President Deathspank: A Scrum Hybrid

Jeremy Evert and ChatGPT

November 18, 2025

Contents

1 Introduction **1**

1.1 Overview 1

2 Universe Description: *Presidents of Virtue* in the DeathSpank World **3**

2.1 Setting: SpankTopia in Political Chaos 3

2.1.1 Core Inspiration 3

2.1.2 Cast of Roles 4

2.2 Rules of Play 4

2.2.1 Components and Objective 4

2.2.2 Dealing and Initial Titles 5

2.2.3 Between-Round Trading: Thongs of Virtue 5

2.2.4 Turn Structure: Quests and Justice Bursts 5

2.2.5 Justice Bursts and Bacon Revolutions 6

2.2.6 End of Round and Campaign Play 6

2.3 Player and User Choices 7

2.3.1 Choices for Players Around the Table 7

2.3.2 Choices for Users of a Digital or Classroom Version 7

2.4 Constraints and Design Goals 8

2.4.1 Mechanical Constraints 8

2.4.2 Thematic and Pedagogical Constraints 8

3 Math Writeup: Counting Presidents of Virtue **11**

3.1 Permutations: Ordering the Chain of Command 11
3.2 Combinations: Choosing Sets of Thongs of Virtue 13
3.3 Stars-and-Bars: Distributing Justice Points 14
3.4 Probability: Justice Bursts in a Starting Hand 16

4 Presidents of Virtue: Engine, Code, and Experiments 19

4.1 Getting the Code and Running It 19
4.2 File Layout 20
4.3 Core Engine Listing 21
4.4 Guided Tour of the Engine 29
4.4.1 Cards, Ranks, and Suits 29
4.4.2 Players and the Strategy Base Class 30
4.4.3 Legal Moves: Leads and Responses 30
4.4.4 The Round Engine 31
4.4.5 Why This Structure is Useful 32
4.5 Next Steps 32

5 Reading the Presidents of Virtue Log Like a Data Scientist 33

5.1 Where the CSV Comes From 33
5.2 What the Columns Mean 34
5.3 First Contact: Opening the CSV 35
5.3.1 Spreadsheet View 35
5.3.2 Python View with `csv` 35
5.4 Exercise 1: Warm–Up Data Slurp 36
5.5 Exercise 2: Strategy Scorecards 37
5.6 Exercise 3: Justice Bursts and Bomb Endings 38
5.7 Optional: Using `pandas` 39
5.8 Big Picture 40

Bibliography **41**

Chapter 1

Introduction

1.1 Overview

This document presents a solution to the current assignment. It shows one way the problem could be done and sets some expectations. You don't have to match it exactly, but it provides a starting point.

Chapter 2

Universe Description: *Presidents of Virtue* in the DeathSpank World

2.1 Setting: SpankTopia in Political Chaos

SpankTopia has never exactly been a bastion of calm, but things have gotten especially weird.

After recovering the mysterious Artifact and doing battle over the six legendary Thongs of Virtue, the heroic DeathSpank has accidentally destabilized the entire realm.[2, 4] Each time a Thong changes hands, the balance of power shifts: mayors become peasants, peasants become tyrants, and somewhere in the shadows the AntiSpank grins behind a curtain of sizzling bacon.[3]

To keep the world from collapsing into full-scale ridiculousness, the Council of Virtue has adopted a new, highly official, completely serious system of governance:[1] a card-driven political contest known as *Presidents of Virtue*. Every round of the game reshuffles the hierarchy of SpankTopia. Heroes rise, villains fall, and whoever empties their hand first becomes the new leader of the land—at least until the next round of chaos.[1]

2.1.1 Core Inspiration

Presidents of Virtue is a narrative reskin and mechanical variant of the shedding-type card game commonly known as *President*, *Scum*, or *Asshole*, in which players race to shed all of their cards to claim the top social rank for the next round.[1]

The universe skin, titles and card powers are inspired by the action role-playing series *DeathSpank*, *DeathSpank: Thongs of Virtue*, and *The Baconing*, where DeathSpank dispenses justice, hunts for mystical underwear, and eventually confronts his evil counterpart,

[1] It is neither official nor serious.

the AntiSpank.[2, 4, 3]

2.1.2 Cast of Roles

Each round, the players earn in-world titles based on the order in which they empty their hands. In the canonical four-player version, the titles are:

- **Hero of SpankTopia** (President): first to go out. Wields the greatest power between rounds.
- **Deputy of Justice** (Vice-President): second to go out. Still powerful, but slightly less sparkly.
- **Adventurers** (Citizens): any middle positions. They are the working heroes of the realm.
- **Minion of AntiSpank** (High-Scum): next-to-last; clearly one bad decision away from full villainy.
- **AntiSpank** (Scum): last player with cards. They represent the forces of corruption and are treated accordingly.

With five or more players, multiple *Adventurer* ranks may exist; with six or more, tables may insert extra titles such as *Clerk of Bacon*, *Thong Custodian*, or *Intern of Mildly Evil Paperwork* as desired. Titles matter because they determine trading privileges and turn order in the next round.

2.2 Rules of Play

2.2.1 Components and Objective

- Standard French deck of 52 cards; add up to 2 jokers if desired.
- Recommended player count: 3–6 (more players may use multiple decks).
- Default rank order (low to high): 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A, 2; jokers (if used) are higher than 2 and act as special bombs.[1]

Objective: Be the first to play all the cards in your hand. The relative finishing order determines titles for the next round.

2.2.2 Dealing and Initial Titles

For the very first round, all players are treated as *Adventurers*. Choose a dealer randomly. The dealer shuffles and deals all cards face-down, one at a time, clockwise. Slightly uneven hand sizes are permitted.

For subsequent rounds, the **Hero of SpankTopia** (President) deals. Seating is arranged in title order, clockwise, starting from the Hero and going down the hierarchy. This reinforces the social ladder both in fiction and in gameplay, similar to the original *President*.[1]

2.2.3 Between-Round Trading: Thongs of Virtue

After the cards are dealt (except in the first round), a structured trading phase occurs:

- The **AntiSpank** must give their *two highest* cards to the **Hero of SpankTopia**.
- The **Hero of SpankTopia** returns any *two* cards of their choice (often weak cards, nicknamed "Cursed Thongs").
- The **Minion of AntiSpank** gives their single highest card to the **Deputy of Justice**, who returns one card of their choice.

With more players and extra intermediate ranks, this can be extended (for example, the lowest two ranks might each owe tribute to the top two, parallel to the richer multi-title variants of *President*[1]). The theme is clear: virtue flows up, junk flows down.

Optional variant (*Communal Bacon*): the Hero may declare "Thongs for the People" and reverse the direction of generosity: the Hero gives two best cards to the AntiSpank and receives two worst cards in return, mirroring the "Communism" variants of the original game.[1] This is usually triggered as an act of mercy, chaos, or comedy.

2.2.4 Turn Structure: Quests and Justice Bursts

Play proceeds clockwise. The first leader of the first round is the player holding the 3 of Clubs (or another agreed-upon lowest card). In later rounds, the Hero of SpankTopia leads the first trick.

1. **Lead a Quest.** On your turn, you may start a new *quest* by playing a group of cards of the same rank:
    - Single (one card), pair (two of a kind), triple, or four-of-a-kind.
    - Optional rule: longer sequences (runs) of consecutive ranks of the same length may be allowed, e.g., 5-6-7 vs 9-10-J.[1]

2. **Respond or Pass.** Each subsequent player may:
   - Play the *same number* of cards of a *higher rank*, or
   - Pass and sit out the rest of that quest (trick), unless using a special revolution option.
3. **Ending the Quest.** When all players in sequence pass, the last player to successfully play a set wins the quest. They collect the pile into a "won" stack (for flavor, not scoring), and then lead the next quest with any legal group from their hand.

2.2.5 Justice Bursts and Bacon Revolutions

To reinforce the DeathSpank feel, several special rules layer on top of the standard climbing structure:

Justice Burst (2s and Jokers). Twos (and jokers if used) are ultra-powerful cards. They can be played only *in pattern*: a single 2 over a single card, two 2s over a pair, etc. A Justice Burst immediately clears the table; the player who triggered it wins the quest and leads the next one. Players may *never* lead a quest with a 2 or joker.

Ending the round by playing a Justice Burst is dangerous: if your last card is a 2 or joker, you *automatically become the AntiSpank* for the next round, even if you technically finished first. This mirrors harsh variants where ending on a bomb card forces a player into last place.[1]

Bacon Revolution (Four-of-a-Kind). If a player ever plays four of a kind in one move (either as a lead or as a legal response), a *Bacon Revolution* triggers. From that moment until the end of the current round, the rank order is completely reversed: 2 becomes the weakest card and 3 becomes the strongest, or vice versa depending on table convention. This is adapted from "revolution" variants in *President* and is thematically tied to DeathSpank's universe-scale silliness and the consequences of wearing too many Thongs of Virtue.[1, 4, 3]

2.2.6 End of Round and Campaign Play

When a player plays their last card, they immediately claim the highest remaining title (e.g., Hero, Deputy, Adventurer, Minion, AntiSpank). Play continues among the remaining players until all titles are assigned or until only one player still holds cards.

A *campaign* of *Presidents of Virtue* is simply a sequence of rounds with evolving titles. Tables may:

- Score each round (e.g., Hero earns +3, Deputy +2, Adventurer +1, Minion 0, Anti-Spank −1).

- Track long-term achievements (e.g., “Most Times as Hero”, “Most Times Accidentally AntiSpank”, “Longest Bacon Revolution Streak”).
- Introduce narrative events between rounds, such as side quests inspired by locations and NPCs from the *DeathSpank* games.[2, 4, 3]

2.3 Player and User Choices

2.3.1 Choices for Players Around the Table

Players face several meaningful decisions every round:

- **Risk vs. Safety in Leading.** Do you lead small, conservative singles to gently drain your hand, or unleash pairs and triples to try to seize control early?
- **Timing of Justice Bursts.** Do you burn powerful 2s and jokers early to escape a bad position, or hoard them to swing a late-game quest?
- **Embracing or Avoiding the AntiSpank Role.** Some players may intentionally drift toward the bottom of the hierarchy to enjoy the drama of climbing back up in later rounds, or to exploit variants like reversed trading.
- **Triggering Bacon Revolutions.** Holding four-of-a-kind, you can flip the entire power structure of the deck. Is it worth confusing everyone—including yourself—to rescue a weak hand?
- **Social Bluffing and Table Talk.** The DeathSpank universe almost demands ridiculous in-character banter. Players may negotiate, taunt, or role-play their titles, adding emergent narrative on top of the core card play.

2.3.2 Choices for Users of a Digital or Classroom Version

If *Presidents of Virtue* is implemented as a digital or classroom activity, non-tabletop “users” (students, designers, or players interacting with a UI) have additional configuration choices:

- Toggle optional rules: Bacon Revolutions, Communal Bacon (reversed trading), runs/straights, or special Justice Burst constraints.
- Configure scoring models for a course: number of rounds, how titles map onto participation points or bonuses.
- Decide whether roles carry minor special powers (e.g., the Hero can declare one rule toggle per round, the AntiSpank always leads the very first quest, the Minion may look at one opponent’s hand once per round, etc.).

- Adjust deck size and player caps for accessibility (e.g., limit to single deck for smaller groups, enable multi-deck chaos for large events).

These choices allow instructors or designers to tune the experience: fast and light for ice-breakers, or strategic and campaign-based for longer narrative sessions.

2.4 Constraints and Design Goals

2.4.1 Mechanical Constraints

- **Player Count:** The base rules assume 3–6 players. More players require additional decks and possibly simplified role ladders to keep the between-round trading manageable.[1]
- **Card Visibility:** Hands are private; only played cards and discarded piles are public. Any "table talk" or information sharing is optional and stylistic rather than mechanical.
- **Pacing:** The shedding core keeps each round relatively short (5–15 minutes), matching both the original *President* and the bite-sized feel of the *DeathSpank* games.[1, 2]
- **Complexity Ceiling:** Optional rules (runs, revolutions, special bombs) are modular. Groups can add or remove them to match their experience level.

2.4.2 Thematic and Pedagogical Constraints

- **Tone:** The tone aims to echo DeathSpank's blend of heroic fantasy and self-aware absurdity without requiring prior knowledge of the games.[2, 4, 3]
- **Narrative Flexibility:** The roles and titles are deliberately loose so they can be adapted for storytelling, classroom activities, or light role-playing.
- **Reusability:** Because the underlying mechanics closely track a well-known card game, *Presidents of Virtue* can slide neatly into contexts where *President* is already familiar, adding theme without reinventing the rules from scratch.

Overall, *Presidents of Virtue* wraps a familiar shedding game in the heroic nonsense of the DeathSpank universe: players climb a social ladder built from Thongs of Virtue, bursts of justice, and the occasional bacon-fueled revolution.

Bibliography

[1] *President (card game)*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/President_(card_game).

[2] *DeathSpank*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/DeathSpank.

[3] *The Baconing*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/The_Baconing.

[4] *DeathSpank: Thongs of Virtue*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/DeathSpank:_Thongs_of_Virtue.

Chapter 3

Math Writeup: Counting Presidents of Virtue

This chapter connects the story and rules of *Presidents of Virtue* to four core counting ideas from discrete mathematics:

- permutations (when order matters),
- combinations (when order does not matter),
- stars-and-bars (distributing identical things into labeled boxes),
- and probability (using counts to measure how likely an event is).

Each section below includes a clearly labeled **Problem**, the relevant **Formula**, and a fully **Worked Example** in the DeathSpank / Presidents of Virtue universe.

3.1 Permutations: Ordering the Chain of Command

In each round of *Presidents of Virtue*, players race to empty their hands. The order in which they finish determines the social ladder for the next round: Hero of SpankTopia, Deputy of Justice, Adventurers, Minion of AntiSpank, and finally the AntiSpank.

Problem (Permutations)

Suppose there are n distinct players at the table. At the end of a round, all titles are assigned by finishing order: first place becomes Hero of SpankTopia, second becomes Deputy of Justice, and so on, down to the AntiSpank in last place.

Question. How many different ways can the titles be assigned after one round, assuming every finishing order is possible?

Formula (Permutations)

A *permutation* of n distinct objects is any ordering of them in a line. The number of permutations of n distinct objects is

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

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

In words: for the first position we have n choices; for the second, $n-1$ choices; and so on, until only one choice remains. By the product principle, we multiply these choices together to get $n!$ possible orders.

Worked Example

Let $n = 5$ players sit down to play: DeathSpank, Sparkles, Steve, a Random Adventurer, and the Mysterious Clerk of Bacon.

At the end of the round, the finishing order determines:

1st	Hero of SpankTopia
2nd	Deputy of Justice
3rd	Adventurer
4th	Minion of AntiSpank
5th	AntiSpank

How many possible ways can these titles be assigned?

Because all five characters are distinct, this is just the number of permutations of 5 distinct players:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120.$$

So there are 120 different ways the round could end in terms of titles. Even with only five players, the space of possible "political timelines" for SpankTopia is already quite large.

Design note. If a designer wanted to guarantee that some specific character (say, DeathSpank) *never* becomes AntiSpank, they would be forbidding all permutations with that character in last place. That is a structural change to the game, and permutations give us a precise language for describing it.

3.2 Combinations: Choosing Sets of Thongs of Virtue

Between rounds, the Hero of SpankTopia may be allowed to equip special artifacts or Thongs of Virtue that grant small advantages (extra card trades, one-time Justice Bursts, or fancy decorative glow).

In many situations, it is the set of artifacts that matters, not the order in which they are chosen.

Problem (Combinations)

Suppose there are n distinct Thongs of Virtue stored in the Sacred Drawer of Laundry. At the start of a campaign night, the Hero is allowed to choose k of them as their *loadout* for the evening.

> **Question.** How many different loadouts of k Thongs can the Hero choose, if the order of selection does not matter?

Formula (Combinations)

A *combination* answers the question: "In how many ways can we choose k objects from n distinct objects when order does not matter?"

The number of such combinations is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

read as "n choose k."

The denominator divides out over-counting, since each *set* of k objects can be listed in $k!$ different orders.

Worked Example

Assume there are $n = 7$ distinct Thongs of Virtue:

> Justice, Stealth, Bacon, Looting, Mana, Friendship, and Mildly Confusing Glow.

Before the first round, the Hero may choose $k = 3$ of these to wear (in a safe, family-friendly way). Only the set matters; wearing Justice–Bacon–Mana is the same loadout as Mana–Justice–Bacon.

The number of possible loadouts is

$$\binom{7}{3} = \frac{7!}{3!\,4!} = \frac{7 \cdot 6 \cdot 5}{3 \cdot 2 \cdot 1} = 35.$$

So there are 35 distinct sets of three Thongs the Hero could bring into the session. If you later tweak the balance by adding more artifacts, you can use the same combination formula to see how quickly the loadout universe grows.

3.3 Stars-and-Bars: Distributing Justice Points

Many role-playing games let a character spread points across several abilities. In *Presidents of Virtue*, we might imagine DeathSpank distributing *Justice Points* among different powers before the game starts.

Problem (Stars-and-Bars)

DeathSpank has J Justice Points to allocate among k special powers:

- Smite of Righteousness,
- Flaming Bacon Shield,
- and Loot Sense.

Let x_1, x_2, x_3 be the number of points assigned to these three powers, respectively.

> **Question A.** If $J = 6$ and $k = 3$, and each power may receive any nonnegative number of points (including zero), how many different allocations (x_1, x_2, x_3) are possible?
>
> **Question B.** Answer the same question, but now require that every power must receive at least one point.

Formula (Stars-and-Bars)

We are counting the number of nonnegative integer solutions to

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

with $x_i \geq 0$.

The classic stars-and-bars result says that the number of such solutions is

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

If instead every power must receive at least one point, we set $y_i = x_i - 1$ so $y_i \geq 0$ and

$$y_1 + y_2 + \cdots + y_k = J - k.$$

Then the number of solutions with $x_i \geq 1$ is

$$\binom{(J-k)+k-1}{k-1} = \binom{J-1}{k-1}.$$

Worked Example

Question A: Some powers may get zero. Here $J = 6$ and $k = 3$, so we count nonnegative integer solutions to

$$x_1 + x_2 + x_3 = 6.$$

By stars-and-bars, the number of allocations is

$$\binom{6+3-1}{3-1} = \binom{8}{2} = \frac{8 \cdot 7}{2 \cdot 1} = 28.$$

So there are 28 ways for DeathSpank to distribute 6 Justice Points among three powers if some powers are allowed to be left at zero.

Question B: Every power must get at least one point. Now we require $x_1, x_2, x_3 \geq 1$ and $x_1 + x_2 + x_3 = 6$.

Set $y_i = x_i - 1$, so $y_i \geq 0$ and

$$y_1 + y_2 + y_3 = 6 - 3 = 3.$$

By stars-and-bars, the number of such allocations is

$$\binom{3+3-1}{3-1} = \binom{5}{2} = \frac{5 \cdot 4}{2 \cdot 1} = 10.$$

So there are 10 allocations in which every power gets at least one Justice Point. In game-design language: enforcing “no dump stats” shrinks the build universe from 28 to 10 possibilities.

3.4 Probability: Justice Bursts in a Starting Hand

Counting lets us describe how *big* a universe of possibilities is. Probability uses those counts to describe how *likely* certain events are, assuming all hands are equally likely.

Problem (Probability with Combinations)

In *Presidents of Virtue*, the card 2 in each suit is treated as a *Justice Burst*—a very powerful card that often clears the trick.

Suppose we are using a standard 52-card deck with four suits and four Justice Bursts (the four 2s). A player is dealt a 5-card starting hand.

> **Question.** What is the probability that the player's starting hand contains *exactly one* Justice Burst?

Formula (Probability from Counting)

When all 5-card hands are equally likely, we can write

$$\Pr(\text{exactly one Justice Burst}) = \frac{\text{number of hands with exactly one 2}}{\text{number of all 5-card hands}}.$$

- The number of all 5-card hands from a 52-card deck is
$$\binom{52}{5}.$$
- To have *exactly one* Justice Burst:
  - choose which one of the four 2s appears: $\binom{4}{1}$ ways;
  - choose the remaining 4 cards from the 48 non-2 cards: $\binom{48}{4}$ ways.

  So the number of favorable hands is
$$\binom{4}{1}\binom{48}{4}.$$

Therefore

$$\Pr(\text{exactly one Justice Burst}) = \frac{\binom{4}{1}\binom{48}{4}}{\binom{52}{5}}.$$

Worked Example

We can either leave the answer in binomial form (which is already meaningful), or evaluate it numerically.

First, keep it symbolic:

$$\Pr(\text{exactly one Justice Burst}) = \frac{\binom{4}{1}\binom{48}{4}}{\binom{52}{5}}.$$

If we expand:

$$\binom{4}{1} = 4, \qquad \binom{48}{4} = \frac{48 \cdot 47 \cdot 46 \cdot 45}{4 \cdot 3 \cdot 2 \cdot 1}, \qquad \binom{52}{5} = \frac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48}{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}.$$

Computing these (by hand or with a calculator) gives approximately

$$\Pr(\text{exactly one Justice Burst}) \approx 0.2995,$$

or about a 30% chance.

In other words, if you sit down and are repeatedly dealt random 5-card hands from a fresh deck, you should expect to see *exactly one* Justice Burst in roughly 3 out of every 10 hands, on average.

Design note. If you want Justice Bursts to be more common in starting hands, you could:

- add more special cards to the deck, or
- increase the starting hand size.

Either way, the combination formulas above let you recompute the exact probabilities and make data-informed design decisions.

Summary

In this chapter we:

- used **permutations** to count possible chains of command at the end of a round;
- used **combinations** to count how many artifact loadouts or Thong sets a Hero can choose;

- used **stars-and-bars** to count how many Justice Point builds are possible for a character;
- and used **probability** to quantify how often special cards (Justice Bursts) appear in random hands.

These four tools give a mathematical backbone to the *Presidents of Virtue* universe. They also serve as a template: any time you invent a new mini-game or variant, you can ask the same questions:

> How many different configurations are possible, and how likely are the ones I care about?

Answering those questions is where discrete mathematics and game design shake hands.

Chapter 4

Presidents of Virtue: Engine, Code, and Experiments

In this chapter we step behind the curtain and look at the actual Python code that drives the *Presidents of Virtue* simulations. This is our concrete instance of the "Design Your Own Universe" project: we have a world (the card game), a set of rules, and a Python engine that can both *play* the game and *generate data* for later analysis or machine learning.

4.1 Getting the Code and Running It

The code for this chapter lives in the course Git repository under:

`Discrete_Structures/Ch06/jeremy_solution/`

A typical workflow for a student at a terminal:

Step 1: Clone the Repository

```
# Replace this with the URL your instructor posts on Canvas
git clone <REPO-URL-FOR-SwosuCsPythonExamples>.git

cd SwosuCsPythonExamples/Discrete_Structures/Ch06/jeremy_solution
```

Step 2: Build the LaTeX Book (Optional but Recommended)

```
# From the jeremy_solution directory:
make
```

```
# This should produce PresidentDeathspank.pdf
```

Step 3: Run the Python Engine

The Python scripts live in the `scripts/` subdirectory:

```
cd scripts

# Sanity check for the engine
python test_presidents_engine.py

# Run several rounds and write a CSV play log
python simulate_game.py
```

If everything is wired up correctly, you should see console output describing the rounds and a file `presidents_of_virtue_plays.csv` created in the project folder. This CSV can be used later for statistics, visualizations, or machine learning models that try to learn good play from the log data.

4.2 File Layout

For this chapter, the important files are:

- `scripts/presidents_engine.py`
  Core card game engine: cards, players, strategies (via the base class), and the `PresidentsOfVirtueRo`
  class that simulates a single round and logs every action.
- `scripts/pov_strategies.py`
  Concrete strategy classes (Cautious, Greedy, Chaos, Random, Human, ...) that subclass the common `Strategy` base.
- `scripts/pov_players.py`
  Helper functions to build a table of named players with chosen strategies.
- `scripts/simulate_game.py`
  Top–level script: chooses players, runs multiple rounds, prints summaries, and writes the CSV play log.
- `scripts/pov_logging.py`
  Utilities for turning the in–memory play log into a CSV on disk.

In this chapter we focus on `presidents_engine.py`, because that is where the core combinatorial universe is defined.

4.3 Core Engine Listing

Below is the complete source for the engine. It is included directly from the course repository, so if the code changes and you rebuild the book, the PDF will always show the current version.

Listing 4.1: Core engine for Presidents of Virtue

```
# scripts/presidents_engine.py
"""
Core engine for the 'Presidents of Virtue' card game.

Contains:
- Card + deck utilities
- Strategy base class (concrete strategies live in pov_strategies.py)
- Player dataclass
- Round engine (PresidentsOfVirtueRound) that:
  * prints table state
  * logs all actions into a play_log list of dicts
"""

import random
from dataclasses import dataclass, field
from typing import List, Optional, Dict

# ---------------------------------------------------------------------------
# Card model
# ---------------------------------------------------------------------------

RANKS_NORMAL = ['3', '4', '5', '6', '7', '8',
               '9', '10', 'J', 'Q', 'K', 'A', '2']
SUITS = ['♣', '♢', '♡', '♠']


def rank_index(rank: str, order: List[str]) -> int:
    return order.index(rank)


@dataclass(frozen=True)
class Card:
    rank: str
    suit: str

    def __str__(self) -> str:
        return f"{self.rank}{self.suit}"


# ---------------------------------------------------------------------------
# Player + Strategy base
```

```
# ------------------------------------------------------------------------

@dataclass
class Player:
    name: str
    strategy: "Strategy" # concrete strategies come from pov_strategies
    hand: List[Card] = field(default_factory=list)
    finished: bool = False
    finish_position: Optional[int] = None
    ended_on_bomb: bool = False # True if final play contained a 2

    def remove_cards(self, cards: List[Card]) -> None:
        for c in cards:
            self.hand.remove(c)


class Strategy:
    """Base class that all strategies (NPC + human) should subclass."""

    def description(self) -> str:
        """One-line explanation for humans."""
        return "Mysterious strategy. (Probably chaos.)"

    def short_label(self) -> str:
        """Short tag to show in logs."""
        return self.__class__.__name__.replace("Strategy", "")

    def choose_play(
        self,
        player: Player,
        legal_plays: List[List[Card]],
        can_lead: bool,
        rank_order: List[str],
        revolution: bool,
    ) -> Optional[List[Card]]:
        """
        Must be implemented by subclasses.

        Returns:
            - a list of Card objects to play, OR
            - None to indicate PASS.
        """
        raise NotImplementedError


# ------------------------------------------------------------------------
# Helper functions: deck + move generation
```

```
89  # ----------------------------------------------------------------------------
90
91  def make_deck() -> List[Card]:
92      return [Card(rank, suit) for rank in RANKS_NORMAL for suit in SUITS]
93
94
95  def deal_deck(deck: List[Card], players: List[Player]) -> None:
96      random.shuffle(deck)
97      n_players = len(players)
98      for p in players:
99          p.hand.clear()
100     for i, card in enumerate(deck):
101         players[i % n_players].hand.append(card)
102
103
104 def group_by_rank(cards: List[Card]) -> Dict[str, List[Card]]:
105     grouped: Dict[str, List[Card]] = {}
106     for c in cards:
107         grouped.setdefault(c.rank, []).append(c)
108     return grouped
109
110
111 def generate_leads(hand: List[Card]) -> List[List[Card]]:
112     """
113     Generate all legal leading plays from a given hand.
114
115     Normal rule:
116       - You cannot lead with 2s.
117
118     Exception:
119       - If your hand consists ONLY of 2s, you ARE allowed to lead with them,
120         otherwise the game can get stuck in an infinite "everyone passes with 2s"
              loop.
121     """
122     grouped = group_by_rank(hand)
123     plays: List[List[Card]] = []
124
125     # Do we have any non-2 ranks?
126     non_two_ranks = [rank for rank in grouped.keys() if rank != '2']
127
128     if non_two_ranks:
129         ranks_to_consider = non_two_ranks
130     else:
131         # Hand is ONLY 2s -> allow leading with 2s
132         ranks_to_consider = ['2']
133
134     for rank in ranks_to_consider:
```

```
        cards = grouped[rank]
        for size in range(1, min(4, len(cards)) + 1):
            plays.append(cards[:size])

    return plays


def generate_responses(
    hand: List[Card],
    current_size: int,
    current_rank: str,
    rank_order: List[str],
) -> List[List[Card]]:
    """Generate all responses that beat the current table play."""
    grouped = group_by_rank(hand)
    plays: List[List[Card]] = []
    current_idx = rank_index(current_rank, rank_order)
    for rank, cards in grouped.items():
        if len(cards) < current_size:
            continue
        if rank_index(rank, rank_order) > current_idx:
            plays.append(cards[:current_size])
    return plays


def hand_to_str(hand: List[Card], rank_order: List[str]) -> str:
    return " ".join(
        str(c)
        for c in sorted(hand, key=lambda c: rank_index(c.rank, rank_order))
    )


# ---------------------------------------------------------------------------
# Round engine (with in-memory logging)
# ---------------------------------------------------------------------------

class PresidentsOfVirtueRound:
    """
    Simulates a single round of Presidents of Virtue.

    Public attributes after run():
    - finish_order (list of Players by finishing position)
    - play_log (list of dicts; each dict describes one action)
    - rank_order, revolution (final state)
    """

    def __init__(self, players: List[Player], round_index: int):
```

```
182         self.players = players
183         self.round_index = round_index
184         self.rank_order = list(RANKS_NORMAL)
185         self.revolution = False
186         self.finish_order: List[Player] = []
187         self.play_log: List[Dict[str, object]] = []
188 
189     # -------------------- internal helpers --------------------
190 
191     def _find_starting_player_index(self) -> int:
192         for i, p in enumerate(self.players):
193             for c in p.hand:
194                 if c.rank == '3' and c.suit == '♣':
195                     return i
196         return 0
197 
198     def _all_finished(self) -> bool:
199         return all(p.finished for p in self.players)
200 
201     def _active_players_indices(self) -> List[int]:
202         return [i for i, p in enumerate(self.players) if not p.finished]
203 
204     # -------------------- main round loop ---------------------
205 
206     def run(self) -> None:
207         # Reset per-round state
208         for p in self.players:
209             p.finished = False
210             p.finish_position = None
211             p.ended_on_bomb = False
212 
213         self.finish_order = []
214         self.play_log.clear()
215         self.rank_order = list(RANKS_NORMAL)
216         self.revolution = False
217 
218         # Print strategy blurbs once at the top of the round
219         print(f"\n=== ROUND {self.round_index} -Strategy overview ===")
220         for p in self.players:
221             print(f"{p.name:18s} [{p.strategy.short_label():12s}] "
222                   f"- {p.strategy.description()}")
223 
224         # Show starting hands
225         print(f"\n=== ROUND {self.round_index} -Starting hands ===")
226         for p in self.players:
227             print(f"{p.name:18s}: {hand_to_str(p.hand, self.rank_order)}")
228 
```

```
        leader_index = self._find_starting_player_index()
        next_finish_pos = 1
        trick_id = 0

        while not self._all_finished():
            trick_id += 1
            print(f"\n--- Round {self.round_index}, Trick {trick_id} "
                  f"(leader: {self.players[leader_index].name}) ---")

            current_size: Optional[int] = None
            current_rank: Optional[str] = None
            last_player_to_play: Optional[int] = None
            passed = {i: False for i in self._active_players_indices()}
            pile_cards: List[Card] = []
            step_in_trick = 0

            turn_index = leader_index

            while True:
                player = self.players[turn_index]
                if player.finished:
                    turn_index = (turn_index + 1) % len(self.players)
                    continue

                step_in_trick += 1
                hand_before = list(player.hand)

                if current_size is None:
                    legal_plays = generate_leads(player.hand)
                    can_lead = True
                else:
                    legal_plays = generate_responses(
                        player.hand, current_size, current_rank, self.rank_order
                    )
                    can_lead = False

                play = player.strategy.choose_play(
                    player, legal_plays, can_lead, self.rank_order, self.revolution
                )

                current_size_before = current_size if current_size is not None else
                    0
                current_rank_before = current_rank if current_rank is not None else
                    ""

                if play is None:
                    # PASS
```

```
                    passed[turn_index] = True
                    print(f"{player.name:18s} [{player.strategy.short_label():12s}] "
                          f"passes. (hand: {hand_to_str(player.hand, self.rank_order)})")

                    self.play_log.append({
                        "round": self.round_index,
                        "trick": trick_id,
                        "step": step_in_trick,
                        "player_name": player.name,
                        "strategy": player.strategy.short_label(),
                        "action": "pass",
                        "cards_played": "",
                        "hand_size_before": len(hand_before),
                        "hand_size_after": len(player.hand),
                        "current_size_before": current_size_before,
                        "current_rank_before": current_rank_before,
                        "is_lead": can_lead,
                        "is_justice_burst": False,
                        "is_revolution_trigger": False,
                        "revolution_state_after": self.revolution,
                        "finish_position": None,
                        "ended_on_bomb": None,
                    })

                    active_idxs = self._active_players_indices()
                    if last_player_to_play is not None and all(
                        (idx == last_player_to_play)
                        or passed.get(idx, False)
                        or self.players[idx].finished
                        for idx in active_idxs
                    ):
                        leader_index = last_player_to_play
                        break

                    turn_index = (turn_index + 1) % len(self.players)
                    continue

                # PLAY
                player.remove_cards(play)
                pile_cards.extend(play)

                is_revolution_trigger = False
                ranks_set = {c.rank for c in play}
                if len(play) == 4 and len(ranks_set) == 1:
                    # Bacon Revolution
```

```
                    self.rank_order = list(reversed(self.rank_order))
                    self.revolution = not self.revolution
                    is_revolution_trigger = True
                    print(f"*** BACON REVOLUTION triggered by {player.name}! "
                          f"Rank order reversed. ***")

                is_justice_burst = any(c.rank == '2' for c in play) and not
                    can_lead

                if not is_justice_burst:
                    current_size = len(play) if current_size is None else
                        current_size
                    current_rank = play[0].rank
                    last_player_to_play = turn_index

                table_str = " ".join(str(c) for c in pile_cards)
                play_str = " ".join(str(c) for c in play)
                lead_tag = " (lead)" if can_lead else ""
                extra_note = ""
                if is_justice_burst:
                    extra_note = " [Justice Burst: clears table]"
                elif is_revolution_trigger:
                    extra_note = " [Bacon Revolution: rank order flipped]"

                print(f"{player.name:18s} [{player.strategy.short_label():12s}] "
                      f"plays: {play_str:<12s}{lead_tag} | table: {table_str}{
                          extra_note}")

                self.play_log.append({
                    "round": self.round_index,
                    "trick": trick_id,
                    "step": step_in_trick,
                    "player_name": player.name,
                    "strategy": player.strategy.short_label(),
                    "action": "play",
                    "cards_played": play_str,
                    "hand_size_before": len(hand_before),
                    "hand_size_after": len(player.hand),
                    "current_size_before": current_size_before,
                    "current_rank_before": current_rank_before,
                    "is_lead": can_lead,
                    "is_justice_burst": is_justice_burst,
                    "is_revolution_trigger": is_revolution_trigger,
                    "revolution_state_after": self.revolution,
                    "finish_position": None,
                    "ended_on_bomb": None,
                })
```

```
363
364                 # Did the player just go out?
365                 if not player.hand and not player.finished:
366                     player.finished = True
367                     player.finish_position = next_finish_pos
368                     if any(c.rank == '2' for c in play):
369                         player.ended_on_bomb = True
370                     bomb_note = " (ended on a 2: bomb!)" if player.ended_on_bomb
                            else ""
371                     print(f"--> {player.name} is OUT and takes position "
372                           f"{next_finish_pos}{bomb_note}")
373                     self.finish_order.append(player)
374                     next_finish_pos += 1
375
376                     if len(self._active_players_indices()) == 1:
377                         last_idx = self._active_players_indices()[0]
378                         last_player = self.players[last_idx]
379                         last_player.finished = True
380                         last_player.finish_position = next_finish_pos
381                         print(f"--> {last_player.name} is last and takes position "
382                               f"{next_finish_pos}.")
383                         self.finish_order.append(last_player)
384                         return
385
386                 if is_justice_burst:
387                     print(f"*** JUSTICE BURST by {player.name}! "
388                           f"Table cleared; {player.name} leads next trick. ***")
389                     leader_index = turn_index
390                     break
391
392                 turn_index = (turn_index + 1) % len(self.players)
```

4.4 Guided Tour of the Engine

This section walks through the major moving parts of Listing 4.1. The goal is that every student can explain, in words, what each block of code is doing and how it connects to the game rules.

4.4.1 Cards, Ranks, and Suits

At the top of the file we fix a rank order `['3', ..., 'A', '2']` and a list of suits. The `Card` dataclass bundles a rank and a suit together, and defines a small `__str__` method so that cards print as things like 7♡ or 2♣ instead of raw Python objects.

The helper function `make_deck()` builds a full standard deck by taking all rank–suit pairs. The function `rank_index` lets us compare ranks according to the current ordering (which flips during a "Bacon Revolution").

4.4.2 Players and the Strategy Base Class

The `Player` dataclass represents one seat at the table. It has:

- a **name** (for readable logs),
- a **strategy** object that decides what to play,
- a list of **cards** in `hand`,
- flags for whether the player has **finished** and whether they **ended on a bomb** (their last play contained a 2).

The `Strategy` base class is intentionally simple: it provides a human–friendly `description()` and a short label (`short_label()`), and it requires subclasses to implement a single method:

```
choose_play(player, legal_plays, can_lead, rank_order, revolution)
```

This method returns either a list of `Card` objects (the chosen play) or `None` to indicate a pass. Human strategies and bot strategies both plug into this same interface.

4.4.3 Legal Moves: Leads and Responses

Two helper functions generate all legal plays from a given hand:

`generate_leads(hand)` Used when a player is *leading* a trick. It groups cards by rank and then creates all singletons, pairs, triples, or quads for each rank except 2s (you may not lead with 2s).

`generate_responses(hand, current_size, current_rank, rank_order)` Used when the table already has a play. It groups the hand by rank and keeps only sets that (a) have the same size as the current play and (b) strictly beat the current rank according to `rank_order`.

This separates *rules* (what is legal) from *strategy* (which of the legal plays we choose).

4.4.4 The Round Engine

The class `PresidentsOfVirtueRound` encapsulates the logic for a single round of the game:

- The constructor stores the players, the round index, and sets up the initial rank order and revolution flag.
- The public method `run()`:
  1. Resets per–round state on all players.
  2. Prints a strategy overview and starting hands.
  3. Finds the starting player (the one holding 3♣).
  4. Loops over tricks until all players have finished.

Within each trick, the engine:

1. Tracks whose turn it is and what the current table play looks like (size and rank).
2. Asks the current player's strategy for a move via `choose_play`.
3. Either logs a **pass** or removes cards from the player's hand and adds them to the table pile.
4. Checks for special events:
   - **Bacon Revolution**: a four–of–a–kind flips the rank order and toggles the `revolution` flag.
   - **Justice Burst**: playing a 2 (without leading) clears the table and gives that player the lead in the next trick.
5. Logs every action into `self.play_log` as a dictionary with fields such as round, trick, player, action, hand sizes, and flags.
6. Detects when a player goes out, assigns them the next finish position, and records whether they ended on a bomb.

The round ends when there is only one active player left; that last player is assigned the final position.

4.4.5 Why This Structure is Useful

This design is intentionally modular:

- **Game rules** live in `presidents_engine.py`.
- **Strategies** (human or bot) live in `pov_strategies.py`.
- **Table lineups** live in `pov_players.py`.
- **Data export** lives in `pov_logging.py`.

For the Chapter 5 project, you were asked to design a universe, count its structures (permutations, combinations, stars and bars), and then connect it to Python experiments. Here, we go one step further: we have a live engine that can produce large play logs. Those logs become training data for simple models that try to predict, for example, which plays lead to better positions or which strategies tend to win from a given state.

4.5 Next Steps

Once you are comfortable reading and running this engine, you are ready to:

- Add new strategy classes (for example, a player that hoards pairs, or a player that tries to avoid ending on a bomb).
- Change the rules slightly and see how the distribution of finish positions changes in the CSV.
- Use the CSV as input to a notebook or a simple machine learning experiment: can you predict the winner from a snapshot of the table?
- Recreate this pattern for your own designed universes from Chapter 5—a small set of rules, a clean Python engine, and an experiment loop that can be checked against your combinatorics.

This chapter is your bridge from story and counting to full simulation and data. The more you can explain what the code is doing in plain language, the more powerful your future universes (and your debugging sessions) will become.

Chapter 5

Reading the Presidents of Virtue Log Like a Data Scientist

In the previous chapters we designed the game universe, implemented the `PresidentsOfVirtueRound` engine, and wired up strategies and simulations. Now we do something deliciously modern: we treat the play log as a *dataset*.

This chapter is about reading the CSV file `presidents_of_virtue_plays.csv` and asking questions like a data scientist:

- Which strategies tend to finish earlier?
- How often do players trigger a Justice Burst?
- What happens to hand sizes over the course of a round?

We are not trying to build a full machine learning model (yet), but we *are* trying to get comfortable with turning a log of events into structured questions and answers.

5.1 Where the CSV Comes From

When you run the simulation script,

```
cd SwosuCsPythonExamples/Discrete_Structures/Ch06/jeremy_solution
python scripts/simulate_game.py
```

it will print several rounds of game play to the console and then write a file:

`presidents_of_virtue_plays.csv`

Each row in this CSV represents *one action* taken by one player during one trick of one round. It includes both game state and labels that will be useful for statistics and machine learning.

5.2 What the Columns Mean

Here is a quick tour of the columns in the log; these names come directly from the engine:

- **`round`**: which round of the simulation this action belongs to (1, 2, 3, ...).
- **`trick`**: which trick within the round (starts at 1 each round).
- **`step`**: the order of actions inside a trick (1 = first action in that trick).
- **`player_name`**: the seat at the table (`Cody`, `Savannah`, ...).
- **`strategy`**: the short label for that player's strategy (`Cautious`, `Greedy`, `PairLover`, `ChaosRevolutionary`, `Random`, etc.).
- **`action`**: either `play` or `pass`.
- **`cards_played`**: a space–separated string naming the cards played, for example "Q of hearts, Q of clubs, Q of diamonds" or "2 of spades". This is empty for passes.
- **`hand_size_before`** and **`hand_size_after`**: how many cards the player held before and after that action.
- **`current_size_before`** and **`current_rank_before`**: what the table looked like *before* this action (for example, "there is a pair of 8s out").
- **`is_lead`**: `True` if this action started a new trick (a lead), `False` otherwise.
- **`is_justice_burst`**: `True` if the player used a 2 to slam the table, clear the trick, and take back control.
- **`is_revolution_trigger`**: `True` if this play was a Bacon Revolution (four of a kind) that flipped the rank order.
- **`revolution_state_after`**: `True` when the round is currently in the "revolution" rank order, `False` otherwise.
- **`finish_position`**: when this row was written we also tagged each player with their final finishing place (1 for first, 2 for second, etc.). This is constant for a given player within a given round.
- **`ended_on_bomb`**: whether the player ended the round by playing a 2 (a bomb).

The most important idea is: **each row is one event in time**. If we sort by `(round, trick, step)`, we see the game unfold action by action.

5.3 First Contact: Opening the CSV

There are two very common ways to peek at a CSV:

1. In a spreadsheet program (Excel, LibreOffice, Google Sheets).
2. In Python, using either the built–in `csv` module or a library like `pandas`.

5.3.1 Spreadsheet View

From your file manager or terminal, you can open the CSV directly in a spreadsheet tool. This is great for a quick visual scan:

- Filter by `player_name` to see what one player did.
- Filter by `action==play` to see only plays (no passes).
- Sort by `finish_position` to see which strategies are winning.

For deeper work, though, we want code.

5.3.2 Python View with `csv`

Here is a small, self–contained example script that just prints the first few rows and counts them. Save this as `scripts/pov_peek.py`:

```
import csv
from pathlib import Path

def main():
    # Assume we run from the jeremy_solution folder
    csv_path = Path("presidents_of_virtue_plays.csv")

    if not csv_path.exists():
        print("CSV not found. Did you run simulate_game.py first?")
        return

    with csv_path.open(newline="", encoding="utf-8") as f:
        reader = csv.DictReader(f)
        rows = list(reader)

    print(f"Loaded {len(rows)} actions from {csv_path}.\n")
    print("First 5 actions:")
```

```
    for row in rows[:5]:
        print(
            f"round {row['round']}, trick {row['trick']}, step {row['step']}: "
            f\"{row['player_name']} ({row['strategy']}) "
            f\"{row['action']} {row['cards_played']}\"
        )

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

Run it from the `jeremy_solution` directory:

```
python scripts/pov_peek.py
```

If that works, you now have a tiny analysis pipeline: simulation → CSV → Python → printed insight.

5.4 Exercise 1: Warm–Up Data Slurp

Goal: practice reading the CSV and extracting basic stats.

Task A: Count Actions by Strategy

Write a Python script `pov_count_actions.py` that:

1. Reads `presidents_of_virtue_plays.csv` with `csv.DictReader`.
2. Builds a dictionary mapping:

$$\text{strategy name} \longmapsto \text{number of actions (rows)}$$

3. Prints each strategy and its total number of actions.

You might see output like:

```
Cautious: 130 actions
Greedy: 128 actions
PairLover: 125 actions
ChaosRevolutionary: 132 actions
Random: 129 actions
```

Task B: Count Plays vs Passes

Extend your script or write a second one that counts how many actions were `play` and how many were `pass`. Then break it down by strategy:

$$(\text{strategy}, \text{action}) \longmapsto \text{count}.$$

Questions to reflect on:

- Which strategy passes the most?
- Does this match your intuition from reading the strategy descriptions?

5.5 Exercise 2: Strategy Scorecards

Goal: connect the `finish_position` field to the strategies.

Each player appears many times in the log during a round, but their `finish_position` is the same in all those rows for that round. We can treat `finish_position` as a label for that (player, round) combo.

Write a Python script `pov_strategy_scorecard.py` that:

1. Reads the CSV with `DictReader`.
2. Builds a mapping from:

   $$(\text{round}, \text{player\_name}) \longmapsto (\text{strategy}, \text{finish\_position})$$

   The simplest way is to only record a (round, player) pair the *first* time you see it.
3. After reading all rows, build a table for each strategy:

   $$\text{strategy} \longmapsto \{\text{finish positions seen}\}.$$

4. For each strategy, compute:
   - how many rounds it appeared in,
   - the average finish position (lower is better: 1.0 is best),
   - how many times it finished first.

For example, your script might print something like:

```
Cautious          : avg finish 2.33 over 3 rounds, 1 wins
Greedy            : avg finish 3.00 over 3 rounds, 0 wins
PairLover         : avg finish 1.67 over 3 rounds, 2 wins
ChaosRevolutionary: avg finish 3.67 over 3 rounds, 0 wins
Random            : avg finish 2.33 over 3 rounds, 1 wins
```

Your numbers will depend on how many rounds you simulated.

Reflection

- Which strategies seem strong in this small sample?
- Do you trust the results yet, or do you feel like you need more data?
- What would happen if you changed the strategies slightly and re-ran?

This is a tiny version of an experiment: we are estimating performance from sampled games, and sampling error is very real.

5.6 Exercise 3: Justice Bursts and Bomb Endings

Goal: use the special flags `is_justice_burst` and `ended_on_bomb` to study dramatic events in the game.

Task A: How Often Do Strategies Trigger Justice Bursts?

Write a script `pov_justice_stats.py` that:

1. Reads the CSV.
2. For each strategy, counts how many actions had `is_justice_burst == True`.
3. Optionally, also compute the fraction:

$$\frac{\#\text{ Justice Bursts}}{\#\text{ actions for that strategy}}.$$

Questions:

- Does the Chaos/Revolution strategy really cause more Justice Bursts?
- Do cautious players ever use them, or do they mostly avoid them?

Task B: Who Ends on a Bomb?

The column `ended_on_bomb` is associated with the *final play* that took a player out. Since we carried that label into every row for that (player, round), we can still extract it from the CSV.

Extend your analysis to compute, for each strategy:

- how many rounds that strategy ended on a bomb, and
- the fraction of its rounds that ended on a bomb.

You might discover that some strategies rarely end on bombs (they burn their 2s earlier) while others like to hoard them until the final punch.

5.7 Optional: Using pandas

If you are comfortable installing extra libraries, you can also use the `pandas` library to treat the CSV as a DataFrame. This often makes grouping and aggregation more concise.

A minimal example (saved as `scripts/pov_pandas_demo.py`):

```
import pandas as pd

def main():
    df = pd.read_csv("presidents_of_virtue_plays.csv")

    print(df.head()) # first 5 rows

    # Count actions by strategy
    print("\nActions per strategy:")
    print(df["strategy"].value_counts())

    # Justice bursts per strategy
    jb = df[df["is_justice_burst"] == True]
    print("\nJustice bursts per strategy:")
    print(jb["strategy"].value_counts())

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

If you go this route, be sure you can explain *what* each line is doing in plain language. The point of this chapter is not to memorize pandas, but to get used to treating a play log as a structured, countable universe.

5.8 Big Picture

By this point you have:

- designed a combinatorial universe (the card game rules),
- implemented it as a Python engine,
- generated play logs as CSV,
- and started to ask quantitative questions about strategies and outcomes.

This is exactly the kind of pipeline that many real data scientists and machine learning engineers use:

Universe / Rules → Simulation / Logs → CSV / Tables → Models / Decisions

In the next steps of your journey, you can plug this data into more advanced models, or design entirely new universes and repeat the experiment. For now, enjoy the feeling that you can read a game log like a scientist, not just a spectator.

Bibliography

- Author, A. (Year). *Title of Work*. Publisher.