Probability Paradoxes

A Programming Tutorial

Learn Python by Simulating the Impossible

Liceo Lugano 3 — Giornata Autogestita 2026

In this tutorial, you will learn to **program in Python** by simulating five famous probability paradoxes.

No programming experience needed — just curiosity!

By the end, you will have written your own simulations, generated beautiful plots, and *proven* that math can be surprising.

Contents

Getting Started

What is Python?

Python is one of the most popular programming languages in the world. Scientists, engineers, data analysts, and even artists use it every day. Why?

- It reads almost like English
- It's free and works on any computer
- It has powerful tools for math, plotting, and simulation

Setting Up

The easiest way to start is **Google Colab** — no installation required!

1. Go to https://colab.research.google.com
2. Sign in with your Google account
3. Click **"New Notebook"**
4. You're ready to code!

Important

In Colab, you write code in *cells*. Press **Shift** + **Enter** to run a cell. Try it now!

Your First Program

Type this in a cell and press **Shift** + **Enter**:

```
print("Hello, world!")
print("I am learning Python!")
print(2 + 3)
```

Output

```
Hello, world!
I am learning Python!
5
```

Congratulations — you just wrote your first program!

Python Essentials (5 minutes)

Here's everything you need for this tutorial:

```
# Variables --- store values with a name
name = "Alice"
age = 17
pi = 3.14159

# Print --- show results on screen
print(name)              # Alice
```

```python
print(f"I am {age}")      # I am 17

# Math --- Python is a calculator
print(10 / 3)             # 3.3333...
print(2 ** 10)            # 1024 (powers!)

# Lists --- ordered collections
colors = ["red", "blue", "green"]
print(colors[0])          # red (counting starts at 0!)
print(len(colors))        # 3

# Loops --- repeat actions
for i in range(5):
    print(i)              # prints 0, 1, 2, 3, 4

# Conditions --- make decisions
x = 7
if x > 5:
    print("big")
else:
    print("small")
```

Indentation matters!

In Python, the spaces at the beginning of a line are *not decoration* — they define the structure. Everything inside a `for` loop or an `if` must be indented (4 spaces). Colab does this automatically for you.

Importing Libraries

Python has thousands of *libraries* — pre-built tools you can use. We need two:

```python
import random              # for generating random numbers
import matplotlib.pyplot as plt  # for making plots
```

In Colab, these are already installed. Just run this cell once at the top of your notebook.

The `random` library

`random` lets us simulate chance:

- `random.random()` — a random number between 0 and 1
- `random.randint(1, 6)` — a random integer (like rolling a die)
- `random.choice(["a", "b", "c"])` — pick one randomly from a list

Try it:

```python
import random

# Roll a die 10 times
for i in range(10):
    roll = random.randint(1, 6)
    print(f"Roll {i+1}: {roll}")
```

Run it multiple times — you'll get different results each time! That's randomness.

Paradox I: The Monty Hall Problem

The Story

You're on a game show. There are **3 doors**. Behind one is a **car**, behind the other two are **goats**. You pick a door. The host (who *knows* what's behind each door) opens another door, revealing a goat. He then asks:

"Do you want to switch to the other door?"

Most people say it doesn't matter — it's 50/50, right?
Wrong. Switching wins **2/3** of the time!

That sounds impossible. Let's *prove* it with code.

Step 1: Simulate One Game

Let's think about what happens in one game:

1. The car is placed behind a random door (0, 1, or 2)
2. The player picks a random door
3. The host opens a door with a goat (not the player's door, not the car's door)
4. The player decides: **stay** or **switch**

```python
import random

# Place the car behind a random door
car = random.randint(0, 2)
print(f"Car is behind door {car}")

# Player picks a door
player_pick = random.randint(0, 2)
print(f"Player picks door {player_pick}")

# Did the player win by STAYING?
if player_pick == car:
    print("STAY wins! (player picked the car)")
else:
    print("SWITCH wins! (player didn't pick the car)")
```

The Key Insight

Notice something: you win by **staying** only if your first pick was the car. You win by **switching** whenever your first pick was *not* the car. Since there are 3 doors:

- Probability of picking the car first try: **1/3**
- Probability of *not* picking the car: **2/3**

So switching wins 2/3 of the time!

Step 2: Simulate 1000 Games

One game doesn't prove much. Let's play **1000 games** and count:

```python
import random

stay_wins = 0
switch_wins = 0

for game in range(1000):
    car = random.randint(0, 2)           # place the car
    player_pick = random.randint(0, 2)   # player picks

    if player_pick == car:
        stay_wins += 1       # staying would win
    else:
        switch_wins += 1     # switching would win

print(f"Games played: 1000")
print(f"Stay wins:   {stay_wins}  ({stay_wins/10:.1f}%)")
print(f"Switch wins: {switch_wins} ({switch_wins/10:.1f}%)")
```

Output

```
Games played:  1000
Stay wins:  332 (33.2%)
Switch wins:  668 (66.8%)
```

The numbers speak for themselves: switching wins about **67%** of the time!

Step 3: Watch It Converge

Let's make a beautiful plot that shows how the win rates *converge* to 1/3 and 2/3 as we play more games:

```python
import random
import matplotlib.pyplot as plt

n_games = 5000
stay_wins = 0
switch_wins = 0
stay_rates = []
switch_rates = []

for game in range(1, n_games + 1):
    car = random.randint(0, 2)
    pick = random.randint(0, 2)

    if pick == car:
        stay_wins += 1
    else:
        switch_wins += 1

    # Record the running win rate
    stay_rates.append(stay_wins / game)
    switch_rates.append(switch_wins / game)

# Plot!
plt.figure(figsize=(10, 5))
```

```
plt.plot(stay_rates, color='#DC2626', linewidth=1, label='Stay', alpha=0.8)
plt.plot(switch_rates, color='#16A34A', linewidth=1, label='Switch', alpha=0.8)
plt.axhline(y=1/3, color='#DC2626', linestyle='--', alpha=0.5, label='Theory: 1/3')
plt.axhline(y=2/3, color='#16A34A', linestyle='--', alpha=0.5, label='Theory: 2/3')
plt.xlabel('Number of games', fontsize=12)
plt.ylabel('Win rate', fontsize=12)
plt.title('Monty Hall: Stay vs Switch', fontsize=14, fontweight='bold')
plt.legend(fontsize=11)
plt.ylim(0.15, 0.85)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

You should see the red line (stay) settling around 33% and the green line (switch) settling around 67%. The more games you play, the closer they get to the theoretical values.

What you just did

You used a **Monte Carlo simulation**: using random experiments to estimate probabilities. This is a real technique used by scientists, engineers, and financial analysts every day!

Step 4: Bar Chart Comparison

Let's make a clean bar chart to summarize the final results:

```
strategies = ['Stay', 'Switch']
results = [stay_wins / n_games, switch_wins / n_games]
theory = [1/3, 2/3]
colors = ['#DC2626', '#16A34A']

fig, ax = plt.subplots(figsize=(6, 4))
x = [0, 1]
bars = ax.bar([i - 0.15 for i in x], results, 0.3,
              color=colors, alpha=0.8, label='Simulation')
ax.bar([i + 0.15 for i in x], theory, 0.3,
       color=colors, alpha=0.3, label='Theory')

for i, (r, t) in enumerate(zip(results, theory)):
    ax.text(i - 0.15, r + 0.02, f'{r:.1%}', ha='center',
            fontweight='bold', fontsize=11)
    ax.text(i + 0.15, t + 0.02, f'{t:.1%}', ha='center',
            fontsize=10, color='gray')

ax.set_xticks(x)
ax.set_xticklabels(strategies, fontsize=12)
ax.set_ylabel('Win Rate', fontsize=12)
ax.set_title('Monty Hall: Final Results', fontsize=14, fontweight='bold')
ax.set_ylim(0, 0.85)
ax.legend()
ax.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()
```

Challenge: 100 Doors!

What if there were **100 doors** instead of 3? You pick one, the host opens 98 doors with goats, leaving your door and one other. Would you switch?
Modify the simulation:

```
n_doors = 100
car = random.randint(0, n_doors - 1)
pick = random.randint(0, n_doors - 1)
# Staying wins only if you picked right: 1/100
# Switching wins the other 99/100 of the time!
```

Try it with 10, 50, and 100 doors. What pattern do you see?

Paradox II: The Boy or Girl Problem

The Story

A parent tells you: *"I have two children. At least one of them is a boy."*

What is the probability that *both* are boys?

Most people say 1/2. After all, the other child is either a boy or a girl, right?
The answer is 1/3.

Why 1/3? Let's Think First

A family with two children has four equally likely possibilities:

Case	Child 1	Child 2	At least one boy?
1	Girl	Girl	No
2	Girl	Boy	Yes
3	Boy	Girl	Yes
4	Boy	Boy	Yes

We know "at least one is a boy," so Case 1 is eliminated. Of the 3 remaining cases, only 1 has both boys. So the probability is $\frac{1}{3}$!

Step 1: Simulate Random Families

```python
import random

def make_family():
    """Create a random family with 2 children."""
    child1 = random.choice(["Boy", "Girl"])
    child2 = random.choice(["Boy", "Girl"])
    return (child1, child2)

# Generate 10 families and show them
for i in range(10):
    family = make_family()
    print(f"Family {i+1}: {family[0]}, {family[1]}")
```

Output

```
Family 1:  Boy, Girl
Family 2:  Girl, Girl
Family 3:  Boy, Boy
Family 4:  Girl, Boy
Family 5:  Boy, Girl
...
```

Step 2: Filter and Count

Now let's generate many families, keep only those with at least one boy, and count how many have *both* boys:

```python
import random

n_families = 10000
at_least_one_boy = 0
both_boys = 0

for i in range(n_families):
    child1 = random.choice(["Boy", "Girl"])
    child2 = random.choice(["Boy", "Girl"])

    # Does this family have at least one boy?
    if child1 == "Boy" or child2 == "Boy":
        at_least_one_boy += 1

        # Are BOTH boys?
        if child1 == "Boy" and child2 == "Boy":
            both_boys += 1

probability = both_boys / at_least_one_boy
print(f"Families with at least one boy: {at_least_one_boy}")
print(f"Of those, both boys: {both_boys}")
print(f"Probability: {probability:.4f}")
print(f"Theory: {1/3:.4f}")
```

Output

```
Families with at least one boy:  7509
Of those, both boys:  2498
Probability:  0.3327
Theory:  0.3333
```

Step 3: Convergence Plot

```python
import random
import matplotlib.pyplot as plt

n = 10000
at_least_one_boy = 0
both_boys = 0
probabilities = []

for i in range(n):
    c1 = random.choice(["Boy", "Girl"])
    c2 = random.choice(["Boy", "Girl"])

    if c1 == "Boy" or c2 == "Boy":
        at_least_one_boy += 1
        if c1 == "Boy" and c2 == "Boy":
            both_boys += 1

    # Record running probability (avoid division by zero)
    if at_least_one_boy > 0:
        probabilities.append(both_boys / at_least_one_boy)
    else:
        probabilities.append(0)

```

```
plt.figure(figsize=(10, 5))
plt.plot(probabilities, color='#2563EB', linewidth=1, alpha=0.8,
         label=f'Simulation (final: {probabilities[-1]:.4f})')
plt.axhline(y=1/3, color='#DC2626', linestyle='--', linewidth=2,
            label='Theory: 1/3')
plt.xlabel('Number of families', fontsize=12)
plt.ylabel('P(both boys | at least one boy)', fontsize=12)
plt.title('Boy or Girl Paradox', fontsize=14, fontweight='bold')
plt.legend(fontsize=11)
plt.ylim(0.2, 0.5)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

Challenge: A Different Question

What if the parent says: "*My **older** child is a boy*"? Now what is the probability both are boys?

Hint: This time we fix Child 1 = Boy. The only question is about Child 2. Write the simulation and see — the answer changes to $\mathbf{1/2}$!

Why does specifying *which* child changes the answer?

Paradox III: The Birthday Problem

The Story

How many people do you need in a room for there to be a **greater than 50% chance** that two of them share a birthday?
Most people guess around 180 (half of 365). The real answer?

Just 23 people!

With only 23 people, there is a 50.7% chance that at least two share a birthday.
With 50 people, it's 97%!

Step 1: Simulate One Room

Let's generate random birthdays for a group and check for matches:

```python
import random

def has_birthday_match(n_people):
    """Check if any two people in a group share a birthday."""
    birthdays = []
    for i in range(n_people):
        birthday = random.randint(1, 365)  # day of the year
        if birthday in birthdays:
            return True  # match found!
        birthdays.append(birthday)
    return False  # no match

# Try it with 23 people
result = has_birthday_match(23)
print(f"Match found: {result}")
```

Run this cell several times. Sometimes you get `True`, sometimes `False`. That's because 23 people gives roughly a coin flip (50%).

Step 2: Estimate the Probability

To estimate the probability, we repeat the experiment many times:

```python
import random

def birthday_probability(n_people, n_trials=10000):
    """Estimate P(match) for a group of n_people."""
    matches = 0
    for trial in range(n_trials):
        if has_birthday_match(n_people):
            matches += 1
    return matches / n_trials

# Test for different group sizes
for n in [10, 20, 23, 30, 40, 50, 60]:
    p = birthday_probability(n)
    print(f"{n:3d} people: P(match) = {p:.1%}")
```

Output

```
10 people:  P(match) = 11.8%
20 people:  P(match) = 41.0%
23 people:  P(match) = 50.4%
30 people:  P(match) = 70.5%
40 people:  P(match) = 89.2%
50 people:  P(match) = 97.1%
60 people:  P(match) = 99.4%
```

Step 3: The Exact Formula

We can also compute the *exact* probability mathematically:

$$P(\text{no match with } n \text{ people}) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{365 - n + 1}{365}$$

$$P(\text{at least one match}) = 1 - P(\text{no match})$$

In Python:

```python
def birthday_theory(n):
    """Exact P(at least one shared birthday) for n people."""
    p_no_match = 1.0
    for k in range(1, n):
        p_no_match *= (365 - k) / 365
    return 1 - p_no_match

print(f"P(match with 23 people) = {birthday_theory(23):.4f}")
```

Output

```
P(match with 23 people) = 0.5073
```

Step 4: The Beautiful Plot

Now let's create the classic Birthday Problem curve, comparing simulation with theory:

```python
import random
import matplotlib.pyplot as plt

# Simulation
group_sizes = list(range(2, 61))
simulated = [birthday_probability(n, 5000) for n in group_sizes]

# Theory
theoretical = [birthday_theory(n) for n in group_sizes]

# Plot
plt.figure(figsize=(10, 5))
plt.scatter(group_sizes, simulated, s=20, color='#2563EB',
            alpha=0.6, zorder=3, label='Simulation (5000 trials)')
plt.plot(group_sizes, theoretical, color='#DC2626',
         linewidth=2, zorder=2, label='Exact theory')
plt.axhline(y=0.5, color='gray', linestyle=':', linewidth=1)
plt.axvline(x=23, color='#16A34A', linestyle='--', linewidth=1.5,
            alpha=0.8, label='n = 23 (50.7%)')
```

```
plt.xlabel('Number of people in the room', fontsize=12)
plt.ylabel('P(at least one shared birthday)', fontsize=12)
plt.title('The Birthday Problem', fontsize=14, fontweight='bold')
plt.legend(fontsize=10)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

Why is it so surprising?

Our intuition fails because we think about *our own* birthday matching someone. But the question is whether *any two* people match. With 23 people, there are $\binom{23}{2} = 253$ possible pairs to compare — that's a lot of chances for a match!

Challenge: Your Class!

1. How many students are in your class? Use the formula to compute the theoretical probability of a birthday match.
2. Collect real birthdays from your classmates. Is there a match? Compare with the theory!
3. What if we include the year? Does it change anything?

Paradox IV: Simpson's Paradox

The Story

In 1973, the University of California, Berkeley was accused of **gender discrimination** in graduate admissions. The overall numbers seemed clear:

	Men	Women
Applied	8,442	4,321
Admitted	**44%**	**35%**

Case closed? Not so fast! When researchers looked at *each department separately*, they found that women had **higher** admission rates in most departments!
How can women do better *everywhere* but worse *overall*?

The Trick: Different Base Rates

The secret: women applied more to *competitive* departments (low admission rates), while men applied more to *easy* departments. Let's simulate this.

Step 1: Create the Data

```python
import random

def simulate_admissions(n_applicants=10000):
    """Simulate Berkeley-like admissions data."""
    results = {"men": {"easy": [], "hard": []},
               "women": {"easy": [], "hard": []}}

    for i in range(n_applicants):
        # Half men, half women
        gender = "men" if i < n_applicants // 2 else "women"

        # Men apply mostly to the easy department (80%)
        # Women apply mostly to the hard department (80%)
        if gender == "men":
            dept = "easy" if random.random() < 0.80 else "hard"
        else:
            dept = "easy" if random.random() < 0.20 else "hard"

        # Admission rates: easy dept ~60%, hard dept ~20%
        # Women get a SLIGHT ADVANTAGE in both (+2%)
        if dept == "easy":
            rate = 0.62 if gender == "women" else 0.60
        else:
            rate = 0.22 if gender == "women" else 0.20

        admitted = random.random() < rate
        results[gender][dept].append(admitted)

    return results

```

```
data = simulate_admissions()
```

Step 2: Analyze the Data

```
def admission_rate(lst):
    """Calculate admission rate from a list of True/False."""
    if len(lst) == 0:
        return 0
    return sum(lst) / len(lst)

# Department-level rates
print("=== DEPARTMENT LEVEL ===")
for dept in ["easy", "hard"]:
    m = admission_rate(data["men"][dept])
    w = admission_rate(data["women"][dept])
    winner = "Women win!" if w > m else "Men win!"
    print(f"{dept.upper():5s} dept: Men={m:.1%}, Women={w:.1%}  --> {winner}")

# Overall rates
print("\n=== OVERALL ===")
all_men = data["men"]["easy"] + data["men"]["hard"]
all_women = data["women"]["easy"] + data["women"]["hard"]
m_all = admission_rate(all_men)
w_all = admission_rate(all_women)
print(f"Overall:    Men={m_all:.1%}, Women={w_all:.1%}")
print(f"Men applied to easy dept: {len(data['men']['easy'])}")
print(f"Women applied to easy dept: {len(data['women']['easy'])}")
```

Output

```
=== DEPARTMENT LEVEL ===
EASY dept:  Men=60.2%, Women=62.4% -> Women win!
HARD dept:  Men=20.1%, Women=22.3% -> Women win!

=== OVERALL ===
Overall:  Men=52.1%, Women=30.2%
Men applied to easy dept:  4010
Women applied to easy dept:  1005
```

Women win in *every* department, but lose overall! That's Simpson's Paradox.

Step 3: Visualize It

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 3, figsize=(12, 4.5), sharey=True)

categories = ["Men", "Women"]
colors = ['#2563EB', '#DC2626']

# Easy dept
rates = [admission_rate(data["men"]["easy"]),
         admission_rate(data["women"]["easy"])]
axes[0].bar(categories, rates, color=colors, alpha=0.8)
axes[0].set_title("Easy Department", fontweight='bold')
axes[0].set_ylabel("Admission Rate")
```

```
for i, r in enumerate(rates):
    axes[0].text(i, r + 0.01, f'{r:.1%}', ha='center', fontweight='bold')

# Hard dept
rates = [admission_rate(data["men"]["hard"]),
         admission_rate(data["women"]["hard"])]
axes[1].bar(categories, rates, color=colors, alpha=0.8)
axes[1].set_title("Hard Department", fontweight='bold')
for i, r in enumerate(rates):
    axes[1].text(i, r + 0.01, f'{r:.1%}', ha='center', fontweight='bold')

# Overall
rates = [admission_rate(all_men), admission_rate(all_women)]
axes[2].bar(categories, rates, color=colors, alpha=0.8)
axes[2].set_title("Overall", fontweight='bold')
for i, r in enumerate(rates):
    axes[2].text(i, r + 0.01, f'{r:.1%}', ha='center', fontweight='bold')

# Annotations
axes[0].text(0.5, 0.95, 'Women higher!', transform=axes[0].transAxes,
             ha='center', color='green', fontweight='bold', fontsize=11)
axes[1].text(0.5, 0.95, 'Women higher!', transform=axes[1].transAxes,
             ha='center', color='green', fontweight='bold', fontsize=11)
axes[2].text(0.5, 0.95, 'Men higher!?', transform=axes[2].transAxes,
             ha='center', color='red', fontweight='bold', fontsize=11)

for ax in axes:
    ax.set_ylim(0, 0.85)
    ax.grid(axis='y', alpha=0.3)

plt.suptitle("Simpson's Paradox", fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()
```

The Lesson

Aggregated data can be misleading. The overall numbers hide the fact that women were choosing harder departments. The "confounding variable" (department choice) reverses the trend. This is why scientists always look for hidden variables before drawing conclusions!

This happens everywhere: medicine, sports, economics, politics...

Challenge: Create Your Own Paradox

Can you create a scenario with **three** departments where the paradox still holds? What about a sports example: a player with a better batting average in every season but a worse average overall?

Paradox V: Six Degrees of Separation

The Story

The idea is simple and mind-blowing: **any two people on Earth** can be connected through at most about **6 intermediate acquaintances**.
You know someone, who knows someone, who knows someone... and in just 6 steps, you can reach anyone — the President of the United States, a farmer in Mongolia, a fisherman in Brazil.
Facebook confirmed this in 2016: the average distance between any two users was **3.57 steps**.

Step 1: The Power of Exponential Growth

If each person knows about 150 people (Dunbar's number), how many people can you reach?

```python
contacts = 150

print("Steps | People reachable")
print("-" * 30)
for step in range(7):
    reach = contacts ** step
    print(f"  {step}    | {reach:>15,}")

print(f"\nWorld population: 8,000,000,000")
print(f"Reached in 5 steps: {contacts**5:,}")
```

Output

```
Steps | People reachable
---------------
0 | 1
1 | 150
2 | 22,500
3 | 3,375,000
4 | 506,250,000
5 | 75,937,500,000

World population:  8,000,000,000
Reached in 5 steps:  75,937,500,000
```

By step 5, you can *theoretically* reach more people than exist on Earth!

But wait...

This is a simplification! In reality, your friends know each other (overlap), so the real reach is smaller. That's why we need to simulate an actual *network* to see what really happens.

Step 2: Build a Social Network

We'll build a **small-world network** — a model invented by Watts and Strogatz in 1998 that captures how real social networks work.

```python
import random

def build_network(n_people, n_contacts, rewire_prob=0.1):
    """Build a small-world network."""
    # Start: connect each person to their nearest neighbors
    # (like people sitting in a circle knowing those next to them)
    network = {i: set() for i in range(n_people)}

    # Connect to nearest neighbors (ring lattice)
    for i in range(n_people):
        for j in range(1, n_contacts // 2 + 1):
            neighbor = (i + j) % n_people
            network[i].add(neighbor)
            network[neighbor].add(i)

    # Rewire some connections randomly (this creates "shortcuts")
    for i in range(n_people):
        for j in range(1, n_contacts // 2 + 1):
            if random.random() < rewire_prob:
                neighbor = (i + j) % n_people
                # Remove old connection and add a random one
                if neighbor in network[i] and len(network[i]) > 1:
                    network[i].discard(neighbor)
                    network[neighbor].discard(i)
                    # Pick a random new connection
                    new = random.randint(0, n_people - 1)
                    while new == i or new in network[i]:
                        new = random.randint(0, n_people - 1)
                    network[i].add(new)
                    network[new].add(i)

    return network
```

Step 3: Find the Shortest Path (BFS)

To find the shortest path between two people, we use **Breadth-First Search** (BFS) — one of the most important algorithms in computer science:

```python
def shortest_path(network, start, end):
    """Find shortest path using BFS."""
    if start == end:
        return [start]

    visited = {start}
    queue = [(start, [start])]  # (current node, path so far)

    while queue:
        current, path = queue.pop(0)

        for neighbor in network[current]:
            if neighbor == end:
                return path + [neighbor]  # found it!

            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))

```

```
    return []  # no path found

# Build and test
net = build_network(200, 6, 0.1)
path = shortest_path(net, 0, 100)
print(f"Shortest path from person 0 to person 100:")
print(f"Path: {path}")
print(f"Steps: {len(path) - 1}")
```

Output

```
Shortest path from person 0 to person 100:
Path:  [0, 3, 47, 98, 100]
Steps:  4
```

Step 4: Measure All Distances

Let's measure the average distance between many random pairs:

```
import random

# Build a network
N = 500
net = build_network(N, 10, 0.1)

# Sample random pairs and measure distances
distances = []
for trial in range(2000):
    a = random.randint(0, N - 1)
    b = random.randint(0, N - 1)
    if a != b:
        path = shortest_path(net, a, b)
        if path:
            distances.append(len(path) - 1)

avg = sum(distances) / len(distances)
print(f"Network: {N} people, 10 contacts each")
print(f"Average distance: {avg:.2f} steps")
print(f"Maximum distance: {max(distances)} steps")
```

Step 5: Histogram of Distances

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 4.5))
plt.hist(distances, bins=range(1, max(distances) + 2),
         color='#16A34A', alpha=0.8, edgecolor='white',
         linewidth=1.2, density=True)
plt.axvline(x=avg, color='#DC2626', linestyle='--', linewidth=2,
            label=f'Mean = {avg:.1f} steps')
plt.xlabel('Shortest path length (steps)', fontsize=12)
plt.ylabel('Fraction of pairs', fontsize=12)
plt.title(f'Six Degrees: Network of {N} People', fontsize=14,
          fontweight='bold')
plt.legend(fontsize=11)
plt.grid(axis='y', alpha=0.3)
```

```
plt.tight_layout()
plt.show()
```

Small-World Effect

Even though each person only knows 10 others, the average path is very short! This is because the random "shortcuts" (rewired connections) dramatically shrink the distances. This is exactly what happens in real life: your friend who moved abroad connects you to an entirely different social circle.

Challenge: Explore the Parameters

Try changing the network parameters and observe:

1. What happens if `rewire_prob = 0`? (No shortcuts — a pure ring)
2. What happens if `rewire_prob = 1`? (Fully random network)
3. How does increasing `n_contacts` affect the average distance?
4. Can you find a combination where the average distance is less than 3?

Make a plot showing average distance vs. rewiring probability!

Putting It All Together

You've now programmed five probability paradoxes! Let's make a final summary plot:

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 2, figsize=(12, 9))

# (a) Monty Hall
ax = axes[0, 0]
# (re-run the Monty Hall simulation from Section 2)
stay_r, switch_r = [], []
sw, stw = 0, 0
for g in range(1, 5001):
    car = random.randint(0, 2)
    pick = random.randint(0, 2)
    if pick == car: stw += 1
    else: sw += 1
    stay_r.append(stw / g)
    switch_r.append(sw / g)
ax.plot(stay_r, color='#DC2626', linewidth=1)
ax.plot(switch_r, color='#16A34A', linewidth=1)
ax.axhline(1/3, color='#DC2626', linestyle='--', alpha=0.5)
ax.axhline(2/3, color='#16A34A', linestyle='--', alpha=0.5)
ax.set_title('(a) Monty Hall', fontweight='bold')
ax.set_ylabel('Win rate')
ax.set_ylim(0.15, 0.85)
ax.grid(True, alpha=0.3)

# (b) Boy or Girl
ax = axes[0, 1]
boy_cnt, both_cnt = 0, 0
probs = []
for i in range(5000):
    c1 = random.choice([0, 1])
    c2 = random.choice([0, 1])
    if c1 == 1 or c2 == 1:
        boy_cnt += 1
        if c1 == 1 and c2 == 1:
            both_cnt += 1
    probs.append(both_cnt / boy_cnt if boy_cnt > 0 else 0)
ax.plot(probs, color='#2563EB', linewidth=1)
ax.axhline(1/3, color='#DC2626', linestyle='--', linewidth=2)
ax.set_title('(b) Boy or Girl', fontweight='bold')
ax.set_ylim(0.2, 0.5)
ax.grid(True, alpha=0.3)

# (c) Birthday Problem
ax = axes[1, 0]
sizes = list(range(2, 61))
theory = [birthday_theory(n) for n in sizes]
ax.plot(sizes, theory, color='#DC2626', linewidth=2)
ax.axhline(0.5, color='gray', linestyle=':')
ax.axvline(23, color='#16A34A', linestyle='--')
ax.set_title('(c) Birthday Problem', fontweight='bold')
ax.set_xlabel('People in room')
ax.set_ylabel('P(shared birthday)')
```

```
ax.grid(True, alpha=0.3)

# (d) Six Degrees
ax = axes[1, 1]
ax.hist(distances, bins=range(1, max(distances) + 2),
        color='#16A34A', alpha=0.8, edgecolor='white', density=True)
ax.axvline(avg, color='#DC2626', linestyle='--', linewidth=2)
ax.set_title('(d) Six Degrees', fontweight='bold')
ax.set_xlabel('Path length')
ax.grid(axis='y', alpha=0.3)

plt.suptitle('Five Probability Paradoxes -- Simulated!',
             fontsize=15, fontweight='bold')
plt.tight_layout()
plt.show()
```

What You Learned

Python Skill	Where You Used It
Variables & math	Everywhere!
`for` loops	Running thousands of simulations
`if`/`else` conditions	Checking if the player won
Functions	`has_birthday_match()`, `build_network()`
Lists	Storing results, birthdays, paths
Dictionaries	Network adjacency, grouped data
`random` library	Simulating chance events
`matplotlib` plotting	Convergence plots, bar charts, histograms
BFS algorithm	Finding shortest paths in networks
Monte Carlo simulation	Estimating probabilities empirically

Key Takeaways

1. **Monty Hall:** Always switch — you win 2/3 of the time, not 1/2.
2. **Boy or Girl:** *How* you get information changes the probability. The answer is 1/3, not 1/2.
3. **Birthday Problem:** With just 23 people, P(match) $> 50\%$. Our intuition about combinations is terrible.
4. **Simpson's Paradox:** Aggregated data can reverse trends. Always look for hidden variables.
5. **Six Degrees:** A few random connections create shortcuts that connect entire networks. The world is small.

Where to Go Next

- **More Python:** https://www.codecademy.com/learn/learn-python-3
- **Probability:** 3Blue1Brown on YouTube — beautiful visual explanations
- **Data Science:** Kaggle.com — practice with real datasets
- **Network Science:** "Linked" by Albert-László Barabási (popular science book)

Happy coding!

The best way to learn is to *experiment*.
Change the numbers. Break things. Ask "what if?"
That's what programming — and science — is all about.