# When Maths meets Python: 100 Coding Experiments for Problem Solvers

Robin Wang

**Preface**

Welcome to a journey where mathematics and Python programming come together to reveal new dimensions of problem-solving. In this book, we'll explore intriguing mathematical challenges, each paired with Python code that makes complex ideas tangible. Through this approach, you'll see how coding can transform abstract maths concepts into interactive, engaging problems. Many of the quizzes in this book are inspired by or adapted from esteemed sources such as the Australian Mathematics Competition (AMC), Presh Talwalkar's *Maths Puzzles*, and mathematics textbooks such as Cambridge IB Mathematics High Level and VCE Specialist Mathematics. These carefully chosen problems highlight how programming can be a powerful tool for solving real-world mathematical challenges.

Maths is a language of patterns, logic, and creativity. When paired with Python, it becomes even more powerful, unlocking new ways to explore and experiment with mathematical ideas. Keep an open mind, stay curious, and enjoy the journey – there's beauty in mathematics that often reveals itself when you least expect it, and coding will help you uncover insights that might otherwise remain hidden.

**What does this book cover?**

This book leads you through some of the most fascinating areas of mathematics, using Python to tackle a wide range of problems. Whether you're new to coding or already comfortable with Python, the pages ahead will guide you in applying code to approach challenging mathematical questions. Here's what you can look forward to:

- **Chapter 1: Probability**
  We begin with the world of chance and uncertainty, exploring probability through game play to make abstract concepts come to life. Probability is central to understanding randomness in daily life, and by approaching it as a series of interactive games, you'll gain hands-on experience with randomness and strategy. From simulating dice rolls to modelling dynamic events, you'll use Python to run simulations, calculate probabilities, and gain a richer understanding of randomness and statistical patterns – all while engaging in interactive and enjoyable experiments.
- **Chapter 2: Combinatorics**
  This chapter ventures into combinatorics, or the art of counting. We'll solve problems ranging from simple combinations to complicated arrangements. With Python, you'll see how coding can make light work of combinatorial calculations and reveal insights that would be difficult to uncover with manual counting.
- **Chapter 3: Sequences and Series**
  Here, we explore the beauty of mathematical patterns. You'll encounter challenges that require recognising patterns, extending sequences, or deriving series. Python will help you visualise these patterns, making abstract concepts more concrete and allowing you to appreciate the elegance of mathematical structure.
- **Chapter 4: Equations and Relations**
  In this final chapter, we'll tackle problems involving equations and relations. Python offers tools to solve complex equations and relations, simplifying problems that would otherwise

be too tedious to solve by hand. You'll see firsthand how coding can enhance your understanding of algebraic principles and the connections between numbers.

**Why does code matter?**

Coding turns theory into practice. When we write Python code to solve a maths problem, we aren't just finding a solution; we're building a versatile tool that can solve similar problems repeatedly with ease and efficiency. Coding allows us to automate processes, experiment with different scenarios, and explore mathematical ideas more dynamically than with pen and paper alone. In this way, coding enriches the modelling experience and helps uncover deeper insights.

In this book, certain coding techniques, such as list comprehensions, are used to make the examples more concise and efficient. If these concepts are new to you, I recommend exploring my other book, *A Not Too Short Introduction to Python*, where these foundational techniques are explained in an accessible manner, making them easy to grasp.

**Ready for the challenge?**

In this book, each chapter is designed to challenge you while demonstrating Python's power in maths problem-solving. Along the way, you'll learn how coding can simplify complex concepts, whether by visualising a pattern, generating sequences, or calculating probabilities.

So, whether you're drawn to probability, fascinated by combinatorics, curious about patterns, or interested in equations, this book has something for you. I invite you to join me as we explore the synergy between maths and programming. Let's uncover the beauty of mathematics through Python, and above all, enjoy the journey!

**Why are answers not provided?**

I've intentionally chosen not to provide the answer to each question. This approach encourages you to engage more deeply with problem-solving itself. By focusing on modelling and reaching solutions through code, you'll develop stronger analytical skills and better understand Python's applications in real-world challenges. Rather than offering the answer, you'll experience the process of discovery, building resilience and curiosity – qualities that are key in both mathematics and programming.

Dr. Robin's Python Lab

Melbourne

# Chapter 1. Probability: The Art of Chance

Probability, or probabilistic modelling, is, in my opinion, one of the best topics for learning how to code. Through coding, especially with simulations, students can see firsthand why the mathematical calculations in probability make sense. This is particularly valuable when exploring more complex ideas, like conditional probability.

In real life, simulations offer a practical way to estimate the likelihood of events, as the complexity of real-world scenarios can make direct mathematical analysis difficult. For this reason, I've prepared a large variety of questions in this chapter to help you see how simulations can simplify these problems. But before we start, it is important to understand that while simulations are useful, they are not a replacement for mathematical analysis.

**Why not use simulations for everything?**

With the incredible processing power of modern computers, one might wonder why we need mathematical analysis at all when simulations can often provide quick answers. While simulations are undoubtedly useful, they cannot fully replace mathematical reasoning for several reasons.

First, even the fastest computers can encounter difficulties with complex simulations. When modelling large systems or running detailed analysis, simulations can become highly time consuming. Certain problems, such as those classified as NP-hard, are so computationally intensive that simulation can be impractical or impossible within a reasonable time frame.

Second, mathematical analysis offers deeper insights that simulations often cannot. While simulations may produce results, they do not always explain why something happens. Through analysis, we can explore the underlying principles and patterns behind the results, gaining a better understanding of the problem at hand. It's the difference between merely watching a game unfold and fully understanding the rules that govern how it is played.

Additionally, mathematical reasoning provides a solid foundation for meaningful data interpretation. In fields like engineering, economics, and computer science, a deep understanding of probability and statistics is essential for conducting accurate data analysis. Simulations may be valuable for experimentation, but without a strong mathematical foundation, it becomes difficult to correctly interpret and apply the results to real-world situations.

In coding probability, the beauty lies in the art of modelling. When you code a probability question, you are transforming an abstract idea into a living model that behaves according to real-world rules. Through code, concepts like randomness, chance, and statistical likelihood aren't just theoretical; they're interactive. Each line of code becomes a step in building a system that can simulate experiment, test hypotheses, and bring abstract probabilities to life. Modelling through coding doesn't just make probability problems solvable, it makes them understandable, dynamic, and fun to explore.

## 1.1 Basic probability and simple events

### *Experiment 1.*

**Find the probability of getting at least 1 head when you toss 4 fair coins.**

We will estimate the probability by simulating the tossing of 4 coins 1000 times. Before we start coding, let's break down how we can model this problem and then translate it into Python.

- *Random coin toss:* We use *random.randint(1, 2)* to simulate each coin toss. A value of 1 represents heads, and 2 represents tails.
- *Checking for heads:* In each trial, we simulate tossing 4 coins. If we get a head (value 1) on any coin, we stop checking the other coins using a *break* statement and count this trial as a success.
- *Final calculation:* After running the simulation for 1000 trials, we calculate the probability by dividing the number of trials that resulted in at least 1 head by the total number of trials.

Here's the code:

```
import random

n = 1000  # Number of simulations
count = 0  # Count how many times we get at least 1 head

for i in range(n):
    # Simulate tossing 4 coins
    for j in range(4):
        # Randomly simulate heads or tails (1 for heads, 2 for tails)
        if random.randint(1, 2) == 1:
            count += 1  # We got at least 1 head
            break  # No need to check the other coins, move to the next trial

# Calculate and print the probability
print("The probability is: {}% ({}/{})".format((count / n) * 100, count, n))
```

### *Experiment 2.*

**Jack has a ring of ten keys in the same shape among which only one key can open the door of his home. One night he was drunk and randomly picked one key from the ring to open the door. What is the probability that he would open the door on the fifth try?**

Assume he randomly selects on key at a time and replaces it each time. Let's model this problem:

- *Randomly picking a key: We'll* use the function *random.randint(1, 10)* to simulate Jack randomly choosing one of the 10 keys. Each key has an equal chance of being picked, and we keep repeating this until Jack picks the correct key (1).
- *Counting the attempts:* We track the number of attempts Jack makes before finding the correct key. If Jack opens the door on the fifth attempt, we mark it as a successful trial.
- *Final calculation: After* running 1000 trials, we estimate the probability by dividing the number of successful trials by the total number of trials.

Now let's write the code:

```
 1. import random
 2.
 3. n = 1000  # Number of simulations
 4. count = 0  # Count how many times Jack opens the door on the 5th try
 5.
 6. for i in range(n):
 7.     a = random.randint(1, 10)  # Randomly pick a key (1 to 10)
 8.     j = 1  # Track the number of attempts
 9.     while a != 1:  # Keep picking until the correct key (key 1) is
selected
10.         a = random.randint(1, 10)  # Randomly pick again
11.         j += 1  # Increment the number of attempts
12.     if j == 5:  # Check if Jack opened the door on the 5th try
13.         count += 1  # Increment the count for successful 5th try
openings
14.
15. # Calculate and print the probability
16. print("The probability is: {}% ({}/{})".format((count / n) * 100,
count, n))
```

### *Experiment 3.*

**A system has 100 accounts, two of which have bad passwords (let's call these bad accounts). If someone could only test 20 accounts, what are the chances that one will net a bad account?**

The question is asking about the chances of selecting at least one bad account from the 20 tested accounts. Let's model the problem first:

- *Simulating accounts:* We represent the 100 accounts with a list, where 0 and 1 are the accounts with bad passwords. We shuffle the list in each simulation to randomise the order of the accounts.
- *Selecting the accounts:* After shuffling, we take the first 20 accounts to simulate a selected subset of the accounts. If either 0 or 1 (the bad accounts) are in the selected subset, we count this simulation as a success.
- *Final calculations:* After running 1000 simulations, we calculate the probability of finding a bad account by dividing the number of successes by the total number of simulations.

Let's write the code:

```
 1. import random
 2.
 3. n = 1000  # Number of simulations
 4. count = 0  # Count of successful attempts (finding a bad account)
 5. a = list(range(100))  # List of 100 accounts (0 and 1 are bad
accounts)
 6.
 7. for i in range(n):
 8.     random.shuffle(a)  # Shuffle the accounts
 9.     b = a[:20]  # Select 20 accounts to test
```

```
    if 0 in b or 1 in b:  # If one of the bad accounts is in the selection
        count += 1

# Calculate and print the probability
print('The probability that they contain the bad password is: {}%, {}/{}'.format(count / n * 100, count, n))
```

## 1.2 Combinatorial probability

***Experiment 4.***

**A committee of 4 is randomly chosen from 6 girls and 5 boys. What is the probability that the committee contains exactly 3 girls?**

We will estimate the probability by simulating the random selection of the committee 1000 times. Before we start to code, let's break down how we can model this problem:

- *Random committee selection*: We have a pool of 6 girls and 5 boys, represented as a list where each girl is denoted by 1 and each boy by 2. To form a committee, we shuffle this list and select the first 4 members.
- *Checking for 3 girls*: After each trial, we check if exactly 3 girls are selected by counting the number of 1s in the chosen committee. If the count equals 3, we mark the trial as a success.
- *Final calculation*: We estimate the probability of selecting exactly 3 girls by dividing the number of successful trials by the total number of trials.

```
import random

n = 1000  # Number of simulations
count = 0  # Count how many times the committee has exactly 3 girls

# Represent the pool: 1 for girls, 2 for boys
pool = [1] * 6 + [2] * 5

for i in range(n):
    random.shuffle(pool)  # Randomly shuffle the pool of girls and boys
    com = pool[:4]  # Select the first 4 members to form the committee
    if com.count(1) == 3:  # Check if there are exactly 3 girls
        count += 1  # Increment count for a successful trial

# Calculate and print the probability
print("The probability is: {}% ({}/{})".format((count / n) * 100, count, n))
```