

SCIENCE COURSE

Genetics and Heredity Fundamentals

10 Chapters

Philip Osztromok

Generated with Claude

GENETICS AND HEREDITY FUNDAMENTALS

Contents

Chapter 1	Mendel and Inheritance	2
Chapter 2	Genes, DNA and Chromosomes	8
Chapter 3	From Gene to Protein	14
Chapter 4	Cell Division and Meiosis	21
Chapter 5	Two Genes at Once	29
Chapter 6	Beyond Dominant and Recessive	37
Chapter 7	Sex and Linked Traits	45
Chapter 8	Populations and Alleles	54
Chapter 9	Reading Genomes	63
Chapter 10	Genetics in Life	72

Genetics and Heredity Fundamentals

Course 1 · Chapter 1 · Mendel and Inheritance

Children resemble their parents, but not exactly, and a trait can skip a generation. For thousands of years breeders used that knowledge without understanding it. In the 1850s and 1860s a monk named Gregor Mendel, working in a monastery garden in Brno (then Brünn, in the Austrian Empire), counted thousands of pea plants and found that inheritance follows rules that can be written as simple arithmetic. This course follows that arithmetic into modern genetics. Because the rules are arithmetic, we will test them with short Python 3 programs.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run and the output shown is what it printed. The history and the pea counts are written from general knowledge, not checked against a source while writing; items marked (check) are the ones to verify first.

Mendel's Experiment

Garden peas suited the job. They grow quickly, have clearly different forms (round or wrinkled seeds, tall or short plants, purple or white flowers), and a gardener can control which plant fertilises which. Mendel studied seven such traits. He first made sure his plants were **true-breeding**: when crossed with themselves for generations, they always gave the same form. Then he crossed plants that differed in one trait, for example round-seeded with wrinkled-seeded.

The first generation of offspring (called F1) were all round. The wrinkled form seemed to have vanished. When he let the F1 plants fertilise themselves, the wrinkled form came back in the next generation (F2), in about one plant in four. For seed shape his counts are usually quoted as 5,474 round and 1,850 wrinkled (check).

GENERATION	HOW PRODUCED	WHAT MENDEL SAW (SEED SHAPE)
P (parents)	true-breeding round x true-breeding wrinkled	two clearly different forms
F1	offspring of the parents	all round
F2	F1 plants crossed with each other	round and wrinkled, about 3 : 1

Alleles, Dominant and Recessive

Mendel's explanation was that each plant carries **two** copies of the factor for a trait, one from each parent, and passes on only **one** of them to each offspring, chosen at random. Today we call the factors **genes** and the versions of a gene **alleles** (the word "gene" was coined later, and was not Mendel's; check the date, often given as 1909).

We write alleles as letters. Use `A` for the round allele and `a` for the wrinkled one. A plant's pair of letters is its **genotype**; the visible result is its **phenotype**. The round allele is **dominant**: one copy is enough to show the trait. The wrinkled allele is **recessive**: it shows only when both copies are `a`.

GENOTYPE	NAME	PHENOTYPE
AA	homozygous dominant	round
Aa	heterozygous	round
aa	homozygous recessive	wrinkled

This explains everything above. True-breeding round plants are `AA`, wrinkled are `aa`. Every F1 plant gets `A` from one parent and `a` from the other, so all are `Aa` and all look round. The wrinkled allele is hidden, not lost.

The Punnett Square as Code

A Punnett square (devised by Reginald Punnett in the early 1900s; check) lists every pairing of one allele from each parent. That is exactly what `itertools.product` does, so we can write it in a few lines. Each pairing is equally likely, so counting pairings gives the probabilities.

```
from itertools import product
from collections import Counter

def cross(p1, p2):
    """Count offspring genotypes for a one-gene cross, e.g. 'Aa' x 'Aa'."""
    kids = Counter()
    for a, b in product(p1, p2):
        kids["".join(sorted(a + b))] += 1
```

```
    return kids

c = cross("Aa", "Aa")
print(dict(c))
total = sum(c.values())
for g in ("AA", "Aa", "aa"):
    print(g, c[g], "/", total)
```

Output:

```
{'AA': 1, 'Aa': 2, 'aa': 1}
AA 1 / 4
Aa 2 / 4
aa 1 / 4
```

The sorting in `"".join(sorted(a + b))` makes `aA` and `Aa` the same genotype. Three of the four squares contain at least one `A`, so three-quarters of the offspring look round and one quarter look wrinkled. That is the 3 : 1 ratio.

```
def phenotype_ratio(counts, dominant="A"):
    dom = sum(n for g, n in counts.items() if dominant in g)
    return dom, sum(counts.values()) - dom

print(phenotype_ratio(cross("Aa", "Aa")))   # (3, 1)
print(dict(cross("Aa", "aa")), phenotype_ratio(cross("Aa", "aa")))
```

```
(3, 1)
{'Aa': 2, 'aa': 2} (2, 2)
```

The second line shows a heterozygous plant crossed with a wrinkled one: half the offspring are round and half wrinkled. Breeders use this to find out whether a round plant is `AA` or `Aa`.

Chance and Sample Size

A 3 : 1 ratio is a *probability*, not a promise. Each offspring independently has a three-in-four chance of looking round. Simulating it shows how counts behave:

```
import random

def f2_simulation(n, seed):
    rng = random.Random(seed)
    dom = 0
    for _ in range(n):
        g = rng.choice("Aa") + rng.choice("Aa")
        if "A" in g:
            dom += 1
    return dom, n - dom

for n in (10, 100, 10000):
    d, r = f2_simulation(n, 1)
    print(n, d, r, round(d / n, 3))
```

```
10 7 3 0.7
100 78 22 0.78
10000 7515 2485 0.751
```

With 10 offspring the share of round plants was 0.7; with 10,000 it was 0.751, close to the expected 0.75. Using a fixed seed makes the run repeatable. Mendel's habit of counting thousands of plants is a large part of why his ratios were so clear.

> **RATIOS ARE NOT GUARANTEES FOR ONE FAMILY**
>
> If two heterozygous parents have four children, that does not mean exactly one will show the recessive trait. Each child is an independent draw with a one-in-four chance. A family of four can easily have none, or two.

Challenges

EXERCISE 1

Using `cross`, work out the offspring genotypes and the dominant : recessive appearance counts for `Aa x aa`, `AA x aa` and `AA x Aa`. Predict each result on paper first.

View solution

EXERCISE 2

Taking Mendel's seed-shape counts as 5,474 round and 1,850 wrinkled, compute the total, the round : wrinkled ratio, the counts expected under an exact 3 : 1 split, and the difference.

View solution

EXERCISE 3

Write `dominant_fraction(n, seed)` that simulates n offspring of an `Aa x Aa` cross. For n = 8, 80, 800, 8000 and 80000, run five seeds each and report the largest miss from 0.75. Describe the pattern.

View solution

QUICK REFERENCE

- **Gene:** a unit of inheritance; **allele:** a version of a gene
- **Genotype:** the alleles carried; **phenotype:** the visible trait
- **Dominant** shows with one copy; **recessive** needs two
- **Aa x Aa** gives 1 AA : 2 Aa : 1 aa, so 3 dominant : 1 recessive
- **Each offspring** is an independent draw; ratios emerge only in large samples
- **Check:** the pea counts, the date "gene" was coined, the Punnett date

COMING NEXT

Genetics and Heredity Fundamentals 2 looks at what Mendel's factors physically are: DNA, genes and chromosomes.

Genetics and Heredity Fundamentals

Course 1 · Chapter 2 · Genes, DNA and Chromosomes

Genetics and Heredity Fundamentals 1 treated Mendel's "factors" as letters: `A` and `a`. This chapter asks what those letters physically are. The short answer is that an allele is a stretch of **DNA**, DNA is packaged into **chromosomes**, and the whole collection in a cell is its **genome**. As before, the code is Python 3, run before it is shown, and facts written from general knowledge are marked (check).

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> All programs here were run and the output shown is what they printed. The biology (numbers of chromosomes, genome sizes, dates) is from general knowledge and not checked against a source while writing; items marked (check) are the ones to verify first.

What DNA Is

DNA (deoxyribonucleic acid) is a long molecule built from a chain of small units called **nucleotides**. Each nucleotide has a sugar, a phosphate group and one of four **bases**: adenine (`A`), thymine (`T`), guanine (`G`) and cytosine (`C`). The sugar and phosphate form a backbone; the bases stick out sideways. The order of the bases is the information.

That makes DNA convenient for programmers: to a good approximation a piece of DNA is just a string over a four-letter alphabet.

The Double Helix and Base Pairing

DNA normally exists as two strands wound around each other in a double helix, a structure worked out in 1953 by James Watson and Francis Crick, building on X-ray data from Rosalind Franklin and Maurice Wilkins (check the details of who contributed what). The strands are held together by bonds between bases, and the pairing is strict:

BASE	PAIRS WITH	BONDS BETWEEN THEM (CHECK)
A (adenine)	T (thymine)	2 hydrogen bonds

BASE	PAIRS WITH	BONDS BETWEEN THEM (CHECK)
G (guanine)	C (cytosine)	3 hydrogen bonds

Because of this rule, one strand completely determines the other. The two strands also run in opposite directions: each strand has a start (called the 5' end) and a finish (the 3' end), and the partner strand is read the other way round. So the partner of `ATGC`, written in the usual 5' to 3' direction, is `GCAT`, the **reverse complement**.

```
COMP = str.maketrans("ACGT", "TGCA")

def complement(s):
    return s.translate(COMP)

def reverse_complement(s):
    return complement(s)[::-1]

strand = "ATGCGTACCTGA"
print(strand)
print(complement(strand))
print(reverse_complement(strand))
print(reverse_complement(reverse_complement(strand)) == strand)
```

Output:

```
ATGCGTACCTGA
TACGCATGGACT
TCAGGTACGCAT
True
```

`str.translate` swaps every letter at once using the table from `str.maketrans`. The plain complement lines up letter for letter under the original; the reverse complement is the partner strand as it would be written in its own 5' to 3' direction. Taking the reverse complement twice gives back the original, which is the check on the last line.

Counting Bases and GC Content

Pairing has a consequence: in double-stranded DNA the number of `A` bases equals the number of `T` bases, and `G` equals `C`. This was noticed by Erwin Chargaff in the late 1940s (check), before the structure was known, and is called Chargaff's rule. It holds for the two strands taken together, not for a single strand.

The share of bases that are `G` or `C` is the **GC content**. It varies between species and between regions of a genome, and because a G–C pair has three bonds against two for A–T, GC-rich DNA is a little harder to pull apart (check).

```
from collections import Counter

def gc_content(s):
    return (s.count("G") + s.count("C")) / len(s)

seq = "ATGCGTACCTGAGGCCATTA"
print(len(seq), dict(Counter(seq)))
print(round(gc_content(seq), 3))
print(round(gc_content("ATATATATAT"), 3), round(gc_content("GCGCGCGCGC"), 3))

other = "".join({"A": "T", "T": "A", "G": "C", "C": "G"}[b] for b in seq)
both = Counter(seq + other)
print(both["A"] == both["T"], both["G"] == both["C"])
```

```
20 {'A': 5, 'T': 5, 'G': 5, 'C': 5}
0.5
0.0 1.0
True True
```

This particular strand happens to have equal numbers of all four bases, which is a coincidence of the example, so the final line is only a real test because we build the partner strand and count both together. Try it on a lopsided strand such as `AAAAGC`: the single strand fails the A = T test, but the pair of strands passes.

Genes, Chromosomes and the Genome

A **gene** is a stretch of DNA that carries the instructions for making a product, usually a protein (chapter 3 follows how). The gene is where an allele lives: two alleles of the same gene are two versions of the same stretch of DNA, differing in some of their bases. Genes are not evenly packed; in humans most of the DNA lies between genes or inside them in non-coding regions, and the exact share that does something useful is debated (check).

A cell's DNA is divided among several long pieces, the **chromosomes**, each one a single enormous DNA molecule wrapped around proteins so that it fits inside the cell. The complete set of DNA in a cell is its **genome**. A chromosome holds many genes, and each gene sits at a fixed position on a particular chromosome, called its **locus**.

Mendel's rule that every plant has two copies of each factor now has a physical meaning: most cells of a plant or animal are **diploid**, holding chromosomes in matching pairs, one of each pair from each parent. The sex cells (eggs and pollen or sperm) are **haploid**, with one of each pair. The numbers differ by species:

```
# (name, diploid chromosome number); values from general knowledge
organisms = [("pea", 14), ("fruit fly", 8), ("human", 46), ("dog", 78)]
for name, n2 in organisms:
    print(name, n2, "chromosomes =", n2 // 2, "pairs; gamete carries", n2 // 2)
```

```
pea 14 chromosomes = 7 pairs; gamete carries 7
fruit fly 8 chromosomes = 4 pairs; gamete carries 4
human 46 chromosomes = 23 pairs; gamete carries 23
dog 78 chromosomes = 39 pairs; gamete carries 39
```

All four numbers are from general knowledge (check). The count says nothing about complexity: it is a fact about how the DNA happens to be packaged. A pea has seven pairs, and Mendel studied seven traits; this is a well-known coincidence, and the question of whether his traits sat on different chromosomes comes back in chapter 5 (check the details there).

GENE, ALLELE, CHROMOSOME: KEEP THEM APART

A chromosome is a physical package of DNA. A gene is a region of it. An allele is one version of that region. A diploid cell has two copies of each chromosome (apart from the sex chromosomes, chapter 7), so two alleles of each gene, which may be the same or different.

How Big Is a Genome?

The human haploid genome is usually quoted as about 3.1 billion base pairs (check). If each base needs two bits (four choices), a quick calculation gives the raw information content:

```
bases = 3_100_000_000
print(bases * 2, "bases in a diploid cell")
bits = bases * 2
print(round(bits / 8 / 1e9, 3), "GB at 2 bits per base (haploid)")
print(round(bases * 0.34 / 1e9, 2), "metres of DNA if 0.34 nm per base (haploid)")
```

```
6200000000 bases in a diploid cell
0.775 GB at 2 bits per base (haploid)
1.05 metres of DNA if 0.34 nm per base (haploid)
```

So a haploid human genome would fit in under a gigabyte, and the DNA of one set of chromosomes, laid end to end, is about a metre long (using 0.34 nanometres per base, check). All of that sits inside a cell nucleus a few micrometres across, which is why DNA is so tightly wound and packaged.

Challenges

EXERCISE 1

Write `reverse_complement(s)`. For `ATGC`, `GGGATTT` and `AAAACCCGGT`, print each sequence, its reverse complement, and whether applying the function twice returns the original. Then test whether `GAATTC`

and `GGATCC` equal their own reverse complement.

View solution

EXERCISE 2

Given four sequences, print them in order of increasing GC content with the value to two decimals, and then name the one with the highest GC content.

View solution

EXERCISE 3

Write `dna_bytes(bases, bits_per_base)` and use it to estimate, in megabytes, the raw storage for the haploid genomes of three organisms, then the human figure in gigabytes if each base were stored as one 8-bit character instead.

View solution

QUICK REFERENCE

- **DNA:** a chain of nucleotides with bases A, T, G, C; the base order is the information
- **Pairing:** A with T, G with C; one strand fixes the other
- **Reverse complement:** the partner strand written 5' to 3'
- **GC content:** the fraction of bases that are G or C
- **Gene:** a region of DNA; **allele:** a version of it; **chromosome:** a whole DNA package
- **Diploid** cells have pairs of chromosomes; **haploid** sex cells have one of each
- **Check:** the chromosome numbers, genome size, bond counts, and who contributed what to 1953

COMING NEXT

Genetics and Heredity Fundamentals 3 follows a gene's instructions from DNA to protein, and what happens when a base changes.

Genetics and Heredity Fundamentals

Course 1 · Chapter 3 · From Gene to Protein

Genetics and Heredity Fundamentals 2 described a gene as a stretch of DNA that carries instructions. This chapter follows those instructions to the product: a **protein**, a chain of smaller units called amino acids, whose shape lets it do a job in the cell. The route has two steps, and both are a kind of text processing, which makes them easy to write as Python 3 programs.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run and the output shown is what it printed. The biology is written from general knowledge; items marked (check) are the ones to verify first.

Two Steps: Transcription and Translation

The DNA stays put (in plants and animals, inside the nucleus). To use a gene, the cell first makes a working copy in a related molecule, **RNA**. This copying is **transcription**. RNA differs from DNA in three ways that matter here: it is usually a single strand, its sugar is different, and it uses the base **U** (uracil) where DNA uses **T**. The copy is built by the pairing rules of chapter 2, so it matches the coding strand of the gene letter for letter, with U in place of T.

The RNA message is then read by cell machinery called the ribosome, which builds the protein one amino acid at a time. This reading is **translation**. The message is read in blocks of three bases, called **codons**, and each codon names one amino acid.

STEP	FROM	TO	UNIT
Transcription	DNA	RNA	one base at a time (T becomes U)
Translation	RNA	protein	one codon (three bases) per amino acid

Real cells add steps that this chapter skips. In plants and animals the first RNA copy is often trimmed before use, with some stretches removed (check), and the beginning and end of a message have extra features. We will write the central idea and leave those details aside.

The Codon Table as Code

Three bases, four choices each, gives 4 x 4 x 4 = 64 codons. Sixty-one of them name one of the 20 standard amino acids; three are **stop** codons that end the message (UAA, UAG and UGA; check). One codon, AUG, names the amino acid methionine and also usually marks where reading begins (check).

Rather than typing 64 entries, we can write the table as one string of 64 letters in a fixed order and build the dictionary with loops. The order is T, C, A, G for each of the three positions, and each letter is the one-letter code for an amino acid, with `*` meaning stop. We work with the DNA letters (T), since our genes are written as DNA. The one-letter amino acid codes and the table itself are written from general knowledge (check), which is why the program also tests the table against facts we know.

```
B = "TCAG"
AA = "FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG"
CODON = {a + b + c: AA[16 * i + 4 * j + k]
         for i, a in enumerate(B) for j, b in enumerate(B) for k, c in enumerate(B)}

def transcribe(dna):
    return dna.replace("T", "U")

def translate(dna):
    protein = []
    for i in range(0, len(dna) - 2, 3):
        aa = CODON[dna[i:i + 3]]
        if aa == "*":
            break
        protein.append(aa)
    return "".join(protein)

gene = "ATGGCCTTGAAATGGTAA"
print(transcribe(gene))
print(translate(gene))
print(len(CODON), list(CODON.values()).count("*"), len(set(CODON.values()) - {"*"}))
```

```
AUGGCCUUGAAAUGGUAA
```

```
MALKW
64 3 20
```

The gene is six codons: ATG GCC TTG AAA TGG TAA. The first five name M, A, L, K and W; the sixth is a stop, so `translate` halts and the protein is `MALKW`, five amino acids long. The last line confirms the counts above: 64 codons, 3 of them stops, 20 amino acids. (The sequence itself is invented for the example, not a real gene.)

One Code, Many Spellings

There are 61 codons but only 20 amino acids, so most amino acids have more than one codon. The code is said to be **degenerate** (or redundant): several codons, called **synonymous**, name the same amino acid. This is not ambiguity, because each codon still names exactly one amino acid. Let us count how many codons each amino acid has.

```
from collections import Counter

sizes = Counter(CODON.values())
print(sizes["L"], sizes["R"], sizes["S"], sizes["M"], sizes["W"])
print(Counter(sizes[a] for a in sizes if a != "*"))
print(CODON["CTT"], CODON["CTC"], CODON["CTA"], CODON["CTG"])
```

```
6 6 6 1 1
Counter({2: 9, 4: 5, 6: 3, 1: 2, 3: 1})
L L L L
```

Three amino acids (L, R, S) have six codons each, and two (M and W) have only one. Reading the second line as "how many amino acids have this many codons": nine have two, five have four, three have six, two have one, and one (I) has three. The third line shows a common pattern: the four codons `CTN` all name L, so a change in the *third* base often changes nothing. That pattern is why the next section matters.

Mutations

A **mutation** is a change in the DNA sequence. Most that matter for a protein fall into a few kinds, and they can be told apart by comparing the protein before and after:

KIND	WHAT CHANGES IN THE DNA	EFFECT ON THE PROTEIN
Silent	one base swapped	none: the new codon names the same amino acid
Missense	one base swapped	one amino acid replaced by another
Nonsense	one base swapped	a stop codon appears early; the protein is cut short
Frameshift	bases inserted or deleted, not a multiple of three	every codon after the change is read differently
In-frame indel	a multiple of three bases inserted or deleted	amino acids added or lost; the rest is unchanged

The one-base swaps are called **point mutations** (or substitutions). We can classify a change automatically by translating both versions and comparing them.

```
def classify(old, new):
    if len(old) != len(new):
        return "frameshift" if (len(new) - len(old)) % 3 else "in-frame indel"
    a, b = translate(old), translate(new)
    if a == b:
        return "silent"
    return "nonsense" if len(b) < len(a) else "missense"

cases = {
    "original":      gene,
    "G>T at 9":      gene[:8] + "T" + gene[9:],
    "C>G at 6":      gene[:5] + "G" + gene[6:],
    "G>A at 14":     gene[:13] + "A" + gene[14:],
    "delete base 5": gene[:4] + gene[5:],
    "insert GGG":    gene[:6] + "GGG" + gene[6:],
}
for label, seq in cases.items():
```

```
    print(f"{label:14} {translate(seq):8} {classify(gene, seq) if seq != gene else ''}")
```

```
original       MALKW
G>T at 9       MAFKW    missense
C>G at 6       MALKW    silent
G>A at 14      MALK     nonsense
delete base 5  MA       frameshift
insert GGG     MAGLKW   in-frame indel
```

Positions are counted from 1. Changing the ninth base turns TTG (L) into TTT (F): one amino acid replaced. Changing the sixth turns GCC into GCG, which still names A: silent. Changing the fourteenth turns TGG (W) into TAG, a stop, so the protein loses its last amino acid. Deleting one base shifts the reading frame, so the later codons are read in a different grouping; here the new message hits a stop early and the protein is just `MA`. Inserting three bases adds exactly one amino acid (G) and leaves the rest alone.

SILENT DOES NOT ALWAYS MEAN HARMLESS

"Silent" only means the amino acid is unchanged. Some silent changes can still affect how a gene is used (check), and a "missense" change can range from harmless to serious depending on which amino acid is swapped and where. Our classifier counts changes; it does not judge their effect.

A Famous Single-Letter Change

The textbook example of a missense mutation is sickle cell disease, in which one base change in the gene for part of haemoglobin swaps the sixth amino acid from glutamic acid (E) to valine (V) (check). Below is the beginning of the normal sequence, in the form usually quoted (check, and note that the leading methionine is often removed in the finished protein), together with a changed version. We also try some other changes to the same stretch.

```
orig = "GTGCACCTGACTCCTGAGGAGAAG"
variants = [("substitution", orig[:15] + "GTG" + orig[18:]),
            ("delete 1", orig[:15] + orig[16:]),
```

```
            ("insert 3", orig[:15] + "AAA" + orig[15:]),
            ("stop", orig[:15] + "TAG" + orig[18:])]
for label, mut in variants:
    print(label, translate(orig), "->", translate(mut), classify(orig, mut))
```

```
substitution VHLTPEEK -> VHLTPVEK missense
delete 1 VHLTPEEK -> VHLTPRR frameshift
insert 3 VHLTPEEK -> VHLTPKEEK in-frame indel
stop VHLTPEEK -> VHLTP nonsense
```

The first variant changes one letter, GAG to GTG, and one amino acid: the sixth letter of the protein goes from E to V, the change described above. The frameshift garbles everything after the deletion, the three-base insertion adds one amino acid, and the stop variant ends the protein after five. Note how little the DNA changed in the first case: one base in 24, yet a different protein.

Challenges

EXERCISE 1

Write `find_orfs(dna)`, which returns a list of `(start, protein)` pairs for every `ATG` that is followed, in the same frame, by a stop codon (use the codon table from this chapter; positions count from 0). Test it on `ATGAAATTTGGGTAA`, `ATGTGGTGA` and `GGATGCCCTAGATGAAATAG`.

View solution

EXERCISE 2

For the gene `ATGGCCTTGAAATGGTAA`, change each of the first 15 bases (the five codons before the stop) to each of the three other bases. Classify all 45 resulting mutations and count how many are silent, missense and nonsense.

View solution

EXERCISE 3

Write `report(orig, mut)` that prints the original protein, the mutated protein and the mutation kind, and also the position (counting from 1) of the first amino acid that differs, or `none` if the proteins are identical. Run it on four variants of the sequence `GTGCACCTGACTCCTGAGGAGAAG`.

View solution

QUICK REFERENCE

- **Transcription:** DNA to RNA, with U in place of T
- **Translation:** RNA read in codons of three bases, each naming one amino acid
- **Codon table:** 64 codons; 61 name 20 amino acids, 3 are stops; the code is redundant but never ambiguous
- **Silent / missense / nonsense:** same amino acid / different amino acid / early stop
- **Frameshift:** insertion or deletion not a multiple of three; **in-frame indel:** a multiple of three
- **Check:** the codon table string, the stop codons, the sickle cell details, skipped steps such as trimming of RNA

COMING NEXT

Genetics and Heredity Fundamentals 4 looks at how cells divide and how sex cells are made, and why brothers and sisters differ.

Genetics and Heredity Fundamentals

Course 1 · Chapter 4 · Cell Division and Meiosis

Genetics and Heredity Fundamentals 3 followed a gene to its protein. This chapter asks a different question: how does DNA get from one cell to the next, and from parent to child? The answer is two kinds of cell division. One makes exact copies; the other shuffles and halves the DNA to make sex cells. The shuffling is the reason Genetics and Heredity Fundamentals 1 could talk about chance at all, and it is why brothers and sisters are alike but not identical. All the programs here are Python 3.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run and the output shown is what it printed. The biology is written from general knowledge; items marked (check) are the ones to verify first. Where a program uses random numbers it fixes the seed, so the same output appears each time.

Copying DNA First

Before a cell divides it copies its DNA. The pairing rules from Genetics and Heredity Fundamentals 2 make this easy: if the two strands of the double helix are pulled apart, each one can serve as a template for building a new partner. The result is two double helices, each made of one old strand and one newly built strand. This is called **semi-conservative** copying (check). In code, the new strand is just the complement of the old one.

```
PAIR = {"A": "T", "T": "A", "C": "G", "G": "C"}

def replicate(strand):
    return "".join(PAIR[b] for b in strand)

old = ("ATGCCGTA", "TACGGCAT")
new1 = (old[0], replicate(old[0]))
new2 = (replicate(old[1]), old[1])
print(new1)
print(new2)
print(new1 == old, new2 == old)
```

```
('ATGCCGTA', 'TACGGCAT')
('ATGCCGTA', 'TACGGCAT')
True True
```

Each new double helix matches the original exactly. (Real copying makes rare mistakes, and cells have proofreading and repair that catch most of them; check. Those rare mistakes are the source of the mutations in Genetics and Heredity Fundamentals 3.) After copying, each chromosome consists of two identical halves joined together, called **sister chromatids** (check).

Mitosis: Two Identical Cells

Mitosis is the division used for growth and repair. One cell copies its DNA and then splits into two **daughter cells**, each receiving a full set of chromosomes. Each daughter cell has the same chromosome number as the parent cell and the same DNA. A human body cell has 46 chromosomes in 23 pairs, one of each pair from each parent (check), and after mitosis both daughters have 46.

A cell with two sets of chromosomes is called **diploid**, written **2n**. A cell with one set is **haploid**, written **n**. For humans n = 23 and 2n = 46 (check). We can model a cell as a list of chromosome pairs, with `m` marking the one from the mother and `p` the one from the father.

Meiosis: Halving the Set

Meiosis makes the sex cells, or **gametes**: sperm and egg cells in animals. If gametes carried the full 2n set, a child would get 2n from each parent, 4n in all, and the number would double every generation. Instead meiosis cuts the number in half, so a gamete is haploid, and fertilisation joins two gametes to restore 2n.

The process has two rounds of division after a single copying of the DNA (check). In the first round the **homologous pairs** (the maternal and paternal copy of each chromosome) line up together and are separated, so each daughter gets one chromosome of each pair. In the second round the sister chromatids are separated. One starting cell ends up as four gametes (in humans, check that the egg-making version divides unequally and gives one working egg rather

than four). The important point for us is the first round: for each pair, which of the two chromosomes goes into a given gamete is a matter of chance.

	MITOSIS	MEIOSIS
Purpose	growth, repair	making gametes
DNA copying rounds	one	one
Divisions	one	two
Cells produced	two	four
Chromosome number	stays 2n (46)	halves to n (23)
Daughter cells	identical to the parent	different from the parent and from each other

```
import random

def mitosis(cell):
    return [list(cell), list(cell)]

def meiosis(cell, rng):
    return [[rng.choice(pair) for pair in cell] for _ in range(2)]

cell = [("m1", "p1"), ("m2", "p2"), ("m3", "p3")]
rng = random.Random(4)
a, b = mitosis(cell)
print(len(cell), len(a), len(b), a == b == cell)
for gamete in meiosis(cell, rng):
    print(gamete, len(gamete))
```

```
3 3 3 True
['m1', 'p2', 'm3'] 3
['p1', 'p2', 'm3'] 3
```

Mitosis returns two copies equal to the parent (the pairs are kept; the lists hold three entries each). Meiosis returns gametes of length 3, one chromosome per pair, and the two gametes differ: here they disagree on the first pair. This is a simplified model: it makes only two gametes where real meiosis makes four, and it ignores crossing over, which comes next.

Counting the Possible Gametes

Because each pair is sorted independently, the number of different gametes doubles with every extra pair. With 3 pairs there are 2 x 2 x 2 = 8 possible gametes; with the human 23 pairs there are 2^{23}. This shuffling of whole chromosomes is called **independent assortment**, and Genetics and Heredity Fundamentals 5 returns to it for two genes at once.

```
from itertools import product

pairs = [("m1", "p1"), ("m2", "p2"), ("m3", "p3")]
combos = list(product(*pairs))
print(len(combos), combos[0], combos[-1])
print(2 ** 3, 2 ** 23, (2 ** 23) ** 2)
```

```
8 ('m1', 'm2', 'm3') ('p1', 'p2', 'p3')
8 8388608 70368744177664
```

A single person can therefore make 8,388,608 different gametes by assortment alone. Two parents can combine any of one person's gametes with any of the other's: 8,388,608 squared is 70,368,744,177,664, about 70 trillion different children, before counting crossing over. (The number is an upper bound on what chance allows; it treats every chromosome as distinct and ignores real-world constraints. check)

Crossing Over

During the first round of meiosis, the maternal and paternal copies of a chromosome lie alongside each other and can swap matching stretches. This is called **crossing over** (or recombination). After a swap, a chromosome is a mosaic: part from the grandmother's line and part from the grandfather's. In human meiosis there are typically a few crossovers per chromosome pair (check), so the gametes' chromosomes are mixtures, not copies of either original.

We can model two chromosomes as strings and swap the tails at a chosen point. Here the mother's copy is all `A` and the father's all `b`, so you can see exactly what came from where.

```
def cross_over(mat, pat, point):
    return mat[:point] + pat[point:], pat[:point] + mat[point:]

mat, pat = "AAAAAAAAAA", "bbbbbbbbbb"
print(cross_over(mat, pat, 4))
print(cross_over(mat, pat, 0))
print(cross_over(mat, pat, 10))
```

```
('AAAAbbbbbb', 'bbbbAAAAAA')
('bbbbbbbbbb', 'AAAAAAAAAA')
('AAAAAAAAAA', 'bbbbbbbbbb')
```

A swap at position 4 gives two new chromosomes, each with the first four letters from one parent line and the last six from the other. Swapping at position 0 or 10 exchanges everything or nothing, which is the same as no crossing over at all (the two results are just the originals, possibly swapped). Crossing over does not change what is on a chromosome overall; it changes which pieces travel together. That is why two genes close together on one chromosome are usually inherited as a pair, and why genes far apart behave almost as if on different chromosomes (check). Genetics and Heredity Fundamentals 5 covers this under the name linkage.

Why Siblings Differ

Every child of the same two parents gets one gamete from each, and each gamete is one random draw from millions of possibilities. We can put numbers on this. Write a gamete as a string of 23 letters, `m` or `p`, saying which chromosome of each pair it contains, and count how many positions two gametes share. (This ignores crossing over, so the real figures are fuzzier than the model; check.)

```
def gamete(rng, n=23):
    return "".join(rng.choice("mp") for _ in range(n))

def shared(x, y):
    return sum(i == j for i, j in zip(x, y))

rng = random.Random(1)
g1, g2 = gamete(rng), gamete(rng)
print(g1)
print(g2)
print(shared(g1, g2))
trials = 20000
mean = sum(shared(gamete(rng), gamete(rng)) for _ in range(trials)) / trials
print(round(mean, 1), 23 / 2)
```

```
mmpmppppmmpmppmppmmpmmm
mpmpmmppmpmmppmpmmpmppm
11
11.5 11.5
```

Two gametes from the same parent share 11 of 23 positions in this example, and on average they share half, 11.5 of 23, as chance predicts. So two children of the same parents get, from each parent, a set of chromosomes that matches the sibling's about half the time at each chromosome. The chance that two gametes from one person are identical is 1 in 2^{23}, about one in 8.4 million, and for two children to match from both parents it is the square of that. So, apart from identical twins (who come from a single fertilised egg that splits; check), siblings are very unlikely to carry the same combination.

> **"HALF" IS AN AVERAGE, NOT A RULE**

Siblings are often said to share half their DNA. That is the average over many pairs of siblings; a particular pair can be above or below it, and crossing over spreads the figure further (check). Full siblings can also look much more alike or much less alike than the average suggests, because appearance depends on many genes and on environment as well.

Challenges

EXERCISE 1

For n = 1 to 5 pairs of chromosomes (named `m1`, `p1`, `m2`, `p2` and so on), list every possible gamete with `itertools.product`, count the distinct ones, and check that the count equals 2^n.

View solution

EXERCISE 2

Write `cross_over_many(mat, pat, points)` that swaps tails at each point in the list, one after another, and returns the two resulting chromosomes. Run it on `"AAAAAAAAAA"` and `"bbbbbbbbbb"` with points `[4]`, `[3, 7]`, `[2, 5, 8]` and an empty list, and check at every position that the two results together still hold one `A` and one `b`.

View solution

EXERCISE 3

Model parents who each have only 3 chromosome pairs. Each child gets one gamete from each parent. Simulate 100,000 pairs of children (use `random.Random(11)`) and find the fraction of pairs in which both children carry exactly the same chromosomes. Compare it with the exact chance, $(1/2^3)^2$.

View solution

QUICK REFERENCE

- **Mitosis:** one cell becomes two identical cells; chromosome number stays 2n
- **Meiosis:** two divisions after one copying; makes gametes with n chromosomes

- **Diploid / haploid:** 2n (two sets, 46 in humans) / n (one set, 23)
- **Independent assortment:** each pair is sorted separately, giving 2^n gametes
- **Crossing over:** matching stretches of paired chromosomes are swapped, creating mosaics
- **Check:** the counts of chromosomes, the number of crossovers, the egg-making details, the statement about siblings sharing half

COMING NEXT

Genetics and Heredity Fundamentals 5 follows two genes at once: when they are inherited independently, and when crossing over cannot separate them.

Genetics and Heredity Fundamentals

Course 1 · Chapter 5 · Two Genes at Once

Genetics and Heredity Fundamentals 1 followed one gene at a time. Real organisms have thousands, and they are inherited together. This chapter asks what happens when we follow two genes in the same cross. There are two possible answers. If the genes sit on different chromosomes, they are shuffled independently, as Genetics and Heredity Fundamentals 4 described. If they sit close together on the same chromosome, they tend to travel as a block. All the programs here are Python 3.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run and the output shown is what it printed. The biology is written from general knowledge; items marked (check) are the ones to verify first. Where a program uses random numbers it fixes the seed, so the same output appears each time.

Independent Assortment

Take two genes. Gene 1 has alleles `A` (dominant) and `a`; gene 2 has alleles `B` (dominant) and `b`. A parent of genotype `AaBb` is heterozygous for both, and is called a **dihybrid** (check). If the two genes are on different chromosomes, the choice of `A` or `a` in a gamete has no effect on the choice of `B` or `b`. This is the principle of **independent assortment**, Mendel's second law (check). So the parent makes four kinds of gamete, all equally likely.

```
from itertools import product

print(list(product("Aa", "Bb")))
```

```
[('A', 'B'), ('A', 'b'), ('a', 'B'), ('a', 'b')]
```

Each gamete takes one allele of each gene, so the four combinations are `AB`, `Ab`, `aB` and `ab`, each with chance 1/4. Two of these, `Ab` and `aB`, are combinations that neither of the parent's own parents necessarily had together; they are made by the shuffle.

The Dihybrid Cross

Now cross two `AaBb` parents. Each makes four gamete types, so the Punnett square from Genetics and Heredity Fundamentals 1 has 4 x 4 = 16 cells. We can build it by pairing every gamete of one parent with every gamete of the other, and then work out what each child looks like. A child shows the dominant form of a trait if it has at least one dominant allele of that gene.

```
from itertools import product
from collections import Counter

def phenotype(alleles):
    a = "A" if "A" in alleles[:2] else "a"
    b = "B" if "B" in alleles[2:] else "b"
    return a + b

g = list(product("Aa", "Bb"))
counts = Counter()
for g1 in g:
    for g2 in g:
        child = g1[0] + g2[0] + g1[1] + g2[1]
        counts[phenotype(child)] += 1
print(len(g) ** 2)
print(sorted(counts.items(), reverse=True))
print(9 / 16, 3 / 4 * 3 / 4)
```

```
16
[('ab', 1), ('aB', 3), ('Ab', 3), ('AB', 9)]
0.5625 0.5625
```

Of the 16 equally likely children, 9 show both dominant forms (`AB`), 3 show only the first (`Ab`), 3 show only the second (`aB`) and 1 shows neither (`ab`). This is the famous **9:3:3:1** ratio (check). Notice where it comes from. For each gene on its own, the chance of the dominant form is 3/4, from Genetics and Heredity Fundamentals 1. Because the genes are independent, the chances multiply: 3/4 x 3/4 = 9/16, which is what the count gave. The same product rule gives 3/4 x 1/4 = 3/16 for each of the middle two classes and 1/4 x 1/4 = 1/16 for the last.

PHENOTYPE	CHANCE (PRODUCT RULE)	OUT OF 16
both dominant (`AB`)	3/4 x 3/4	9
first dominant only (`Ab`)	3/4 x 1/4	3
second dominant only (`aB`)	1/4 x 3/4	3
neither (`ab`)	1/4 x 1/4	1

Seeing the Ratio by Simulation

The exact count is a statement about chances; real offspring come in whole numbers that wobble around it, as in Genetics and Heredity Fundamentals 1. A simulation of 16,000 children should land near 9, 3, 3 and 1 per 16.

```
import random
from collections import Counter

def make_gamete(rng):
    return rng.choice("Aa") + rng.choice("Bb")

def child_pheno(rng):
    g1, g2 = make_gamete(rng), make_gamete(rng)
    a = "A" if "A" in (g1[0], g2[0]) else "a"
    b = "B" if "B" in (g1[1], g2[1]) else "b"
    return a + b

rng = random.Random(5)
n = 16000
counts = Counter(child_pheno(rng) for _ in range(n))
for k in ["AB", "Ab", "aB", "ab"]:
```

```
    print(k, counts[k], round(counts[k] / n * 16, 2))
```

```
AB 8897 8.9
Ab 3011 3.01
aB 3092 3.09
ab 1000 1.0
```

The last column rescales each count to "out of 16": 8.9, 3.01, 3.09 and 1.0, close to 9, 3, 3 and 1. With only a few dozen children the ratio would look much rougher.

> **9:3:3:1 IS NOT A LAW OF NATURE**
>
> The ratio holds only when the genes assort independently, each trait has simple dominance, and the numbers are large. Other ratios mean something else is going on: linked genes (below), or interactions between genes and more complicated dominance, covered in Genetics and Heredity Fundamentals 6 (check).

The Test Cross

How can a breeder tell whether a plant that shows both dominant forms is really `AaBb` or something else such as `AABB`? The answer is a **test cross**: cross it with an individual that is homozygous recessive for both genes, `aabb` (check). That partner can only give `ab` gametes, so every offspring shows exactly what the tested parent's gamete carried. The offspring are a direct readout of the gametes.

```
def linked_gamete(rng, r):
    # parent is AB / ab: one chromosome carries A and B, the other a and b
    if rng.random() < r:
        return rng.choice(["Ab", "aB"])
    return rng.choice(["AB", "ab"])

rng = random.Random(8)
for r in (0.5, 0.1, 0.0):
    c = Counter(linked_gamete(rng, r) for _ in range(20000))
```

```
    print(r, [(k, c[k]) for k in ("AB", "ab", "Ab", "aB")])
```

```
0.5 [('AB', 4878), ('ab', 5001), ('Ab', 5053), ('aB', 5068)]
0.1 [('AB', 9159), ('ab', 8855), ('Ab', 999), ('aB', 987)]
0.0 [('AB', 10119), ('ab', 9881), ('Ab', 0), ('aB', 0)]
```

This block, which reuses `random` and `Counter` from the previous program, previews the next section. It shows what a test cross reveals when the genes are on the same chromosome. The number `r` is explained below; read the three lines as three different situations and compare them.

Linkage

Genes on the same chromosome are said to be **linked** (check). Picture a parent whose one chromosome carries `A` and `B` and whose other chromosome carries `a` and `b`, written `AB / ab`. If the chromosomes were never cut and rejoined, the only gametes would be `AB` and `ab`: the alleles on one chromosome would always stay together. Crossing over (Genetics and Heredity Fundamentals 4) breaks the association. A gamete that carries `Ab` or `aB` has a chromosome made of parts of both, and is called **recombinant**; the gametes `AB` and `ab` are **parental** (check).

The fraction of recombinant gametes is the **recombination frequency**, called `r` in the code. Read the three lines of output above:

- `r = 0.5`: all four classes are about equal, about 5,000 each of 20,000. This is exactly the independent pattern. Genes on different chromosomes behave as if `r` were 0.5, and so do genes so far apart on one chromosome that a crossover between them is nearly certain to happen in about half of cases (check).
- `r = 0.1`: the parental classes `AB` and `ab` dominate (about 9,000 each) and the recombinant classes are small (about 1,000 each). The genes tend to stay together.
- `r = 0.0`: no recombinants at all. The genes are inherited as if they were a single gene.

The closer two genes lie on a chromosome, the less likely a crossover falls between them, and so the smaller `r` is. A value of `r` can never usefully exceed 0.5, because by then the genes look unlinked.

Measuring Distance with r

The relationship runs the other way too: from a test cross we can count the recombinant offspring and estimate `r`, which tells us how far apart the genes are. In genetics a recombination frequency of 1% is called one **map unit**, or one **centimorgan** (check). Arranging genes along a chromosome from such numbers gives a **genetic map**. This is how maps were first drawn, long before genomes could be read directly (check). The code below imitates a test cross of 5,000 offspring where the true value is 0.12.

```
rng = random.Random(21)
c = Counter(linked_gamete(rng, 0.12) for _ in range(5000))
recomb = c["Ab"] + c["aB"]
print(dict(c))
print(recomb, sum(c.values()), round(recomb / sum(c.values()) * 100, 1))
```

```
{'ab': 2204, 'AB': 2188, 'Ab': 288, 'aB': 320}
608 5000 12.2
```

There are 288 + 320 = 608 recombinant offspring out of 5,000, so the estimate is 12.2%, or about 12.2 map units, close to the true 12. With more offspring the estimate gets closer. The genes in this example would be drawn about 12 units apart on a map.

> **MAP UNITS ARE ONLY APPROXIMATE**
>
> Map distance adds up well over short stretches, but over long ones two crossovers between the same pair of genes cancel each other, so some recombinants are missed and the real distance is underestimated (check). Distances in map units are also not the same as distances in DNA letters; crossing over happens more often in some regions than others (check).

Challenges

EXERCISE 1

Cross two parents that are heterozygous for `n` independent genes (`n` = 1, 2 and 3). For each `n`, enumerate every pair of gametes with `itertools.product`, count the children that show the dominant form of every gene, and compare the fraction with $(3/4)^n$ using `fractions.Fraction`.

View solution

EXERCISE 2

Simulate a test cross of `AaBb` with `aabb` for independent genes. Use `random.Random(13)` and 40,000 offspring. Print the count and fraction of each of the four classes, and the difference between the largest and the smallest class.

View solution

EXERCISE 3

Write `estimate(rng, r, n)` that simulates `n` test-cross offspring from an `AB / ab` parent with true recombination frequency `r`, and returns the fraction that are recombinant. With `random.Random(5)` and 100,000 offspring, run it for `r` = 0.05, 0.2 and 0.5 and print the estimate and its distance from the true value.

View solution

QUICK REFERENCE

- **Independent assortment:** genes on different chromosomes are sorted separately, so their chances multiply
- **Dihybrid cross** (`AaBb` x `AaBb`): 16 equally likely cells; phenotypes 9 : 3 : 3 : 1
- **Test cross:** cross with the homozygous recessive; offspring reveal the gametes directly
- **Linkage:** genes on the same chromosome tend to be inherited together
- **Recombination frequency r:** fraction of recombinant gametes; 0 to 0.5; 1% is one map unit

- **Check:** the names and statements of the laws, the map unit definition, and the limits of map distances

COMING NEXT

Genetics and Heredity Fundamentals 6 looks at traits that do not follow the simple dominant and recessive pattern.

Genetics and Heredity Fundamentals

Course 1 · Chapter 6 · Beyond Dominant and Recessive

Genetics and Heredity Fundamentals 1 used the simplest rule: one allele hides the other. That rule is real, but it is only one of several. In this chapter we look at three other patterns: alleles that blend, alleles that both show, and traits built from many genes at once. The genotype counting from earlier chapters still works; what changes is how a genotype is turned into a look. All the programs here are Python 3.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run and the output shown is what it printed. The biology is written from general knowledge; items marked (check) are the ones to verify first. Where a program uses random numbers it fixes the seed, so the same output appears each time.

Incomplete Dominance

In **incomplete dominance**, the heterozygote looks like a mixture of the two homozygotes rather than like one of them (check). The textbook example is a flower with a red allele `R` and a white allele `r`: `RR` is red, `rr` is white, and `Rr` is pink (check). Neither allele is hiding the other. The pink flower has half the red-making product of the red one.

In code the change is small. The look now depends on how many copies of `R` the genotype has, not only on whether there is one.

```
from itertools import product
from collections import Counter

def flower(genotype):
    return {0: "white", 1: "pink", 2: "red"}[genotype.count("R")]

kids = Counter(flower(a + b) for a, b in product("Rr", "Rr"))
print(kids)
```

```
Counter({'pink': 2, 'red': 1, 'white': 1})
```

Crossing two pink flowers gives 1 red, 2 pink and 1 white out of 4. That is a **1:2:1** ratio. Compare it with the 3:1 ratio from Genetics and Heredity Fundamentals 1: the genotype counts are the same (1 `RR`, 2 `Rr`, 1 `rr`), but now each genotype has its own look, so the look ratio equals the genotype ratio. This is also a useful clue in practice: a 1:2:1 look ratio in a cross suggests incomplete dominance (check).

PATTERN	`RR` LOOKS LIKE	LOOK RATIO, `RR` X `RR`
complete dominance	the same as `RR`	3 : 1
incomplete dominance	in between `RR` and `rr`	1 : 2 : 1
codominance	both at once	1 : 2 : 1

Codominance

In **codominance**, the heterozygote shows both alleles fully and separately, not a blend (check). The look is not "in between"; it is "both". A frequently quoted example is an animal with a red coat allele and a white coat allele whose heterozygote has patches of red and patches of white (check). Because the two patterns have the same 1:2:1 count, the table above lists them together; what differs is the biology, not the arithmetic.

> **THE WORDS ARE USED LOOSELY**
>
> Whether a trait is called incomplete dominance or codominance often depends on which level you look at. The same heterozygote can show a blended outward look (incomplete) while the cells make both gene products fully (codominant) (check). Textbooks differ, so treat the names as labels for the observed pattern.

Blood Groups: Three Alleles and Both Patterns

The ABO blood groups show several ideas together (check). The gene has three common alleles, usually called `A`, `B` and `O`. Up to now every gene had two alleles. A person still carries only

two copies, one from each parent, but the population has three to choose from, so there are six genotypes.

- `A` and `B` are **codominant** with each other: a person with both is group **AB**, showing both markers (check).
- `O` is **recessive** to both: it only shows in a person with two copies, `OO`, who is group **O** (check).
- So `AA` and `AO` are both group A, and `BB` and `BO` are both group B (check).

```
def blood_group(genotype):
    alleles = set(genotype)
    if alleles == {"A", "B"}:
        return "AB"
    if "A" in alleles:
        return "A"
    if "B" in alleles:
        return "B"
    return "O"

for g in ["AA", "AO", "BB", "BO", "AB", "OO"]:
    print(g, blood_group(g))
```

```
AA A
AO A
BB B
BO B
AB AB
OO O
```

Six genotypes, but only four looks. That is the key difference from Genetics and Heredity Fundamentals 1: here you cannot always read the genotype from the look. A person in group A might be `AA` or `AO`. Now ask what a pair of parents can have as children. Each parent gives one of their two alleles, so we try every pairing, as in the Punnett squares of earlier chapters, and collect the groups.

```
def child_groups(p1, p2):
    groups = set()
    for a in p1:
        for b in p2:
            groups.add(blood_group(a + b))
    return sorted(groups)

print(child_groups("AB", "OO"))
print(child_groups("AO", "BO"))
print(child_groups("AA", "BB"))
print(child_groups("OO", "OO"))
```

```
['A', 'B']
['A', 'AB', 'B', 'O']
['AB']
['O']
```

Read the lines in turn. An AB parent and an O parent cannot have an AB child or an O child, only A or B. Two parents who show A and B (here both heterozygous, `AO` and `BO`) can have all four groups. Two O parents can only have O children. Facts like these are why blood groups were once used as one piece of evidence in family questions, with the strong caution that they can only rule things out, never prove a relationship (check).

TWO THINGS THIS CODE LEAVES OUT

Real blood typing also has the separate Rh positive and negative factor, which is controlled by a different gene (check). And rare exceptions to the usual rules exist, so a program like this describes the standard textbook model, not every real family (check).

Many Genes, One Trait

So far a trait has been the work of one gene. Many familiar traits, such as height, are instead shaped by a great many genes, each adding a small push, together with the environment

(check). These are **polygenic** traits. They do not fall into neat groups like "red, pink, white" but form a smooth range.

A simple model shows why. Suppose there are `n` genes, and each has two alleles, one "tall" (worth 1) and one "short" (worth 0). An individual has two copies of each gene, so the total score runs from 0 to 2n. Take a parent that is heterozygous for every gene, so every allele is equally likely to be passed on. How many of the 2n allele slots carry a tall allele? This is the same as counting the heads in 2n fair coin tosses, and we can count the ways exactly with `math.comb`. With `n` = 4 genes there are 8 slots.

```
from math import comb
from fractions import Fraction

n = 4
for k in range(2 * n + 1):
    ways = comb(2 * n, k)
    print(k, ways, Fraction(ways, 2 ** (2 * n)), "#" * ways)
```

```
0 1 1/256 #
1 8 1/32 ########
2 28 7/64 ############################
3 56 7/32 ########################################################
4 70 35/128 ######################################################################
5 56 7/32 ########################################################
6 28 7/64 ############################
7 8 1/32 ########
8 1 1/256 #
```

The bars form a bell shape: the middle score (4) is the most common, with 70 of the 256 equally likely combinations, and the extremes 0 and 8 each have just 1. A person at an extreme needs luck on every single gene. More genes make the shape smoother and the extremes rarer. Now simulate individuals with 1, 2, 5 and 10 genes (so 2, 4, 10 and 20 slots).

```
import random
```

```
from collections import Counter

def height_score(rng, genes):
    return sum(rng.choice((0, 1)) + rng.choice((0, 1)) for _ in range(genes))

rng = random.Random(3)
n = 20000
for genes in (1, 2, 5, 10):
    scores = [height_score(rng, genes) for _ in range(n)]
    mean = sum(scores) / n
    c = Counter(scores)
    print(genes, round(mean, 2), len(c), max(c.values()))
```

```
1 1.0 3 10065
2 2.0 5 7498
5 5.01 11 4860
10 10.01 18 3534
```

The columns are the number of genes, the average score, how many different scores appeared, and how many individuals shared the most common score. The average is always about the number of genes (half of the 2n slots), whatever n is. The number of different scores grows with n (3, 5, 11 and 18, of a possible 3, 5, 11 and 21; with 10 genes the three most extreme scores never turned up in 20,000 individuals). And the most common score holds a smaller and smaller share: about half of individuals with 1 gene, about a sixth with 10. The more genes, the finer the steps, and the closer the picture gets to a smooth curve.

> **A MODEL, NOT A MEASUREMENT**
>
> Real polygenic traits do not have equal genes or exactly two alleles each, and the environment (diet, health and so on) adds its own spread on top of the genes (check). Our model only explains the shape: many small independent contributions add up to a bell-like distribution. A related idea, **heritability**, describes how much of the variation in a group comes from genes; it says nothing about how much of one person's trait is "caused by" genes (check).

Challenges

EXERCISE 1

For incomplete dominance with `R` and `r`, take every unordered pair of parent genotypes from `RR`, `Rr`, `rr` (use `itertools.combinations_with_replacement`). For each pair, print the chance of each flower colour as a `fractions.Fraction`. Which pair gives all three colours?

View solution

EXERCISE 2

Using `blood_group` and `child_groups`, list the unordered pairs of the six genotypes (21 pairs). Print how many pairs can have a group O child, which pairs can have children in all four groups, and which pairs can have both an AB child and an O child.

View solution

EXERCISE 3

Simulate 50,000 individuals with 10 genes using `random.Random(7)` and the `height_score` idea. Print the mean score, the fraction scoring between 8 and 12 inclusive, how many scored exactly 0 or 20, and the lowest and highest scores seen.

View solution

QUICK REFERENCE

- **Complete dominance:** heterozygote looks like one homozygote; cross of heterozygotes gives 3 : 1
- **Incomplete dominance:** heterozygote is in between; cross gives 1 : 2 : 1
- **Codominance:** heterozygote shows both alleles; also 1 : 2 : 1
- **ABO blood groups:** three alleles; `A` and `B` codominant, `O` recessive; six genotypes, four looks
- **Polygenic traits:** many genes add small effects; the result is a smooth, bell-shaped range
- **Check:** the names of the patterns, the blood group details, and what heritability means

COMING NEXT

Genetics and Heredity Fundamentals 7 looks at the chromosomes that decide sex, and at traits carried on them.

Genetics and Heredity Fundamentals

Course 1 · Chapter 7 · Sex and Linked Traits

Until now every gene sat on a chromosome that comes in two matching copies. One pair of chromosomes is different. In humans it decides whether a person is female or male, and any gene on it follows its own rules. In this chapter we count the children of a cross with the same enumeration tools as before, then turn the question around: given a family tree, what can we say about the gene? All the programs here are Python 3.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run and the output shown is what it printed. The biology is written from general knowledge; items marked (check) are the ones to verify first. Where a program uses random numbers it fixes the seed, so the same output appears each time. The chapter describes the usual human pattern; other species do it differently (check).

The Sex Chromosomes

Humans have 46 chromosomes in 23 pairs (check). Twenty-two pairs are the same in everyone and are called **autosomes**. The last pair is the **sex chromosomes**, named **X** and **Y**. A typical female has two X chromosomes (`XX`) and a typical male has one X and one Y (`XY`) (check). The Y is much smaller than the X and carries far fewer genes (check). A gene on the Y called `SRY` has a large part in starting male development (check).

Meiosis (Genetics and Heredity Fundamentals 4) makes the next step easy. A mother is `XX`, so every egg carries an X. A father is `XY`, so half of his sperm carry an X and half a Y. Whether a child is a girl or a boy depends on which sperm wins. We can enumerate it exactly as we did for a Punnett square in Genetics and Heredity Fundamentals 1.

```
from itertools import product
from collections import Counter

def sex(pair):
    return "female" if pair == "XX" else "male"

kids = Counter(sex(a + b) for a, b in product("XX", "XY"))
```

```
print(kids)
```

```
Counter({'female': 2, 'male': 2})
```

Two of the four cells are female and two are male, a 1:1 ratio. The mother's two X chromosomes are identical in this simple count, so each of the four cells is a pairing of one egg type with one sperm type. The father alone decides the child's sex, because the mother can only give an X. In real populations the ratio at birth is close to, but not exactly, equal (check).

> **XX AND XY ARE THE USUAL PATTERN, NOT THE ONLY ONE**
>
> Some people have a different number of sex chromosomes, and sex in the wider sense (body, hormones, identity) is not fully captured by chromosomes alone (check). This chapter only follows the chromosomes, because that is what decides how a gene on them is passed on.

Why Sons and Daughters Differ

The X chromosome carries many genes that have nothing to do with sex, such as genes needed for seeing certain colours or for making blood clot (check). A daughter has two copies of each of these genes, one on each X, so everything from Genetics and Heredity Fundamentals 1 applies to her: a recessive allele is hidden if the other copy is the dominant one. A son has only **one** X, which he always gets from his mother. His Y does not carry a matching copy of most X genes (check). So whatever allele sits on his one X shows, whether it is dominant or recessive. There is nothing to hide it.

The result is called **X-linked recessive** inheritance (check). A recessive allele on the X is expressed:

- in a son, if his one X carries it;
- in a daughter, only if both her X chromosomes carry it.

A daughter with one copy of the allele and one of the normal allele is a **carrier**: she does not show the trait herself but can pass the allele on (check). Writing the alleles as `H` (normal) and `h` (the recessive one), the possible people are:

PERSON	X CHROMOSOMES	LOOK
daughter	`H H`	unaffected
daughter	`H h`	carrier, unaffected
daughter	`h h`	affected
son	`H` (and a Y)	unaffected
son	`h` (and a Y)	affected

A classic textbook example is red-green colour vision deficiency, where the genes involved lie on the X (check). It is much more common in men than in women, which is exactly what the table predicts: a man needs one copy of the allele, a woman needs two (check).

Counting Children

The program below crosses a mother with the pair `(m1, m2)` on her two X chromosomes and a father with one X allele. Each child is one egg (one of the mother's two X chromosomes) combined with one sperm: an X from the father (a daughter) or his Y (a son). Notice that a son takes only the mother's allele; the father's X never reaches a son.

```
from itertools import product
from collections import Counter

def child_look(sex, x_alleles):
    # x_alleles: the X chromosomes the child has, "H" = normal, "h" = recessive
    if sex == "male":
        return "affected" if x_alleles == ("h",) else "unaffected"
    n = x_alleles.count("h")
    return {0: "unaffected", 1: "carrier", 2: "affected"}[n]

def cross(mother, father_x):
    # mother: two X alleles, e.g. ("H", "h"); father_x: his one X allele
    out = Counter()
    for m, f in product(mother, ("X", "Y")):
        if f == "X":
            out["daughter " + child_look("female", tuple(sorted((m, father_x))))] += 1
        else:
```

```
            out["son " + child_look("male", (m,))] += 1
    return out

for name, mom, dad in [("carrier mother x unaffected father", ("H","h"), "H"),
                       ("unaffected mother x affected father", ("H","H"), "h"),
                       ("carrier mother x affected father", ("H","h"), "h")]:
    print(name)
    for k, v in sorted(cross(mom, dad).items()):
        print("   ", k, v)
```

```
carrier mother x unaffected father
    daughter carrier 1
    daughter unaffected 1
    son affected 1
    son unaffected 1
unaffected mother x affected father
    daughter carrier 2
    son unaffected 2
carrier mother x affected father
    daughter affected 1
    daughter carrier 1
    son affected 1
    son unaffected 1
```

- **Carrier mother, unaffected father:** half the sons are affected (they got her `h`), and no daughter is affected, because each daughter also gets the father's normal X. Half of the daughters are carriers.
- **Unaffected mother, affected father:** no child is affected, but every daughter is a carrier, because each gets her father's one X, which is `h`. Every son is unaffected, because he gets no X from his father.
- **Carrier mother, affected father:** now a daughter can be affected, since she may get `h` from both parents.

These crosses make a pattern worth remembering: an affected father cannot pass the trait directly to a son, because he gives a son his Y, not his X. The trait can skip a generation through

a carrier daughter. This is sometimes described as passing from grandfather through a daughter to a grandson (check).

Simulating Families

Enumeration gives exact fractions. A simulation shows how those fractions look in real families. The program below makes 40,000 children of a carrier mother and an unaffected father, with a fixed seed, and reports the share of sons and of daughters who are affected.

```
import random
from collections import Counter

def child(rng, mother, father_x):
    sex = rng.choice("XY")
    m = rng.choice(mother)
    if sex == "Y":
        return "son", m == "h"
    return "daughter", (m, father_x).count("h") == 2

rng = random.Random(11)
c = Counter()
for _ in range(40000):
    s, aff = child(rng, ("H", "h"), "H")
    c[(s, aff)] += 1
for s in ("son", "daughter"):
    tot = c[(s, True)] + c[(s, False)]
    print(s, tot, c[(s, True)], round(c[(s, True)] / tot, 3))
```

```
son 19935 9998 0.502
daughter 20065 0 0.0
```

The columns are the group, the number of children in it, how many were affected, and the fraction. About half the sons are affected (0.502, close to 1/2) and none of the daughters, matching the first cross above. The sex split is also near even: 19,935 sons and 20,065 daughters.

Because the sex of each child is a separate chance, a single small family can look very different from these averages, as we saw with sample size in Genetics and Heredity Fundamentals 1.

Reading a Pedigree

Real families are not crossed on purpose. What we have is a **pedigree**: a family tree that records who is affected (check). Geneticists read pedigrees backwards, asking which patterns of inheritance could produce the one they see. The two clues we have found give a way to do this for an X-linked recessive trait:

- **Flagging:** an affected father with an affected son is a warning sign, since the son's X came from his mother and not from the father. It can still happen if the mother is a carrier or affected, but that is a less likely explanation, so a careful analysis looks at the mother too (check).
- **Ruling in:** a son who is affected must have got the allele from his mother, so she is a carrier or affected herself.

A pedigree is a good fit for a dictionary: each person maps to their father, mother and sex. The program below applies the two clues in their simplest form. If a father and son are both affected, it flags the pair as a pattern that does not fit the simple model.

```
# pedigree: name -> (father, mother, sex)
pedigree = {
    "Ann":   (None, None, "F"),
    "Bob":   (None, None, "M"),
    "Cara":  ("Bob", "Ann", "F"),
    "Dan":   ("Bob", "Ann", "M"),
    "Eli":   (None, None, "M"),
    "Fay":   ("Eli", "Cara", "F"),
    "Gus":   ("Eli", "Cara", "M"),
}
affected = {"Dan", "Gus"}

def father_to_son(pedigree, affected):
    # an affected father with an affected son: a warning sign for a trait on X
    hits = []
    for name, (dad, mom, sx) in pedigree.items():
        if sx == "M" and name in affected and dad in affected:
            hits.append((dad, name))
```

```
    return hits

def must_be_carrier(pedigree, affected):
    # mothers of affected sons carry the allele (X-linked recessive, check)
    carriers = set()
    for name, (dad, mom, sx) in pedigree.items():
        if sx == "M" and name in affected and mom is not None:
            carriers.add(mom)
    return sorted(carriers)

print(father_to_son(pedigree, affected))
print(must_be_carrier(pedigree, affected))
pedigree["Hal"] = ("Dan", "Fay", "M")
affected.add("Hal"); affected.add("Dan")
print(father_to_son(pedigree, affected))
```

```
[]
['Ann', 'Cara']
[('Dan', 'Hal')]
```

The first line is an empty list: in this family no affected father has an affected son, so the pedigree does not contradict an X-linked recessive trait. The second line lists the mothers who must be carriers: Ann (mother of the affected Dan) and Cara (mother of the affected Gus). Cara herself shows no trait here, which is just what a carrier looks like. Then we add Hal, son of Dan, and mark him affected. The last line reports the pair (Dan, Hal), a father and son who are both affected, which this simple model flags (a carrier mother could explain it, and so could other patterns, such as a gene on an autosome, so the mother needs checking and other patterns need considering) (check).

A PEDIGREE RULES PATTERNS OUT MORE THAN IT PROVES THEM

A check like this can reject a pattern, but passing it does not prove the trait is X-linked. Several patterns can produce the same small family, and real pedigrees also include new mutations, incomplete penetrance (a person with the genotype who does not show the trait), and mistaken records (check). Genetic counsellors use pedigrees as a first step, not as a final answer (check).

Challenges

EXERCISE 1

For each mother genotype `HH`, `Hh`, `hh` and each father (`H` or `h`), enumerate the children with `itertools.product`. Print the fraction of sons that are affected and the fraction of daughters that are affected, as `fractions.Fraction` values. Which pairings can have an affected daughter?

View solution

EXERCISE 2

A carrier mother and an unaffected father have three children. Simulate 20,000 such families with `random.Random(4)`, count the families with at least one affected son, and compare the fraction with the exact value $1 - (3/4)^3$ as a `Fraction`.

View solution

EXERCISE 3

Write `check(pedigree, affected)` that returns a message: either that the pedigree does not fit the simple X-linked recessive model, naming the affected father and son, or that it is consistent, listing the mothers of affected sons. Test it on a pedigree of eight people with three different sets of affected people.

View solution

QUICK REFERENCE

- **Sex chromosomes:** typical female `XX`, typical male `XY`; the father decides the child's sex, 1 : 1
- **X-linked recessive:** a son needs one copy of the allele, a daughter needs two
- **Carrier:** a daughter with one copy; unaffected, but can pass it on
- **Fathers and sons:** a son gets his father's Y, never his X, so no direct father-to-son passing
- **Pedigree:** a family tree used to rule patterns in or out; it rarely proves one

- **Check:** the chromosome facts, the colour vision example, and the pedigree caveats

COMING NEXT

Genetics and Heredity Fundamentals 8 moves from families to whole populations, and asks how allele frequencies stay the same or change.

Genetics and Heredity Fundamentals

Course 1 · Chapter 8 · Populations and Allele Frequencies

Every chapter so far followed one family at a time. Now we step back and look at a whole population: a group of individuals who breed with each other. The question changes from "what will this couple's children look like?" to "how common is each allele in the group, and does that change from one generation to the next?" This is the starting point of **population genetics** (check). All the programs here are Python 3.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run and the output shown is what it printed. The biology is written from general knowledge; items marked (check) are the ones to verify first. Where a program uses random numbers it fixes the seed, so the same output appears each time.

Counting Alleles

An **allele frequency** is the share of all the copies of a gene, in the whole population, that are a given allele. Each person carries two copies of an ordinary gene, so a population of 100 people has 200 copies. Take a gene with alleles `A` and `a`, and count them by walking through the genotypes. It is the same counting idea as in Genetics and Heredity Fundamentals 1, but done over a group.

```
from collections import Counter

people = ["AA"] * 36 + ["Aa"] * 48 + ["aa"] * 16
counts = Counter(allele for g in people for allele in g)
total = sum(counts.values())
p = counts["A"] / total
q = counts["a"] / total
print(counts, total)
print(p, q)
```

```
Counter({'A': 120, 'a': 80}) 200
0.6 0.4
```

By convention the frequency of one allele is written `p` and the other `q`. With only two alleles, `p + q = 1`. Here `p` = 0.6 and `q` = 0.4. Notice that a heterozygote `Aa` contributes one copy of each allele, so the 48 heterozygotes are counted in both totals. A frequency is a share of *alleles*, not a share of people.

The Hardy-Weinberg Principle

Suppose the allele frequencies are `p` and `q`, and individuals pair up at random (no one chooses a mate by genotype). Picking two alleles from the gene pool is like the coin tosses and Punnett squares we have used since Genetics and Heredity Fundamentals 1: the chance of `AA` is `p` times `p`, of `aa` is `q` times `q`, and of `Aa` is `2pq`, because there are two orders (`A` then `a`, or `a` then `A`). This gives the **Hardy-Weinberg** proportions (check):

$$p^2 + 2pq + q^2 = 1$$

The principle goes further. If a population is large, mates at random, and nothing pushes the allele frequencies (no new mutations, no migration, no selection), then the frequencies stay exactly the same in every generation, and the genotype proportions settle into p^2, $2pq$, q^2 after one generation of random mating (check). It is best read as a baseline: a population that matches it is not evolving at that gene, and a population that does not match is a clue that one of the assumptions fails. We can compute the expected proportions exactly with fractions.

```
from fractions import Fraction

def expected(p):
    q = 1 - p
    return {"AA": p * p, "Aa": 2 * p * q, "aa": q * q}

for p in (Fraction(1, 2), Fraction(7, 10), Fraction(9, 10)):
    e = expected(p)
```

```
    print(p, {g: str(f) for g, f in e.items()}, sum(e.values()))
```

```
1/2 {'AA': '1/4', 'Aa': '1/2', 'aa': '1/4'} 1
7/10 {'AA': '49/100', 'Aa': '21/50', 'aa': '9/100'} 1
9/10 {'AA': '81/100', 'Aa': '9/50', 'aa': '1/100'} 1
```

Each row sums to 1, as it must. Look at the last row: when `a` is rare (`q` = 1/10), only 1 person in 100 is `aa`, but 18 in 100 (9/50) carry one copy. **Most copies of a rare recessive allele are hidden in heterozygotes**, which is why such alleles are hard to remove from a population (check).

Testing a Population Against the Baseline

Given a real count of genotypes, we compute `p` from the alleles, work out the expected counts, and compare. A common measure of the gap is the **chi-square** statistic: for each genotype, take (observed minus expected) squared, divide by expected, and add them up (check). The larger the number, the worse the fit. For this test, with two alleles, a value above about 3.84 is usually taken as a poor fit at the 5% level (check).

```
from fractions import Fraction

def expected(p):
    q = 1 - p
    return {"AA": p * p, "Aa": 2 * p * q, "aa": q * q}

observed = {"AA": 30, "Aa": 60, "aa": 10}
n = sum(observed.values())
p = Fraction(2 * observed["AA"] + observed["Aa"], 2 * n)
print("p =", p)
chi = 0
for g, f in expected(p).items():
    e = float(f * n)
    chi += (observed[g] - e) ** 2 / e
    print(g, observed[g], e)
```

```
print("chi-square", round(chi, 2))
```

```
p = 3/5
AA 30 36.0
Aa 60 48.0
aa 10 16.0
chi-square 6.25
```

This population has too many heterozygotes (60 seen, 48 expected) and too few of both homozygotes. The statistic is 6.25, above 3.84, so it does not fit the baseline. Possible reasons include non-random mating, selection that favours heterozygotes, or two populations that were mixed together (check). The test says that something differs, not which thing it is. Small samples also give unreliable results, as sample size did in Genetics and Heredity Fundamentals 1.

Genetic Drift

The principle assumed a very large population. In a small one, chance matters. Each generation, the next generation's alleles are a random sample of the current gene pool, and a random sample does not match its source exactly (just as 10 coin tosses are rarely exactly 5 heads). The allele frequency therefore wanders from generation to generation, with no cause other than luck. This is **genetic drift** (check).

The simplest model: a population of `size` individuals has `2 * size` allele copies. For the next generation, draw `2 * size` copies at random, each being `A` with the current probability `p`. This is a simplified version of the Wright-Fisher model (check). The program follows `p` for 50 generations, starting at 0.5, in populations of 10, 100 and 1000, printing it every 10 generations.

```
import random

def next_gen(rng, p, size):
    alleles = sum(rng.random() < p for _ in range(2 * size))
    return alleles / (2 * size)
```

```
for size in (10, 100, 1000):
    rng = random.Random(5)
    p = 0.5
    trail = []
    for gen in range(1, 51):
        p = next_gen(rng, p, size)
        if gen % 10 == 0:
            trail.append(round(p, 3))
    print(size, trail)
```

```
10 [0.2, 0.0, 0.0, 0.0, 0.0]
100 [0.625, 0.5, 0.435, 0.445, 0.45]
1000 [0.495, 0.476, 0.522, 0.553, 0.625]
```

In the population of 10, the frequency fell to 0.2 by generation 10 and then hit 0.0 by generation 20: the allele `A` was **lost**, and since no new alleles arrive in this model, it stays lost (the frequency 0 cannot grow back). The opposite end, a frequency of 1.0, is called **fixation**: everyone carries only that allele. In the population of 100 the frequency moved between about 0.43 and 0.63 and was still not fixed after 50 generations. The population of 1000 also drifted, within a fairly narrow range at first, but this run happened to end at 0.625 (a single run, not a rule). Drift is stronger the smaller the population. We can measure it by repeating the experiment many times and recording which allele ends up fixed, and how long that takes.

```
import random

def to_fixation(rng, p, size):
    gens = 0
    while 0 < p < 1:
        k = sum(rng.random() < p for _ in range(2 * size))
        p = k / (2 * size)
        gens += 1
    return p, gens

for size in (5, 20, 50):
    rng = random.Random(1)
    results = [to_fixation(rng, 0.5, size) for _ in range(300)]
```

```
    fixed_A = sum(1 for p, g in results if p == 1)
    mean_gens = sum(g for p, g in results) / len(results)
    print(size, fixed_A, 300 - fixed_A, round(mean_gens, 1))
```

```
5 152 148 13.6
20 147 153 53.4
50 166 134 140.0
```

The columns are the population size, the runs where `A` became fixed, the runs where it was lost, and the average number of generations until one or the other happened. Starting from 0.5, the two outcomes are about equally likely (roughly half each, not exactly), so drift does not favour either allele; it only removes variation. The average time grows with size: about 14 generations for 5 individuals, 53 for 20 and 140 for 50. Small populations lose variation quickly; large ones keep it much longer (check). This is also why a small group that starts a new population can differ in its allele frequencies from the group it came from (check).

> **DRIFT IS NOT "FOR" ANYTHING**
>
> Drift is blind. Which allele wins in a small population is luck, not fitness. A rare helpful allele can be lost, and a harmless or slightly harmful one can spread (check). When a population changes in a way that favours some alleles because of how they affect survival or reproduction, that is a different force, covered next.

Selection

Natural selection acts when genotypes differ in how many offspring they leave. We give each genotype a **fitness**, a number for its relative reproductive success: 1.0 for the best, lower for the worse (check). After one round of selection, each genotype's share is multiplied by its fitness and the shares are rescaled to total 1. In the model below, the frequency of `A` in the next generation is the share of `A` copies among all the surviving, reproducing genotypes.

Let `w_AA`, `w_Aa` and `w_aa` be the fitnesses. Genotypes start in Hardy-Weinberg proportions. The mean fitness is the sum of each proportion times its fitness. The copies of `A` come from all

`AA` (every copy) and half of the `Aa` copies. This gives the update below. It is deterministic (no random numbers), so it describes a very large population where drift can be ignored.

```
def step(p, w_AA, w_Aa, w_aa):
    q = 1 - p
    mean_w = p * p * w_AA + 2 * p * q * w_Aa + q * q * w_aa
    return (p * p * w_AA + p * q * w_Aa) / mean_w

p = 0.01
for gen in range(0, 301):
    if gen in (0, 25, 50, 100, 150, 200, 300):
        print(gen, round(p, 4))
    p = step(p, 1.0, 1.0, 0.9)
```

```
0 0.01
25 0.1114
50 0.4572
100 0.8127
150 0.897
200 0.9302
300 0.958
```

Here `aa` individuals leave only 90% as many offspring as the others, a mild disadvantage, and `A` starts rare at 0.01. The pattern is typical. At first `A` moves slowly, because when it is rare almost all its copies are in heterozygotes, which have full fitness like `AA`, so most of the selection falls on the few `aa` individuals that exist. The change speeds up in the middle (from 0.11 at generation 25 to 0.46 at generation 50) and then slows again as `a` becomes rare and hides in heterozygotes. After 300 generations `a` has not disappeared; it is still about 4%. Selection against a recessive allele is slow at removing the last copies (check). This ties back to the rare-allele row in the Hardy-Weinberg table.

FORCE	CHANGES ALLELE FREQUENCIES?	DIRECTION	STRONGEST WHEN
random mating, nothing else	no (Hardy-Weinberg)	none	-

FORCE	CHANGES ALLELE FREQUENCIES?	DIRECTION	STRONGEST WHEN
genetic drift	yes	random	the population is small
natural selection	yes	towards higher fitness	fitness differences are large
mutation, migration	yes	depends	(not simulated here)

MODELS ARE SIMPLER THAN NATURE

Real fitness depends on the environment and can change; many traits are polygenic (Genetics and Heredity Fundamentals 6), and populations are not perfectly random-mating or isolated. The fitness numbers here are made up to show the shape of the behaviour, not measurements of any real gene (check). Also, "fitness" in this sense means reproductive success, not strength or health in everyday language.

Challenges

EXERCISE 1

Three populations have these genotype counts (AA, Aa, aa): north 49, 42, 9; south 20, 60, 20; east 5, 30, 65. For each, compute `p` and `q` as `Fraction` values, the expected counts under Hardy-Weinberg, and the chi-square statistic. Say which fit the baseline using the cut-off 3.84.

View solution

EXERCISE 2

Using `random.Random(2)`, run 400 populations of 25 individuals starting at `p` = 0.2, each until the allele is fixed or lost. Print how many ended fixed, how many lost, the fixed fraction, and the mean number of generations. Compare the fixed fraction with the starting `p`.

View solution

EXERCISE 3

Using the `step` function with fitnesses 1.0, 1.0 and 1 - s for `aa`, start with `q` = 0.5 and count how many generations it takes for `q` to fall below 0.1 when s is 0.2 and when s is 0.05. Then run 100 generations with s = 0.2 and print the final `q`.

View solution

QUICK REFERENCE

- **Allele frequency:** share of all gene copies; `p + q = 1`; a heterozygote counts once for each allele
- **Hardy-Weinberg:** with random mating and no other forces, genotypes are p^2, $2pq$, q^2 and frequencies do not change
- **Rare recessives:** most copies hide in heterozygotes
- **Drift:** random change in allele frequency, stronger in small populations; ends in loss or fixation
- **Selection:** fitness differences push frequencies in a direction; slow against a rare recessive
- **Check:** the model details, the 3.84 cut-off, and the drift and fixation vocabulary

COMING NEXT

Genetics and Heredity Fundamentals 9 turns from alleles to the sequences themselves, with string tools for reading and comparing DNA.

Genetics and Heredity Fundamentals

Course 1 · Chapter 9 · Reading Genomes

Genetics and Heredity Fundamentals 2 showed that a gene is a stretch of DNA written in four letters, and Genetics and Heredity Fundamentals 3 showed how a cell reads those letters three at a time. In this chapter we do the reading ourselves. A DNA sequence is just a string, so Python's string tools, dictionaries and counters are enough to measure it, search it and compare it with another one. All the programs here are Python 3.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run and the output shown is what it printed. The sequences are short invented examples, not real genes. The biology is written from general knowledge; items marked (check) are the ones to verify first. Where a program uses random numbers it fixes the seed, so the same output appears each time.

What Sequencing Gives You

Sequencing is reading the order of the bases in a piece of DNA. Current machines do not read a whole chromosome in one go. They read many short pieces, called **reads**, and software then fits the overlapping pieces back together into longer sequences, a step called assembly (check). The reads can be hundreds of letters long, or much longer with some newer methods (check). The result is stored as text.

The most common plain-text format is **FASTA** (check). Each record starts with a line that begins with `>` followed by a name and a free-text description. The following lines hold the sequence, usually wrapped at a fixed width. The parser below keeps the first word of each name, joins the wrapped lines and forces upper case, so that `a` and `A` are treated alike.

```
fasta = """>seq1 demo gene
ATGGCCATTG
TAATGGGCCG
>seq2 other
GGGCCCATAT
ATATAT
"""
```

```
def parse_fasta(text):
    records = {}
    name = None
    for line in text.splitlines():
        line = line.strip()
        if not line:
            continue
        if line.startswith(">"):
            name = line[1:].split()[0]
            records[name] = []
        else:
            records[name].append(line.upper())
    return {n: "".join(parts) for n, parts in records.items()}

seqs = parse_fasta(fasta)
for name, s in seqs.items():
    print(name, len(s), s)
```

```
seq1 20 ATGGCCATTGTAATGGGCCG
seq2 16 GGGCCCATATATATAT
```

The two records come back as a dictionary from name to sequence, each already joined into a single string. A real genome file is the same idea at a huge scale: the human genome is roughly three billion letters (check), which is why real tools read files line by line rather than loading everything at once.

Counting Letters and GC Content

The simplest question about a sequence is what it is made of. `collections.Counter` tallies the letters. A particularly useful summary is the **GC content**: the fraction of letters that are `G` or `C`. Because G pairs with C by three hydrogen bonds and A with T by two (Genetics and Heredity Fundamentals 2), GC content is related to how tightly the two strands hold together, and it differs between species and between regions of a genome (check).

The same program also builds the **reverse complement**. The two strands of the double helix run in opposite directions, so the partner strand, read in its own direction, is the complement written backwards. `str.maketrans` and `str.translate` swap letters in one pass, and `[::-1]` reverses the result.

```
from collections import Counter

def gc_content(s):
    return (s.count("G") + s.count("C")) / len(s)

COMP = str.maketrans("ACGT", "TGCA")
def revcomp(s):
    return s.translate(COMP)[::-1]

s = "ATGGCCATTGTAATGGGCCG"
print(Counter(s))
print(round(gc_content(s), 3))
print(revcomp(s))
print(revcomp(revcomp(s)) == s)
```

```
Counter({'G': 7, 'T': 5, 'A': 4, 'C': 4})
0.55
CGGCCCATTACAATGGCCAT
True
```

There are 11 `G` or `C` letters out of 20, so the GC content is 0.55. Taking the reverse complement twice returns the original, a handy sanity check for the function. Reverse complements matter because a gene can sit on either strand, so a search for a pattern usually has to look at both.

> **REAL DATA IS MESSIER THAN THESE STRINGS**
>
> Real sequence files contain `N` for a base that could not be read, other letters for ambiguous bases, and sometimes lower-case letters that mark repeated regions (check). Code that assumes only `ACGT` will quietly give wrong answers on such input. Exercise 1 asks you to check for this.

Finding Patterns

Searching a sequence for a short pattern, or **motif**, is the everyday job. Python gives two tools. `str.find(pattern, start)` returns the position of the first match at or after `start`, or `-1`. The `re` module finds every match, with a catch: a normal `re.finditer` does not report matches that overlap, so in `AGGGGA` it finds `GG` only twice (positions 1 and 3). Wrapping the pattern in a zero-width lookahead, `(?=GG)`, reports every position where the pattern begins, including overlapping ones.

A famous motif is the start codon `ATG` (Genetics and Heredity Fundamentals 3). A stretch that begins with `ATG` and continues codon by codon until a stop codon (`TAA`, `TAG` or `TGA`) with no earlier stop is called an **open reading frame**, or ORF, a candidate for a protein-coding gene (check). The function below tries every `ATG`, steps along in threes, and records the stretch if it reaches a stop and has at least a minimum number of codons before it. It only looks at the strand it is given. A full gene finder would repeat the search on the reverse complement and be far more careful (check).

```
import re
s = "ATGGCCATTGTAATGGGCCGATGA"
print(s.find("ATG"), s.find("ATG", 1), s.find("CCC"))
print([m.start() for m in re.finditer("ATG", s)])
print([m.start() for m in re.finditer("GG", "AGGGGA")])
print([m.start() for m in re.finditer("(?=GG)", "AGGGGA")])

def find_orfs(s, min_codons=2):
    stops = {"TAA", "TAG", "TGA"}
    out = []
    for m in re.finditer("(?=ATG)", s):
        i = m.start()
        for j in range(i, len(s) - 2, 3):
            if s[j:j+3] in stops:
                if (j - i) // 3 >= min_codons:
                    out.append((i, j + 3, s[i:j+3]))
                break
    return out
print(find_orfs(s))
```

```
0 12 -1
[0, 12, 20]
[1, 3]
[1, 2, 3]
[(0, 24, 'ATGGCCATTGTAATGGGCCGATGA'), (12, 24, 'ATGGGCCGATGA')]
```

Go through the output in order. The first line shows `find` returning 0, then 12 once we start searching from position 1, then -1 for a missing pattern. The second line shows all three `ATG` positions (0, 12 and 20). The third and fourth lines show the overlap difference: `[1, 3]` against `[1, 2, 3]`. The last line holds the two ORFs: the whole sequence from position 0 to 24 (eight codons, ending in the stop `TGA`), and a shorter one that starts at 12, in the same reading frame, and ends at the same stop. The `ATG` at 20 gives no ORF because only four letters remain, so no stop codon can be reached in its frame.

```
2 20 0.9
[(5, 'C', 'T'), (14, 'G', 'C')]
```

Two of 20 positions differ, so the sequences are 90% identical. The list gives the position (counting from 0) and the two letters. The first change, `C` to `T` at position 5, is in the third position of codon 2 (`GCC` becomes `GCT`), and both codons code for the same amino acid, alanine (check), a kind of change called silent (Genetics and Heredity Fundamentals 3). The second change, `G` to `C` at position 14, turns the codon `ATG` (methionine) into `ATC` (isoleucine), which does change the amino acid (check).

The Hamming distance has a weakness. If one letter is deleted, every letter after it shifts by one place, and the two strings look different almost everywhere. The **edit distance** (also called Levenshtein distance) counts the fewest single-letter insertions, deletions and substitutions needed to turn one string into the other, so it copes with shifts (check). It is computed by filling a table, row by row, keeping only the previous row in memory.

```
def edit_distance(a, b):
    prev = list(range(len(b) + 1))
```

```
    for i, x in enumerate(a, 1):
        cur = [i]
        for j, y in enumerate(b, 1):
            cur.append(min(prev[j] + 1, cur[j-1] + 1, prev[j-1] + (x != y)))
        prev = cur
    return prev[-1]

print(edit_distance("kitten", "sitting"))
print(edit_distance(a, b), edit_distance("ACGTACGT", "ACGACGT"), hamming("ACGTACGT", "AC
```

```
3
2 1 4
```

The first line is the textbook example, "kitten" to "sitting", which needs 3 edits. In the second line, the distance between our two DNA strings is 2, matching the Hamming distance. The last two numbers compare a deletion: `ACGTACGT` with the `T` removed becomes `ACGACGT`, so the edit distance is 1, while the Hamming distance (over the shorter length) is 4. Real aligners use refinements of this table, with different costs for different kinds of change (check).

k-mers and Sliding Windows

A **k-mer** is any run of `k` consecutive letters. Counting all the k-mers of a sequence is a fast way to summarise it, and it is the basis of many tools (check). A sequence of length `n` has `n - k + 1` k-mers, which overlap. A **sliding window** is the same trick applied to a measurement: slide a fixed-width window along the sequence and compute something, such as GC content, at each step. It shows where along a sequence a property rises and falls.

```
from collections import Counter

def kmers(s, k):
    return Counter(s[i:i+k] for i in range(len(s) - k + 1))

def window_gc(s, w, step):
```

```
    return [(i, round((s[i:i+w].count("G") + s[i:i+w].count("C")) / w, 2))
            for i in range(0, len(s) - w + 1, step)]

s = "ATATATGCGCATAT"
print(kmers(s, 2).most_common(3))
print(kmers(s, 3).most_common(2))

t = "ATATATATATGCGCGCGCGCATATATAT"
print(window_gc(t, 8, 4))
```

```
[('AT', 5), ('TA', 3), ('GC', 2)]
[('ATA', 3), ('TAT', 3)]
[(0, 0.0), (4, 0.25), (8, 0.75), (12, 1.0), (16, 0.5), (20, 0.0)]
```

The 2-mer `AT` occurs 5 times in the 14-letter string and the 3-mers `ATA` and `TAT` tie at 3. The window output reads as a profile: GC content is 0 at the start, climbs to 1.0 in the middle where the sequence is all `GC` repeats, and drops back to 0 at the end. In real genomes, dips and peaks in GC content can point to regions with a different history or function, though the meaning depends on the genome (check).

QUESTION	TOOL	PYTHON
What letters are present, and how many?	counting	`Counter(s)`
How GC-rich is it?	GC content	`(s.count("G") + s.count("C")) / len(s)`
Where does a motif occur?	search	`re.finditer("(?=...)", s)`
How many positions differ?	Hamming distance	`sum(x != y for x, y in zip(a, b))`
How different if letters shift?	edit distance	row-by-row table
How does a property vary along it?	sliding window	slicing in a `range` loop

TOY CODE, NOT A GENOME PIPELINE

These functions are fine for short sequences but would be far too slow for a whole genome. Real work uses specialised, heavily optimised tools and well-tested libraries (check). The aim here is to see what the tools are doing, not to replace them.

Challenges

EXERCISE 1

Write a FASTA string with four records, one of them in lower case and wrapped over two lines, and one containing the letter `N`. Parse it, then print each record's name, length and GC content. Skip any record that has letters other than A, C, G and T, saying which letters were the problem, and finish by printing the name of the record with the highest GC content.

View solution

EXERCISE 2

A site is a reverse-complement palindrome if it equals its own reverse complement. In a 32-letter sequence, find every 6-letter window that is such a palindrome and print the positions and the sites, then the count. Also print whether `GAATTC` is a palindrome, and the reverse complements of `GGATCC` and `ATGGCC`.

View solution

EXERCISE 3

With `random.Random(3)`, build a 60-letter reference sequence, then make three samples by changing 2, 5 and 9 different positions to a different base. Print the reference, then for each sample the Hamming distance, the identity and the list of changed positions. Finish with a table of the Hamming distances between every pair of samples.

View solution

QUICK REFERENCE

- **FASTA:** a name line starting with `>`, then the sequence over one or more lines
- **GC content:** the fraction of letters that are G or C
- **Reverse complement:** swap A with T and C with G, then reverse
- **Motif search:** `str.find` for one match, `re.finditer("(?=...)", s)` for every start, overlaps included
- **ORF:** from `ATG` in steps of three to the first stop codon
- **Hamming distance:** positions that differ; **edit distance:** fewest insertions, deletions and substitutions
- **k-mer and window:** overlapping runs of `k` letters; a fixed-width window slid along the sequence
- **Check:** assembly, FASTA details, GC meaning, ORF caveats and the silent change

COMING NEXT

Genetics and Heredity Fundamentals 10 is the capstone: testing, ancestry claims, ethics and how to read genetics in the news.

Genetics and Heredity Fundamentals

Course 1 · Chapter 10 · Capstone: Genetics in Everyday Life

This last chapter takes what the course built and points it at the places where most people actually meet genetics: a test result, an ancestry report, a news headline about a "gene for" something. The biology from Genetics and Heredity Fundamentals 1 to 9 gives you the vocabulary. What matters now is the arithmetic and the habits of thought that stop you being misled. All the programs here are Python 3.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run and the output shown is what it printed. The numbers in the examples are invented to make a point, not real test or disease figures. The biology and ethics are written from general knowledge; items marked (check) are the ones to verify first. Where a program uses random numbers it fixes the seed, so the same output appears each time. Nothing here is medical advice: for a real result, talk to a doctor or a genetic counsellor.

What a Genetic Test Can and Cannot Tell You

A genetic test looks for something specific: one changed letter, a missing piece, an unusual number of chromosomes, or a pattern across many positions at once. Different tests answer different questions (check).

- **Diagnostic:** does this person have the change that explains their symptoms?
- **Carrier:** does this healthy person carry one copy of a recessive allele (Genetics and Heredity Fundamentals 1 and 8)?
- **Screening:** a cheap test applied to many people to pick out those who need a closer look.
- **Predictive or risk:** does this person have variants that raise the chance of a condition later, without making it certain?

Two numbers describe how good a test is. **Sensitivity** is the share of people who have the thing that the test correctly flags. **Specificity** is the share of people who do not have it that the test correctly clears. A test with 99% of both sounds nearly perfect, and people often assume that a positive result then means a 99% chance of having the condition. That is wrong, and the reason is how common the condition is. The share of positive results that are real is the **positive**

predictive value, and it depends on the **prevalence**, how common the condition is in the group tested.

```
def ppv(prev, sens, spec):
    tp = prev * sens
    fp = (1 - prev) * (1 - spec)
    return tp / (tp + fp)

for prev in (0.5, 0.1, 0.01, 0.001):
    print(prev, round(ppv(prev, 0.99, 0.99), 3))

n = 100000
prev, sens, spec = 0.001, 0.99, 0.99
have = n * prev
true_pos = have * sens
false_pos = (n - have) * (1 - spec)
print(round(have), round(true_pos), round(false_pos))
print(round(true_pos / (true_pos + false_pos), 3))
```

```
0.5 0.99
0.1 0.917
0.01 0.5
0.001 0.09
100 99 999
0.09
```

The same test gives a 99% chance that a positive is real when half the group has the condition, but only 9% when one person in a thousand has it. The last three lines show why using "natural frequencies": of 100,000 people, 100 have the condition and the test finds 99 of them, but it also wrongly flags 1 in 100 of the 99,900 healthy people, which is 999 false alarms. So about 99 of every 1,098 positives are real. Rare conditions and screening tests are a poor mix for this reason, and it is why a positive screen is normally followed by a second, more specific test (check).

> **THE BASE RATE IS EASY TO FORGET**

> The common mistake is to remember the accuracy of the test and forget how rare the condition is. Before trusting a positive (or a negative) result, ask two questions: how accurate is the test, and how common is this in people like me? The answer to the second one can change the meaning of the first completely.

Ancestry Claims and the Shrinking Share

Ancestry reports compare your DNA with reference groups and give percentages, or list possible relatives (check). They rest on real genetics, but a few facts make the headline claims easy to misread. You inherit half your DNA from each parent, about a quarter from each grandparent, an eighth from each great-grandparent, and so on: the expected share from one ancestor `n` generations back is `1 / 2^n`. Crucially that is an *average*. Genetics and Heredity Fundamentals 4 showed that each chromosome is a random mixture, so the real share varies, and from far enough back a particular ancestor may have left you no DNA at all, even though they are certainly your ancestor.

The simplified model below pretends each of the 22 non-sex chromosomes is passed down whole, with no crossing over. At each generation a chromosome comes from one of the two parents with probability one half, so a chromosome comes from a chosen ancestor with probability `1 / 2^n`. The first table is the chance, by formula, that none of the 22 does. The second runs 1,000 simulated people for each of three depths.

```
import random
for n in range(1, 8):
    print(n, 1 / 2**n, round((1 - 1 / 2**n) ** 22, 3))

rng = random.Random(5)
def shared_fraction(n, chroms=22):
    hits = sum(1 for _ in range(chroms) if all(rng.random() < 0.5 for _ in range(n)))
    return hits / chroms

for n in (2, 4, 6):
    results = [shared_fraction(n) for _ in range(1000)]
    zero = sum(1 for r in results if r == 0) / 1000
    print(n, round(sum(results) / 1000, 4), round(1 / 2**n, 4), zero)
```

```
1 0.5 0.0
2 0.25 0.002
3 0.125 0.053
4 0.0625 0.242
5 0.03125 0.497
6 0.015625 0.707
7 0.0078125 0.842
2 0.2488 0.25 0.002
4 0.066 0.0625 0.234
6 0.0161 0.0156 0.7
```

In the second block the columns are the depth, the average simulated share, the formula's expected share, and the fraction of simulated people who got nothing from that ancestor. The averages match the formula. But at six generations back (the great-great-great-great-grandparents), about 70% of people would share no whole chromosome with a given ancestor. In real DNA, crossing over cuts chromosomes into pieces, so ancestors tend to leave many small segments rather than a few whole chromosomes, and the exact chances differ (check). The lesson holds either way: the further back you go, the less reliably DNA can confirm a particular ancestor, so a claim such as "you are descended from this famous person" is mostly unfalsifiable by the DNA alone.

CLAIM	WHAT IT CAN MEAN	QUESTION TO ASK
"You are 30% from region X"	your DNA resembles a reference group of people living there now	how was the reference group chosen, and how big is it? (check)
"A relative match"	a stretch of shared DNA larger than chance	how much is shared, and what range of relationships fits it? (check)
"Descended from a famous person"	perhaps a paper trail, rarely the DNA	what is the evidence apart from the DNA?
"No DNA from that ancestor"	quite possible many generations back	does it contradict the family records at all?

Risk Headlines: Relative and Absolute

"Gene variant doubles the risk of X" is a classic headline. A **relative risk** compares two groups: here, 2 means twice as likely. It says nothing about how likely X was in the first place. The **absolute risk** is the actual chance. The difference between the new and the old absolute risk is the extra risk, and its reciprocal is the number of people among whom one extra case is expected.

```
def risks(base, rr):
    new = base * rr
    return new, new - base

for base, rr in ((0.01, 2.0), (0.10, 2.0), (0.0001, 1.5)):
    new, extra = risks(base, rr)
    print(base, rr, round(new, 5), round(extra, 5), round(1 / extra))
```

```
0.01 2.0 0.02 0.01 100
0.1 2.0 0.2 0.1 10
0.0001 1.5 0.00015 5e-05 20000
```

The columns are the baseline risk, the relative risk, the new risk, the extra risk, and one extra case per that many people. "Doubles" turns a 1-in-100 risk into 2-in-100 (one extra case in 100 people), but turns a 1-in-10 risk into 2-in-10 (one in 10). A relative risk of 1.5 on a baseline of 1 in 10,000 means an extra case in 20,000 people. Note too that most common traits and conditions are **polygenic** (Genetics and Heredity Fundamentals 6) and depend on environment as well, so a single variant rarely behaves like the single gene of a pea. A "gene for" a complex trait is almost always shorthand for a small statistical nudge (check).

Ethics and Privacy

Genetic data is unusual because it is permanent, it is partly shared with relatives, and it can say things the person did not ask about. Several questions follow, and they do not have simple answers (check).

- **Privacy and consent.** Your result also says something about your parents, siblings and children, who did not agree to be tested. Who may see it, and who may keep it?
- **Discrimination.** Should an insurer or an employer be allowed to use genetic results? Laws differ between countries and change over time (check).
- **The right not to know.** Some people want to know their risk of a serious late-onset condition, and some do not. Tests can reveal findings that were not being looked for.
- **Children.** Testing a child for a condition that would only appear in adulthood takes the choice away from the adult they will become. Practice and guidance vary (check).
- **Fairness of the science.** Reference data has historically over-represented some groups, so tests and risk scores can work less well for others (check).
- **Reading too much into DNA.** Genetic determinism, the idea that genes alone fix traits, is not what the science says. Genes and environment work together, and a high-risk variant is not a sentence.

> **COUNSELLING EXISTS FOR A REASON**
>
> Many health services offer genetic counselling before and after a test, to explain what a result can and cannot say and what the options are. The code in this course is for understanding the arithmetic. It is not a substitute for that conversation (check what is offered where you live).

A Checklist for Reading Genetics in the News

Pulling the whole course together, here are the questions to put to a story before believing it.

1. **Who and how many were studied?** A small sample can give striking results by chance, as the coin and pea simulations in Genetics and Heredity Fundamentals 1 showed.
2. **Relative or absolute?** Find the baseline risk. "Doubles" of a tiny risk is still tiny.
3. **What does "gene for" mean here?** One gene, as with a pea colour, or a small nudge among many, as with height?
4. **Correlation or cause?** A variant being more common in a group does not by itself show the variant causes the trait (check).
5. **Which population?** Results from one group do not always carry over to another, because allele frequencies differ (Genetics and Heredity Fundamentals 8).

6. **How good is the test, and how common is the condition?** Remember the 9% positive predictive value above.
7. **Was it in humans, in mice, or in a dish?** Many headlines describe work in animals or cells (check).
8. **Is it peer reviewed, and has anyone repeated it?** A single study is a starting point.
9. **What are the limits the authors state?** Good reports and good papers say what they cannot show.

CHAPTER	THE IDEA YOU NOW HAVE	WHERE IT PROTECTS YOU
Genetics and Heredity Fundamentals 1	chance and sample size	small studies and "my family proves it"
Genetics and Heredity Fundamentals 2 and 3	DNA, genes and what proteins do	knowing what a "mutation" is and is not
Genetics and Heredity Fundamentals 4	shuffling by meiosis	ancestry claims and why siblings differ
Genetics and Heredity Fundamentals 6 and 7	beyond dominant and recessive, and sex-linked traits	"a gene for" headlines and pedigree questions
Genetics and Heredity Fundamentals 8	allele frequencies	carrier numbers and population claims
Genetics and Heredity Fundamentals 9	sequences as data	understanding what a test actually reads

Challenges

EXERCISE 1

A recessive allele has frequency `q` = 0.01 in a population of 200,000. Using Hardy-Weinberg, print the fraction of carriers and of affected people. A carrier test has sensitivity 0.95 and specificity 0.98. Print the number of carriers, true positives and false positives, and the share of positive results that are real. Finish by printing the chance that a random couple are both carriers, and the chance that a child of a random couple is affected.

View solution

EXERCISE 2

Write a headline decoder. Given a list of (headline, baseline risk, relative risk) tuples, for each print the headline, the new absolute risk, the extra risk, and "about 1 extra case per N people". Use three headlines with baseline risks 0.01, 0.20 and 0.0002, and print a warning if any new risk would be above 100%.

View solution

EXERCISE 3

With `random.Random(11)` and 5,000 simulated people per depth, use the whole-chromosome model to estimate, for 3 to 8 generations back, the share of people who inherit no chromosome from a chosen ancestor. Print it next to the formula's value, and finish by printing the first depth at which more than half of people share nothing.

View solution

QUICK REFERENCE

- **Sensitivity and specificity:** correct flags among those who have it; correct clears among those who do not
- **Positive predictive value:** the share of positives that are real; it falls when the condition is rare
- **Ancestry:** the expected share from an ancestor `n` generations back is `1 / 2^n`, but the real share varies and can be zero
- **Relative and absolute risk:** always find the baseline; extra risk is new minus baseline
- **Ethics:** privacy, discrimination, the right not to know, testing children, fairness of the data
- **In the news:** sample size, baseline, population, correlation or cause, repeated or not
- **Check:** test types, ancestry methods, the model's limits and the ethics and law details

COURSE COMPLETE

That is the end of Genetics and Heredity Fundamentals. You can now follow a trait from a pair of alleles through the chromosomes, the protein and the population, read a sequence as data, and ask

sensible questions of a genetics story. A good next step is to take one real health or ancestry article and run the checklist above against it, writing down your answers.