbing, or simple Monte Carlo rollouts).
- If you are curious, you may treat builds as feature vectors and try a small model that predicts which builds will be strong, but this is not required.

Deliverables Checklist

Each pair submits the following (one submission per pair):

☐ **Universe description** (1–2 pages). Story, rules, what choices players or users make, and any constraints.

☐ **Math writeup** (1–2 pages). For each required situation (permutation, combination, stars-and-bars, probability): clearly labeled problems, formulas, and worked examples.

☐ **Python code**. At least one script with comments, using formulas and either enumeration or simulation. Include a short text summary of what you learned from running the code.

☐ **Reflection paragraph** (about half a page). Answer prompts like:

- What surprised you about the size of your universe?
- Where did the counting get messy, and how did you handle that?
- How might someone use counting or simple AI tools to balance or explore your universe?

Pair Programming Guidelines

During your work sessions, you should intentionally practice pair programming.

Driver Has hands on the keyboard. Types the code, edits the LaTeX, and narrates what they are doing.

Navigator Watches for bugs, asks "why" questions, and thinks ahead:

- Does this match the math model?
- Are we naming variables clearly?
- Are we testing the interesting cases?

Good practice:

- Switch roles regularly (every 10–15 minutes, or at natural breakpoints).
- Both partners must be able to explain every part of the code and the math.
- If you disagree, pause and explain your reasoning in words before changing code.

I will be looking for evidence that you both contributed and that you can both talk about the universe, the counting, and the code.

Grading Rubric (Guide)

This project will be graded using roughly the following criteria (example point weights in parentheses, adjust as needed):

Criterion	**Strong Performance**	**Needs Improvement**
Universe design (0–8)	Universe is clear, coherent, and interesting. Rules are well specified and easy to imagine playing.	Universe is vague, inconsistent, or very small; rules are hard to follow or incomplete.
Counting correctness (0–10)	Permutation, combination, and stars-and-bars problems are clearly stated; correct formulas and computations with explanations.	Frequent mistakes in identifying or applying formulas; explanations missing or unclear.
Use of multiple tools (0–6)	All required structures (permutation, combination, stars-and-bars, probability) appear naturally in the universe and are well motivated.	One or more required structures missing, forced, or not clearly connected to the story.
Python component (0–8)	Code is clean, commented, runs successfully, and clearly connects to the math; comparison between theory and experiment is explained.	Code is incomplete, hard to read, does not run, or has a weak connection to the math; little or no analysis of results.
Communication and reflection (0–8)	Writing is organized and readable. Reflection shows genuine insight into counting, complexity, or possible AI uses. Both partners can explain the work.	Writing is disorganized or very brief. Reflection is superficial. It is unclear whether both partners understand the project.

Total suggested: 40 points. I may also award small bonus credit for creative, well-executed AI or simulation extensions.

Chapter 1

Tiny Tactics Arena: Universe Overview and Design Brief

1.1 Story Hook: Welcome to the Arena

You and your friend have accidentally invented a tactics game.

It started as a joke in the Discord:

> "What if we made a tiny turn-based arena where every character is wildly unbalanced, but we use math to pretend it is fair?"

Two hours later, there is:

- a roster of heroes (some cool, some cursed),
- a pile of artifacts with questionable side effects,
- a skill tree that looks like it was designed at 3am,
- and a spreadsheet named `balance_vFINAL_final_ACTUAL.xlsx`.

This chapter is where we stop guessing and actually *specify* the universe we are playing in. We will give it a name:

Tiny Tactics Arena

and we will turn the chaos into symbols, parameters, and precise questions. If someone reads only this chapter, they should understand:

- what objects exist in our world (heroes, artifacts, skills, matches),
- what can be customized by a player,
- where the math lives (permutations, combinations, stars-and-bars, probability),
- and what code we will eventually write to explore and analyze the game.

1.2 Entities and Parameters

At the heart of Tiny Tactics Arena are a few types of objects.

Heroes

We have a roster of heroes. Each hero has:

- a name (for flavor: "Nova", "Glitch", "Tanktop Tim"),
- a *role* (e.g., damage, support, tank),
- a *skill tree* they can invest points into.

We will treat the roster size as a parameter:

$$H = \text{number of available heroes in the full roster.}$$

Teams

A player enters a match with a team of fixed size:

$$T = \text{number of heroes on a team.}$$

For example, in a "3v3" mode, $T = 3$.

Sometimes we will care only which heroes are on the team (as a *set* of size T). Other times, we will care about the *order* in which they act (initiative order).

Artifacts

Artifacts are items that can be equipped before a match:

- Each artifact has a name and effect ("+2 damage", "double heal", "50% chance to explode").
- A player can bring up to $A_{\max}$ artifacts into a match.

We let:

$$A = \text{number of distinct artifact types in the game,}$$

and

$$A_{\max} = \text{maximum number of artifacts a team can equip.}$$

We will consider simple rules first:

- At most one copy of each artifact.
- Order of artifacts does not matter (you either bring it or you do not).

Skill Points

Each hero has a small skill tree. To keep things combinatorially friendly, we will say:

- A hero has k skill lines (for example: attack, defense, utility).
- Each hero gets S total skill points to distribute across those lines.

We let:

$$k = \text{number of skill lines for a hero,} \qquad S = \text{total skill points available for that hero.}$$

The skill tree, for counting purposes, is just a way to distribute S identical points into k labeled buckets (lines).

Matches and Randomness

A match consists of:

- Two teams (you vs. opponent), each of size T.
- An initiative order telling us who acts first, second, etc.
- Random events like critical hits, misses, or random artifact effects.

We will model some of this randomness in a very simplified way (e.g., each attack hits with probability p, crits with probability q, etc.) so that we can:

- compute exact probabilities in small toy versions, and
- simulate matches using Monte Carlo in Python.

1.3 Core Mechanics in Math-Friendly Language

Now we strip away the flavor and look at the *combinatorial skeleton* of the game.

Teams as Sets or Sequences

A team configuration can be viewed in two ways:

1. As a **set** of T heroes chosen from H: order does not matter, only who is on the team.
2. As a **sequence** (permutation) of T heroes: we care about the order in which they act.

This means we will use:

- *combinations* to count the number of possible teams,
- *permutations* to count initiative orders and lineups.

Artifact Loadouts

An artifact loadout is:

- a subset of the A artifact types,
- with size at most $A_{\max}$,
- where order does not matter.

So loadouts are also counted with combinations, although we may have to add up several cases (size 0, 1, 2, ...)

In more advanced versions of the project, you could allow:

- multiple copies of the same artifact (a multiset),
- per-hero artifact slots (combinatorics inside combinatorics),
- caps or categories (at most one "legendary", etc.).

Skill Allocations as Stars and Bars

Skill point allocations are a perfect place to use the *stars and bars* technique.

We imagine:

- S identical "stars" (skill points),
- k labeled "buckets" (skill lines),
- and we count how many ways to drop the S stars into k buckets.

This is exactly the classic stars-and-bars setup you see in counting problems:

$$\text{number of allocations} = \binom{S+k-1}{k-1},$$

under the assumption that each skill line can receive zero or more points.

We can also add constraints:

- minimum points in some line (e.g., at least 1 point in defense),
- caps on lines (e.g., at most 3 points in any single line).

From Counting to Probability

Once we know how many configurations or outcomes there are, we can ask questions like:

- What is the probability that a randomly generated team has at least one healer?
- What is the probability that a randomly chosen artifact loadout includes a particular legendary item?
- Under a simple model of combat, what is the probability that a fight ends in 3 turns or fewer?

For small cases, we will compute probabilities exactly using:

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

where Ω is the sample space of all equally likely outcomes and E is the event we care about.

For larger cases, we will use Python to approximate $P(E)$ by simulation: run many matches, count how often E happens, and look at the relative frequency.

1.4 Required Math Questions for This Universe

To keep the project focused, Tiny Tactics Arena is designed to support at least the following five mathematical questions.

1. Permutation Question

Example prompt: Given a team of T distinct heroes, in how many ways can we order them in an initiative track from first to Tth?

Possible extensions:

- Some heroes must act before others (partial order).
- One hero always acts last because they are slow but dramatic.

2. Combination Question

Example prompt: Given A distinct artifacts and a cap of $A_{\max}$ on how many a team can bring, in how many ways can a team choose its artifact loadout?

Variants:

- Exactly $A_{\max}$ artifacts, no fewer.
- At most one artifact of each type vs. allowing duplicates.
- Artifacts restricted by hero role (certain artifacts only usable by tanks, etc.).

3. Stars-and-Bars Question

Example prompt: A hero has k skill lines and S total skill points. In how many ways can we allocate those S points across the k lines?

Variants:

- Each line must receive at least one point.
- No line can receive more than some cap C.

4. Probability Question

Example prompt: Assume a very simple combat model: each attack has probability p of hitting and, on a hit, probability q of being a critical hit. For a fixed sequence of n attacks in a match, what is the probability that we see:

- exactly k hits,
- at least one critical hit,
- or some other event your story cares about?

You might use binomial coefficients here: the probability of exactly k hits in n independent trials is:

$$\binom{n}{k} p^k (1-p)^{n-k}.$$

5. AI / Machine Learning Question

Example prompt: We generate a large dataset of simulated matches under different team and artifact configurations. Can we:

- detect imbalanced heroes or artifacts,
- or train a simple classifier to predict which team is favored to win?

In the full project, this could involve:

- encoding matches as rows in a CSV file (`data/matches.csv`),
- using Python (and possibly `scikit-learn`) to fit a simple model,
- interpreting the learned parameters as "balance hints" for the game.

1.5 Planned Code Artifacts and File Outputs

To support these questions, we will create a small collection of Python scripts.
Each script will:

- live in the `scripts/` directory,
- take simple command-line arguments (e.g., `--heroes 6 --team-size 3`),
- print a short summary line to standard output,
- and write the interesting results to files in `data/`, `figures/`, or `files/`.

Here is the planned lineup.

Script 1: `turn_orders.py`

Purpose:

- Compute and/or enumerate possible initiative orders for a team.
- Optionally write all orders (for small T) to a CSV file (`data/turn_orders_T3.csv`, etc.).

Console output example:

```
Team size T=3:  6 possible initiative orders.  Data saved to data/turn_orders_T3.csv
```

Script 2: `artifact_loadouts.py`

Purpose:

- Count and (for small cases) list all artifact loadouts.
- Save loadouts to `data/artifact_loadouts_A5_Amax2.csv`.

Console output example:

```
A=5 artifacts, Amax=2:  16 possible loadouts.  Results in data/artifact_loadouts_A5_Amax2.
```

Script 3: `skill_allocations.py`

Purpose:

- Compute the number of ways to allocate S skill points into k lines.
- Optionally enumerate allocations for small S and k and write them to CSV.

Console output example:

```
k=3 lines, S=4 points:  15 allocations.  Sample saved to data/skill_allocations_k3_S4.csv
```

Script 4: `match_simulator.py`

Purpose:

- Simulate many Tiny Tactics Arena matches under a simplified combat model.
- Record key statistics (winner, turns taken, damage dealt, etc.).
- Write all simulated matches to `data/matches.csv`.

Console output example:

```
Simulated 10,000 matches.  Win rate:  Team A 57.2%, Team B 42.8%.  Data in
data/matches.csv
```

Script 5: `ai_balance_checker.py`

Purpose:

- Read `data/matches.csv`.
- Train a simple model (even just basic statistics at first) to detect:
  - which heroes or artifacts appear most often on winning teams,
  - whether some configuration is overwhelmingly strong.
- Write summary results to `files/balance_report.txt` and possibly plots to `figures/`.

Console output example:

```
Balance report written to files/balance_report.txt; hero win rates plot saved
to figures/hero_winrates.png
```

1.6 Project Roadmap and Expectations

This chapter is the "design doc" for your universe.

In later chapters, we will:

- turn each core question into:
  - a clean mathematical problem,
  - a worked-out solution with formulas and explanations,
  - and a small Python script that generates matching results;
- connect the pieces together into a coherent story about the game;
- optionally, push into AI/ML territory by analyzing the simulated data.

Think of it this way:

- The **math** tells you how big and complex your universe is.
- The **code** lets you actually wander around that universe, sample it, and collect data.
- The **story** is what keeps humans (including you) caring about the answers.

Checklist for Your Brain

As you read this and start planning your own version of the project, keep an eye on:

- Which parts of your universe are naturally modeled as:
  - permutations (order matters),
  - combinations (order does not),
  - stars-and-bars (distributing points/resources).
- Which questions are purely counting, and which turn into probabilities.
- What data you might want to generate and analyze from simulations.
- How you could tell a story around all of this ("hero balance council", "artifact nerf emergency", etc.).

If you can explain your game to a friend in under two minutes, and they can point to at least three places where "we should really count how many possibilities there are", you are in the right place.

Welcome to Tiny Tactics Arena. Time to start counting.

Chapter 2

Permutation Scenario — Turn Order Math

What we're solving (the setup)

We have a team of T distinct heroes. A *turn order* is an ordered list of those T heroes from first to Tth. Since order matters, this is a *permutation* problem: sequences, not sets.

Assignment link. This chapter addresses the "Turn Order / Initiative" counting task from the assignment: count all possible orders for a team; then handle restricted variants (e.g., someone fixed first, two heroes required to be adjacent, two heroes forbidden from being adjacent), and show small concrete cases.

2.1 Outcome space = permutations

Label the heroes $\{h_1, \ldots, h_T\}$. Any legal turn order is a sequence $(h_{\pi(1)}, h_{\pi(2)}, \ldots, h_{\pi(T)})$ where π is a permutation of $\{1, \ldots, T\}$. Therefore:

$$|\Omega| = T! \quad \text{(all orders equally likely in the pure-counting model).}$$

Tiny worked example: $T = 3$

Let the heroes be A, B, C. The $3! = 6$ possible orders are

$$ABC,\ ACB,\ BAC,\ BCA,\ CAB,\ CBA.$$

For students: it helps to *list* these once, then use the factorial rule afterwards.

2.2 Common restrictions (your spice rack)

The real fun is when we add constraints. Below are three patterns you'll reuse all over the course.

2.2.1 One hero fixed first ("Tank must act first")

If a specific hero is fixed in position 1, the remaining $T-1$ distinct heroes can be arranged arbitrarily:

$$\# = (T-1)!.$$

Check: For $T = 3$ with A fixed first, the orders are ABC, ACB — indeed $(3-1)! = 2$.

2.2.2 Two heroes must be adjacent ("BFFs together")

Treat the two adjacent heroes as a single *block*. There are two internal orders for that block, and there are $(T-1)!$ ways to arrange the block plus the other $T-2$ solo heroes:

$$\# = 2\,(T-1)!.$$

Check: For $T = 4$ and BFFs (A, B), valid sequences count to $2 \cdot 3! = 12$.

2.2.3 Two heroes may *not* be adjacent ("keep them apart")

Use complement counting.

$$\#(\text{not adjacent}) \;=\; T! \;-\; \#(\text{adjacent}).$$

We already know $\#(\text{adjacent}) = 2\,(T-1)!$, so

$$\#(\text{not adjacent}) = T! - 2\,(T-1)! = (T-1)!\,(T-2).$$

Check: For $T = 4$, that's $24 - 12 = 12$.

2.3 Partial orders / precedence constraints (leveled-up)

Sometimes you're told "X must act before Y" (but not necessarily adjacent). With distinct elements, exactly half of the $T!$ permutations put X before Y:

$$\# = \frac{T!}{2}.$$

For multiple constraints, you can:

a) collapse forced equalities (if any) into blocks and then permute, or

b) count linear extensions (harder in general), or

c) for a few constraints, use symmetry/conditioning.

2.4 Decision talk: set vs sequence (why this is permutations)

A quick habit: ask "Am I counting *sets* or *sequences*?" If it's sets, think *combinations*. If it's sequences (like initiative), think *permutations*. This avoids the classic "does order matter?" trap and lines up with the quotient/product principles you've seen.

2.5 Mini-exercises (with quick answers)

1. Team size $T = 5$. How many turn orders? *Answer:* $5! = 120$.
2. $T = 5$, hero Z must act first. *Answer:* $(5-1)! = 24$.
3. $T = 5$, heroes X and Y must be adjacent (in either order). *Answer:* $2\,(5-1)! = 48$.
4. $T = 5$, heroes X and Y may not be adjacent. *Answer:* $5! - 2\,(5-1)! = 120 - 48 = 72$.
5. $T = 6$, A must act before B (not necessarily adjacent). *Answer:* $\frac{6!}{2} = 360$.

2.6 What the code will verify (preview)

Later, a small script will:

- enumerate all orders for small T and dump them to `data/turn_orders_T#.csv`;
- filter by constraints (fixed-first, adjacent/not-adjacent pairs);
- print a one-line summary and save the real work to files.

Takeaway

Turn order is the *canonical* permutation arena. Master these three moves—fix, block, and complement—and you'll style on 90% of initiative puzzles.

Chapter 3

Permutation Scenario — Turn Order Code (Question 1)

3.1 Purpose and Big Picture

This is the code-focused companion to the turn-order mathematics. We turn the factorial idea ($T!$ permutations for a team of size T) into a small, replicable pipeline:

1. **Inputs** (team size, names, constraints).
2. **Compute/Enumerate** permutations and filter by constraints.
3. **Write outputs** to stable files (CSV, summary, and an optional figure).
4. **Document** everything here so the analysis is reproducible.

3.2 Script Overview

The script lives at `scripts/turn_orders.py`. It supports:

- `-T / -team-size` (required),
- `-names` (comma-separated; defaults to `A,B,C,...`),
- `-list-small` (enumerate only when $T \leq 8$; otherwise just report $T!$),
- `-must-before` (e.g. `A>B,B>C` means A acts before B, B before C),
- `-together` (groups that must be contiguous, e.g. `C+D` or `C+D+E`),
- `-apart` (groups that must *not* appear adjacent, e.g. `A+B`).

Outputs (files).

- `data/turn_orders_T{T}.csv` — when enumerating. Columns: `order_id`, `order`, `valid_must_before`, `valid_together`, `valid_apart`, `valid_all`.
- `files/turn_orders_summary.txt` — always written; parameters and counts.
- `figures/turn_orders_lead_freq_T{T}.png` — optional bar chart if `matplotlib` is available and we enumerated.

3.3 How to Run (examples)

```
# Tiny team: enumerate all
python scripts/turn_orders.py -T 3 --list-small

# Named heroes, precedence + block constraints, and enumerate
python scripts/turn_orders.py -T 4 \
  --names A,B,C,D \
  --must-before A>B,B>D \
  --together C+D \
  --list-small

# Larger T (no full enumeration): just report T!
python scripts/turn_orders.py -T 8
```

The console prints a single, quiet summary line. The real artifacts live in `data/`, `files/`, and `figures/` so students can re-use results later.

3.4 Design Choices (why this way?)

- **Separation of concerns.** The console stays minimal for demonstration, while durable results are written to files for grading, plotting, or sharing.
- **Safety for big T.** Full enumeration scales as $T!$; we cap enumeration to $T \leq 8$ unless you explicitly force it. Larger T uses the clean formula $T!$.
- **Composable constraints.** Precedence (`must-before`), adjacency (`together`), and non-adjacency (`apart`) cover many natural "house rules" in turn-based systems.

3.5 The Code (current version)

```
#!/usr/bin/env python3
"""
turn_orders.py - enumerate / count initiative orders (permutations) for a team.

Console output: a single summary line.
Files:
  data/turn_orders_T{T}.csv (only when enumerating)
  files/turn_orders_summary.txt (always)
  figures/turn_orders_lead_freq_T{T}.png (if matplotlib present and enumerating)

Examples:
  python scripts/turn_orders.py -T 3 --list-small
  python scripts/turn_orders.py -T 4 --names A,B,C,D --must-before A>B,B>D --together C+D
      --list-small
  python scripts/turn_orders.py -T 8 # count only (no full enumeration)
"""

import argparse
import csv
```

```
import itertools
import math
import os
import sys
from collections import Counter

# ------------------------------------------------------------
# Helpers
# ------------------------------------------------------------

def ensure_dirs():
    for d in ("data", "files", "figures"):
        os.makedirs(d, exist_ok=True)

def default_names(T: int):
    # A, B, C, ... then A1, B1... if we run out
    base = [chr(ord('A') + i) for i in range(min(T, 26))]
    if T <= 26:
        return base
    names = base[:]
    i = 1
    while len(names) < T:
        for ch in base:
            names.append(f"{ch}{i}")
            if len(names) >= T:
                break
        i += 1
    return names

def parse_pairs(s: str):
    # "A>B,B>C" -> [("A","B"),("B","C")]
    if not s:
        return []
    parts = [p.strip() for p in s.split(",") if p.strip()]
    pairs = []
    for p in parts:
        if ">" not in p:
            raise ValueError(f"must-before pair '{p}' must look like X>Y")
        a, b = [x.strip() for x in p.split(">")]
        if not a or not b:
            raise ValueError(f"Bad must-before pair '{p}'")
        pairs.append((a, b))
    return pairs

def parse_groups(s: str):
    # "A+B, C+D+E" -> [["A","B"], ["C","D","E"]]
    if not s:
        return []
    groups = []
    for g in s.split(","):
        g = g.strip()
        if g:
            groups.append([x.strip() for x in g.split("+") if x.strip()])
    return groups
```

```
def is_contiguous_block(order, group):
    # group appears as a consecutive block in 'order' (any internal order allowed)
    idxs = sorted(order.index(x) for x in group)
    return all(idxs[i] + 1 == idxs[i+1] for i in range(len(idxs)-1))

def is_any_adjacent_pair(order, group):
    # does ANY adjacent pair from group occur? (used for "apart" = forbid adjacency)
    pos = {name: i for i, name in enumerate(order)}
    group_set = set(group)
    for a in group:
        i = pos[a]
        if i > 0 and order[i-1] in group_set:
            return True
        if i < len(order)-1 and order[i+1] in group_set:
            return True
    return False

def check_constraints(order, must_before, together_groups, apart_groups):
    ok_mb = True
    for a, b in must_before:
        if order.index(a) >= order.index(b):
            ok_mb = False
            break

    ok_tog = True
    for g in together_groups:
        if not is_contiguous_block(order, g):
            ok_tog = False
            break

    ok_apart = True
    for g in apart_groups:
        if is_any_adjacent_pair(order, g):
            ok_apart = False
            break

    ok_all = ok_mb and ok_tog and ok_apart
    return ok_mb, ok_tog, ok_apart, ok_all

# ------------------------------------------------------------
# Main
# ------------------------------------------------------------

def main():
    ap = argparse.ArgumentParser(description="Turn order (permutation) counter/enumerator
        .")
    ap.add_argument("-T", "--team-size", type=int, required=True, help="Number of heroes
        on the team")
    ap.add_argument("--names", type=str, default="", help="Comma-separated hero names (
        length must equal T)")
    ap.add_argument("--list-small", action="store_true",
                    help="Enumerate and write CSV only for small T (<=8). Otherwise just
                        print formula count.")
```

```
ap.add_argument("--must-before", type=str, default="", help="Constraints like 'A>B,B>
    C'")
ap.add_argument("--together", type=str, default="", help="Groups that must be
    contiguous like 'A+B, C+D+E'")
ap.add_argument("--apart", type=str, default="", help="Groups that must NOT be
    adjacent like 'A+B, C+D'")
args = ap.parse_args()

T = args.team_size
if T < 1:
    print("Team size T must be >= 1", file=sys.stderr)
    sys.exit(2)

names = [n.strip() for n in args.names.split(",") if n.strip()] if args.names else
    default_names(T)
if len(names) != T:
    print(f"Provided {len(names)} names but T={T}.", file=sys.stderr)
    sys.exit(2)

must_before = parse_pairs(args.must_before)
together_groups = parse_groups(args.together)
apart_groups = parse_groups(args.apart)

ensure_dirs()

# Always compute the total unconstrained count.
total_unconstrained = math.factorial(T)

enumerating = args.list_small and T <= 8
csv_path = f"data/turn_orders_T{T}.csv"
summary_path = "files/turn_orders_summary.txt"
fig_path = f"figures/turn_orders_lead_freq_T{T}.png"

valid_count = None
lead_counter = Counter()

if enumerating:
    order_id = 0
    with open(csv_path, "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(["order_id", "order", "valid_must_before", "valid_together", "
            valid_apart", "valid_all"])
        for perm in itertools.permutations(names, T):
            perm_list = list(perm)
            ok_mb, ok_tog, ok_ap, ok_all = check_constraints(perm_list, must_before,
                together_groups, apart_groups)
            if ok_all:
                lead_counter[perm_list[0]] += 1
            w.writerow([order_id, "-".join(perm_list), int(ok_mb), int(ok_tog), int(
                ok_ap), int(ok_all)])
            order_id += 1

    # Count how many were valid overall
    with open(csv_path, newline="") as f:
```

```
            r = csv.DictReader(f)
            valid_count = sum(1 for row in r if row["valid_all"] == "1")

        # Try to plot if matplotlib is available
        try:
            import matplotlib.pyplot as plt
            if lead_counter:
                heroes, counts = zip(*sorted(lead_counter.items()))
                plt.figure()
                plt.bar(heroes, counts)
                plt.title(f"Lead-slot frequency among valid orders (T={T})")
                plt.xlabel("Leading hero")
                plt.ylabel("Count")
                plt.tight_layout()
                plt.savefig(fig_path, dpi=150)
                plt.close()
        except Exception:
            # Soft fail: no figure if matplotlib missing
            fig_path = None

    # Write summary
    with open(summary_path, "w") as s:
        s.write("=== Turn Orders Summary ===\n")
        s.write(f"T = {T}\n")
        s.write(f"Names = {names}\n")
        s.write(f"Unconstrained count = {total_unconstrained}\n")
        s.write(f"must_before = {must_before}\n")
        s.write(f"together = {together_groups}\n")
        s.write(f"apart = {apart_groups}\n")
        if enumerating:
            s.write(f"Enumerated all orders and wrote {csv_path}\n")
            s.write(f"Valid (all constraints) = {valid_count}\n")
            if fig_path:
                s.write(f"Figure (lead frequency) saved to {fig_path}\n")
        else:
            s.write("Large T or no --list-small: skipped full enumeration.\n")
            s.write("Reported only the unconstrained factorial count above.\n")

    # Console: one clean line
    if enumerating:
        print(f"Turn order count (valid/all) = {valid_count}/{total_unconstrained} | CSV:
            {csv_path} | Summary: {summary_path}")
    else:
        print(f"Turn order count (unconstrained) = {total_unconstrained} | Summary: {
            summary_path}")


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

Listing 3.1: `scripts/turn_orders.py`

3.6 Generated Data (conditionally included)

If you ran the script, the following artifacts are pulled into this PDF (when present).

Summary (text)

Run the script to generate `files/turn_orders_summary.txt`, then recompile.

Lead-slot Frequency Figure

If you enumerate (e.g. `-T 3 -list-small`) and have `matplotlib`, a figure will appear here.

CSV (head preview)

For big CSVs we don't typeset the whole file in the PDF. Students should open `data/turn_orders_T{T}.csv` in a spreadsheet to explore rows and filters.

3.7 Checks Against the Math

Without constraints, the count is $T!$. With constraints, this chapter's script *filters* the enumerated permutations and reports the valid count in the summary. This lets students compare the *computed* valid count to small-case hand proofs from the math chapter.

3.8 Rubric Alignment

This chapter produces:

- **Code** (`scripts/turn_orders.py`) with clear flags and comments,
- **Data** (`data/turn_orders_T{T}.csv`) for small T,
- **Report** (`files/turn_orders_summary.txt`) suitable for quoting in the writeup,
- **Optional Figure** (`figures/*.png`) if plotting is available.

These map cleanly to the assignment's code + outputs deliverables. Pair partners can split responsibilities (driver = coding; navigator = test cases/constraints and report checking) and swap midway.