

THE BREEDER'S CODE

From Dogs to DNA to Algorithms

By

MARTIN CARR

INDEPENDENT PUBLISHER

Copyright © 2026 by Martin Carr.

All rights reserved.

No part of this book may be reproduced, digitalized, or transmitted in any form without written permission from the publisher, except in the case of brief quotations embodied in critical reviews.

Publisher: Independent Publisher

ISBN Placeholder

TABLE OF CONTENTS

Who this book is for	4
What this book is about	5
How to read it	7
Part One — Dogs	8
The Oldest Algorithm	9
Breeding Dogs in Code	14
Part Two — Plants	26
The Monk in the Garden	27
Peas in Code	35
Part Three — DNA	47
The Book of Four Letters	48
DNA in Code	55
Part Four — Algorithms	69
Breeding an Answer Nobody Can Design	70
The Salesman in Code	78
Breeding the Formula Itself	91

CHAPTER 1

WHO THIS BOOK IS FOR

You can already write a little bit of code. Maybe a lot. You know what a variable is, you've written a for loop, and you're comfortable running a Python file and seeing what it prints. You do not need to know any biology, any statistics, or any maths beyond adding numbers up. If a chapter ever leans on something more, it teaches it first.

What you need to bring is curiosity, and a machine that can run Python 3. Every program in this book is pure, plain Python — no libraries to install, nothing to download, nothing that will stop working in two years. You can type them in, run them, break them, and fix them. That is the whole point.

CHAPTER 2

WHAT THIS BOOK IS ABOUT

There is one idea in this book. It is very old, and it is very simple, and once you can see it you will start seeing it everywhere.

The idea is this: if you have a population of things that vary, and some of them are better at something than others, and the better ones get to make copies of themselves that resemble but don't perfectly match their parents — then, given time, that population will get better at that something, all on its own, with nobody in charge.

Nobody designed the greyhound to be fast. Nobody sat down and engineered the pea plant that feeds half the world. Nobody wrote your DNA. And — this is where we're going — nobody needs to hand-design the solution to a hard problem in code, either. You can breed it.

We're going to arrive at that last claim slowly and honestly, by watching the same idea do its work in four different places:

1. Dogs. Where selection is easy to see, because humans have been doing it on purpose for ten thousand years.
2. Plants. Where a monk with a garden full of peas worked out the hidden rules — the first real look under the hood at how traits are actually passed on.
3. DNA. Where we finally meet the code itself: four letters, in a long string, that every living thing is written in — and how a spit-in-a-tube test can find a cousin you've never met.
4. Algorithms. Where we take everything the first three parts taught us and write programs that evolve their own answers to problems we don't know how to solve by hand.

By the last chapter you will have built a working evolutionary algorithm from scratch and understood every line of it — because you'll have already met every idea inside it, wearing fur, then leaves, then a double helix.

HOW TO READ IT

Read it at the keyboard. When you hit a program, type it in and run it before you read my explanation of what it does. Guess what it'll print. You'll be wrong sometimes; that's the fun part. Then change a number and run it again. The reader who fiddles learns twice as fast as the reader who nods along.

Each part ends with a short "Breed your own" challenge that stretches the code you just wrote. None of them are homework. All of them are more interesting than they look.

Let's go and meet the oldest algorithm in the world. It has four legs.

PART ONE — DOGS

• • •

CHAPTER 3

THE OLDEST ALGORITHM

Somewhere between fifteen and forty thousand years ago, long before farming, before writing, before metal, a wolf came a little closer to a human fire than the other wolves dared to. Maybe it was after scraps. Maybe it was just less frightened than its brothers and sisters. Nobody planned what happened next. But that one small difference — this wolf is a bit less afraid of people than that wolf — is the seed of every dog that has ever lived.

The bold wolves ate better near the humans. Wolves that eat better raise more pups. Those pups, on average, inherited a little of their parents' boldness. The next generation crept a little closer to the fire. And the one after that. Nobody was breeding them. Nobody had a plan. But a population that varied was being filtered by an advantage, and the advantage was heritable, and there was time — thousands of years of it. That's all it takes.

By the time humans started doing it on purpose, the raw machinery had already been running for millennia. And when we did start doing it on purpose, we got fast. Look at what we made:

- A greyhound, built for one thing — a sprint — that can hit 45 miles an hour.
- A border collie so tuned to movement and instruction that it can move a flock of sheep across a hillside on whistles alone.
- A St. Bernard the size of a person, and a chihuahua you can hold in one hand — both, underneath, the same animal, the same species, descended from the same wolf.

No one engineered a greyhound the way you'd engineer a car. Nobody drew a blueprint. What people did was much lazier and much more powerful: for hundreds of generations, they let the fastest dogs have the puppies. That's it. That is the entire technique. And out the other end came an animal so specialised it looks designed.

THE FOUR INGREDIENTS

Slow down and name what actually happened, because we are going to need these four words for the rest of the book — and by the end you'll be writing each of them as a few lines of Python.

1. A population. Not one dog — many. Evolution has nothing to work with if there's only one of a thing. It needs a crowd to choose

from.

2. Variation. The dogs in that crowd are not identical. Some are faster, some slower; some bold, some shy. This variation is the raw material. No differences, nothing to select. A population of perfect clones goes nowhere, forever.

3. Selection. Something decides who gets to reproduce. With wolves at the fire, it was survival — the well-fed raise more young. With racing dogs, it was a human choosing: you two, the fast ones, you'll be the parents. The thing doing the choosing is the only part that changes from story to story. Keep your eye on it; it's the most important character in the book.

4. Heredity with change. The puppies resemble their parents — fast parents tend to make fast puppies — but they are not exact copies. They're a shuffle of both parents, plus the odd small random difference all their own. Resemblance carries the advantage forward. The small differences are new variation, fresh raw material for the next round of selection.

Population, variation, selection, heredity-with-change. Run that loop over and over and the population drifts, generation by generation, toward whatever selection favours. Toward speed. Toward tameness. Toward herding sheep.

Here's the sentence to tattoo on the inside of your eyelids:

Selection + variation + heredity + time = a population that improves itself, with nobody in charge.

NOBODY IN CHARGE

That last part — nobody in charge — is the part that trips people up, and it's the part that makes this so useful to a programmer. There is no designer inside the process. The breeder chooses which dogs mate, yes, but the breeder does not design the puppy. Nobody decides which genes recombine, or that this pup will have slightly longer leg bones. The improvement emerges from the loop. It is a bottom-up result, not a top-down plan.

Hold onto that, because it's the exact trick we're going to steal. In Part Four we'll face problems where nobody knows how to design the answer — where the "blueprint" approach fails because the thing is too complicated to design by hand. And instead of designing it, we'll breed it. We'll make a population of candidate answers, let them vary, select the ones that work best, breed those, and repeat. We will play the role of the fire, or the racing breeder: we choose what counts as good, and then we get out of the way and let the loop do the building.

But that's getting ahead of ourselves. Right now there's a much more

satisfying thing to do, which is to make it happen with our own hands. Not with real dogs — real dogs take years, and I'd like you to see this work before dinner. We're going to build a population of pretend dogs inside the computer, give them a trait, choose the good ones, breed them, and watch — in actual printed numbers — the whole pack get faster, generation by generation, with nobody in charge.

Turn the page and open your editor.

CHAPTER 4

BREEDING DOGS IN CODE

We're going to build a kennel inside the computer. Our dogs won't have faces or names or tails. Each one will be a small bundle of numbers, and the only thing we'll care about is how fast it can run. Then we'll do to them exactly what a racing breeder does: let the fastest ones have the puppies, over and over, and watch what happens.

Open a new file. Call it `dogs.py`. We'll build it up a piece at a time, and by the end you'll be able to run the whole thing and watch a pack of dogs get faster before your eyes.

WHAT IS A DOG, TO A COMPUTER?

A real dog's speed comes from hundreds of things: leg length, heart size, muscle type, the shape of a shoulder. No single "speed gene."

Speed is the sum of lots of small contributions, some helping a little, some helping a lot.

We'll copy that idea directly. Each of our dogs is a list of eight numbers. Think of each number as one "gene" — one small contribution to how fast the dog is. A dog's speed is just those eight genes added together.

```
import random

GENES = 8          # each dog has 8 genes that add up to how fast it
is

def random_dog():
    return [random.uniform(0, 5) for _ in range(GENES)]

def speed(dog):
    return sum(dog)
```

`random_dog()` makes a brand-new dog with eight random genes, each somewhere between 0 and 5. `speed(dog)` reads a dog's genes and adds them up. A dog whose genes happen to be all high is fast; a dog with low genes is slow; most dogs are somewhere in the middle. That spread — that's our variation, ingredient number two, and we got it for free just by making the genes random.

Try it. Add this at the bottom, run it, and you'll see a random dog and its speed:

```
d = random_dog()
print(d)
print("speed:", speed(d))
```

Run it a few times. Different dogs, different speeds. Good. Delete those two print lines before moving on — they were just to look.

MAKING A PACK

One dog is no use to evolution. We need a population — ingredient one. That's just a list of dogs:

```
POP = 20          # dogs in a generation

pack = [random_dog() for _ in range(POP)]
```

Twenty random dogs. Some fast, some slow, all of them accidents of `random.uniform`. This is generation zero: the starting crowd, before anyone has selected anything. It is, on average, mediocre. That's fine. Mediocre is where every breeding program starts.

CHOOSING WHO BREEDS

Now the important part — selection, ingredient three, the character to keep your eye on. Our rule is the racing breeder's rule, about as simple as it gets: the fastest half of the pack get to be parents; the slow half don't.

To do that we sort the pack by speed, fastest first, and keep the top half:

```
pack.sort(key=speed, reverse=True)
parents = pack[:POP // 2]    # the fastest 10 dogs
```

`pack.sort(key=speed, reverse=True)` lines the dogs up fastest-to-slowest. `pack[:POP // 2]` takes the front of that line — the fastest ten. Those ten are our breeding stock. The other ten don't pass anything on. Harsh, but that's selection: it always means somebody doesn't get to reproduce.

This one line — what counts as good enough to breed? — is the single most important decision in every program in this book. Today it's raw speed. Later it'll be much cleverer things. But the shape never changes: score everyone, keep the best, throw the rest.

MAKING PUPPIES

The parents need to make the next generation. This is heredity with change — ingredient four — and it has two parts, and both matter.

First, a puppy gets each of its genes from one parent or the other. Not a blend, not an average — for each of the eight gene-slots, we flip a coin and take mum's gene or dad's gene. This is called crossover, and it's a real thing that happens when actual sperm and egg are made. It shuffles the parents' genes into a new combination.

Second, once in a while a gene changes slightly — a small random nudge up or down. This is mutation. It's rare, it's small, and it's the only source of genuinely new material in the whole system.

Crossover just reshuffles what's already there; mutation invents. Without it, a population can never get better than its best starting gene. With it, there's always a trickle of something new to select on.

MUTATION = 0.15 # chance a puppy's gene changes a little

```
def breed(mum, dad):
```

```
    puppy = []
```

```
    for i in range(GENES):
```

```
        gene = mum[i] if random.random() < 0.5 else dad[i]    #
```

```
    crossover: take each gene from one parent
```

```
        if random.random() < MUTATION:
```

```
            gene = max(0.0, gene + random.uniform(-1, 1))    #
```

```
    mutation: a small random change
```

```
        puppy.append(gene)
```

```
    return puppy
```

Walk through breed. For each of the eight gene-slots: flip a coin (`random.random() < 0.5`) and take that gene from mum or from dad — that's the crossover. Then, with a 15% chance, nudge that gene by a small random amount — that's the mutation. (`max(0.0, ...)` just stops a gene going negative, which would be a nonsense dog.) Do that eight times and you've got a whole puppy.

THE LOOP THAT DOES EVERYTHING

Now we put the four ingredients in a loop and let it run. This is the heart of the book — read it slowly, because the same shape reappears in every part, all the way to the evolutionary algorithms at the end.

GENERATIONS = 12

for gen in range(GENERATIONS):

 pack.sort(key=speed, reverse=True)

 best = speed(pack[0])

 avg = sum(speed(d) for d in pack) / POP

 print(f"Generation {gen:2d}: fastest {best:5.1f} average
 {avg:5.1f}")

parents = pack[:POP // 2] # SELECTION: keep the fastest
half

 puppies = []

 while len(puppies) < POP:

```
mum, dad = random.sample(parents, 2) # pick two different
parents
puppies.append(breed(mum, dad))      # HEREDITY +
CHANGE: make a puppy
pack = puppies                        # the puppies become the new
pack
```

Each time round the loop is one generation. We sort the pack, print how the fastest and the average dog are doing, keep the fastest half as parents, then fill a new pack of twenty puppies by repeatedly picking two parents and breeding them. The puppies replace their parents, and round we go again. Twelve generations. That's the whole program.

RUN IT

Here's the complete file. Put `random.seed(8)` at the top so your numbers match mine exactly — it fixes the "random" choices to the same sequence every run, which is priceless when you're learning, because you and I see the same thing.

```
import random

random.seed(8) # same results every run — and a nod to where this
book is headed
```

```
GENES = 8
POP = 20
GENERATIONS = 12
MUTATION = 0.15

def random_dog():
    return [random.uniform(0, 5) for _ in range(GENES)]

def speed(dog):
    return sum(dog)

def breed(mum, dad):
    puppy = []
    for i in range(GENES):
        gene = mum[i] if random.random() < 0.5 else dad[i]
        if random.random() < MUTATION:
            gene = max(0.0, gene + random.uniform(-1, 1))
        puppy.append(gene)
    return puppy

pack = [random_dog() for _ in range(POP)]

for gen in range(GENERATIONS):
    pack.sort(key=speed, reverse=True)
    best = speed(pack[0])
    avg = sum(speed(d) for d in pack) / POP
```

```
print(f"Generation {gen:2d}: fastest {best:5.1f} average  
{avg:5.1f}")  
parents = pack[:POP // 2]  
puppies = []  
while len(puppies) < POP:  
    mum, dad = random.sample(parents, 2)  
    puppies.append(breed(mum, dad))  
pack = puppies
```

Run it. You should see exactly this:

```
Generation 0: fastest 27.9 average 18.9  
Generation 1: fastest 33.0 average 21.7  
Generation 2: fastest 30.9 average 24.5  
Generation 3: fastest 34.8 average 27.3  
Generation 4: fastest 34.5 average 30.2  
Generation 5: fastest 36.9 average 33.0  
Generation 6: fastest 37.8 average 34.4  
Generation 7: fastest 39.1 average 35.9  
Generation 8: fastest 39.1 average 37.3  
Generation 9: fastest 39.8 average 38.0  
Generation 10: fastest 41.0 average 38.9  
Generation 11: fastest 41.8 average 39.9
```

Sit with that for a second, because you just made evolution happen.

The average dog started at 18.9 and finished at 39.9 — the whole pack more than doubled in speed. And look at how the average chases the fastest and closes the gap: generation 0 has a slow pack with one lucky sprinter way out front (18.9 average, 27.9 best — a huge gap). By generation 11 the whole pack is fast and bunched up right behind the leader (39.9 average, 41.8 best). Selection didn't just pull up the champion; it dragged the entire population up behind it.

And notice — nowhere in that program did you tell a dog how to be fast. You never wrote "make the legs longer" or "grow a bigger heart." You wrote down four ingredients — a population, some variation, a rule for who breeds, and a way to make imperfect copies — and speed built itself. Nobody was in charge. That's the whole trick, and you just ran it.

TWO THINGS WORTH BREAKING

Before you leave this program, break it on purpose. Breaking things is how you find out what each part was really doing.

Turn off mutation. Set `MUTATION = 0.0` and run again. The pack still improves for a few generations — crossover is reshuffling the good genes that happened to be there at the start — and then it stalls. It hits a ceiling and sits there. That ceiling is the best combination of the genes that existed in generation zero. With no mutation, nothing new can ever enter the pool, so the pack can never climb past the

luck of its founders. Mutation is what keeps the ceiling rising.

Turn off selection. Change `parents = pack[:POP // 2]` to `parents = pack` — now everyone breeds, fast and slow alike. Run it. The improvement vanishes. The average wobbles around near where it started and goes nowhere. No selection, no direction. This is the experiment that proves selection was doing the work all along: variation and heredity are just the raw materials; selection is the thing that points them somewhere.

Put both back the way they were. You've now felt, in your own hands, what each ingredient contributes — and that intuition is worth more than any diagram.

BREED YOUR OWN

Stretch the code you just wrote. No right answers; just interesting ones.

1. The elite always survives. Right now the fastest dog might, by bad luck of the coin-flips, produce only slow puppies and vanish. Add "elitism": copy the single fastest dog straight into the new pack, unchanged, before breeding the rest. Does the fastest line ever go down now? Should it be able to?
2. Two traits at once. Give each dog a second trait — say the last two

genes are "speed" and the first six are "stamina" — and select on speed + stamina instead. Watch both climb. What happens if you select on speed minus stamina — can you breed a fast dog that has no staying power?

3. A bigger, slower world. Raise POP to 200 and GENERATIONS to 100. Does a bigger pack reach a higher ceiling, or just get there more smoothly? (Both, a little. Think about why more dogs means more variation to select from.)

You've now built the loop that the rest of this book runs on. In Part Two we go looking for the rules underneath it — because so far we've been cheating slightly. Real genes don't blend and nudge quite the way ours did. To find out how they actually work, we're going to visit a monk, a monastery garden, and about twenty-nine thousand pea plants.

PART TWO — PLANTS

• • •

CHAPTER 5

THE MONK IN THE GARDEN

In Part One we cheated a little, and I promised we'd come back and own up to it. Here's the cheat.

When our dogs bred, each puppy took each gene from one parent or the other, then sometimes nudged it up or down by a small random float. That worked — the pack got faster — and the "one parent or the other" part was actually closer to the truth than most people believed for most of history. But the nudging hid something. It made a gene look like a dial that could be turned to any value, smoothly, forever. Real inheritance isn't a smooth dial. It's much stranger, much simpler, and much more powerful, and to see it we have to leave the racing track and go and stand in a vegetable garden.

THE PROBLEM NOBODY COULD SOLVE

For thousands of years people bred plants and animals without having the faintest idea how traits were passed on. They could see that they were — tall parents tend to give tall children, this is obvious — but the mechanism was a total mystery. And the best guess everyone had was wrong.

The best guess was blending. The idea that a child is an average of its two parents, the way a bucket of white paint and a bucket of black paint give you grey. Tall crossed with short gives medium. Medium crossed with medium gives medium. It feels right. It even looks right, a lot of the time.

But it has a fatal flaw, and it's a flaw a programmer will smell immediately. If every generation averages its parents, then variation halves every generation. Mix all the paint together long enough and the whole world turns the same shade of grey. Every difference gets diluted away. After enough generations everybody should be identical — and then evolution has nothing left to work with, because remember from Part One: no variation, nothing to select. Blending inheritance quietly destroys the raw material that the whole loop depends on.

This genuinely worried the people thinking hardest about it, Charles Darwin among them. He'd worked out that selection shapes populations — the dog loop from Part One — but the inheritance theory of his day was busy erasing the variation his theory needed to

eat. Something was wrong with the picture, and nobody could see what.

The answer was sitting in a monastery garden in a town called Brunn, being worked out with almost unbelievable patience by a friar the world would ignore for another thirty-five years.

A FRIAR, A GARDEN, AND TWENTY-NINE THOUSAND PEA PLANTS

Gregor Mendel was an Augustinian friar with a head for numbers and a monastery plot full of peas. Between about 1856 and 1863 he did something almost nobody had thought to do with breeding: he counted. Not a few plants. Nearly twenty-nine thousand of them, over eight years, one careful cross at a time.

Peas were a shrewd choice. They come in clean, either-or varieties — a pea plant is tall or short, its seeds round or wrinkled, yellow or green, with nothing much in between. No blending-friendly middle ground to muddy the water. And peas usually fertilise themselves, so Mendel could keep pure lines going — plants whose short parents and grandparents were all short, "true-breeding" — and then cross them by hand exactly when he chose.

He started with pure-breeding parents that differed in one trait. Pure tall crossed with pure short. And here's the first thing that should

have been impossible under blending:

The children were not medium. Every single one was tall.

Not tallish. Not a bit shorter than the tall parent. Tall, full stop, as if the short parent had contributed nothing at all. The shortness hadn't blended in and made them medium. It had vanished.

Except it hadn't. And this is the moment the whole thing cracks open. Mendel took that all-tall generation — he called it the first filial generation, F1 — and let it self-fertilise to make a second generation, F2. If the shortness were truly gone, F2 should be all tall too. It wasn't. The shortness came back:

Of the F2 plants, 787 were tall and 277 were short.

Short had disappeared for a whole generation and then reappeared, in almost exactly one plant in four. Blending can't do that. You cannot un-mix grey paint back into black. Whatever was carrying "short" hadn't been diluted at all — it had been hidden, whole and intact, and passed down through a generation of tall plants unchanged, waiting.

THE RULES UNDER THE LOOP

Look at that ratio. 787 to 277. Divide it out: 2.84 to 1. Near enough three tall for every one short. Mendel saw the same near-3-to-1 in

trait after trait — round versus wrinkled seeds, yellow versus green, over thousands of plants — and he did the thing that makes him the hero of this book. He didn't just record it. He worked out the rule that would produce it, a rule so simple you can hold it in one hand.

Here it is, in four pieces.

1. Traits come from factors that stay whole. Each plant carries, for each trait, two factors — we call them alleles now. Not a blended value: two separate, intact things. A tall factor and a short factor sit side by side without mixing, like two cards in a hand, not two paints in a bucket. This is the idea that kills blending dead. Factors are particles, not fluids. They don't average; they keep their identity forever.
2. One is dominant, one is recessive. When a plant carries one of each — one tall factor, one short factor — you only see one of them. Tall is dominant: one tall factor is enough to make the whole plant tall. Short is recessive: it only shows when there's nothing else to hide behind, when the plant carries two short factors and no tall one at all. That's why the F1 were all tall (every one had exactly one tall factor and one short) — and why short could hide for a generation without being destroyed.
3. Each parent gives one factor, at random. When a plant makes a seed, it puts in just one of its two factors — a coin-flip which one —

and the other parent does the same. The seed ends up with two factors again, one from each side. This is the engine. This single fair coin-flip, repeated, is what generates the ratio.

4. That's the whole machine. Now watch it produce 3-to-1 with nothing up its sleeve. Take two F1 plants. Each carries one tall (call it T) and one short (t) — we'd write it Tt. Each gives one factor at random. So a seed can get:

		gets T from dad	gets t from dad	
-----	:	-----:	:	-----:
T from mum		TT		Tt
t from mum		Tt		tt

Four equally likely combinations. TT, Tt, Tt, tt. Three of them contain at least one T, so three of them are tall. Only the last, tt — short factor doubled up, no tall to hide behind — is short. Three tall, one short. There's your 787 to 277. Not a coincidence, not roughly, but the exact and inevitable output of a fair coin flipped by particles that never blend.

That little table has a name — a Punnett square — and it is, though nobody called it this for another century, an algorithm. A tiny, exact procedure that takes two parents in and gives the odds of every possible child out. Mendel found a rule that computes heredity.

WHY THIS MATTERS FOR THE CODE

Sit with what just happened, because it changes how you should think about the dog loop from Part One.

The variation that selection feeds on is not fragile. Blending theory said differences wash out; Mendel proved they don't. A recessive factor can go completely invisible for generations — silent, unselectable, doing nothing — and still be sitting there whole, ready to reappear the moment two carriers meet. Populations quietly hoard hidden variation. This is why a species can seem uniform for ages and then, under new pressure, suddenly produce forms nobody had seen. The material was never gone. It was just recessive.

And notice the shape of Mendel's factor: it isn't a smooth dial you nudge, like our dog genes. It's a discrete choice — this allele or that one, inherited whole, dominant or recessive, present or hidden. That's the honest version of heredity, and it's the version we're going to build in the next chapter. We'll drop the float-nudging cheat and code inheritance the way Mendel actually found it: two alleles per plant, one passed on at random, dominance deciding what you see. And when we run it over a few thousand plants, the 3-to-1 will fall out of our own coin-flips exactly as it fell out of his garden — because it's the same algorithm, and now you'll be the one running it.

Mendel published all this in 1866. The world's biologists read it, or

didn't, and shrugged, and forgot it for thirty-four years. He died an abbot, not a scientist, with no idea that he'd found the rulebook underneath everything Darwin was fighting to explain. Then in 1900 three different researchers, working separately, rediscovered his paper and realised what had been sitting in it the whole time.

We're not going to wait thirty-four years. Turn the page and open your editor. We're going to grow his peas in Python and count them ourselves.

CHAPTER 6

PEAS IN CODE

Mendel needed eight years and twenty-nine thousand plants. We're going to reproduce the heart of his result in about twenty lines of Python and half a second, because we can flip his coins a hundred thousand times before the kettle boils. Open a new file. Call it `peas.py`.

The point of this chapter isn't just to see 3-to-1 appear (though it will, and it's satisfying). It's to write down the rule from the last chapter as code — and in doing so, to drop the cheat from Part One and build heredity the honest way: whole alleles, inherited one at a time, with dominance deciding what you actually see.

WHAT IS A PEA, TO A COMPUTER?

In the dog chapter a creature was a list of eight numbers. A pea is

simpler and, in an important way, truer to life. For one trait — height — a plant carries exactly two alleles, and each allele is just a letter: 'T' for the dominant tall factor, 't' for the recessive short one. So a plant is a pair:

('T', 'T') # pure tall

('T', 't') # tall, but secretly carrying short

('t', 't') # short

That's the whole organism. No numbers to add up, no dial to nudge — just two cards in a hand. Everything Mendel found is going to come out of how those two cards get dealt to the next generation.

THE ONE RULE THAT MATTERS: MAKING A SEED

Here is Mendel's engine, rule three from the last chapter, in one line: when a plant makes a seed, it puts in one of its two alleles, chosen at random.

```
import random
```

```
def gamete(plant):  
    return random.choice(plant)
```

A gamete is the biology word for the single-allele contribution a parent makes to a seed — think pollen, or egg. `random.choice(plant)`

reaches into the plant's pair and pulls out one allele, a fair coin-flip between the two. That's it. That one function is the coin Mendel flipped twenty-nine thousand times. We're about to flip it a hundred thousand.

A seed gets one gamete from each parent, so a cross is just two coin-flips:

```
def cross(mum, dad):  
    return (gamete(mum), gamete(dad))
```

One allele from mum, one from dad, paired up into a brand-new plant. Notice what's not here: no blending, no averaging, no nudging. Each allele arrives whole and unchanged, exactly as the last chapter insisted. This is the honest heredity we promised.

DOMINANCE: WHAT YOU SEE VERSUS WHAT YOU CARRY

A plant carries two alleles but only shows one trait. Rule two: tall is dominant, so a plant is tall if it has even a single 'T'. It's short only when it has no 'T' at all — two 't's, nothing to hide behind.

```
def is_tall(plant):  
    return 'T' in plant  
  
def phenotype(plant):
```

```
return "tall" if is_tall(plant) else "short"
```

"T" in plant is True for ("T","T") and for ("T",'t') — both tall — and False only for ('t','t'). That single in check is dominance. The word phenotype means "what you can see" (tall or short); the pair of alleles itself is the genotype (TT, Tt, or tt). The whole magic trick of Chapter 3 — short vanishing then reappearing — lives in the gap between those two words: two different genotypes, TT and Tt, show the exact same phenotype, so Tt can smuggle a hidden t through a generation completely unseen.

THE F1: WHERE DID SHORT GO?

Let's run Mendel's first cross. Pure tall meets pure short:

```
pure_tall = ('T', 'T')
pure_short = ('t', 't')

f1 = [cross(pure_tall, pure_short) for _ in range(1000)]
tall = sum(1 for p in f1 if is_tall(p))
print(f"F1: {tall} tall, {1000 - tall} short  every plant is {f1[0]}")
```

Think before you run it. `pure_tall` can only ever hand over a 'T' — both its alleles are 'T', so the coin-flip is rigged. `pure_short` can only ever hand over a 't'. So every seed is forced to be ("T", 't') — one of each, guaranteed. And every Tt plant is tall. So all 1000 should be

tall, every one of them secretly carrying the short allele that seems to have disappeared. Run it and check you were right.

THE F2: SHORT COMES BACK, 3 TO 1

Now the cross that made Mendel famous. Take those Tt F1 plants and breed them with each other. Each parent is ('T', 't'), so now the coin-flip is fair — a seed can get T or t from either side. This is the Punnett square from the last chapter, except instead of drawing four boxes we'll flip the actual coins, thousands of times, and count.

And we'll do it at four different sample sizes, because the size is the lesson:

```
print("F2 (Tt x Tt), more plants each time:")
parent = ('T', 't')
for n in (100, 1000, 10000, 100000):
    f2 = [cross(parent, parent) for _ in range(n)]
    tall = sum(1 for p in f2 if is_tall(p))
    short = n - tall
    print(f"  {n:6d} plants: {tall:6d} tall {short:6d} short  ratio
{tall/short:.2f} : 1")
```

THE WHOLE FILE

Put `random.seed(8)` at the top, as before, so your numbers match

mine exactly. Here's everything together, plus a last block that cracks open the tall plants to show what genotypes were hiding inside them:

```
import random

random.seed(8)

def gamete(plant):
    return random.choice(plant)

def cross(mum, dad):
    return (gamete(mum), gamete(dad))

def is_tall(plant):
    return 'T' in plant

def phenotype(plant):
    return "tall" if is_tall(plant) else "short"

pure_tall = ('T', 'T')
pure_short = ('t', 't')

f1 = [cross(pure_tall, pure_short) for _ in range(1000)]
tall = sum(1 for p in f1 if is_tall(p))
print("F1 (pure tall x pure short)")
print(f"  {tall} tall, {1000 - tall} short  every plant is {f1[0]}")
```

```
print()

print("F2 (Tt x Tt), more plants each time:")
parent = ('T', 't')
for n in (100, 1000, 10000, 100000):
    f2 = [cross(parent, parent) for _ in range(n)]
    tall = sum(1 for p in f2 if is_tall(p))
    short = n - tall
    print(f"  {n:6d} plants: {tall:6d} tall {short:6d} short  ratio
{tall/short:.2f} : 1")
print()

counts = {'TT': 0, 'Tt': 0, 'tt': 0}
for p in f2:
    counts[".".join(sorted(p))] += 1
print("Genotypes in that last F2 batch:")
for g, c in counts.items():
    print(f"  {g}: {c:6d}  ({phenotype(tuple(g))})")
```

Run it. You should see exactly this:

F1 (pure tall x pure short)

1000 tall, 0 short every plant is ('T', 't')

F2 (Tt x Tt), more plants each time:

100 plants: 77 tall 23 short ratio 3.35 : 1

1000 plants: 779 tall 221 short ratio 3.52 : 1
10000 plants: 7466 tall 2534 short ratio 2.95 : 1
100000 plants: 75250 tall 24750 short ratio 3.04 : 1

Genotypes in that last F2 batch:

TT: 25151 (tall)
Tt: 50099 (tall)
tt: 24750 (short)

There it is. You just re-ran Mendel's eight years of gardening, and out of nothing but a coin-flipped random.choice came his 3-to-1.

READ THE NUMBERS SLOWLY

Two things in that output are worth more than the whole program.

First, the ratio homes in on 3-to-1 — but not politely. Look at the sequence: 3.35, then 3.52, then 2.95, then 3.04. It does not march smoothly toward 3. At 1000 plants it's actually further off than it was at 100. That's the real texture of randomness, and it's worth feeling in your bones now, because it'll matter for the rest of the book: more samples don't make every step better — they make the answer reliable in the long run. A hundred plants can land anywhere from 2-ish to 4-ish on any given day; a hundred thousand plants lands on 3 and stays there, run after run. This is exactly why Mendel couldn't have found his rule with a dozen plants. The rule was always there

— it only becomes visible through big numbers. Small samples lie, cheerfully and often. Remember that.

Second, look at the genotypes behind that last batch. Roughly TT 25,000, Tt 50,000, tt 25,000 — a clean 1 : 2 : 1. But TT and Tt both look identical: both just "tall". Out in the garden you could never tell them apart by eye. Fully half your tall plants are secretly carrying a short allele they'll pass on. That hidden Tt layer — invisible, unselectable, a quarter of the whole population smuggling t forward — is the variation-hoard from the last chapter, made of numbers you can now see. It's why short can never be bred away just by pulling out the short plants: you'd be missing every carrier.

And notice, one more time, the thing that ties this book together: nobody told the peas to come out 3-to-1. You wrote down one honest rule — each parent gives one whole allele at random, and one letter dominates the other — and the ratio built itself. Same as the dogs. A simple local rule, repeated across a population, producing a precise global pattern that nobody designed. That's the whole trick, wearing leaves this time.

TWO THINGS WORTH BREAKING

Break it on purpose, the way you broke the dog code.

Kill dominance. Change `is_tall` to `return plant == ('T', 'T')` — now a

plant only counts as tall if it's purely tall, Tt counts as short. Re-run and watch the F2 ratio flip from 3-to-1 toward 1-to-3, because now only the TT quarter shows the trait. You've just turned tallness recessive. This is the whole reason dominant traits look common and recessive ones look rare, even when the underlying alleles are split fifty-fifty: dominance decides how many of the carriers you can actually see.

Break segregation. What if a parent, instead of handing over one random allele, always handed over the same one? Change gamete to return plant[0] — always the first allele, no coin-flip. Re-run and watch what happens: every Tt parent now always passes on its T, so every F2 plant comes out ('T', 'T') — 100,000 tall, zero short. In fact the program crashes, with a ZeroDivisionError, the moment it tries to divide by that zero. Sit with why: there are no short plants left to divide by, because you removed the only thing that ever made a short plant — the fair coin-flip that occasionally deals two ts. Mendel's 3-to-1 depends entirely on that random pick between a parent's two alleles. Segregation — the honest coin-flip in gamete — isn't a detail. It's the whole engine, and the crash is the engine's silence. Put it back.

BREED YOUR OWN

Stretch the code. As always, no right answers — just interesting ones.

1. A second trait. Give each plant a second gene — say seed colour, 'Y' (yellow, dominant) and 'y' (green) — as its own pair of alleles, inherited independently. Cross two Tt Yy plants a hundred thousand times and count the four combinations: tall-yellow, tall-green, short-yellow, short-green. Mendel found these come out 9 : 3 : 3 : 1. Can you make that ratio fall out of your code? (This was his second great discovery — that separate traits are shuffled independently.)
2. A carrier hunt. Among the tall F2 plants, you know from the genotype count that about two-thirds are secretly Tt carriers. Write a function that takes a tall plant and test-crosses it with a pure short plant (tt) many times: if any short offspring appear, the parent must have been a hidden Tt. This is exactly how a real breeder smokes out carriers — you've just automated a genetic test.
3. Selection returns. Combine this chapter with Part One. Instead of one static cross, run generations: each generation, remove all the short plants (select against tt) before breeding the rest. Does short ever fully disappear? Watch how slowly the t allele drains away even under total selection — because it keeps hiding inside the tall Tt carriers. This is one of the most important facts in all of genetics, and you can watch it happen in your terminal.

You've now met the rules under the loop: heredity isn't a smooth blend, it's whole particles, dealt one at a time, some dominant over others. In Part Three we go one level deeper still — down to the

actual code those particles are written in. Four letters, in a line, in every living thing. We're going to read some real DNA, and find out how a tube of spit can locate a cousin you've never met.

PART THREE — DNA

. . . .

CHAPTER 7

THE BOOK OF FOUR LETTERS

We've been circling something for two whole parts without ever looking at it directly. In Part One a "gene" was a number that helped a dog run fast. In Part Two it got sharper — Mendel's factor, an allele, a whole particle that's either T or t and never blends. But that only pushes the question back a step. What is an allele? What is a factor actually made of? When a pea hands a T to its seed, what physical thing gets handed over?

For all of history nobody could answer that. Mendel found the rules of a code he couldn't read — like working out the grammar of a language from its patterns without ever seeing the alphabet. In this part we finally see the alphabet. And it turns out to be almost insultingly simple: four letters.

EVERY LIVING THING IS WRITTEN IN THE SAME FOUR

LETTERS

Inside almost every cell of your body there is a molecule called DNA, and if you could stretch it out and read it, it would look like an impossibly long string spelled out of just four characters:

...A C G T T A G C C A A T G A T C ...

A, C, G and T. Adenine, cytosine, guanine, thymine — but the names don't matter, and that's rather the point. To us they're just four letters. Your entire genome — the full instruction set for building and running a human being — is one long line of those four letters, about three billion of them. A dog's genome is the same four letters in a different order. A pea's, an oak's, a bacterium's, a blue whale's — same four-letter alphabet, every one. Life is not written in millions of different scripts. It's all written in the same one, and it's the smallest alphabet you could imagine that still carries real information. Two bits per letter.

So here, at last, is what an allele is. Mendel's T factor isn't a mysterious particle. It's a stretch of these letters — a specific run of A's, C's, G's and T's sitting at a specific spot on the string. His t is the same spot, spelled slightly differently. Tall and short peas differ by a few letters in one word of a three-billion-letter book. That's all a "factor" ever was: a passage of text. Mendel was reading the story by its effects; we're now looking at the ink.

COPYING THE BOOK — AND GETTING IT SLIGHTLY WRONG

Now, the crucial thing about this book isn't just that it exists. It's that it gets copied, constantly. Every time a cell divides, it copies its entire three-billion-letter string so each half gets a complete edition. Every time a living thing reproduces, it passes a copy on. Life is, more than anything else, a copying machine for a four-letter text.

And no copier is perfect. Copy three billion letters and, however careful the machinery, a few will come out wrong — an A where there should have been a G, a letter dropped, a letter doubled. A typo. In biology a typo in the DNA has a name you already know from Part One: a mutation.

Stop and feel the whole book snap together here. Back in Chapter 2 we gave our dogs a MUTATION chance — a small random nudge — and I told you it was the only source of genuinely new material in the whole system. I wasn't being poetic. That is literally, molecularly, true. A mutation is a copying error in the four-letter text. It is the one and only way a letter that nobody's ancestors had can appear in the book. Crossover — Chapter 2's coin-flip that takes each gene from one parent or the other — is also real, and also molecular: when a parent packages up DNA to pass on, it physically shuffles together stretches from its mum and its dad. Crossover reshuffles letters that already exist. Mutation writes new ones. Everything Darwin needed,

everything the dog loop ran on, comes down to those two events happening to a string of A, C, G and T.

Which means variation — the raw material selection feeds on, the thing blending inheritance would have destroyed — is just accumulated typos. Every difference between you and the person next to you traces back to some copy error in some ancestor's DNA, preserved and passed down because nothing removed it. The tower of life is built on mistakes that stuck.

YOUR DNA IS A PATCHWORK OF YOUR ANCESTORS

Here's the consequence that makes the rest of this part possible.

When you were made, you got a copy of half of your mother's DNA and half of your father's — spliced together by crossover into one new string. But their DNA was itself a patchwork of their parents'. So your genome is a quilt: this stretch came from your father's mother, that stretch from your mother's father, this next bit from a great-grandfather you never met. You are a physical mosaic of scraps of text copied down from everyone you descend from, shuffled a little more at each step.

And that gives us something we can measure. If your DNA is a patchwork of your ancestors' DNA, then two people who share an ancestor are carrying some of the same scraps — the same runs of

letters, inherited from that shared source. The closer the relative, the more recently you shared an ancestor, and the more scraps you still have in common. The further apart, the more generations of shuffling have chopped the shared scraps smaller and scattered them.

You can put actual numbers on it, and they are beautifully regular. You share:

- 50% of your DNA with a parent, or with a full sibling.
- 25% with a grandparent, an aunt, an uncle. (One generation further — half of a half.)
- 12.5% with a first cousin. (Half again.)
- ~0% with a stranger you're not related to — no recent shared ancestor, no shared scraps.

Every step out along the family tree roughly halves the DNA you share. That halving is not a rule of thumb someone invented. It falls straight out of the coin-flip: each parent passes on a random half of their book, so each generation of distance cuts the shared fraction in half. It's the same fair coin from Mendel's peas, now counting cousins.

HOW A TUBE OF SPIT FINDS A COUSIN YOU'VE NEVER MET

Now you can see exactly how a home DNA test works — there's no

magic left in it.

You spit in a tube and send it off. A machine reads the letters at the spots where humans commonly differ from one another (it doesn't need all three billion — just the variable ones, the accumulated-typo positions, because the letters everyone shares tell you nothing). That gives the company your personal string of variable letters. Then they do the obvious thing: they compare your string to everyone else's in their database — millions of people — and look for anyone who shares long scraps with you.

And because of the halving, the amount you share tells them how you're related. Share about 50%? Parent or sibling. About 25%? Grandparent, aunt, or uncle. About 12.5%? First cousin. A stranger in the database who happens to share 12% of your DNA is not a coincidence and not a stranger — they are, whether either of you knew it, your cousin. The test didn't need to know your family. It reconstructed a piece of your family tree from nothing but shared scraps of a four-letter text.

(One honest wrinkle, so you're not surprised later: the percentage alone can't always tell relatives apart — you'll notice an aunt and a grandparent both sit at about 25%, because both are exactly one coin-flip-generation removed. Untangling which 25%-relative someone is takes a bit more — how the shared scraps are arranged, and old-fashioned family knowledge. The number narrows it down;

it doesn't always finish the job.)

This is real, and it is powerful enough to be sobering. These databases have reunited adopted children with birth parents, found grandparents nobody knew existed, and — because you share scraps even with cousins you've never heard of — helped police identify people who never took a test at all, through the relatives who did. A four-letter book, copied with occasional typos and shuffled every generation, remembers your whole ancestry whether you want it to or not.

WHAT WE'RE ABOUT TO BUILD

In the next chapter we'll build all of this in Python, and it's less code than you'd think. We'll write DNA as an honest string of A, C, G and T. We'll copy it with the odd random typo and watch mutations appear. Then we'll build a little family — grandparents, parents, children, cousins — by passing DNA down through crossover exactly the way real reproduction does, and we'll measure how much each person shares with each other. And out of nothing but coin-flips, the halving will appear: 50%, 25%, 12.5%, zero. You'll have built, in about thirty lines, the engine that finds a cousin in a tube of spit.

Turn the page and open your editor. We're going to read some DNA.

CHAPTER 8

DNA IN CODE

Time to build the tube of spit. By the end of this file you'll have a program that grows a small family, passes DNA down through it, and reads off who's a parent, who's a cousin and who's a stranger — from nothing but shared letters. Open a new file. Call it `dna.py`.

We'll build it in two parts. First we'll get comfortable with DNA as a real string of letters, and watch mutation happen. Then we'll pass that DNA down through a family and measure relatedness.

PART A: DNA IS A STRING, MUTATION IS A TYPO

There's no clever data structure here, and that's the lesson — DNA really is just a string of four letters. Let's make some:

```
import random
```

```
random.seed(8)
```

```
BASES = "ACGT"
```

```
def random_dna(length):
```

```
    return "".join(random.choice(BASES) for _ in range(length))
```

`random_dna(40)` builds a 40-letter string by picking a random base forty times and joining them up. Real genomes are three billion letters; forty is plenty to see what's going on.

Now the important bit — copying with mistakes. We walk along the string letter by letter, and almost always we copy the letter faithfully. But every so often, with some small probability, we fumble it and write a random base instead. That fumble is a mutation:

```
def copy_with_mutations(dna, rate):
```

```
    out = []
```

```
    for base in dna:
```

```
        if random.random() < rate:
```

```
            out.append(random.choice(BASES)) # a copy error: a
```

```
random base
```

```
        else:
```

```
            out.append(base) # copied faithfully
```

```
    return "".join(out)
```

And a way to see how many letters ended up different between two strings:

```
def differences(a, b):  
    return sum(1 for x, y in zip(a, b) if x != y)
```

`zip(a, b)` walks the two strings in step, handing us matching pairs (`x`, `y`); we count the pairs where the letters disagree. Let's copy a strand with a 5% error rate and look:

```
original = random_dna(40)  
child = copy_with_mutations(original, 0.05)  
print("original:", original)  
print("copy   :", child)  
print(f"{differences(original, child)} letters changed out of  
{len(original)}")
```

This is Chapter 2's `MUTATION` nudge, except now you can see it — an actual letter in an actual string, changed by an actual copy error. Same idea, made of ink.

PART B: PASSING DNA DOWN A FAMILY

Now the real machine. We want to measure how much DNA relatives share. To do that cleanly we're going to use a little trick: instead of tracking the letters a person carries, we'll track where each

stretch of their DNA came from — which ancestor it was inherited from. That way "shared DNA" means "inherited from the same source", which is exactly what a real DNA test measures, and it keeps the numbers clean.

Each person's genome will be a list of N tags, one per position. A brand-new, unrelated person — a founder — gets their own unique tag stamped on every position:

```
N = 20000          # positions in the genome we track
next_id = 0

def founder():
    global next_id
    person = [next_id] * N      # every position stamped with this
    founder's id
    next_id += 1
    return person
```

So founder 0's genome is [0, 0, 0, ...], founder 1's is [1, 1, 1, ...], and so on — each founder's DNA is entirely, recognisably theirs. Now reproduction. A child copies each position from one parent or the other, a fair coin-flip per position — this is crossover, the same coin from Chapter 2, now shuffling real inherited DNA:

```
def child_of(mum, dad):
```

```
return [mum[i] if random.random() < 0.5 else dad[i] for i in
range(N)]
```

A child of founder 0 and founder 1 comes out as a patchwork of 0s and 1s — about half and half — a physical mosaic of both parents, exactly like the quilt from the last chapter. And measuring relatedness is now trivial: two people share DNA at every position where their tags match, because a matching tag means that scrap came down from the same ancestor:

```
def shared(a, b):
    same = sum(1 for i in range(N) if a[i] == b[i])
    return same / N
```

BUILDING THE FAMILY

Let's grow a family big enough to have cousins in it. Two grandparents have two children (who are therefore siblings); each of those siblings marries an outsider and has one child; those two children are first cousins.

```
A = founder(); B = founder()      # grandparents, couple one
P1 = child_of(A, B)                # their two children...
P2 = child_of(A, B)                # ...P1 and P2 are full siblings
X = founder(); Y = founder()       # the outsiders P1 and P2 marry
C1 = child_of(P1, X)               # P1's child
```

```
C2 = child_of(P2, Y)          # P2's child -> C1 and C2 are first
                                cousins
stranger = founder()          # nobody's relative
```

Read that until the family tree is clear in your head — C1 and C2 are cousins because their parents P1 and P2 are siblings, sharing the grandparents A and B. Now we ask how much C1 shares with everyone:

```
print("How much DNA does C1 share with each person?")
print(f" parent (P1) : {shared(C1, P1)*100:5.1f}%")
print(f" grandparent (A) : {shared(C1, A)*100:5.1f}%")
print(f" aunt/uncle (P2) : {shared(C1, P2)*100:5.1f}%")
print(f" first cousin (C2) : {shared(C1, C2)*100:5.1f}%")
print(f" stranger : {shared(C1, stranger)*100:5.1f}%")
print()
print(f" full siblings P1&P2: {shared(P1, P2)*100:5.1f}%")
```

RUN IT

Here's the whole file — Part A then Part B — with `random.seed(8)` at the top so your numbers match mine:

```
import random

random.seed(8)
```

```
BASES = "ACGT"
```

```
def random_dna(length):  
    return "".join(random.choice(BASES) for _ in range(length))
```

```
def copy_with_mutations(dna, rate):  
    out = []  
    for base in dna:  
        if random.random() < rate:  
            out.append(random.choice(BASES))  
        else:  
            out.append(base)  
    return "".join(out)
```

```
def differences(a, b):  
    return sum(1 for x, y in zip(a, b) if x != y)
```

```
original = random_dna(40)  
child = copy_with_mutations(original, 0.05)  
print("original:", original)  
print("copy   :", child)  
print(f"{differences(original, child)} letters changed out of  
{len(original)}")  
print()
```

```
N = 20000
```

```
next_id = 0
```

```
def founder():
```

```
    global next_id
```

```
    person = [next_id] * N
```

```
    next_id += 1
```

```
    return person
```

```
def child_of(mum, dad):
```

```
    return [mum[i] if random.random() < 0.5 else dad[i] for i in  
range(N)]
```

```
def shared(a, b):
```

```
    same = sum(1 for i in range(N) if a[i] == b[i])
```

```
    return same / N
```

```
A = founder(); B = founder()
```

```
P1 = child_of(A, B)
```

```
P2 = child_of(A, B)
```

```
X = founder(); Y = founder()
```

```
C1 = child_of(P1, X)
```

```
C2 = child_of(P2, Y)
```

```
stranger = founder()
```

```
print("How much DNA does C1 share with each person?")
```

```
print(f"  parent  (P1)    : {shared(C1, P1)*100:5.1f}%")
```

```
print(f"  grandparent (A)  : {shared(C1, A)*100:5.1f}%")
print(f"  aunt/uncle (P2)  : {shared(C1, P2)*100:5.1f}%")
print(f"  first cousin (C2) : {shared(C1, C2)*100:5.1f}%")
print(f"  stranger          : {shared(C1, stranger)*100:5.1f}%")
print()
print(f"  full siblings P1&P2: {shared(P1, P2)*100:5.1f}%")
```

Run it. You should see exactly this:

original:

CGTCCAACCCTATTTTTCTATCAGTTTAGAATTAAGCATC

copy :

CGTCCAACCCTATTTTTCTATCAGTTTAGAAGTGAGCATC

2 letters changed out of 40

How much DNA does C1 share with each person?

parent (P1) : 50.1%

grandparent (A) : 25.2%

aunt/uncle (P2) : 25.0%

first cousin (C2) : 12.4%

stranger : 0.0%

full siblings P1&P2: 49.6%

There it is — the tube of spit, in thirty lines.

READ THE NUMBERS SLOWLY

Look down that column: 50, 25, 25, 12.5, 0. Every step out along the family tree halves the shared DNA, and it fell straight out of the per-position coin-flip in `child_of` — nobody put the halving in on purpose. A parent hands on a random half of their genome, so a child shares half with them; a grandchild is a half of a half, 25%; a cousin a half of that, 12.5%; and a stranger, sharing no recent ancestor, shares essentially nothing. This is the whole basis of every home DNA test, and you just reproduced it from scratch.

Now notice two honest things in that output, because they're where the real understanding is.

The grandparent and the aunt both read 25%. A and P2 are completely different relationships — one is two generations straight up, the other is a sideways branch — yet the percentage can't tell them apart, because both sit exactly one extra coin-flip-generation away from C1. This is exactly the wrinkle the last chapter warned about: the amount of shared DNA pins down how many generations of shuffling separate two people, but not the shape of the connection. A real testing company untangles a 25% match into "aunt" versus "grandmother" using the arrangement of the shared scraps and plain family knowledge — the number narrows it to a short list and stops there.

The stranger reads 0.0%, and that's a simplification we chose. Because we gave every founder a totally unique tag, unrelated people share nothing in our model. Real humans are about 99.9% identical whatever their ancestry, so a real "stranger" match isn't 0 — it's a low baseline of coincidental sharing that the test has learned to see through by only comparing the variable positions. Our tag trick quietly did that filtering for us. Worth knowing that the clean zero is a convenience, not quite the truth. (There's a second simplification too: real chromosomes are inherited in long continuous chunks, which is why real tests measure shared segments, not just a percentage. Our per-position coin-flip gets the percentages spot-on but chops the DNA finer than biology does. The headline numbers are right; the texture is smoothed.)

And once more, the refrain of the whole book: nobody designed the halving. You wrote one honest local rule — a child copies each position from one parent or the other, at random — and a precise, useful global pattern built itself, the same pattern that reunites families every day. Selection built speed in Part One. Fair coin-flips built Mendel's 3-to-1 in Part Two. And here the same coin-flip, counting scraps of a four-letter book, builds a family tree.

TWO THINGS WORTH BREAKING

Shrink the genome. Change $N = 20000$ to $N = 20$ and run a few times without the fixed seed (comment out `random.seed(8)`). The

percentages go haywire — a cousin might read 5% one run and 30% the next. Twenty positions is far too few letters to measure sharing reliably, exactly as a hundred pea plants were too few to see 3-to-1. This is the sampling lesson from Part Two returning in new clothes, and it's a real constraint on real tests: reading more of the genome is what makes the relationship estimate trustworthy. Put N back to 20000.

Add another generation. Give C1 a child with a fresh outsider, and give C2 a child with another. Those two new children are second cousins. Measure their sharing. You should find it's dropped to around 3% — halved twice more from the first cousins' 12.5%. Keep going and watch the shared fraction fall toward the stranger's zero. Somewhere out there is the distance at which real family becomes statistically invisible — where a true relative shares so little that a test can't tell them from a stranger. Find roughly where that happens in your model. It's the horizon of what a tube of spit can see.

BREED YOUR OWN

1. Reconstruct the tree. Take the whole family — A, B, P1, P2, X, Y, C1, C2, stranger — and print a full table of how much every pair shares. Then cover up the code that built them and try to reconstruct the family tree from the numbers alone: who are the 50% pairs (parents/siblings)? The 25%? The 12.5%? This is precisely the puzzle a genealogy company solves — they only ever see the sharing

table, never the tree, and they work backwards to it. You've just written the front half of their whole business.

2. The molecular clock. Take one DNA string and copy it down two separate lines for many generations, each with a small mutation rate, never letting the lines mix. Count how many letters differ between the two lines over time. You'll find the differences pile up at a steady average rate — so if you know the rate, you can turn a count of differences back into an estimate of how long ago the two lines split. That's the "molecular clock", and it's how scientists date when two species — or two human populations — last shared an ancestor, using nothing but accumulated typos.

3. A test that can be fooled. Our shared function is honest, but real matching only reads the variable positions. What happens to your relatedness numbers if two founders happen to share some tags by chance? Give founders genomes that are mostly unique but occasionally collide, and see how a little coincidental sharing muddies the readings — and think about why real tests need millions of people in the database before a match means anything.

You've now seen all three layers: selection that shapes a population (dogs), the rules that pass traits down (peas), and the four-letter code those rules are actually written in, with its crossover and its typos (DNA). Every piece of the machine is on the table. In Part Four we pick them all up at once and do the thing this whole book has been

walking toward: point evolution at a problem nobody knows how to solve by hand, and let it breed the answer.

PART FOUR — ALGORITHMS

. . . .

CHAPTER 9

BREEDING AN ANSWER NOBODY CAN DESIGN

Here is where the whole book has been heading.

For three parts we've watched one loop do its work in three disguises. We bred dogs for speed. We watched Mendel's coin-flip deal out alleles. We passed DNA down a family and measured the scraps. And every single time I kept whispering the same thing in your ear: nobody was in charge. No one designed the greyhound, no one designed the 3-to-1 ratio, no one designed your family tree. A simple loop — population, variation, selection, heredity-with-change — built all of it from the bottom up.

That's a lovely thing to notice about biology. But this is a book for a programmer, and now we cash it in. Because there is a whole category of problem where "nobody can design the answer" isn't a philosophical observation — it's a wall you smack into, hard, in real

work. And the loop we've been studying is the way through the wall.

THE PROBLEM THAT LAUGHS AT CLEVERNESS

Meet a delivery driver. She has a van, a depot, and this morning a list of twenty addresses to deliver to. She wants to drive the shortest possible route that visits all twenty and comes home. That's it. That's the whole problem. It has a friendly name — the Travelling Salesman Problem — and it is one of the most quietly monstrous problems in all of computing.

It doesn't look monstrous. With three stops there's basically one sensible loop. With four or five you can eyeball it. So you might think: fine, let the computer just try all the possible routes and keep the shortest. And for five stops, that's a great plan — there are only twelve possible routes. Easy.

But watch what happens as the list grows, because this is the whole point of the chapter. Every new stop doesn't add to the number of possible routes — it multiplies it. The number of possible routes through N stops is roughly N times $N-1$ times $N-2$... all the way down. Mathematicians write it $(N-1)! / 2$. Let's just look at what it does:

Stops	Possible routes
1	1
2	2
3	3
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800
12	479001600
13	6227020800
14	87178291200
15	1309324672000
16	20901888000000
17	355681472000000
18	6402353280000000
19	121665135360000000
20	2432902008960000000

5	12
10	181,440
15	about 43 billion
20	about 60,000,000,000,000,000

Read that last number again. Twenty stops — a light morning for a delivery driver — has sixty quadrillion possible routes. A fast computer checking a billion routes every second would need about two years to try them all. Add a few more stops and you blow past the number of seconds since the Big Bang. This isn't a matter of waiting for a better computer. "Try everything" is dead on arrival, forever, for any real delivery round.

So try to be clever instead. Sit down and design a rule — an algorithm — that builds the best route directly, without checking them all. People have been attacking this exact problem with serious mathematics for the better part of a century. And here is the humbling truth: for the general problem, nobody has ever found a fast, guaranteed-perfect method, and there are deep reasons to suspect nobody ever will. The Travelling Salesman Problem is the friendly face of a whole class of problems — the "NP-hard" problems — that sit at the edge of what computers can do. You cannot check every answer. And you cannot design the perfect answer by hand.

Stop and feel where that leaves us, because it's exactly the situation

from Part One, now with the stakes made real. No blueprint. No way to engineer the solution directly. The thing is too tangled to design.

Which is precisely when a breeder stops trying to design, and starts to breed.

LOWERING THE BAR ON PURPOSE

Here's the mental shift that unlocks everything, and it's worth saying out loud because it feels like cheating.

We are going to stop demanding the perfect route, and start hunting for a very good one, fast. The delivery driver does not need the mathematically optimal loop delivered on a silver platter after two years of computation. She needs a route that's excellent — near enough the best that the difference is a minute or two — and she needs it before she leaves. Evolution is outstanding at this. It doesn't search by being clever. It searches by breeding: it keeps a crowd of okay answers, mixes and mutates them, keeps the ones that came out better, and repeats — climbing toward good solutions without ever needing to understand the problem, exactly as a breeder improves a dog without understanding genetics.

We already have every piece we need. Let's map our four ingredients straight onto the driver's problem.

Population. Not one route — a whole crowd of them. A hundred, two hundred different possible orderings of the twenty stops. Most of them bad. That's fine; generation zero is always mediocre.

Variation. They're all different orderings, so they're already varied — we get it for free from shuffling, exactly as we got varied dogs for free from random.uniform.

Selection. Score every route by its total length, and keep the shortest ones to breed. Notice the small but important flip: with dogs we kept the highest score (speed), here we keep the lowest (distance).

Selection doesn't care which direction "better" points — it only cares that you can rank them and throw away the worse half. Score everyone, keep the best, throw the rest. Same sentence as Chapter 2. It always is.

Heredity with change. Breed two good routes into a child route that resembles both, and occasionally mutate it. And here — right here — is the one genuinely new idea in this whole part, and it's a callback to Mendel: heredity has rules, and if you break them you get nonsense.

THE WRINKLE: YOU CAN'T BREED ROUTES CARELESSLY

With dogs, breeding was easy: take each gene from one parent or the other, done. Try that with routes and watch it explode. Suppose

mum's route is [London, Leeds, Bath, York] and dad's is [Bath, York, London, Leeds], and for each slot we flip a coin like we did with dogs. We might get [London, York, London, Leeds] — a "route" that visits London twice and never goes to Bath at all. That's not a worse route. It's not a route at all. It's gibberish.

A valid route is a permutation: every stop appears exactly once. That's a hard rule, like Mendel's alleles being whole and countable — and our breeding has to respect it. So we need a slightly cleverer way to mix two routes that always produces a legal one. It's called order crossover, and the idea is simple even though it sounds fancy:

Take a run of stops from mum, in her order. Then fill the empty slots with the remaining stops, in the order dad lists them.

Because we take each remaining stop from dad exactly once, and we only take stops not already copied from mum, every stop ends up present exactly once. Legal route, guaranteed, and it genuinely inherits structure from both parents — a chunk of mum's ordering, threaded through with dad's. Mutation is easier: just swap two stops. Swapping two entries in a permutation always leaves a permutation, so it's always legal, and it nudges the route into something new — the fresh variation that lets the population climb past its starting luck, exactly as mutation did for the dogs.

That's the whole design. It is the Chapter 2 loop, wearing a delivery

uniform, with one careful new breeding rule so we never make a gibberish child.

ONE MORE TRICK, SO WE CAN CHECK THE MACHINE'S HOMEWORK

There's a problem with testing a Travelling Salesman solver: for twenty real scattered stops, we don't know the perfect answer either — that's the whole reason we're breeding one. So how would we ever know if our GA did well or badly? Near enough the best... compared to what?

We'll use a trick. We'll place our stops evenly around a circle. Because then we know something the GA doesn't: the shortest possible route through points on a circle is obviously the one that goes smoothly round the rim, never cutting across the middle. We can even calculate its exact length. So we hand the GA the stops in a scrambled order, with no hint that they lie on a circle, and we set it breeding. If it rediscovers the rim — if the route it breeds matches the perimeter we secretly calculated — then it found the genuine, provable optimum, and we'll know, because we can check its homework against an answer it was never given.

That's a real technique, by the way, not just a teaching gimmick: giving your optimiser a problem whose answer you already know, to prove it works, before you turn it loose on the real problems whose

answers you don't.

Turn the page and open your editor. We're going to breed the shortest route through sixty quadrillion, and watch it find the one the mathematicians would have to grind two years to confirm.

CHAPTER 10

THE SALESMAN IN CODE

We're going to breed the shortest delivery route through twenty stops. Open a new file, call it `salesman.py`, and keep Chapter 2's dog program open in another window if you can — because the loop at the heart of this one is the same loop, and I want you to see that with your own eyes.

THE STOPS, ON A SECRET CIRCLE

First we need somewhere to deliver. We'll place twenty stops evenly around a circle, then shuffle the order we list them in — so the tidy "go round the rim" answer is hidden from the algorithm, but still secretly true.

```
import random, math
```

```
random.seed(8)

NUM_CITIES = 20

cities = []
for i in range(NUM_CITIES):
    angle = 2 * math.pi * i / NUM_CITIES
    cities.append((math.cos(angle), math.sin(angle)))
random.shuffle(cities)
```

cities is now a list of twenty (x, y) points sitting on a circle of radius 1, but jumbled up so that stop number 0 and stop number 1 are almost certainly not neighbours on the rim. The GA will have to discover the neighbours for itself.

We need to measure distance between two stops — plain straight-line distance — and the total length of a whole route:

```
def dist(a, b):
    ax, ay = cities[a]
    bx, by = cities[b]
    return math.hypot(ax - bx, ay - by)

def tour_length(tour):
    return sum(dist(tour[i], tour[(i + 1) % NUM_CITIES]) for i in
range(NUM_CITIES))
```

A tour is just a list of stop-numbers in the order we visit them, like [3, 17, 0, 8, ...]. `tour_length` walks along it adding up the hops, and the $(i + 1) \% \text{NUM_CITIES}$ is a tidy trick to make the last stop connect back to the first — the `%` wraps the final index around to 0, closing the loop so the driver gets home.

A ROUTE, AND A CROWD OF THEM

A single random route is just the numbers 0 to 19, shuffled:

```
def random_tour():
    tour = list(range(NUM_CITIES))
    random.shuffle(tour)
    return tour
```

And our population is a crowd of them — same as the pack of dogs, just routes instead:

```
pop = [random_tour() for _ in range(POP)]
```

Every route in that crowd is legal (each visits all twenty stops once) and almost all of them are bad — long, criss-crossing tangles. That's generation zero.

BREEDING ROUTES WITHOUT MAKING GIBBERISH

Now the careful part from the last chapter. We cannot breed routes the lazy way or we'll make children that visit a stop twice and skip another. We use order crossover: take a run of stops from mum in her order, then fill the gaps with the rest in dad's order.

```
def crossover(mum, dad):
    a, b = sorted(random.sample(range(NUM_CITIES), 2))
    child = [None] * NUM_CITIES
    child[a:b] = mum[a:b]           # a run of stops, in mum's
order
    fill = [c for c in dad if c not in child] # the rest, in dad's order
    j = 0
    for i in range(NUM_CITIES):
        if child[i] is None:
            child[i] = fill[j]
            j += 1
    return child
```

Walk through it slowly, because it's the one new idea. We pick two cut points *a* and *b*. We copy the slice of mum's route between them straight into the child — that's the run of stops in mum's order. Then *fill* is every stop not already in the child, listed in the order dad visits them. Finally we walk the child's empty slots (the Nones) and drop in the *fill* values one by one. Every stop lands exactly once: the ones from mum are already placed, and *fill* contains precisely the others, each once. A legal route, built from a piece of mum threaded with

dad's ordering.

Mutation is the easy one — just swap two stops, which can never break a route:

```
def mutate(tour, rate):
    tour = tour[:]
    for i in range(NUM_CITIES):
        if random.random() < rate:
            j = random.randrange(NUM_CITIES)
            tour[i], tour[j] = tour[j], tour[i] # swap two stops
    return tour
```

THE LOOP — LOOK HOW FAMILIAR IT IS

Here's the engine. Put Chapter 2's dog loop next to it. It is, line for line, the same shape: sort the population by how good each one is, keep the better half as parents, breed a new population from them, repeat.

POP = 300

GENERATIONS = 400

MUTATION = 0.02

pop = [random_tour() for _ in range(POP)]

```
for gen in range(GENERATIONS):
    pop.sort(key=tour_length)          # SELECTION: shortest
first this time
    parents = pop[:POP // 2]          # keep the better half
    children = [pop[0]]               # elitism: the best route
survives untouched
    while len(children) < POP:
        mum, dad = random.sample(parents, 2)
        child = crossover(mum, dad)    # HEREDITY: mix two
good routes...
        child = mutate(child, MUTATION) # ...with CHANGE
        children.append(child)
    pop = children
```

Two small differences from the dogs, both of which you've already met. First, `pop.sort(key=tour_length)` sorts shortest-first instead of fastest-first — because now smaller is better. Second, that `children = [pop[0]]` line is elitism: we copy the single best route into the next generation untouched, so a great route can never be lost to a bad roll of the mutation dice. (That was one of the "Breed your own" challenges back in Chapter 2 — here it earns its place, because when you're hunting for the best answer, you never want to accidentally throw it away.)

RUN IT

Here's the whole program, with a little printing so we can watch it work and check its homework against the rim length we can calculate but never tell it:

```
import random, math

random.seed(8)

NUM_CITIES = 20

cities = []
for i in range(NUM_CITIES):
    angle = 2 * math.pi * i / NUM_CITIES
    cities.append((math.cos(angle), math.sin(angle)))
random.shuffle(cities)

def dist(a, b):
    ax, ay = cities[a]
    bx, by = cities[b]
    return math.hypot(ax - bx, ay - by)

def tour_length(tour):
    return sum(dist(tour[i], tour[(i + 1) % NUM_CITIES]) for i in
range(NUM_CITIES))

def random_tour():
```

```
    tour = list(range(NUM_CITIES))
    random.shuffle(tour)
    return tour

def crossover(mum, dad):
    a, b = sorted(random.sample(range(NUM_CITIES), 2))
    child = [None] * NUM_CITIES
    child[a:b] = mum[a:b]
    fill = [c for c in dad if c not in child]
    j = 0
    for i in range(NUM_CITIES):
        if child[i] is None:
            child[i] = fill[j]
            j += 1
    return child

def mutate(tour, rate):
    tour = tour[:]
    for i in range(NUM_CITIES):
        if random.random() < rate:
            j = random.randrange(NUM_CITIES)
            tour[i], tour[j] = tour[j], tour[i]
    return tour

POP = 300
GENERATIONS = 400
```

```
MUTATION = 0.02
```

```
best_possible = NUM_CITIES - 2 * math.sin(math.pi / NUM_CITIES)
```

```
pop = [random_tour() for _ in range(POP)]
```

```
start_best = tour_length(min(pop, key=tour_length))
```

```
for gen in range(GENERATIONS):
```

```
    pop.sort(key=tour_length)
```

```
    if gen % 25 == 0 or gen == GENERATIONS - 1:
```

```
        print(f"Generation {gen:3d}: shortest tour  
{tour_length(pop[0]):6.2f}")
```

```
        parents = pop[:POP // 2]
```

```
        children = [pop[0]]
```

```
        while len(children) < POP:
```

```
            mum, dad = random.sample(parents, 2)
```

```
            child = crossover(mum, dad)
```

```
            child = mutate(child, MUTATION)
```

```
            children.append(child)
```

```
        pop = children
```

```
pop.sort(key=tour_length)
```

```
print()
```

```
print(f"Best random tour to start : {start_best:6.2f}")
```

```
print(f"Best tour the GA bred      : {tour_length(pop[0]):6.2f}")
```

```
print(f"The perfect rim tour      : {best_possible:6.2f}")
```

```
print(f"Possible tours it beat : {math.factorial(NUM_CITIES - 1)}  
// 2:.3g}")
```

Run it. You should see the tour get shorter and shorter, then lock onto a number and stay there:

```
Generation 0: shortest tour 19.93  
Generation 25: shortest tour 14.26  
Generation 50: shortest tour 9.89  
Generation 75: shortest tour 6.26  
Generation 100: shortest tour 6.26  
(...it holds at 6.26 all the way to the end...)  
Generation 399: shortest tour 6.26
```

```
Best random tour to start : 19.93  
Best tour the GA bred : 6.26  
The perfect rim tour : 6.26  
Possible tours it beat : 6.08e+16
```

Stop and take that in, because it's the whole book delivering on its promise.

WHAT JUST HAPPENED

The best route in the random starting crowd was 19.93 — a long criss-crossing tangle. By generation 75 the population had bred that

down to 6.26. And 6.26 is not just "good." Look at the last two lines: the perfect rim tour — the length we calculated in secret, the provable best route through all twenty stops — is 6.26 as well. The GA didn't find a good route. It found the optimal one. It rediscovered the rim it was never shown.

And it did that having searched a vanishing sliver of the possibilities.

There are 6.08×10^{16} — sixty quadrillion — possible twenty stops. A computer trying them one by one would still be grinding two years from now. Our little loop, running a crowd of three hundred routes for a few hundred generations — a few hundred thousand route-evaluations, done before the kettle boils — walked straight to the single best one. Not by being clever. Not by understanding circles, or distance, or geometry. It has no idea what a circle is. It only ever did four things: make a crowd, keep the shorter half, breed them with a careful shuffle, and mutate a little.

Population, variation, selection, heredity-with-change. The exact loop that made the dogs fast, pointed at a problem no human can solve by hand — and it solved it.

That is the thing this whole book was built to show you. You do not always have to know how to design the answer. If you can recognise a good answer when you see one — if you can score it — you can often breed the answer instead. The breeder never understood genetics. Your GA never understood geometry. Neither of them needed to.

TWO THINGS WORTH BREAKING

Turn off crossover. Make the child just a mutated copy of one parent: replace the `crossover(mum, dad)` line with `child = mum[:]`. Now the only source of change is swapping stops at random — no mixing of two good routes. Run it. It still improves (random swaps that happen to shorten the route get kept), but far more slowly, and it often gets stuck well short of 6.26, stranded on a decent-but-not-best route it can't swap its way out of. This is crossover earning its keep: mixing two good routes can make a leap that a lone mutation never could, combining the good first-half of one route with the good second-half of another.

Turn off elitism. Change `children = [pop[0]]` to `children = []`, so the best route is not protected. Watch the "shortest tour" number carefully now — every so often it will jump back up, get worse between generations, because the reigning best route got mangled by mutation and nothing saved a copy. The population still climbs overall, but it staggers instead of striding. That one-line safety net is why the good version's number never once went backwards.

BREED YOUR OWN

1. Real, scattered stops. Replace the circle with genuinely random stops: `cities = [(random.random(), random.random()) for _ in range(NUM_CITIES)]`. Now you don't know the perfect answer

either — welcome to the real problem. Run the GA a few times and watch it find short routes anyway. Draw them on paper. This is exactly the situation a real delivery company is in, and exactly why they use methods like this one.

2. A bigger round. Push NUM_CITIES to 50, or 100. The number of possible routes becomes an integer with more digits than the universe has atoms, and the GA... just keeps working, though you'll want more generations. Nothing about the loop changes. That indifference to scale is the whole point — the same four ingredients that bred a dog breed a hundred-stop route.

3. Give it a smarter mutation. Our mutation swaps two random stops. A famously better move for routes is called 2-opt: pick two points in the route and reverse the whole run between them, which is brilliant at untangling crossings. Add it as an occasional alternative to the swap and see if the GA reaches the best route in fewer generations. (Hybrids like this — evolution plus a clever local tweak — are how the strongest real-world solvers are built.)

You've now done the thing the book promised on page one: you've bred the answer to a problem nobody can design by hand. There's one last trick, and it's the strangest of all. We've bred numbers, and we've bred orderings. In the final chapter we're going to breed a program — an actual formula — and watch the machine rediscover a piece of mathematics from nothing but a column of numbers.

CHAPTER 11

BREEDING THE FORMULA ITSELF

We've bred three things now. We bred numbers — the dogs, whose genes were just values to push up. We bred an ordering — the delivery route, a permutation to rearrange. Each time the loop was the same; only the shape of the thing being bred changed.

There's one more shape, and it's the strangest and most wonderful of all. We're going to breed a program. An actual formula. And we're going to watch the machine rediscover a piece of mathematics it was never taught, from nothing but a column of numbers.

THE PROBLEM: GUESS THE FORMULA

Imagine someone hands you this table and asks a single question — what's the rule?

x	!	y
- 3	!	- 3 0
- 1	!	- 2
0	!	0
1	!	2
2	!	1 0
3	!	3 0

You don't get told the formula. You just get the numbers, and you have to find the machine that made them. This is not a toy exercise — it is, more or less, what a scientist does. Kepler stared at columns of numbers about planets and found the formula hiding under them. Every physical law started as somebody looking at data and asking "what's the rule?" The name for finding a formula that fits data is symbolic regression, and normally it takes a clever, patient human. We're going to make the loop do it.

But to breed formulas, we first have to answer a strange question: what does a formula look like, as a thing a program can hold, and copy, and mutate?

A FORMULA IS A LITTLE TREE

Here's the key idea. A formula like $x \times x + x$ isn't really a line of text — it's a tree. To work it out you first do the two multiplications and the... no. You do the innermost bits first, then combine. Written as a

tree, $x \ x + x$ looks like this:

```

+
 / \
 *   x
 / \
x   x

```

The $+$ at the top says "add my two branches." Its left branch is $x \ x$ (a tree with two x s under it); its right branch is just x . To get the answer you work up from the bottom: multiply x by x , then add x . Every formula, however big, is a tree like this: the branches are operations ($+$, $-$, $*$), and the leaves at the very bottom are either x or a plain number.

And a tree is a wonderful thing to breed, because it has parts you can swap. In code we'll write a tree as a little nested list — `['+', ['x', 'x'], 'x']` for the tree above — and give ourselves two helpers: one to evaluate* a tree (work out its answer for a given x) and one to turn it back into readable text.

```
import random, operator
```

```
random.seed(8)
```

```
OPS = {'+': operator.add, '-': operator.sub, '*': operator.mul}
```

```
def evaluate(tree, x):
    if tree == 'x':
        return x
    if isinstance(tree, int):
        return tree
    op, left, right = tree
    return OPS[op](evaluate(left, x), evaluate(right, x))

def to_str(tree):
    if tree == 'x':
        return 'x'
    if isinstance(tree, int):
        return str(tree)
    op, left, right = tree
    return f"({to_str(left)} {op} {to_str(right)})"
```

Both helpers do the same thing trees always ask of us: if the tree is a leaf ('x' or a number), handle it directly; otherwise it's [op, left, right], so deal with the left and right branches first and then combine them. A function that calls itself on its own branches like this is called recursion, and trees and recursion are made for each other.

MAKING, MIXING AND MUTATING TREES

To breed trees we need to make random ones, cross two of them, and mutate one — the same three abilities we've needed every single

time.

A random tree: sometimes a leaf, sometimes an operation with two smaller random trees underneath (we stop it growing forever with a depth limit).

```
def random_tree(depth=0):
    if depth >= 2 or (depth > 0 and random.random() < 0.3):
        return random.choice(['x', 'x', 1, 2])
    op = random.choice(list(OPS))
    return [op, random_tree(depth + 1), random_tree(depth + 1)]
```

Crossover is the beautiful one. To breed two formula-trees, we snip a random branch out of one and graft in a random branch from the other — exactly like grafting a cutting from one plant onto another. The child is mum's tree with one of its branches replaced by a branch of dad's:

```
def subtrees(tree, path=()):
    yield path, tree
    if isinstance(tree, list):
        for i in (1, 2):
            yield from subtrees(tree[i], path + (i,))

def clone(tree):
    return [tree[0], clone(tree[1]), clone(tree[2])] if isinstance(tree,
```

```
list) else tree
```

```
def replace_at(tree, path, new):  
    if not path:  
        return new  
    tree = [tree[0], tree[1], tree[2]]  
    tree[path[0]] = replace_at(tree[path[0]], path[1:], new)  
    return tree
```

```
def crossover(mum, dad):  
    mum = clone(mum)  
    spot = random.choice([p for p, _ in subtrees(mum)])  
    _, branch = random.choice([(p, t) for p, t in subtrees(dad)])  
    return replace_at(mum, spot, clone(branch))
```

Don't sweat every line of the plumbing — `subtrees` just lists every branch in a tree so we can pick one at random, and `replace_at` swaps a chosen branch for a new one. The idea is all that matters: crossover grafts a branch of one formula onto another. Mutation is simpler — with some chance, replace a random branch with a fresh random one:

```
def mutate(tree, rate=0.2):  
    if random.random() < rate:  
        return random_tree()  
    tree = clone(tree)  
    spots = [p for p, _ in subtrees(tree) if p]
```

```
if spots:
```

```
    tree = replace_at(tree, random.choice(spots), random_tree(1))
```

```
return tree
```

SELECTION: HOW WRONG IS THIS FORMULA?

Everything else has been leading here, and it's the easy part. We have our data — the table from the top of the chapter. A formula is good if its numbers come close to the data's numbers, and bad if they're far off. So we score a tree by its total error: run it on every x , compare to the true y , and add up how wrong it is (squared, so misses in either direction count as bad). Zero error means a perfect fit.

```
def hidden(x):
```

```
    return x * x * x + x      # the machine is NEVER told this -- only  
    the numbers
```

```
DATA = [(x, hidden(x)) for x in range(-3, 6)]
```

```
def error(tree):
```

```
    try:
```

```
        return sum((evaluate(tree, x) - y) ** 2 for x, y in DATA)
```

```
    except Exception:
```

```
        return 1e9
```

(The try is there because a random formula can occasionally do

something illegal; if it does, we just call it hopelessly wrong and move on. Evolution doesn't mind a few dead offspring.)

Notice we've quietly upgraded the hidden formula to $x \times x \times x + x$ — a cube plus x — to give the machine something meatier to discover. And now the loop. Say it with me: score everyone, keep the best half, breed them, repeat.

POP = 150

GENERATIONS = 60

```
pop = [random_tree() for _ in range(POP)]
for gen in range(GENERATIONS):
    pop.sort(key=error)
    best = pop[0]
    print(f"Generation {gen:2d}: error {error(best):7.0f}  f(x) =
{to_str(best)}")
    if error(best) == 0:
        break
    parents = pop[:POP // 2]
    children = [best]          # elitism again: keep the best formula
    while len(children) < POP:
        mum, dad = random.sample(parents, 2)
        children.append(mutate(crossover(mum, dad)))
    pop = children
```

Same loop. Always the same loop.

RUN IT

Put the whole thing together (all the functions above, then the loop, then the check below) and run it:

```
print()
best = min(pop, key=error)
print(f"The machine rediscovered: f(x) = {to_str(best)}")
print("Check it against the hidden formula xxx + x:")
for x in range(6):
    print(f"  x = {x}: formula says {evaluate(best, x):3d}, true
answer {hidden(x):3d}")
```

You should see this:

```
Generation 0: error    69  f(x) = (x (x x))
Generation 1: error    69  f(x) = (x (x x))
Generation 2: error     9  f(x) = ((x - 1) + ((x x) x))
Generation 3: error     9  f(x) = ((x - 1) + ((x x) x))
Generation 4: error     0  f(x) = (((x x) x) + (1 * x))
```

The machine rediscovered: $f(x) = (((x \ x) \ x) + (1 * x))$

Check it against the hidden formula $xxx + x$:

```
x = 0: formula says  0, true answer  0
```

x = 1: formula says 2, true answer 2
x = 2: formula says 10, true answer 10
x = 3: formula says 30, true answer 30
x = 4: formula says 68, true answer 68
x = 5: formula says 130, true answer 130

Read the four generations like a little story, because that's exactly what they are.

Generation 0: the best formula in the random crowd is $x(x+x)$ — that's x^3 . A decent guess! It's got the cube right, and it's wrong by 69. Generation 2: breeding has produced $(x-1) + (xxx)$ — it's found x^3 and noticed it needs to add something, though -1 isn't quite right. Error down to 9, warmer. Generation 4: $(xxx) + (1+x)$ — it worked out the missing piece is $+x$ (dressed up as $1+x$, which is the same thing). Error zero. The check underneath confirms it: every single value matches the hidden formula, including values it was never shown, like $x = 5!$ 130.

Sit with what that means. Nobody told the machine the formula. Nobody told it to try a cube. We gave it a column of numbers and four operations, and it bred a piece of mathematics — starting from a rough guess, discovering what was missing, and grafting its way to the exact truth. It rediscovered a formula the way a breeder makes a greyhound: not by understanding it, but by keeping what worked and mixing it, over and over, with nobody in charge.

This is a real and serious technique — it's called genetic programming — and it is genuinely used to rediscover physical laws from experimental data, and to design circuits and antennas and molecules so strange and clever that human engineers, looking at what the machine bred, sometimes cannot work out why they work. Some of them have been patented. You just built the toy version of a tool that invents things.

(A fair warning, so you're not puzzled: this program is more temperamental than the earlier ones. Formulas are a wilder space to search than routes or dogs, so genetic programming leans harder on luck. Change the seed and you'll sometimes see it take many more generations, or land on a correct-but-messy formula like $((x (x x)) + (x (2 - 1)))$ — which is just $x^3 + x$ wearing a silly hat, right but bloated. That untidiness — evolution finding a working answer that no human would ever write — is not a bug. It's the most honest thing in the whole book.)*

THE ONE IDEA, FOUR TIMES OVER

Stop and look back at what you've built, because you've come a very long way from a wolf at a fire.

You have written four programs. One bred a pack of dogs faster. One dealt Mendel's alleles and grew his 3-to-1 out of coin-flips. One passed DNA down a family and found a cousin in the shared scraps.

One bred the shortest route through sixty quadrillion, and then bred a formula out of thin air. Four programs, four completely different domains — animals, plants, molecules, mathematics — and here is the thing I most want you to carry out of this book:

They are all the same program.

Every one of them is the loop from Chapter 2. A population. Some variation. A rule for who gets to breed. And imperfect copying — crossover plus mutation — to make the next generation. Score everyone, keep the best, throw the rest, breed, repeat. That's it. That's the whole engine, and it does not care whether the things in the population are dogs or routes or formulas. Change what a "creature" is and what "good" means, and the same four ingredients will climb toward it.

Selection + variation + heredity + time = a population that improves itself, with nobody in charge.

That was the sentence from Chapter 1, the one I told you to tattoo on the inside of your eyelids. You've now watched it come true four times, in your own terminal, in numbers you generated yourself. It built the dog. It built the ratio. It built your family tree. It built the answer to a problem no human can solve by hand. It is, as far as anyone can tell, how you got here — how a few billion years of copying-with-mistakes, filtered by survival, built an animal clever

enough to write a for` loop and read a book about where it came from.

WHERE YOU GO FROM HERE

You didn't just learn about evolutionary algorithms. You built them, from nothing, and understood every line — because you met every idea first wearing fur, then leaves, then a double helix. That understanding is the real prize, and it opens a lot of doors:

- There are powerful, battle-tested libraries for this — for genetic algorithms, for genetic programming, for evolving neural networks. You'll now recognise every knob they expose, because you built each one by hand: population size, mutation rate, selection pressure, elitism, crossover. Nothing in them will be magic to you.
- The way modern neural networks learn is a close cousin of what you've done — a search that keeps small changes that reduce error and discards the rest. It's not identical, but the family resemblance is unmistakable, and you've now met the family.
- And the mindset travels furthest of all. Any time you face a problem where you can't design the answer but you can recognise a good one — a schedule, a layout, a strategy, a shape — you now know a secret: you don't have to be clever enough to solve it. You have to be clever enough to score it, and then let the loop breed the answer. Score everyone, keep the best, throw the rest.

This book started with a wolf that came a little closer to the fire than the others dared, and a promise that nobody needed to be in charge for something remarkable to get built. You've now felt that be true with your own hands, four times over. The oldest algorithm in the world runs in your machine now. Go and point it at something nobody's solved yet.

It has four legs, and it just went digital.