



www.maynet.com

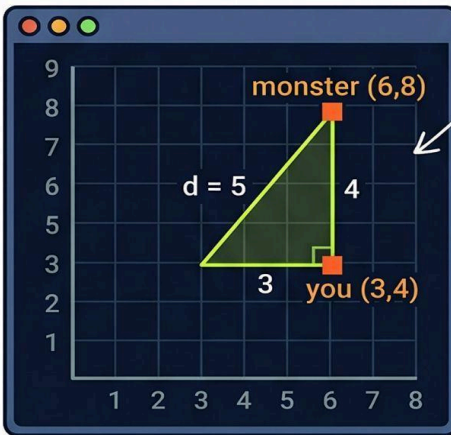
A HANDS-ON GAME BOOK · AGES 13+

PART 1: A HANDS-ON SERIES

THINK LIKE AN AI

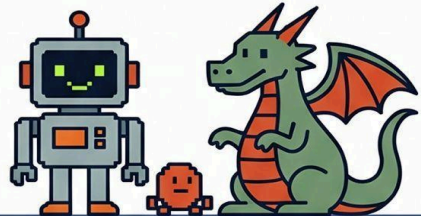
LEARN THE MATH BEHIND AI
BY BUILDING 8 PYTHON GAMES

PYTHON · PLAIN-ENGLISH MATH · YOUR AI CODING PARTNER



How close is the monster?
That triangle is how AI
measures similarity

`random.randint(1, 100) → 42`



KAPILA SOOD

PART 1 · A HANDS-ON SERIES

Think Like an AI

*Learn the Math Behind AI
by Building 8 Python Games*

PYTHON · PLAIN-ENGLISH MATH · YOUR AI CODING PARTNER

Kapila Sood

Think Like an AI, Part 1: Learn the Math Behind AI by Building 8 Python Games

Copyright © 2026 Kapila Sood. All rights reserved.

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the author, except for brief quotations in reviews and other noncommercial uses permitted by copyright law.

Code permission: Readers, parents and teachers may use, copy and adapt the Python code examples in this book for personal and classroom learning.

Trademarks: Python is a registered trademark of the Python Software Foundation. ChatGPT is a trademark of OpenAI. Google, Gemini, Gmail, Chrome, YouTube and NotebookLM are trademarks of Google LLC. Visual Studio Code is a trademark of Microsoft Corporation. Spotify, Netflix, Uber Eats, Meta and other product names mentioned are trademarks of their respective owners. This book is independent and is not affiliated with, sponsored by or endorsed by any of these companies.

Disclaimer: Software, websites and AI tools change often. Instructions were checked against the versions available in October 2026, and your screens may look slightly different. AI chat tools have minimum-age rules; young readers should use them only with a parent or teacher, as described in each chapter's Teacher/Parent Toolkit. The author has made every effort to ensure accuracy but accepts no liability for any loss or damage arising from use of the information in this book.

First edition, 2026

Independently published

*For every curious kid who has ever asked, “But how does it
know?”*

Contents

How to Use This Book	
Chapter 1: Your Lab — First Scripts and Error Messages	1
Chapter 2: Mind-Reading Number Game — Binary Search	15
Chapter 3: Monster Chase — Distance and Vectors	31
Chapter 4: Creature Sorter — Classification with Nearest Neighbors	50
Chapter 5: Rock-Paper-Scissors Mind Reader — Prediction from Probability	69
Chapter 6: Hot-and-Cold Learning — How Models Learn with Gradient Descent	90
Chapter 7: Spam vs Ham Detector — Sorting Text with Word Counts (Naive Bayes)	111
Chapter 8: Word Wizard — How Chatbots Predict the Next Word (Markov Chains)	138
Glossary	164
Enjoyed This Book?	167
Coming Next: Parts 2 and 3	168
About the Author	169

How to Use This Book

This book teaches the ideas inside modern AI by having you build small games in Python. Each chapter starts with a game, works out the math by hand in plain English, turns it into a short script, and then shows you where the same idea runs inside apps you use every day.

Who This Book Is For

Curious beginners with no coding experience, and the parents and teachers who learn alongside them. If you can add, subtract, multiply and divide, you have all the math you need. Every new idea, from square roots to probabilities, is explained the first time it appears.

What You Need

- **A computer.** Windows or Mac, with an internet connection.
- **Python 3 and VSCode.** Both are free. Chapter 1 walks you through the setup step by step.
- **A web browser.** For the interactive widgets that let you see each idea move.
- **A free ChatGPT or Gemini account (optional).** Used in the Level Up sections. Check the age rules first; each Teacher/Parent Toolkit lists them.

How Each Chapter Works

- **Learn it by hand.** A game scene introduces the idea. “Try one by hand” puzzles check your understanding before you write any code.
- **Build it.** You write a short script using only Python’s built-in tools: math, random and json. No heavy libraries.
- **Test it.** You run the script and compare your output with the expected output in the book.
- **Level Up.** You team up with ChatGPT or Gemini to extend the game, and learn to check what the AI gives you.
- **Same code, new job.** You reuse the script on a real problem, from movie recommendations to spam filters.
- **In the real world.** You see how companies like Google, Spotify and Netflix use the same math.
- **Boss Challenges.** Harder puzzles that prove you’ve mastered the chapter.
- **Teacher/Parent Toolkit.** Learning goals, likely roadblocks with fixes, and classroom extensions.

Tips for Success

- **Go in order.** Each chapter reuses ideas and code from the ones before it.
- **Run everything.** If your output differs from the book's, the difference is a clue, not a failure.
- **Read errors from the bottom.** The last line says what went wrong, and the line number says where. Chapter 1 shows you how.

Chapter 1: Your Lab — First Scripts and Error Messages

You will set up Python and VSCode, build a loot chest that rolls dice, and fix a broken game by reading its error messages.

You need a Windows or Mac computer, an internet connection, and about 30 minutes for the setup. Every later chapter runs in the lab you build here.

Set Up Your Lab

Your lab has three parts. **Python** is the program that runs your code. **VSCode** is the editor where you write it. The **Python extension** connects the two, so VSCode knows how to run a `.py` file.

Install Python on Windows

1. Go to python.org/downloads and click the yellow **Download Python** button. On Windows it downloads the **Python install manager**, which installs Python and keeps it updated.
2. Open the downloaded file and click **Install**.
3. Open the Start menu, type `terminal`, and open **Terminal**. Type this and press Enter:

```
python --version
```

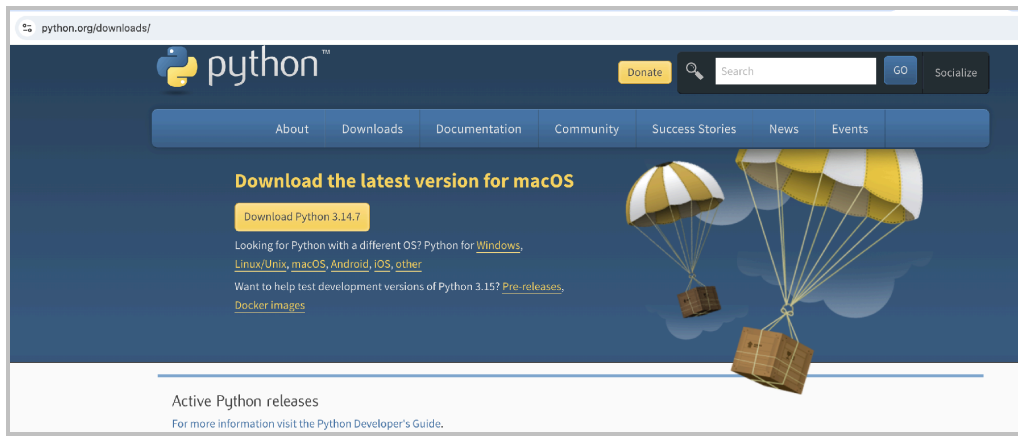
The first time, the install manager downloads the newest Python for you. When it finishes, you should see a version number like `Python 3.14.7`. If it asks whether to add a folder to your PATH, type `y` and press Enter.

Install Python on a Mac

1. Go to python.org/downloads and click the yellow **Download Python** button. You get a `.pkg` file.
2. Open it and click through the installer.
3. When it finishes, a `Python 3.14` folder opens. Double-click **Install Certificates.command** inside it. A Terminal window runs for a few seconds. When it says `update complete`, close it.
4. Open **Terminal** (press `Cmd + Space`, type `terminal`, press Enter). Type this and press Enter:

```
python3 --version
```

You should see `Python 3.14.7` or newer. On a Mac you always type `python3`, never plain `python`.



TEACHER/PARENT TOOLKIT

Likely roadblock (Windows): python opens the Microsoft Store, or the terminal says 'python' is not recognized.

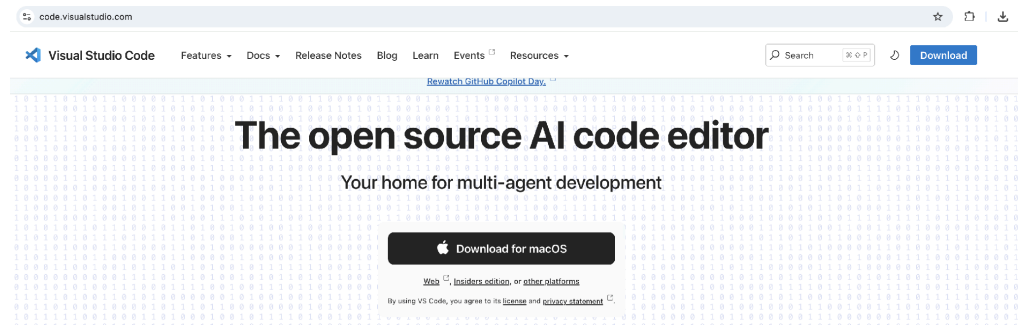
Fix: Make sure the Python install manager finished installing, then close the terminal and open a new one. If it still fails, open Start, search **Manage app execution aliases**, and switch on the entries named **Python (default)** and **Python install manager**.

Likely roadblock (Mac): python3 pops up a box asking to install "command line developer tools".

Fix: Cancel it. That box belongs to Apple's own tools. Run the python.org installer again, then open a fresh Terminal window.

Install VSCode and the Python extension

1. Download VSCode from code.visualstudio.com and install it.
2. Open VSCode. Click the **Extensions** icon on the left bar (four small squares) or press Ctrl + Shift + X (Cmd + Shift + X on a Mac).
3. Search for **Python** and install the one published by **Microsoft**.



Your First Script

1. Make a folder called `loot-lab` in your Documents folder.
2. In VSCode, choose **File** → **Open Folder** and pick `loot-lab`. If VSCode asks whether you trust the authors, click **Yes**. You are the author.
3. Click the **New File** icon next to the folder name and call the file `lab_check.py`.
4. Paste this code and save it with Ctrl + S (Cmd + S on a Mac):

```
lab_name = "Loot Lab"
player_name = "Rookie"
coins = 5
bonus = 10
print("Hello from Python!")
print(f>Welcome to the {lab_name}, {player_name}.")
print(f>You start with {coins} coins and find {bonus} more: {coins +
    bonus} total.")
```

5. Click the **Run Python File** button, the triangle at the top right of the editor. A terminal panel opens at the bottom, and you should see:

```
Hello from Python!
Welcome to the Loot Lab, Rookie.
You start with 5 coins and find 10 more: 15 total.
```

If VSCode asks you to pick a Python interpreter, choose the one that says 3.14. You can also run any script by typing in that terminal: `python3 lab_check.py` on a Mac, or `python lab_check.py` on Windows.

Three ideas carry this script:

- **Variables.** `coins = 5` puts the number 5 in a labelled box called `coins`. The `=` sign means "store this", not "equals". Change 5 to 8, run again, and the total becomes 18.
- **Strings.** Anything inside quote marks is text. Python calls text a **string**. `"Rookie"` is a string, and 5 is a number.
- **f-strings.** The `f` before a quote lets you drop variables into text with curly brackets. Python replaces `{coins + bonus}` with the answer, 15, before it prints.

Every program you build in this book is these pieces, arranged in smarter ways.

How a Loot Chest Decides: Rarity Bands

You open a treasure chest in a game. The game rolls a dice with 100 sides, numbered 1 to 100. A roll from 1 to 70 gives you common loot. A roll from 71 to 95 gives rare loot. A roll from 96 to 100 gives legendary loot. How likely is each one?

Count the numbers in each band. Every number is equally likely, so a band with more numbers wins more often.

1. Common runs from 1 to 70. That's 70 numbers.
2. Rare runs from 71 to 95. Subtracting gives $95 - 71 = 24$, but that's one short. Count on your fingers from 71 to 75: that's 5 numbers, and $75 - 71$ is only 4. Subtraction skips the first number, so you add 1 back: $95 - 71 + 1 = 25$.
3. Legendary runs from 96 to 100: $100 - 96 + 1 = 5$ numbers.
4. Check the total: $70 + 25 + 5 = 100$. Every roll lands in exactly one band.

Now turn each count into a chance. Common owns 70 of the 100 numbers, so its chance is 70 out of 100, or 70%. Rare gets 25%, and legendary gets 5%.

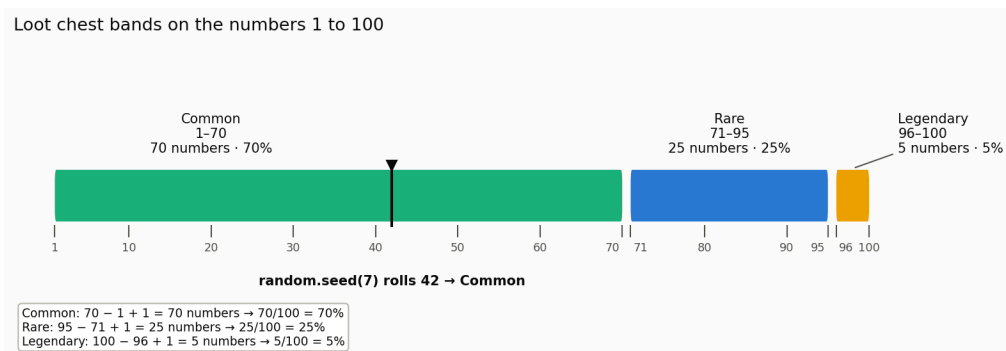


Figure 1.1: The width of each band is its chance. Common takes 70 of the 100 slots, so it wins 70% of the time.

In Figure 1.1, the green band stops at 70, and the blue band starts at 71. The gold legendary band is only 5 slots wide, which is why dragon eggs feel so rare. The black marker at 42 is the roll your script will make in the next section.

The number that says how likely something is has a name: **probability**. Machine learning runs on it. A spam filter says, "this email is 92% likely to be spam". A chatbot gives every possible next word a probability, then picks one, often with the same band trick you just used.

Here is the rule as a formula:

$$P(\text{band}) = \frac{\text{high} - \text{low} + 1}{100}$$

- P stands for probability, the chance of landing in the band.
- low and high are the band's first and last numbers. For rare, 71 and 95.
- The $+ 1$ counts the first number, which plain subtraction skips.
- The 100 on the bottom is how many sides the dice has.

For rare: $(95 - 71 + 1) \div 100 = 25 \div 100 = 25\%$.

Try one by hand. A game designer wants more rare loot. The new bands are 1–60 common, 61–90 rare, 91–100 legendary. What is each chance?

Answer: common $60 - 1 + 1 = 60$, so 60%. Rare $90 - 61 + 1 = 30$, so 30%. Legendary $100 - 91 + 1 = 10$, so 10%. The total is 100, so no roll is lost.

Try it live: open ch01-loot-roller.html and save the code on your machine. Double click the html file to open in your browser. Leave the seed on 7 and press **Roll 100**. The Expected column says 70, 25 and 5. The Actual column shows 70, 27 and 3. A chance tells you what happens on average, not what happens every time.

Loot Chest Roller

Move the band edges and watch every chance recalculate. With seed 7, the first roll matches your Python script.

Common ends at 70

Rare ends at 95

Seed (like random.seed)
7

Book chest 70 / 25 / 5

Stingy chest 90 / 9 / 1

Generous chest 40 / 40 / 20

Fair thirds 33 / 33 / 34

Band	Numbers	Working	Chance	Expected	Actual
Common	1-70	$70 - 1 + 1 = 70$	$70/100 = 70\%$	0.7	1
Rare	71-95	$95 - 71 + 1 = 25$	$25/100 = 25\%$	0.3	0
Legendary	96-100	$100 - 96 + 1 = 5$	$5/100 = 5\%$	0.1	0

Expected = chance $\times$ 1 roll. Actual counts wobble around it; more rolls, smaller wobble.

Roll 1 chest

Roll 10

Roll 100

Reset (start the seed again)

Roll 1: the dice say 42 $\rightarrow$ Common

Last rolls: 42

The loot widget after one roll. Drag either slider and the Working and Chance columns recalculate at once.

Build the Loot Roller

This script asks for your hero's name, rolls the 100-sided dice, and hands out loot by band.

1. In your `loot-1ab` folder, create a file called `loot_roller.py`.

2. Paste this code and save:

```
import random

random.seed(7) # same "random" roll every run, so your screen
               matches the book
player_name = input("Name your hero: ")
roll = random.randint(1, 100)
print(f"{player_name} opens a chest. The dice roll {roll}.")

# Bands: 1-70 common (70 numbers), 71-95 rare (25), 96-100 legendary
# (5)
if roll <= 70:
    print("🟢 Common: a wooden spoon.")
elif roll <= 95:
    print("🟡 Rare: a glowing map.")
else:
    print("🔴 Legendary: a dragon egg!")
```

3. Click **Run Python File**. The terminal asks for a name. Click inside the terminal, type `Rookie`, and press Enter. You should see:

```
Name your hero: Rookie
Rookie opens a chest. The dice roll 42.
🟢 Common: a wooden spoon.
```

Four parts carry the idea:

- `import random` loads Python's dice-rolling toolbox. `random.randint(1, 100)` rolls a whole number from 1 to 100, and both ends can come up.
- `random.seed(7)` fixes the starting point of the dice. With the seed set, the "random" roll is 42 on every computer, every run. Delete the line and each run rolls something new. Scientists and engineers seed their experiments so that someone else can repeat them and get the same numbers.
- `input(...)` prints a question, waits for you to type, and stores your answer in `player_name`.
- `if`, `elif`, `else` pick one branch. Python checks `roll <= 70` first. Only if that is false does it try `roll <= 95`, so a roll of 80 skips common and lands in rare. `else` catches everything left over: 96 to 100. The colon at the end of each line and the 4 spaces before each `print` tell Python which lines belong to which branch.

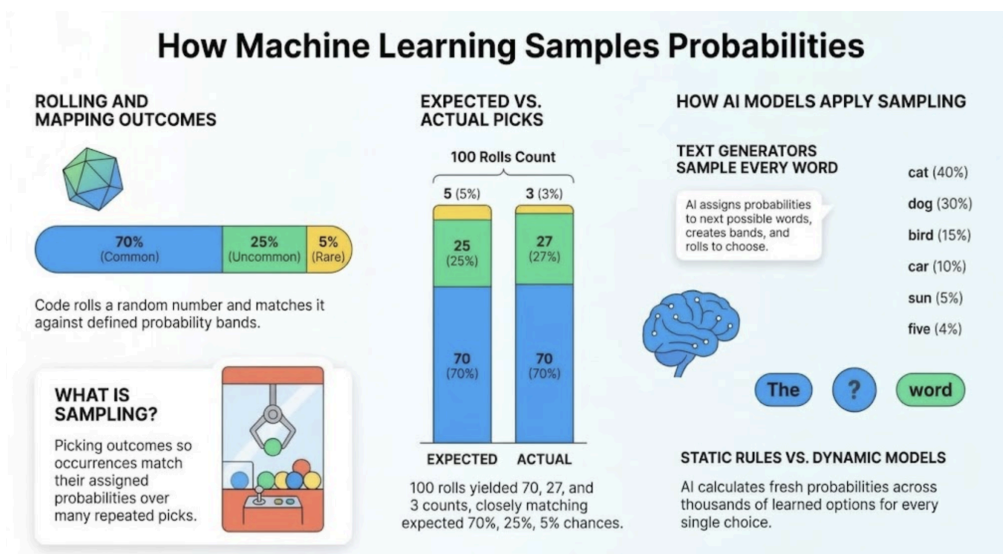
The `<=` sign means "less than or equal to". It makes 70 itself common and 95 itself rare, matching the bands in Figure 1.1. Change `<=` to `<` and 70 would slip into rare. That one symbol moves the band edge.

The ML idea: sampling from probabilities. Two parts of the script work as a team. `random.randint(1, 100)` rolls one number, and the `if` and `elif` bands turn that roll into a choice. Together they turn a list of chances (70%, 25%, 5%) into one concrete outcome. Statisticians call this **sampling**: picking one outcome so that, over many picks, each outcome turns up about as often as its probability says. The loot widget showed it

happening. After 100 rolls the counts were 70, 27 and 3, close to the expected 70, 25 and 5.

Chatbots and other text generators sample every time they write a word. The model gives every possible next word a probability, lays them out as bands, and rolls. Spam filters produce probabilities too, but they compare them with a cut-off instead of rolling.

Your chest has one limit: you typed its chances in by hand, and they never change. A real model calculates fresh probabilities for every single choice, from patterns it learned in huge amounts of text. It also chooses among thousands of options, not three.



Read Error Messages Calmly: The Bug Hunt

Every programmer sees errors every day. An error message is not Python telling you off. It is Python telling you where to look. Beginners panic at the wall of red text, but you only need the last line and one line number.

Read every error from the bottom up:

1. **Last line:** the error's type and a plain message, such as `NameError: name 'rol' is not defined`.
2. **Line number:** find the `line 9` part and jump to that line in VSCode. The line numbers run down the left edge of the editor.
3. **The marks:** the `^^^` or `~~~~~` marks under the code point at the exact part Python choked on.
4. If the line looks fine, check the line above it. Many mistakes only show up on the next line.

Create a file called `bug_hunt.py` and paste this broken game. It has four bugs. Don't fix anything yet.

```
import random

random.seed(7)
print("Bug Hunt: the loot chest is broken.")

hero = input("Name your hero: ")
gold = input("How many gold coins do you carry? ")
roll = random.randint(1, 100)
print(f"{hero} opens a chest. The dice roll {roll}.")

if roll <= 70:
    print("🟢 Common: 10 gold coins.")
    gold = gold + 10
elif roll <= 95:
    print("🟡 Rare: 50 gold coins.")
    gold = gold + 50
else:
    print("🔴 Legendary: 500 gold coins!")
    gold = gold + 500

print(f"{hero} now carries {gold} gold.")
```

VSCode may already draw red squiggles under some lines. Run it anyway. Python finds one bug per run, so you will fix them one at a time. Your file path will be longer than the ones shown here.

Bug 1: SyntaxError

```
File "bug_hunt.py", line 4
    print("Bug Hunt: the loot chest is broken.")
    ^
SyntaxError: '(' was never closed
```

A **SyntaxError** means Python can't read the code at all, so nothing runs. That's why the game didn't even ask your name. The last line says an opening bracket never got its partner, and the `^` points at it on line 4. Add `)` at the end of line 4 and run again.

Bug 2: IndentationError

```
File "bug_hunt.py", line 12
    print("🟢 Common: 10 gold coins.")
    ^^^^^
IndentationError: expected an indented block after 'if' statement on
line 11
```

Python uses spaces at the start of a line to group code. The `if` on line 11 ends with a colon, which promises an indented block, but line 12 starts at the left edge. Put 4 spaces before `print` on line 12 so it lines up with `gold = gold + 10` below it. Run again.

Bug 3: NameError

This time the game gets going. Type `Rookie`, then `5` for the gold:

```
Bug Hunt: the loot chest is broken.
Name your hero: Rookie
How many gold coins do you carry? 5
Traceback (most recent call last):
  File "bug_hunt.py", line 9, in <module>
    print(f"{hero} opens a chest. The dice roll {rol}.")
                                             ^^^
NameError: name 'rol' is not defined. Did you mean: 'roll'?
```

A **NameError** means you used a name Python has never seen. The word `Traceback` marks an error that happened while the program ran, after the first two lines worked. The `^^^` sits under `rol`, and Python even suggests the fix. Change `{rol}` to `{roll}` on line 9 and run again.

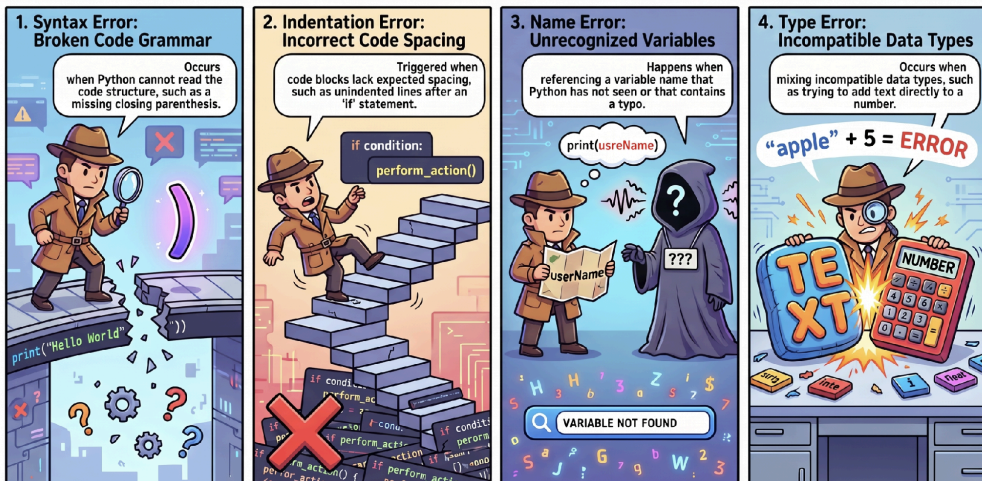
Bug 4: TypeError

```
Bug Hunt: the loot chest is broken.
Name your hero: Rookie
How many gold coins do you carry? 5
Rookie opens a chest. The dice roll 42.
● Common: 10 gold coins.
Traceback (most recent call last):
  File "bug_hunt.py", line 13, in <module>
    gold = gold + 10
           ~~~~~^~~~~
TypeError: can only concatenate str (not "int") to str
```

This one takes more thought. A **TypeError** means you mixed two kinds of data that don't fit together. "Concatenate" means gluing text end to end. `str` is short for string, and `int` is short for integer, a whole number.

So, Python is saying: `gold` holds text, and you tried to glue a number onto it. But you typed `5`, so why is `gold` text? Because `input()` always hands back text, even when you type digits. The `gold` box holds `"5"`, the character, not `5`, the number. Hold that thought, because the next section starts from here.

The Bug Hunt: Understanding 4 Common Code Errors



Level Up: Ask ChatGPT or Gemini to Explain an Error

An AI partner is great at explaining errors. It is risky at fixing them, because a fix you don't understand teaches you nothing and can hide a worse bug. So, ask it to explain, not to fix.

Open ChatGPT (chatgpt.com) or Gemini (gemini.google.com). Paste this prompt. Then add your code and the full error, from Traceback down to the last line.

I'm learning Python. My program crashed with the error below. Don't rewrite my code. Explain in plain words:

1. What this type of error means.
2. Which line caused it, and why.
3. A hint about what to change, without the finished answer.

My code:

[paste bug_hunt.py here]

The full error:

[paste the error here]

This prompt works because it gives the AI everything it needs and sets clear limits. It has the code, the exact error, and your level. It also has a job in three numbered parts. "Why is my code broken?" with no error pasted gets a guess back.

A good answer says that `input()` returns a string, and points you at line 7, where `gold` gets its value. The hint should be something like "convert the text to a number before you do math with it". Check the answer against what you already know. Does it name line 13 and line 7? Does it match the `~~~^~~~` marks?

When the AI Gets It Wrong

Suppose you skip the prompt and type "fix this error" instead. A realistic reply looks like this:

```
gold = gold + str(10)
```

The error goes away. The game runs to the end. Look closely at the last line:

```
Rookie now carries 510 gold.
```

You carried 5 gold and found 10, so you should have 15. The AI turned 10 into text and glued "5" and "10" together into "510". The crash is gone, but the answer is wrong, and nothing on screen warns you. That's worse than a crash.

The real problem sits on line 7, not line 13. `gold` should be a number from the start. Wrap the `input()` in `int()`, which turns text like "5" into the number 5. Here is the fixed line 7.

```
gold = int(input("How many gold coins do you carry? "))
```

Put line 13 back to `gold = gold + 10` and run the game once more. You should see this.

```
Bug Hunt: the loot chest is broken.  
Name your hero: Rookie  
How many gold coins do you carry? 5  
Rookie opens a chest. The dice roll 42.  
● Common: 10 gold coins.  
Rookie now carries 15 gold.
```

You caught the bad fix with one line of hand math: $5 + 10 = 15$. Fix the cause, not the symptom. The AI is your assistant, and you stay the engineer.

Same Bands, Real AI: How a Chatbot Picks Its Next Word

Your loot chest turns a probability into a choice. Chatbots do the same thing with words. Say a model decides the words after "The dragon" have these chances: "sleeps" 60%, "roars" 30%, "sings" 10%. It can lay them out as bands on 1 to 100 and roll.

Save this as `next_word.py` and run it:

```
import random

random.seed(7)
roll = random.randint(1, 100)
# Word bands: "sleeps" 1-60 (60%), "roars" 61-90 (30%), "sings"
# 91-100 (10%)
if roll <= 60:
    next_word = "sleeps"
elif roll <= 90:
    next_word = "roars"
else:
    next_word = "sings"
print(f"Roll {roll}: The dragon {next_word}.")
```

Expected output:

Roll 42: The dragon sleeps.

The same seed gives the same 42, and 42 falls in the "sleeps" band. Delete the seed line and run it a few times. Most runs say "sleeps", but "roars" and even "sings" turn up. That's why a chatbot can answer the same question in different words each time. Real models choose from thousands of words, and they learn the chances from huge amounts of text. You will build a small version of that in Chapter 8.

How Gemini and Gmail Use Probability

Google's [Gemini API documentation](#) explains how its models pick the next piece of text, called a token. The model ranks every token in its vocabulary by probability, and a setting called **temperature** controls how random the pick is. At a temperature of 0, the model always takes the most probable token, like a chest that always drops common. Higher temperatures give less likely tokens a better chance, which the docs say can lead to more diverse or creative results. Two more settings, topK and topP, first cut the list down to the most probable candidates. That's your band trick with a filter in front of it.

Machine-learning models that weigh evidence also guard your inbox. In a [2019 post](#), Google said Gmail's machine-learning models block more than 99.9% of spam, phishing and malware. That year, a new model built with Google's TensorFlow toolkit helped block about 100 million extra spam messages a day. The models check thousands of signals in each email instead of following fixed rules. Google gave the reason: one person's spam can be another person's important message.

Your chest rolls one number against three fixed bands. Gemini makes the same kind of pick for every token it writes, across its whole vocabulary. Randomness has a cost, though. The same question can get a different answer each time, and sometimes a wrong one. And a filter that blocks 99.9% still lets a little spam through.

Boss Challenges

1. Delete `random.seed(7)` from `loot_roller.py`, run it five times, and record each roll and band.
2. Add an **epic** band for 91–95, so rare shrinks to 71–90 and work out every band's chance by hand before you change the code.
3. Swap the 100-sided dice for a normal 6-sided one (`random.randint(1, 6)`) and find out which of the 70%, 25% and 5% chances you can still hit exactly.
4. Find a seed whose first roll is legendary, using the seed box in the loot widget, and prove it in Python.

Teacher/Parent Toolkit

Learning goal: the student installs a working Python lab, computes band probabilities by hand with the +1 rule, and fixes four error types by reading the last line of the traceback first.

Common errors:

- `SyntaxError: invalid syntax` right after typing `python loot_roller.py`: the terminal shows `>>>`, so the student is inside Python itself, not the normal terminal. Type `exit()` and press Enter, then run the command again.
- `EOFError: EOF when reading a line`: the script ran in the **Output** panel (often from the Code Runner extension), which can't accept typing. Use the **Run Python File** button or type the command in the terminal.
- `ValueError: invalid literal for int() with base 10: 'five'`: the student typed a word at the gold question after the `int()` fix. `int()` only understands digits. Type 5.
- Output shows a different roll than 42: `random.seed(7)` is missing, has a different number, or sits below the `randint` line. The seed must run first.
- `can't open file ... [Errno 2] No such file or directory`: the terminal is in a different folder from the script. Use **File** → **Open Folder** on `loot-lab` and run from VSCode's own terminal.

Challenge hints:

1. Five unseeded runs rarely match the 70 / 25 / 5 split exactly. That's the point: five rolls are too few. The widget's **Roll 100** shows the split settling down.
2. Epic 91–95 is 5 numbers (5%), and rare 71–90 is 20 numbers (20%). Common stays 70%, legendary stays 5%, and the total is still 100. The new branch is `elif roll <= 95:` placed after `elif roll <= 90:`.

3. None of them. Each side of a 6-sided dice is $1 \div 6 \approx 16.7\%$, so 70% would need 4.2 sides. The closest split is 4 / 1 / 1 sides: about 66.7%, 16.7% and 16.7%.
4. Seed 23 rolls 100 and seed 26 rolls 96. Both are legendary on the first roll, in the widget and in Python.

Loot widget, 5 minutes: leave the seed on 7 and press **Roll 100**. Ask the class to predict the Actual column from the Expected column, then compare: 70, 27, 3 against 70, 25, 5. Then press **Stingy chest 90 / 9 / 1** without rolling again. The same 100 rolls now count 95, 3 and 2, because only the band edges moved.

Temperature demo, 5 minutes (adult account): in Google AI Studio, set the temperature to 0 and send "Finish this sentence in five words: The dragon..." three times. Then raise the temperature to its highest setting and send it three more times. The class compares the six answers and links the result to the band trick: low temperature keeps picking the widest band.

10-minute extension: write the numbers 1 to 20 on slips of paper and put them in a bag. Declare bands: 1–14 common, 15–19 rare, 20 legendary. Ask the class for each chance (70%, 25%, 5%), then have 20 students draw and replace one slip each. Tally the results on the board and compare with the expected 14, 5 and 1.

Deeper note for curious readers: the band edges 70, 95 and 100 are running totals of the chances: 70, then 70 + 25, then 95 + 5. Turning chances into running totals and rolling one number is how programs sample from any list of probabilities, including a chatbot's next-word list. When people change a chatbot's "temperature" setting, they reshape those chances before the roll: low temperature widens the most likely band, and high temperature evens the bands out.

Tool age rules: ChatGPT allows ages 13–17 with parental consent. The Gemini app allows 13+ in most countries but requires 18+ in the EEA, Switzerland and the UK. Students there should use ChatGPT for the Level Up section, or an adult can run the prompt.

Sources

- [Prompt design strategies](#), Google AI for Developers, Gemini API documentation (accessed September 2026)
- [Spam does not bring us joy—ridding Gmail of 100 million more spam messages with TensorFlow](#), Google Workspace Blog, 2019

Chapter 2: Mind-Reading Number Game — Binary Search

You will build a bot that guesses any number from 1 to 100 in 7 tries, then use its trick to bust a fake high score.

You need the lab from Chapter 1: Python 3, VSCode and a web browser. This chapter adds loops, lists and the `math` toolbox.

How the Bot Reads Your Mind: Halving

Think of a whole number from 1 to 100. Say 73. The bot guesses 50, the exact middle, and you answer "higher". That one word just deleted 1 to 50, so 50 numbers disappeared.

Now only 51 to 100 are left. The bot guesses the middle of those, 75, and you say "lower". Every answer throws away about half of what's left:

Guess	Numbers still possible	Bot guesses	You say	Left after
1	1 to 100 (100)	50	higher	51 to 100 (50)
2	51 to 100 (50)	75	lower	51 to 74 (24)
3	51 to 74 (24)	62	higher	63 to 74 (12)
4	63 to 74 (12)	68	higher	69 to 74 (6)
5	69 to 74 (6)	71	higher	72 to 74 (3)
6	72 to 74 (3)	73	yes	found

To find the middle, add the two ends and halve the total. For guess 3 that's $(51 + 74) \div 2 = 62.5$. The bot drops the .5 and guesses 62. It also never guesses 75 again. After "lower", the new top is $75 - 1 = 74$, the same +1 and -1 edge counting as the loot bands in Chapter 1.

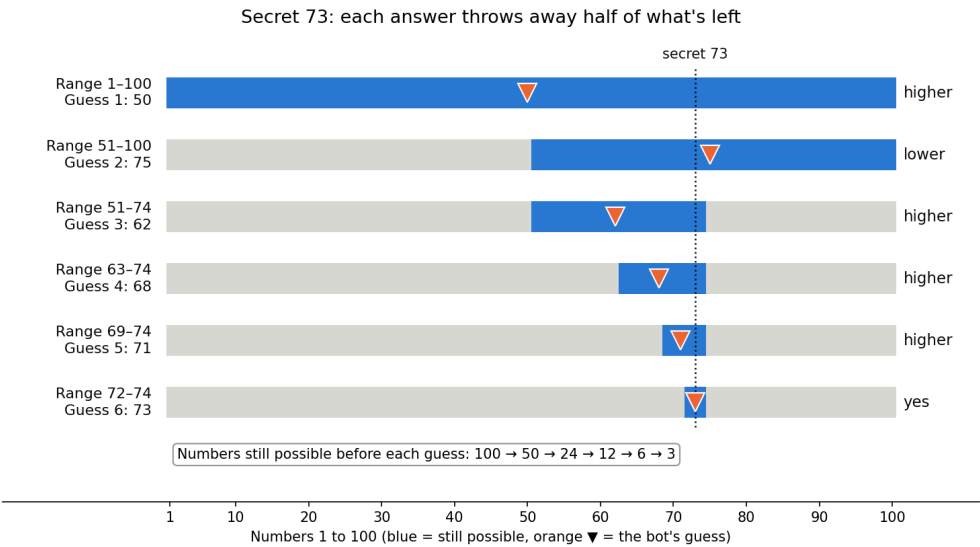


Figure 2.1: Each row is one guess. The blue bar is what's still possible, and it shrinks by half every row until it closes around 73.

In Figure 2.1, follow the orange markers down. Each one sits in the middle of its blue bar. The dotted line at 73 always stays inside the blue, because every answer you give keeps the secret in play. The box at the bottom lists the counts: 100, 50, 24, 12, 6, 3.

This strategy is **binary search**. "Binary" means two: every answer splits the numbers into two halves, and the bot keeps one. It is the fastest way to find something in a sorted list when all you get back is "higher" or "lower".

How Many Guesses Can It Ever Need?

Six guesses found 73. Some secrets take more, so how bad can it get? Build the answer up from tiny ranges.

- One guess can handle 1 number. The range 1 to 1 needs no thinking.
- Two guesses can handle 3 numbers. Guess the middle. If it's wrong, one number is left on each side, and guess 2 names it.
- Three guesses can handle 7 numbers: the middle, plus 3 on each side. Each side of 3 needs 2 more guesses.
- Each extra guess doubles the range and adds 1 for the middle. So 15, then 31, 63 and 127.

Seven guesses cover up to 127 numbers, and six only cover 63. The range 1 to 100 sits between them, so the bot needs 7 guesses at most. Figure 2.2 draws this staircase.

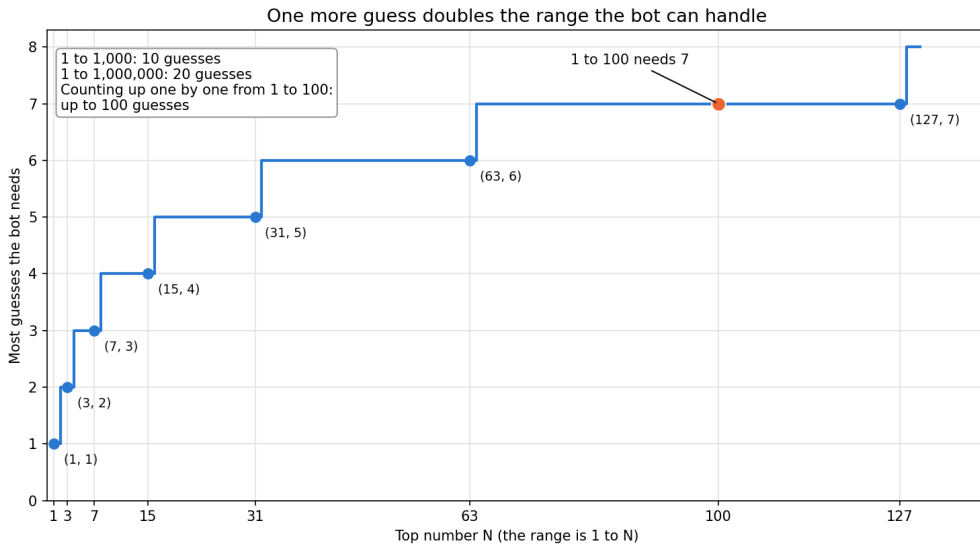


Figure 2.2: The staircase only climbs when the range passes 1, 3, 7, 15, 31, 63 or 127. Between those edges, extra numbers cost no extra guesses.

The orange dot at 100 sits on the step at height 7. The box compares this with counting up from 1, which could take all 100 guesses. At 1,000,000 numbers the gap turns silly: 20 guesses against a million.

The question "how many times do I double 1 before I reach N ?" has a name: the **logarithm base 2**, written $\log_2$. Here is the rule as a formula:

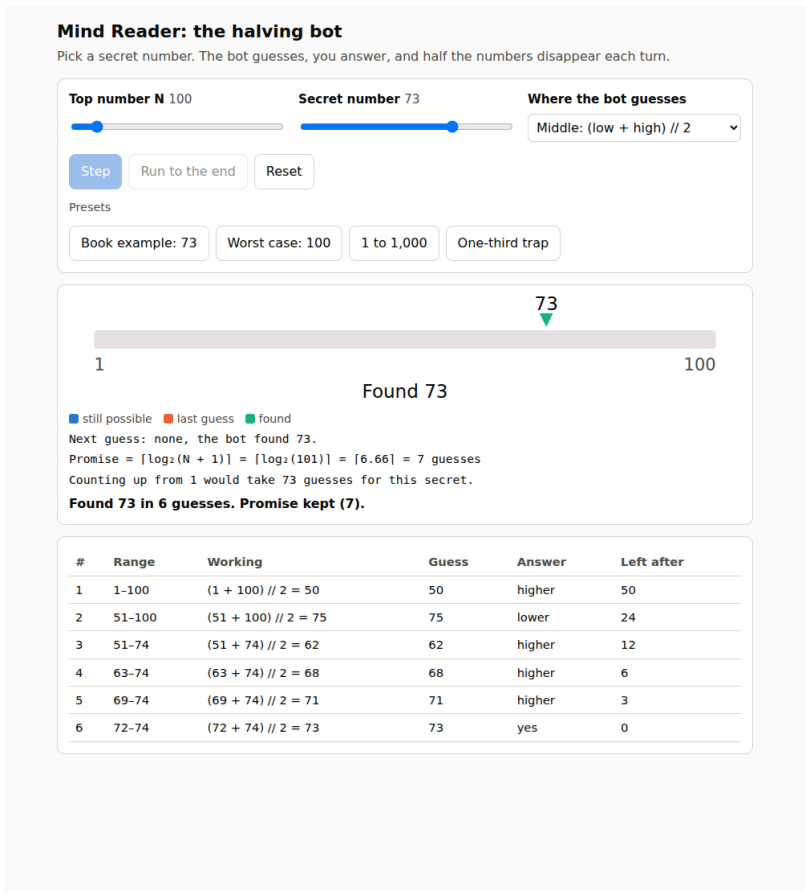
$$\text{most guesses} = \lceil \log_2(N + 1) \rceil$$

- N is the top number, so the range is 1 to N . For this game, 100.
- $\log_2(N + 1)$ asks: 2 to what power makes $N + 1$? $2^6 = 64$ and $2^7 = 128$, so $\log_2 101$ is about 6.66.
- The $+ 1$ matches the staircase edges: 7 guesses cover 127 numbers, which is $2^7 - 1$.
- The $\lceil \rceil$ brackets mean "round up". You can't take 6.66 guesses, so the bot needs 7.

Try one by hand. The range is 1 to 1,000. What's the most guesses the bot needs?

Answer: keep doubling: 7 guesses cover 127, 8 cover 255, 9 cover 511, 10 cover 1,023. Nine is too few, so the answer is 10. The formula agrees: $\log_2 1001 \approx 9.97$, rounded up to 10.

Try it live: open ch02-mind-reader.html and save the code on your machine. Double click the html file to open in your browser. Press **Step** three times. The history table shows 50, 75 and 62, each with its working, such as $(51 + 74) // 2 = 62$. Then press **1 to 1,000** and **Run to the end**. The bot finds 737 in exactly 10 guesses.



The widget after pressing Run to the end with its starting settings. Every line of working updates when you move a slider.**

Machine learning uses this idea in a model called a **decision tree**. A decision tree sorts data by asking yes/no questions, and each answer sends the data down one branch. It picks each question to shrink its uncertainty as fast as possible, the same goal as the bot's middle guess. One yes/no answer carries one **bit** of information, and seven yes/no answers can pick one item out of 128.

Build the Mind Reader

This script plays the bot's side. You think of a number, and it guesses until it's right.

1. Make a folder called `mind-reader` on your computer.

2. In VSCode, choose **File** → **Open Folder** and pick that folder.
3. Create a new file called `mind_reader.py` and paste this code:

```
low = 1
high = 100
guesses = 0
print(f"Think of a whole number from {low} to {high}. I will read
your mind.")
print("Answer h (higher), l (lower) or y (yes, you got it).")

while low <= high:
    guess = (low + high) // 2 # the middle of what's left
    guesses = guesses + 1
    answer = input(f"Guess {guesses}: is it {guess}?
").strip().lower()
    if answer == "y":
        print(f"🎉 Got it in {guesses} guesses.")
        break
    elif answer == "h":
        low = guess + 1 # guess was too low, so throw away guess and
        below
    elif answer == "l":
        high = guess - 1 # guess was too high, so throw away guess
        and above
    else:
        print("Type h, l or y.")
        guesses = guesses - 1
    print(f"    Still possible: {low} to {high} ({high - low + 1} of
    them)")
```

4. Open the terminal in VSCode (**Terminal** → **New Terminal**) and run it:
`python3 mind_reader.py`

On Windows, type `python mind_reader.py` instead. Think of 73 and answer honestly. You should see this, with your own answers after each question mark:

```
Think of a whole number from 1 to 100. I will read your mind.
Answer h (higher), l (lower) or y (yes, you got it).
Guess 1: is it 50? h
    Still possible: 51 to 100 (50 of them)
Guess 2: is it 75? l
    Still possible: 51 to 74 (24 of them)
Guess 3: is it 62? h
    Still possible: 63 to 74 (12 of them)
Guess 4: is it 68? h
    Still possible: 69 to 74 (6 of them)
Guess 5: is it 71? h
    Still possible: 72 to 74 (3 of them)
Guess 6: is it 73? y
🎉 Got it in 6 guesses.
```

The numbers match the table and Figure 2.1 row for row. This game needs no `random.seed`, because nothing in it is random. The same answers always produce the same guesses.

TEACHER/PARENT TOOLKIT

Likely roadblock: `EOFError: EOF when reading a line` as soon as the first guess appears.

Fix: The script ran in VSCode's **Output** panel (often through the Code Runner extension), which can't take typing. Click **Run Python File** or type `python3 mind_reader.py` in the terminal.

Four parts of this code carry the idea:

- `while low <= high:` is a **loop**. Python runs the indented lines, then jumps back to the `while` line and checks again. It repeats while numbers are still possible.
- `(low + high) // 2` finds the middle. `//` divides and throws away the leftover, so `125 // 2` gives 62, not 62.5. Plain `/` would give 62.5.
- `low = guess + 1` and `high = guess - 1` throw away half the range. The `+1` and `-1` also throw away the guess itself, because you just said it's wrong.
- `break` jumps straight out of the loop. Without it, the bot would keep asking after it already won.

`.strip().lower()` tidies your answer. `strip()` removes stray spaces, and `lower()` turns H into h, so both work.

The ML idea: ask the question that splits the most. The line that matters is `guess = (low + high) // 2`. It picks the one question whose answer, either way, throws away half of what's left. That's why 100 numbers never need more than 7 answers: $\lceil \log_2 101 \rceil = 7$, the staircase you drew by hand.

A **decision tree** runs on the same idea. It is a model that predicts by asking yes/no questions about the data, such as "Is it raining?" or "Is it rush hour?". Each answer sends the data down one branch, and the last branch holds the prediction. When the tree learns, it tests many possible questions on past examples and keeps the one that splits them most cleanly. Trees like this are a common choice for predicting prices, delivery times and whether a payment looks like fraud.

Your bot has it easy. Its numbers are sorted, and the middle guess always splits them exactly in half. Real data is messy, so the best question rarely splits it evenly, and one tree often guesses badly. Real systems usually grow hundreds of small trees and add their answers together.

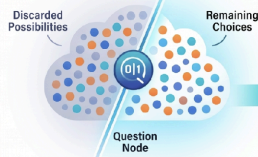
Now try to cheat. Think of 100, but answer h even when the bot guesses 100. The last line says `Still possible: 101 to 100 (0 of them)`, and the game stops without a word. Low has passed high, so the `while` check fails. Your answers contradicted each other, and the bot doesn't know how to say so yet.

Machine Learning Strategy: Ask the Question That Splits the Most

The Core Math: Splitting & Information Gain

Ideal Questions Discard 50% of Possibilities

Choosing the exact middle midpoint guarantees throwing away half of the remaining items.



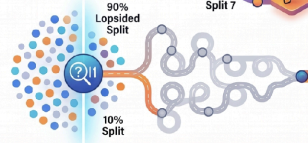
1 Yes/No Answer Delivers 1 Bit of Information

Single yes/no delivers a perfect split of information.



Lopsided Splits Waste Question Efficiency

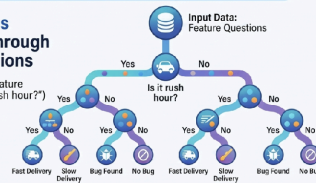
Questions that fail to split data evenly require more total steps to find answers.



How Machine Learning Applies the Splitting Rule

Decision Trees Route Data Through Yes/No Questions

Models ask targeted feature questions (e.g., 'Is it rush hour?') to reach predictions.



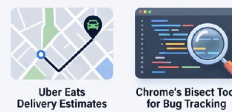
Messy Real Data Requires Ensembles of Small Trees

Real-world data rarely splits evenly, so hundreds of small trees combine their outputs.



Powers Real-World Logistics & Software Debugging

Used in Uber Eats delivery estimates and Chrome's bisect tool for bug tracking.



Test the Bot on Every Number

You tried one secret. A real engineer tests all 100. This script plays every secret from 1 to 100 and answers for you, so it finishes in a blink. Save it as `bot_tester.py`:

```
import math

top = 100
worst = 0
total = 0

for secret in range(1, top + 1):
    low, high = 1, top
    guesses = 0
    while low <= high:
        guess = (low + high) // 2
        guesses = guesses + 1
        if guess == secret:
            break
        elif guess < secret:
            low = guess + 1
        else:
            high = guess - 1
    total = total + guesses
    if guesses > worst:
        worst = guesses

print(f"Played all {top} secret numbers.")
print(f"Worst game: {worst} guesses. Average: {total / top:.2f} guesses.")
promise = math.ceil(math.log2(top + 1))
```

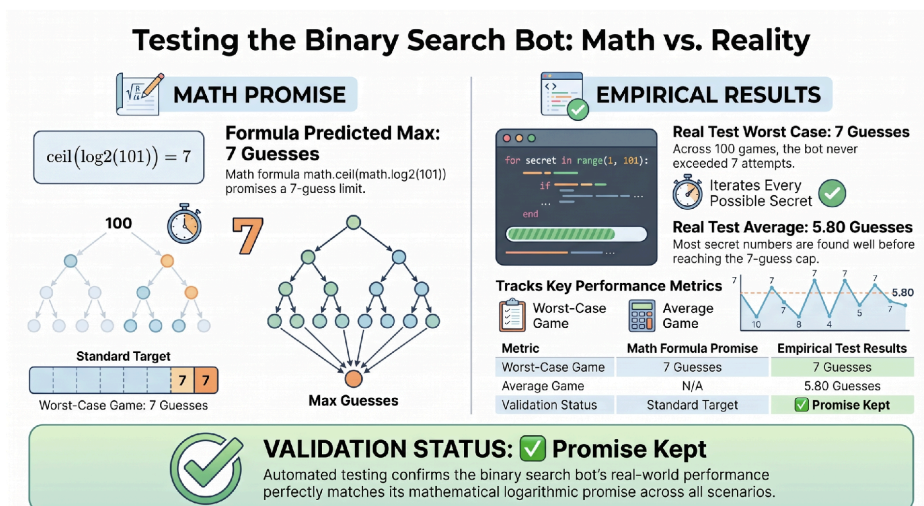
```
print(f"The formula promised at most {promise}.")
if worst <= promise:
    print("✅ The bot keeps its promise.")
else:
    print("❌ The bot broke the promise. Check how it picks its guess.")
```

Run it with `python3 bot_tester.py`. Expected output:

```
Played all 100 secret numbers.
Worst game: 7 guesses. Average: 5.80 guesses.
The formula promised at most 7.
✅ The bot keeps its promise.
```

- `for secret in range(1, top + 1):` is a second kind of loop. It runs once for each number from 1 to 100. `range` stops just before its second number, so you write `top + 1` to include 100.
- The `while` loop inside is the same bot, with `guess == secret` in place of your typed answer.
- `math.log2` and `math.ceil` are the formula, typed out. `import math` loads the toolbox that holds them.

The last lines compare the code with the math. The worst of 100 real games equals the 7 the formula predicted. Two different methods agree, so you can trust both. The average of 5.80 shows that most secrets take fewer than 7 guesses.



Level Up: Add a Lie Detector with ChatGPT or Gemini

Your bot goes silent when someone cheats. You'll ask an AI partner to make it catch the lie and offer another round. Open ChatGPT (chatgpt.com) or Gemini (gemini.google.com) and paste all of `mind_reader.py`. Under it, add this prompt and send it.

```
I have a Python number-guessing bot (code above). It finds my
number from
1 to 100 by always guessing the middle of the numbers still
possible.
Change it so that:
1. Before each guess, it checks whether low has passed high. If
   so, it prints
      that I lied, because no number fits all my answers.
2. It works out the most guesses it could need with math.log2 and
   math.ceil,
   and tells me that promise at the start of each game.
3. After each game, it asks "Play again? (y/n)".
4. Answers other than h, l or y don't count as a guess.
Use only the math library. Keep it at beginner level: no
   functions, no classes,
no list comprehensions. Explain each new part in one sentence.
```

This prompt works because it names what exists, what to add and the limits. It also explains the rule behind the lie detector. "Add cheat detection" alone could come back as anything, such as a bot that counts guesses and accuses you after 8.

Your AI's answer will look different from anyone else's. Here is one version that passes every check below. Save it as `mind_reader_plus.py`:

```
import math

top = 100
promise = math.ceil(math.log2(top + 1)) # most guesses the bot
    should ever need
play_again = "y"

while play_again == "y":
    low, high = 1, top
    guesses = 0
    print(f"\nThink of a number from 1 to {top}. I promise to
    need {promise} guesses or fewer.")
    while True:
        if low > high:
            # No number fits every answer you gave, so one answer
            was wrong
            print("🤖 Lie detected! No number fits all your
            answers.")
            break
        guess = (low + high) // 2
        answer = input(f"Guess {guesses + 1}: is it {guess}?
        (h/l/y) ").strip().lower()
        if answer not in ("h", "l", "y"):
```

```

        print("Please type h, l or y.")
        continue
    guesses = guesses + 1
    if answer == "y":
        print(f"🎉 Got it in {guesses} guesses. Promise was {promise}.")
        break
    elif answer == "h":
        low = guess + 1
    else:
        high = guess - 1
    play_again = input("Play again? (y/n) ").strip().lower()

print("Thanks for playing.")

```

Play one honest game with 73, then one where you answer h every time:

```

Think of a number from 1 to 100. I promise to need 7 guesses or fewer.
Guess 1: is it 50? (h/l/y) h
Guess 2: is it 75? (h/l/y) h
Guess 3: is it 88? (h/l/y) h
Guess 4: is it 94? (h/l/y) h
Guess 5: is it 97? (h/l/y) h
Guess 6: is it 99? (h/l/y) h
Guess 7: is it 100? (h/l/y) h
🙊 Lie detected! No number fits all your answers.
Play again? (y/n) n
Thanks for playing.

```

Three new pieces appear here. `while True:` loops forever, so only a `break` ends it. `continue` skips the rest of the loop and jumps back to the top, which is how a typo doesn't count as a guess. `answer not in ("h", "l", "y")` is true when the answer is none of those three.

Before you trust any AI code, run these four checks:

1. **Does it run?** Play one full game without an error.
2. **Does it use only math?** Read the `import` lines at the top.
3. **Does it keep the promise?** Answer honestly for 1, 100 and 73. No game should pass 7 guesses.
4. **Does the lie detector fire?** Answer h every time. It should accuse you after guess 7.

When the AI Gets It Wrong

AI partners make confident mistakes. Here is a realistic one: a version that "simplified" the two update lines. Think of 100 and answer honestly with h:

```
Guess 6: is it 98? (h/l/y) h
Guess 7: is it 99? (h/l/y) h
Guess 8: is it 99? (h/l/y) h
Guess 9: is it 99? (h/l/y) h
Guess 10: is it 99? (h/l/y) h
```

The bot guesses 99 forever and never reaches 100. Check 3 caught it at guess 8, one past the promise. Press Ctrl + C in the terminal to stop the program. Python prints `KeyboardInterrupt`, which just means you stopped it on purpose. Now find the lines that move the edges:

```
elif answer == "h":
    low = guess
else:
    high = guess
```

The AI dropped the `+ 1` and `- 1`. Walk through it by hand. With low 99 and high 100, the middle is $(99 + 100) // 2 = 99$. You say "higher", so low becomes 99 again, and the range never shrinks. The guess you just ruled out stays in play. Change the lines back to `low = guess + 1` and `high = guess - 1`, and the bot finds 100 on guess 7.

You caught this bug because you know two numbers the AI didn't check: the promise of 7, and the edge rule from Chapter 1. The AI wrote the loop, and you stayed the engineer.

Use the Same Trick to Bust a Fake High Score

Your friend claims they scored 6,120 in the arcade game and made the top 20. The leaderboard runs from best to worst, just like the numbers 1 to 100 are in order. So the bot's trick works on it: look at the middle score and throw away the half that can't hold 6,120.

Save this as `brag_checker.py`:

```
# Arcade top 20, best score first
names = ["Ava", "Raj", "Mia", "Leo", "Zoe", "Sam", "Ivy", "Max",
        "Kai", "Nia",
        "Eli", "Uma", "Ben", "Lua", "Tom", "Ada", "Jin", "Oli",
        "Pia", "Rex"]
scores = [9870, 9415, 9030, 8760, 8310, 7995, 7650, 7340, 7120, 6890,
        6505, 6120, 5875, 5430, 5010, 4780, 4325, 3990, 3605, 3150]

claim = 6120 # your friend says they scored this and made the top 20
```

```

low = 0
high = len(scores) - 1 # the list's last position is 19
looks = 0
found = False

while low <= high:
    middle = (low + high) // 2
    looks = looks + 1
    print(f"Look {looks}: position {middle} holds {scores[middle]}")
    if scores[middle] == claim:
        found = True
        break
    elif scores[middle] > claim:
        low = middle + 1 # scores go down the list, so a smaller
        claim sits further down
    else:
        high = middle - 1

if found:
    print(f"✅ True. {names[middle]} scored {claim} and ranks
    #{middle + 1}.")
else:
    print(f"❌ Nobody in the top 20 scored {claim}. Brag busted after
    {looks} looks.")

```

Expected output:

```

Look 1: position 9 holds 6890
Look 2: position 14 holds 5010
Look 3: position 11 holds 6120
✅ True. Uma scored 6120 and ranks #12.

```

`scores` is a **list**: a row of boxes with numbered positions. Python counts positions from 0, so `scores[0]` is 9870 and the last box, `scores[19]`, is 3150. `len(scores)` counts the boxes: 20. That's why `high` starts at 19 and the rank is `middle + 1`.

One line flips compared with the mind reader. This list runs from big to small, so a claim smaller than the middle score sits further down. That's why `scores[middle] > claim` moves `low` up. Change `claim` to 6200 and run again. The checker needs 4 looks to prove nobody scored it. By the formula, 20 scores never need more than $\lceil \log_2 21 \rceil = 5$ looks.

Binary search has one hard rule: the data must be sorted. On a shuffled list, throwing away half can throw away the answer. Boss Challenge 4 makes you prove it.

Put It in a Google Sheet

Spreadsheets can run binary search for you. Google Sheets' `XLOOKUP` and `XMATCH` functions have a search mode that uses it, as long as you sort the column first.

1. Open a new Google Sheet (sheets.new).

2. Type **Name** in A1 and **Score** in B1.
3. Click A2 and paste this block. Each row lands in column A as one line of text.

Ava,9870
Raj,9415
Mia,9030
Leo,8760
Zoe,8310
Sam,7995
Ivy,7650
Max,7340
Kai,7120
Nia,6890
Eli,6505
Uma,6120
Ben,5875
Lua,5430
Tom,5010
Ada,4780
Jin,4325
Oli,3990
Pia,3605
Rex,3150

With A2 to A21 still selected, choose **Data** → **Split text to columns**.
The scores move into column B.

4. Type **Claim** in D1 and **6120** in E1.
5. Type **Who?** in D2, then this formula in E2:
`=XLOOKUP(E1, B2:B21, A2:A21, "Not on the board", 0, -2)`
6. Type **Rank** in D3, then this formula in E3:
`=IFERROR(XMATCH(E1, B2:B21, 0, -2), "-")`

E2 should say **Uma** and E3 should say **12**, the same as your script. Change E1 to **6200**, and E2 switches to **Not on the board**.

The last two settings in each formula matter. The **0** asks for an exact match. The **-2** asks for binary search on a column sorted from biggest to smallest. Google's help page for **XLOOKUP** says you must sort the range first for that mode. The Sheet trusts you on that, so a wrong order gives wrong answers without any warning.

The sheet and your Python script agree on the same data. In Chapter 3 you'll meet another piece of AI math, distance, and check it the same way.

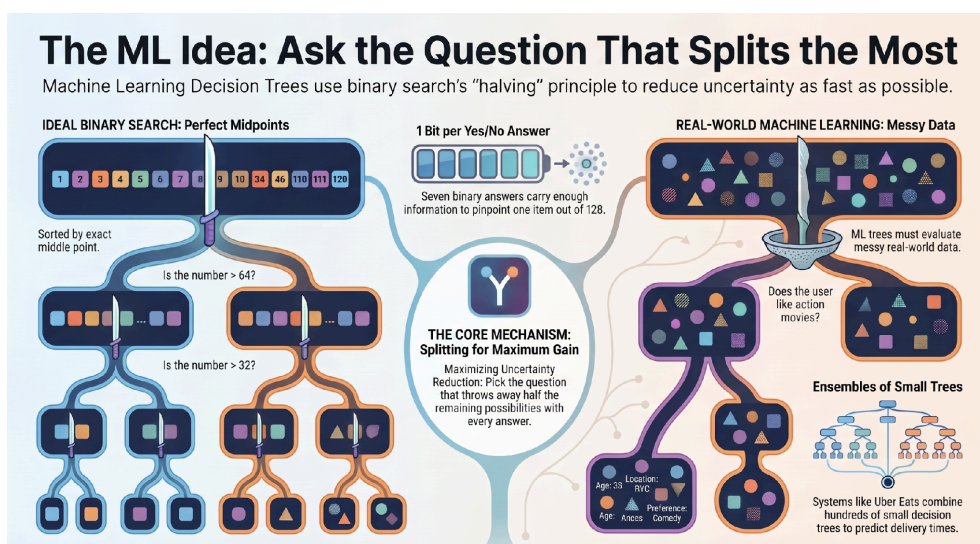
Where Chrome and Uber Eats Use This Idea

Sometimes a new build of Google's Chrome browser breaks something, and nobody knows which code change caused it. The Chromium project, which builds Chrome's code, has a tool for that called [bisection-builds.py](#). It downloads

a build from the suspect range and runs it. You check whether the bug shows up, then answer the script. It uses your answer to drive a binary search, exactly like your bot's "higher" or "lower". It ends with a narrow window of changes, which usually makes the faulty one easy to spot.

Decision trees show up in apps you might order from. In a [2017 blog post](#), Uber's engineers explained how Uber Eats predicted delivery times. The team used gradient boosted decision tree models to estimate the time from order to doorstep. Each tree asks yes/no questions about the data and ends in a number. "Gradient boosted" means many small trees add their answers together. The trees learn their questions from past deliveries instead of having them typed in by hand.

Your bot needs at most 7 questions for 100 numbers. Chrome's tool halves a list of code changes the same way, and Uber's trees split past deliveries into smaller groups. The catch is that both depend on their input. Binary search fails on unsorted data. A tree only knows the patterns in its past examples, so an unusual night can throw its estimate off.



Boss Challenges

1. Change `top` to 1,000 in `bot_tester.py`, predict the worst game with the doubling rule first, then run it.
2. Make the bot guess one third of the way up the range ($\text{low} + (\text{high} - \text{low}) // 3$) in `bot_tester.py`, and compare its worst game with the middle guess.
3. Flip the roles: the computer picks a secret with `random.seed(7)` and `random.randint(1, 100)`, you guess, and it tells you whether you beat the promise of 7.

4. Swap the scores 6890 and 6505 in `brag_checker.py`, then find a true claim that the checker now busts by mistake.

Teacher/Parent Toolkit

Learning goal: the student finds the middle guess by hand, predicts the worst case with $\lceil \log_2(N + 1) \rceil$, codes the search with a `while` loop, and tests it on every possible secret.

Common errors:

- The bot repeats the same guess forever (often 99): `low = guess` or `high = guess` is missing its `+` 1 or `-` 1. Press `Ctrl + C` to stop it, then fix the line.
- `TypeError: list indices must be integers or slices, not float` in `brag_checker.py`: the middle line uses `/` instead of `//`. List positions must be whole numbers.
- `IndexError: list index out of range`: `high` starts at `len(scores)` instead of `len(scores) - 1`. There is no position 20 in a 20-item list.
- `EOFError: EOF when reading a line`: the script ran in the Output panel. Run it from the terminal.
- The mind reader ends with `Still possible: 101 to 100 (0 of them)`: the student gave contradicting answers, often by mixing up `h` and `l`. The Level Up section's lie detector turns this silence into a message.

Challenge hints:

1. Ten guesses cover 1,023 numbers and nine cover 511, so the prediction is 10. The tester prints `Worst game: 10 guesses. Average: 8.99 guesses.`
2. Replace the `guess =` line inside `bot_tester.py`. The worst game jumps to 11 and the average to 6.28, so the  turns into . Ask why: after a "higher", the bot keeps two thirds of the range, not half. The widget's **One-third trap** preset replays the 11-guess game for secret 100.
3. Put `import random` and `random.seed(7)` at the top, then `secret = random.randint(1, 100)`. Seed 7 gives 42, the same roll as the loot chest in Chapter 1. Loop with `int(input(...))` and print "higher" or "lower" until the guess matches.
4. After the swap, a claim of 6890 gets busted in 5 looks. Look 1 lands on 6505, which is smaller than 6890, so the checker throws away the bottom half, where 6890 now sits.

Widget, 5 minutes: project it with the starting settings. Before each press of **Step**, ask the class to call out the next guess from the Range column, then check it against the Working column. Then press **Worst case: 100** and **Run to the end**, and ask why the table stops at exactly 7.

Big-tech discussion, 5 minutes: play a round of "bisect" out loud. Number 16 cards as "versions" and secretly mark the first "broken" one. Students may only ask, "Is version N broken?" and must find the first broken card in 4 questions, the way Chrome's tool narrows its window.

10-minute extension: hand a group a printed dictionary or a thick book and ask them to find a given page number by halving. They open the middle, keep the half that holds the page, and count every opening. A 1,000-page book needs at most 10 openings. Compare with the class's first instinct, which is usually flicking from the front.

Deeper note for curious readers: the formula measures information. Each fair yes/no answer is one bit, and k bits can tell apart 2^k things.

Twenty questions can pick one item from $2^{20} = 1,048,576$, which is why the party game "20 Questions" works so well. A decision tree scores every possible question by how much its answer would tell it, a measure called information gain, and asks the best one first. A question whose answer is nearly always the same tells you little. That's the one-third bot's problem in Challenge 2: its lopsided guesses waste part of each answer.

Tool age rules: ChatGPT allows ages 13–17 with parental consent. The Gemini app allows 13+ in most countries but requires 18+ in the EEA, Switzerland and the UK. Students there should use ChatGPT for the Level Up section, or an adult can run the prompt.

Sources

- [bisect-builds.py](#), The Chromium Projects (accessed September 2026)
- [Meet Michelangelo: Uber's Machine Learning Platform](#), Uber Engineering Blog, 2017

Glossary

Every term below is explained in full in the chapter shown in brackets.

Accuracy (Ch 4) The share of test answers a model gets right: correct answers $\div$ total answers.

Binary search (Ch 2) Finding a target by guessing the middle and throwing away the half it can't be in, again and again. It finds any number from 1 to 100 in at most 7 guesses.

Bit (Ch 2) The information in one yes/no answer. Seven bits are enough to pick one item out of 128.

Classification (Ch 4) Giving a new item a label, such as dragon or slime, based on labelled examples.

Conditional probability (Ch 5) A chance that depends on what just happened, written $P(\text{next} \mid \text{last})$ and read “the chance of next, given last”.

Confidence (Ch 4) How sure a vote is: the winning votes divided by the number of voters.

Cosine similarity (Ch 3) A way to compare two vectors by direction only. It asks “do these point the same way?” rather than “how far apart are they?”.

Decision boundary (Ch 4) The line where a sorter switches from one label to another.

Dictionary (Ch 3) A Python container that pairs each key with a value, such as each friend's name with their ratings.

Estimated probability (Ch 5) A chance measured from data: the times something happened divided by the total tries.

Euclidean distance (Ch 3) The straight-line distance between two points, worked out with the Pythagorean theorem. AI uses it to measure how similar two pieces of data are.

f-string (Ch 1) Text with an f in front of the quote mark, which lets you drop variables into curly brackets.

Gradient (Ch 6) The slope of the loss for every setting at once. It points the way the loss rises fastest.

Gradient descent (Ch 6) How models learn: measure the slope of the loss, then take a small step downhill, again and again.

Greedy picking (Ch 8) Always choosing the most likely option, with no dice roll at all.

JSON (Ch 4) A plain-text format for storing data in a file. Python reads it with the json toolbox.

k-nearest neighbors (kNN) (Ch 4) Labelling a new item by a majority vote of the k known items closest to it.

Learning rate (Ch 6) The number that sets how far each step of gradient descent goes.

Leave-one-out (Ch 4) A test that hides one example's label, asks the model to predict it from the rest, and repeats for every example.

List (Ch 2) A Python container that holds items in order, each at a numbered position.

Logarithm base 2 (log₂) (Ch 2) How many times you double 1 to reach a number. It tells you the most guesses binary search can need.

Loop (Ch 2) Code that repeats until a job is done.

Loss (Ch 6) A measure of how wrong a model is right now. Learning means making it smaller.

Manhattan distance (Ch 3) Distance measured along the grid lines, across plus up, like a taxi driving city blocks.

Markov chain (Ch 8) A system that picks its next step using only where it is right now, such as the next word from the current word.

Naive Bayes (Ch 7) A classifier that treats every word as a separate clue and multiplies their chances. "Naive" because real words depend on each other.

Probability (Ch 1) A number that says how likely something is: the matching outcomes divided by all possible outcomes.

Pythagorean theorem (Ch 3) The triangle rule $a^2 + b^2 = c^2$, which turns an across leg and an up leg into a straight-line distance.

Rescaling (Ch 4) Putting every measurement on the same range, often 0 to 1, so no single measurement swamps the others.

Sampling (Ch 1) Picking one outcome so that, over many picks, each outcome turns up about as often as its probability says.

Seed (Ch 1) A starting number for Python's dice. The same seed gives the same "random" rolls every run.

Slope (Ch 6) How fast something changes as you step: the change in the loss divided by the size of the step.

Smoothing (Ch 5) Adding a small number, often 1, to every count so that no chance is ever exactly 0% or 100%.

String (Ch 1) Text in Python, written inside quote marks.

Temperature (Ch 8) A setting that controls how random a chatbot's next-word pick is. Low is safe and repetitive; high is varied and surprising.

Token (Ch 1) A piece of text, such as a word or part of a word, that a language model predicts one at a time.

Training data (Ch 4) The labelled examples a model learns from.

Variable (Ch 1) A labelled box that stores a value. In `coins = 5`, the `=` sign means "store this".

Vector (Ch 3) A list of numbers that points somewhere, with a direction and a length. AI turns songs, photos and words into vectors.

Enjoyed This Book?

If Think Like an AI helped you, or a young coder you know, please consider leaving a short review where you bought it. Even one sentence helps other parents, teachers and curious beginners decide whether this book is right for them.

Spotted a mistake, or a step that no longer works? Tools change fast, and reader reports keep this book accurate.

Send a message to Kapila on [LinkedIn](#) or email @kapila.sood@gmail.com.

Coming Next: Parts 2 and 3

You've now built eight games and learned the core moves behind modern AI: turn chances into choices, search by halving, measure distance, vote with neighbors, count habits, step downhill, and predict the next word.

Parts 2 and 3 of *Think Like an AI* take those moves further. You'll install your first libraries, give your games real interfaces, and learn to judge AI the way engineers do.

Part 2: Level Up — Libraries and Interfaces

- **Chapter 9: Unbeatable Tic-Tac-Toe.** Build a pygame board that never loses, and discover how game AIs search ahead and plan their moves with minimax.
- **Chapter 10: Make It a Web App.** Turn your Creature Sorter into a Streamlit app, or an HTML/CSS page drafted with ChatGPT or Gemini, then review and debug the code the AI wrote.
- **Chapter 11: Ask Your Data.** Explore a dataset in NotebookLM by asking plain-English questions, and check every AI answer against your own code.

Part 3: Think Like an AI Engineer

- **Chapter 12: When AI Gets It Wrong.** Break the Creature Sorter with biased data, then learn to spot bias and hallucinations and to fact-check AI.
- **Chapter 13: Capstone — Your AI Arcade.** Build a menu that launches every game you've made, then publish and share your arcade.
- **Co-Pilot appendix (adult-run).** Connect the Word Wizard to a real large language model with a free Gemini API key, using only Python's built-in urllib.

About the Author

Kapila Sood is an AI product leader with more than 15 years of experience building data platforms, machine-learning models and AI applications. Kapila has led teams of data scientists and engineers who turn generative AI, conversational intelligence and forecasting models into products used across enterprises.

With deep telecom and healthcare domain expertise, Kapila has built the data systems behind patient care and population-health programs, and has integrated AI into consumer experience platforms. Along the way, Kapila has trained product managers, data scientists and engineers to use generative AI.

This book grew from a simple belief: AI isn't magic. It's counting, measuring and stepping downhill, and anyone who can add and multiply can build it. Kapila graduated from Delhi University as a gold medalist in Philosophy (B.A. Honours), and still thinks the best question in AI is the one in this book's dedication: "But how does it know?"

Connect with Kapila on [LinkedIn](#).