

PRIZE COURSE

Typography Fundamentals

A free prize course, generated with Claude.

8 Chapters | Philip Osztromok | Generated with Claude

TYPOGRAPHY FUNDAMENTALS

Contents

Chapter 1 Why Type Matters 2

Chapter 2 Families and Classes 11

Chapter 3 Measuring Text 22

Chapter 4 Hierarchy and Scale 34

Chapter 5 Pairing Typefaces 44

Chapter 6 Type on Screen 52

Chapter 7 Readability and Language 62

Chapter 8 Putting It Together 73

Typography Fundamentals 🏆

Course 1 · Chapter 1 · Text as a Designed Object

You read thousands of words a day: messages, signs, menus, packaging, web pages. Almost none of it was left to chance. Somebody chose the shape of the letters, how big they are, how far apart the lines sit and how wide the column is, and those choices decide whether you read the thing happily, struggle through it, or skip it. That craft is called **typography**. The word comes from Greek roots meaning "impression" and "writing" (check), a leftover from the days when letters were pressed into paper from metal blocks.

This course is for the curious beginner. You do not need design experience. Because you are also a web-development student, we will keep tying ideas back to CSS, but the ideas belong to paper and signs as much as to screens. This first chapter does three things: it shows that text is *designed*, it teaches the vocabulary for talking about a single letter, and it peeks inside a letter to see what a font really contains.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> The two Python scripts in this chapter were run on Windows with Python 3.14.0 and **fontTools 4.63.0**, reading the fonts that come with Windows (Times New Roman, Georgia, Arial, Consolas). The output shown is exactly what they printed. Your numbers can differ if your copies of those fonts are different versions. The CSS snippets and the live samples were **not** measured in several browsers: how a given browser draws them is the thing to check on your own screen.

Text Is Designed, Even When You Do Not Notice

Here is the same sentence set three ways. Nothing about the words changes. Everything about how it feels does.

A: A NOTICE ON A SHOP DOOR

Closed for stocktaking. We will open again on Monday.

Serif letters, comfortable size, calm.

B: A WARNING SIGN

CLOSED FOR STOCKTAKING. WE WILL OPEN AGAIN ON MONDAY.

Heavy, capitals, tight lines: it shouts.

C: A TERMINAL WINDOW

```
Closed for stocktaking. We will open
again on Monday.
```

Every letter the same width: it sounds technical.

The point is that **type carries a voice** before anyone has read a word. A reader decides in an instant whether a page looks trustworthy, friendly, urgent or boring, and typography is a large part of that. It is also a practical craft: the right choices simply make text easier on the eyes.

Reading Versus Looking

There are two jobs text can do, and good designers know which one they are asking of it.

	LOOKING (DISPLAY)	READING (TEXT)
Examples	A shop sign, a poster headline, a logo, a book title on the cover	A novel's pages, a news article, a manual, a long web page
What the reader does	Takes it in at a glance, from a distance, in a second or two	Settles in and moves along line after line, for minutes or hours
What it needs	Personality, impact, a shape you remember	Evenness, comfort, letters that disappear so the meaning shows through
Typical choices	Large sizes, heavy or unusual shapes, tight spacing	Moderate sizes, regular weights, generous spacing, a sensible line length
CSS you would reach for	Big `font-size`, `font-weight: 800`, a display font	`line-height: 1.5`, a limited `max-width`, a plain body font

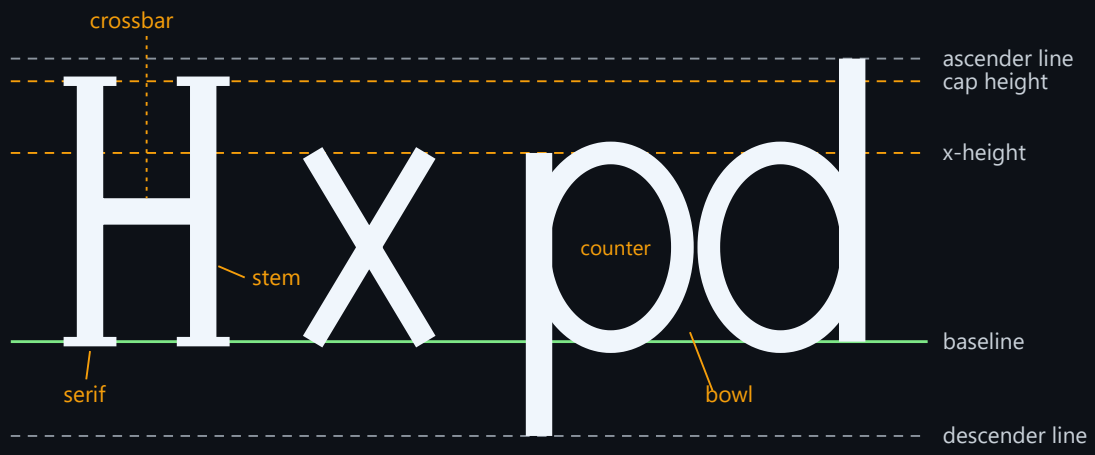
Two words are worth separating now, because they get mixed up. **Legibility** is whether individual letters can be told apart (is that an `l`, a capital `I` or a `1`?). **Readability** is how comfortable a whole stretch of text is to read: the size, spacing, line length and contrast working together. A letter-perfect font can still give unreadable text if the lines are far too long. Later chapters, especially Typography Fundamentals 3 and Typography Fundamentals 7, take both seriously.

COMMON MISTAKE

> Choosing a typeface because it looks interesting in a big heading, then using it for a whole page of body text. Display letters are built to be looked at, not read for ten minutes. If the text is long, pick something plain and let the layout do the talking.

The Vocabulary of a Letter

To talk about type you need names for its parts, the same way you need names for the parts of a bicycle before you can say what is wrong with it. Almost every term comes from a set of invisible horizontal lines that letters are built against.



A schematic: the letters are drawn as simple strokes so that the lines line up exactly, not copied from any real font.

TERM	WHAT IT IS	WHERE YOU MEET IT
Baseline	The invisible line most letters sit on, like writing on ruled paper	Mixing sizes on a line: `vertical-align: baseline` and aligning text across columns
x-height	The height of a plain lowercase letter such as `x`, measured from the baseline	The reason two fonts at the same `font-size` look different sizes; CSS even has an `ex` unit based on it
Cap height	The height of a capital letter such as `H`, measured from the baseline	How tall capitals look next to lowercase; CSS has a `cap` unit (browser support: check)
Ascender	The part of a letter that rises above the x-height: the stem of `d`, `b`, `h`, `l`	Often a little taller than the capitals, though not in every font
Descender	The part that drops below the baseline: `p`, `g`, `y`, `q`	The reason lines of text need some room between them: descenders must not hit the ascenders below

TERM	WHAT IT IS	WHERE YOU MEET IT
Counter	The enclosed or partly enclosed empty space inside a letter: the hole in `o`, `d`, `p`	Open, generous counters keep small text legible; tiny closed ones fill in
Bowl	The curved stroke that makes a counter	The round part of `p`, `d`, `b`
Stem	The main upright stroke of a letter	The vertical of `H`, `l`, `p`
Crossbar	A horizontal stroke linking two others	The bar of `H`, `A`, `e`, `t`
Serif	A small finishing stroke at the end of a main stroke	The reason for the words "serif" and "sans-serif" (sans is French for "without")
Stroke	Any line that makes up a letter; its thickness is the stroke weight	`font-weight` is a request for thicker or thinner strokes

A quick test of whether the vocabulary has stuck: find the letters in this sentence that have a descender, and the letters that have an ascender. (Descenders: the `g`, `y`, `p` and `q` kind. Ascenders: `b`, `d`, `f`, `h`, `k`, `l`. The dot of `i` and `j` counts for neither.) You will never read a page the same way again.

> **WHY X-HEIGHT MATTERS MOST**
>
> Most letters in any line of text are lowercase, so your eye judges "how big is this text?" mostly by the x-height, not by the capitals or the em. That is the key idea behind the next section: `font-size` does not directly tell you how big the letters look.

Measure It: Same Size, Different Letters

A font file stores these measurements. The script below opens four fonts that ship with Windows and prints their x-height and cap height, as a fraction of the **em** (the font's own unit of size: when you ask for `font-size: 16px` you are really asking for an em of 16px). It uses the free `fontTools` package (`py -m pip install fonttools`).

```
"""Read the vertical measurements of a few installed fonts and compare them.

    py font_metrics.py
"""
```

```python
from pathlib import Path
from fontTools.ttLib import TTFont

FONTS = {
    "Times New Roman": "times.ttf",
    "Georgia": "georgia.ttf",
    "Arial": "arial.ttf",
    "Consolas": "consola.ttf",
}
FONT_DIR = Path("C:/Windows/Fonts")

print(f"{'font':<16}{'units/em':>9}{'x-height':>10}{'cap':>6}{'x/em':>7}{'cap/em':>8}")
for name, filename in FONTS.items():
    font = TTFont(FONT_DIR / filename)
    em = font["head"].unitsPerEm
    os2 = font["OS/2"]
    x_height, cap_height = os2.sxHeight, os2.sCapHeight
    print(f"{name:<16}{em:>9}{x_height:>10}{cap_height:>6}{x_height / em:>7.3f}{cap_heig
```

```
font             units/em  x-height   cap   x/em  cap/em
Times New Roman      2048       916  1356  0.447   0.662
Georgia              2048       986  1419  0.481   0.693
Consolas             2048      1004  1307  0.490   0.638
```

Read the `x/em` column. At the same font size, Arial's lowercase letters are about **16% taller** than Times New Roman's (0.519 against 0.447). Same "16px", noticeably different letters. This is the single most common reason a font swap makes a page look suddenly bigger or smaller, and why a design that uses two fonts side by side often has to nudge one of their sizes. (Exercise 3 does the sum for you.)

You can see the effect live. These two lines are both set at exactly the same size:

TIMES NEW ROMAN, 22PX

Handsome sphinx of black quartz

ARIAL, 22PX

Handsome sphinx of black quartz

On a machine without those fonts the browser substitutes others and the difference may shrink or vanish: that is a lesson in itself (Typography Fundamentals 6 returns to fallbacks).

A First Look Inside a Glyph

A single letter as drawn in a font is called a **glyph**. A **character** is the idea ("the lowercase letter a"); a glyph is one particular picture of it, and one font can hold several glyphs for the same character (an italic and a regular `a` look different). A **font** is a collection of glyphs plus the measurements and rules that go with them; a **typeface** is the design as a whole, which can come in many fonts (regular, bold, italic). You will see the typeface/font distinction used loosely everywhere, including on this site.

Inside a modern font, a glyph is not a picture of pixels. It is an **outline**: a closed loop (or several) made of points joined by straight lines and smooth curves, drawn inside an invisible box called the **em square**. Because it is a set of instructions rather than a bitmap, the same glyph can be scaled to a tiny footnote or a billboard without going blocky. The em square is a grid of "font units" (2048 per em in the fonts above; fonts in the other common style use 1000 (check)), so the x-height of 1062 for Arial really means "1062 units up from the baseline, out of 2048".

Each loop is a **contour**. A letter like `l` needs one; a letter like `o` needs two: an outer loop and an inner loop for the counter. Curves are drawn with **control points**, which sit *off* the outline and pull it into a curve, as a magnet would. Let us count them for four letters:

```
"""Look inside the outline of a few letters of one font: how many contours, how many poi

    py glyph_outline.py
"""
from pathlib import Path
from fontTools.ttLib import TTFont

font = TTFont(Path("C:/Windows/Fonts/arial.ttf"))
cmap = font.getBestCmap()
glyf = font["glyf"]

print("Arial, units per em:", font["head"].unitsPerEm)
print(f"{'letter':<8}{'contours':>9}{'points':>8}{'off-curve':>11}")
for letter in "lHog":
    glyph = glyf[cmap[ord(letter)]]
    coords, end_pts, flags = glyph.getCoordinates(glyf)
    off_curve = sum(1 for f in flags if not f & 1)
    print(f"{letter:<8}{glyph.numberOfContours:>9}{len(coords):>8}{off_curve:>11}")
```

```
Arial, units per em: 2048
letter   contours  points  off-curve
l               1       4          0
H               1      12          0
o               2      26         17
g               2      43         25
```

Read it as a story. The `l` is a plain rectangle: one contour, four corner points, no curves. The `H` is one contour too (the outline wraps around the whole letter shape) with twelve corners and still no curves. The `o` has **two contours**, the outer edge and the counter, and most of its 26 points (17) are off-curve control points: curves cost points. Arial's `g` also has two contours (its outer shape and the counter of its bowl) and is the most complicated of the four with 43 points. Counting points is a crude measure of drawing effort, but it shows that "simple" letters are not all equally simple.

> **WHAT THIS TELLS YOU**
>
> The off-curve points are the ones this particular file uses for TrueType-style curves (quadratic curves). Another common outline format uses cubic curves with different control-point rules, so the same letter can have a different point count in another font file even if it looks identical. The measurements, the contours and the counters are the concepts to keep; the exact numbers belong to one file.

Print and Screen: The Same Ideas, Different Names

IDEA	IN PRINT	IN CSS
Size of the em	"Point size" (a point is 1/72 of an inch, in the common computer definition (check))	`font-size`: `px`, `rem`, `em`, `pt`
Strokes thicker or thinner	Choosing a bold or light font	`font-weight`
Space between lines	Leading	`line-height`
Which design to use	Choosing a typeface	`font-family`
How wide a column is	Measure (line length)	`max-width` on the text container

The CSS side of a few of these, in miniature:

```css
/* A readable body-text starting point. Each value is explained in later chapters. */
body {
  font-family: Georgia, "Times New Roman", serif;
  font-size: 1.125rem;
  line-height: 1.5;
  max-width: 40rem;
}

/* A display heading: looked at, not read. */
h1 {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 2.5rem;
  font-weight: 800;
  line-height: 1.1;
}
```

Notice the two jobs from earlier at work: the body asks for comfort (a moderate size, generous `line-height`, a limited width) while the heading asks for impact (large, heavy, tight). Those numbers are starting points, not rules. Typography Fundamentals 3 explains how to choose them properly.

Hands-On Exercises

Three exercises: a code task, a look-around-you design task, and a CSS task that connects them. Each has a solution file that opens in a new tab.

EXERCISE 1

Extend `font_metrics.py` to cover seven fonts (add Verdana, Calibri and Courier New) and rank all of them by x-height as a fraction of the em, largest first. Before you run it, predict which font will have the largest x-height and which the smallest.

View solution

EXERCISE 2

Pick one printed object near you (a packet, a bus timetable, a book cover). List each separate piece of text on it, say for each whether it is meant to be *read* or *looked at*, and name what the designer did to make that so. Use at least three terms from the vocabulary table.

View solution

EXERCISE 3

Make a small HTML page that shows the same sentence in Georgia and Arial at 18px. Using the x-height numbers from this chapter, work out what size Georgia needs to be to match Arial's lowercase height, and add a third line at that size. Then look up `font-size-adjust` as a way to let the browser do it.

View solution

CHAPTER 1 QUICK REFERENCE

- **Type has a voice** before it is read: the same words set three ways say three different things
- **Looking versus reading:** display text is built for impact at a glance; body text for comfort over time
- **Legibility** = letters can be told apart; **readability** = a stretch of text is comfortable to read
- **The lines:** baseline, x-height, cap height, ascender line, descender line
- **The parts:** stem, bowl, counter, crossbar, serif, stroke, ascender, descender
- **x-height decides how big text looks:** at the same `font-size` Arial's lowercase is about 16% taller than Times New Roman's (0.519 against 0.447 of the em)
- **A glyph is an outline** of contours (loops) in an em square; counters are inner contours, and curves use off-curve control points

Typography Fundamentals

Course 1 · Chapter 2 · Type Families and How They Are Sorted

In Typography Fundamentals 1 you learned the parts of a single letter and saw that a font is a collection of glyphs with measurements attached. Now we zoom out. There are tens of thousands of typefaces in the world (check), and nobody can hold them all in their head. Designers cope the way librarians do: by sorting them into a handful of **families and classes**, so that "a transitional serif" or "a monospace" is enough to know roughly what is meant. This chapter gives you that sorting system, the words for how one typeface comes in many weights and widths, the difference between italic and oblique, and a short tour of how we got from metal blocks to font files.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> The two Python scripts in this chapter were run on Windows with Python 3.14.0 and **fontTools 4.63.0**, reading the fonts that come with Windows. The output shown is exactly what they printed. Your numbers can differ if your copies of those fonts are different versions. The CSS snippets and the live samples were **not** measured in several browsers; if a sample looks different on your screen, the likely reason is that your system substitutes other fonts.

The Six Everyday Families

Start with the broad groups you already meet every day. Each specimen below is the same short phrase, so only the letter shapes change. (These are live text samples, so they use whatever fonts your machine has: on a machine without them the browser falls back to something similar.)

SERIF

Quiet Harbour Books

Small finishing strokes on the letters. Books, newspapers, formal documents.

SANS-SERIF

Quiet Harbour Books

No serifs. Signs, interfaces, modern branding.

SLAB SERIF

Quiet Harbour Books

SCRIPT

Quiet Harbour Books

Thick, blocky serifs. Posters, headlines, a sturdy feel.

Imitates handwriting. Invitations and logos; poor for long text.

DISPLAY

Quiet Harbour Books

Made to be looked at in large sizes, with a lot of personality.

MONOSPACE

`Quiet Harbour Books`

Every letter the same width. Code, terminals, typewriter looks.

These groups overlap and are not scientific. A slab serif is really a kind of serif; "display" describes a *job* (Typography Fundamentals 1's "looking" text) rather than a shape; "script" is a handwriting *style*. They are working labels, and that is exactly what makes them useful: they tell you what to expect before you have seen the letters.

FAMILY	GOOD FOR	CAREFUL WITH	CSS GENERIC NAME
Serif	Long reading on paper; a traditional, trustworthy tone	Very small sizes on low-resolution screens (Typography Fundamentals 6)	`serif`
Sans-serif	Interfaces, signs, short text, clean modern looks	Letters that look alike (`I` , `l` , `1`)	`sans-serif`
Slab serif	Headlines, posters, a firm and friendly voice	Dense paragraphs: the heavy serifs add visual noise	`serif` (no separate generic name)
Script	Invitations, signatures, a single word or two	All-capitals (hard to read), long paragraphs	`cursive`
Display	Titles, posters, logos	Body text, small sizes, anything long	`fantasy` (loosely)
Monospace	Code, tables of figures, a typewriter feel	Long prose: wide, uneven-looking letters slow reading	`monospace`

CSS has a short list of **generic family names**: `serif` , `sans-serif` , `monospace` , `cursive` , `fantasy` , plus newer ones such as `system-ui` (check). They are the safety net at the end of every `font-family` list. If none of your named fonts is installed, the browser picks *some* font of that kind. Which one it picks varies by device, so never rely on a generic name for the exact look.

COMMON MISTAKE

Treating `cursive` and `fantasy` as if they gave a predictable result. Each browser and operating system maps them to a different font, or sometimes to nothing special at all (check). They are fallbacks, not design choices. If you want a script look, name a specific script font (or load one: Typography Fundamentals 6), and put `cursive` only at the end of the list.

Measuring a Family: Fixed Width Versus Proportional

One family boundary can be measured rather than judged by eye: monospaced versus proportional. In a **proportional** font a narrow letter such as `i` takes less room than a wide one such as `m`. In a **monospace** font every glyph gets the same advance width. A font file stores each glyph's advance width in its `hmtx` table, so we can just look. The script compares `i`, `m` and `W` in four fonts.

```
"""Is a font monospaced? Compare the width of a narrow letter and a wide one.

    py widths.py
"""
from pathlib import Path
from fontTools.ttLib import TTFont

FONTS = {
    "Times New Roman": "times.ttf",
    "Arial": "arial.ttf",
    "Courier New": "cour.ttf",
    "Consolas": "consola.ttf",
}
FONT_DIR = Path("C:/Windows/Fonts")

print(f"{'font':<16}{'i':>6}{'m':>6}{'W':>6}  {'m / i':>6}  fixed-pitch flag")
for name, filename in FONTS.items():
    font = TTFont(FONT_DIR / filename)
    cmap = font.getBestCmap()
    hmtx = font["hmtx"]
    w = {ch: hmtx[cmap[ord(ch)]][0] for ch in "imW"}
    flag = font["post"].isFixedPitch != 0
    print(f"{name:<16}{w['i']:>6}{w['m']:>6}{w['W']:>6}  {w['m'] / w['i']:>6.2f}  {flag}
```

```
font                i     m     W   m / i  fixed-pitch flag
Times New Roman    569  1593  1933    2.80  False
```

```
Arial              455  1706  1933    3.75  False
Courier New       1229  1229  1229    1.00  True
Consolas          1126  1126  1126    1.00  True
```

The widths are in font units out of 2048 per em, as in Typography Fundamentals 1. Look at the `m / i` column. In Arial an `m` is **3.75 times** as wide as an `i`; in Times New Roman it is 2.80 times. In Courier New and Consolas the ratio is exactly 1.00: the narrow `i` is given the same box as the wide `m`, which is why monospace text has those wide gaps around the `i` and `l`. The last column is a flag the font author sets to tell programs "this is fixed-width", and it agrees with the measurement. A programmer needs it so columns of code line up; a reader of long prose is better served by proportional letters.

Sorting the Serifs: Old-Style, Transitional, Didone

Serif typefaces are the oldest family, and they are often sorted by *when* their design habits developed. Three classes are worth knowing. The key idea is **stroke contrast**: the difference between the thick and thin parts of a letter, and the **stress**, the angle of the axis along which the thinnest part falls. A pen held at an angle makes thick and thin strokes naturally; type designers copied that, then slowly exaggerated or removed it.

CLASS	CONTRAST	STRESS	SERIFS	ROUGHLY FROM	EXAMPLES (CHECK)
Old-style (humanist)	Low to moderate	Tilted, like a pen held at an angle	Bracketed (curved into the stem), often slanted on lowercase	15th and 16th centuries (check)	Garamond, Caslon, Palatino
Transitional	Moderate to high	Nearly vertical	Bracketed, flatter	18th century (check)	Baskerville, Times New Roman
Didone (modern)	Very high	Vertical	Hairline-thin, flat, unbracketed	Late 18th century (check)	Bodoni, Didot

The name "didone" is a blend of two designers' surnames, Didot and Bodoni (check). "Transitional" sits between old-style and didone, which is exactly what the word says. Here are live samples, set large enough to see the contrast. If your machine lacks Garamond or Bodoni,

the browser will substitute a different serif, which tells you something about how much a font choice depends on what is installed.

OLD-STYLE

Rhythm & Cadence

Gentle, tilted stress; the thick and thin parts blend.

TRANSITIONAL

Rhythm & Cadence

More upright, sharper contrast.

DIDONE

Rhythm & Cadence

Hairline thins, heavy thicks, vertical stress.

WHAT THIS TELLS YOU

More contrast gives more sparkle at large sizes and more fragility at small ones. Hairline strokes can almost vanish on a low-resolution screen or in a cheap print, which is one reason high-contrast didones are usually used for headlines and fashion titles rather than small body text. Contrast is a dial, and each class sits at a different setting.

Sans-Serif Gets Its Own Classes

Sans-serifs are sorted too, in a similar spirit. Four loose groups are enough for now:

- **Grotesque**: the early sans-serifs, a little irregular, with some stroke contrast (Akzidenz-Grotesk, Franklin Gothic (check)).
- **Neo-grotesque**: tidier and more uniform, the "neutral" look of much 20th-century signage and branding (Helvetica, Arial (check)).
- **Geometric**: built from circles and straight lines, so an `o` is almost a perfect round (Futura, Avenir (check)).
- **Humanist**: sans-serifs with proportions and stroke variation borrowed from handwriting and old-style serifs, typically very readable (Gill Sans, Frutiger, Verdana (check)).

The labels are fuzzy and people disagree about where a given typeface belongs. Do not memorise the names; learn to notice *why* two sans-serifs feel different (round or squared off, uniform or varying in stroke, tall or short x-height).

One Typeface, Many Fonts: Weight, Width and Style

A single typeface usually ships as a set of fonts that vary in a few controlled ways. Learning the three axes saves you from guessing:

AXIS	WHAT CHANGES	NAMES YOU WILL SEE	CSS
Weight	Stroke thickness	Thin, Light, Regular, Medium, Semibold, Bold, Black	`font-weight: 100` to `900`
Width	How wide the letters are set	Condensed, Normal, Extended	`font-stretch` (as a percentage or keyword)
Style	Upright or slanted	Roman, Italic, Oblique	`font-style: normal | italic | oblique`

CSS weights run in hundreds: `400` is "normal", `700` is "bold", and keywords map onto those two numbers. Inside the font file the same number is stored as the "weight class". The script below reads it, along with the style flag and the **italic angle**, for eight font files that ship with Windows.

```
"""Read the style flags stored inside a few installed font files.

    py font_styles.py
"""
from pathlib import Path
from fontTools.ttLib import TTFont

FILES = [
    "arial.ttf", "arialbd.ttf", "ariali.ttf", "ariblk.ttf",
    "times.ttf", "timesi.ttf", "georgia.ttf", "georgiai.ttf",
]
FONT_DIR = Path("C:/Windows/Fonts")

print(f"{'file':<14}{'family':<18}{'style':<14}{'weight':>7}{'italic angle':>14}")
for filename in FILES:
    path = FONT_DIR / filename
```

```
    if not path.exists():
        print(f"{filename:<14}(not found)")
        continue
    font = TTFont(path)
    name = font["name"]
    family = name.getDebugName(1)
    style = name.getDebugName(2)
    weight = font["OS/2"].usWeightClass
    angle = font["post"].italicAngle
    print(f"{filename:<14}{family:<18}{style:<14}{weight:>7}{angle:>14.1f}")
```

```
file          family            style          weight  italic angle
arial.ttf     Arial             Regular           400           0.0
arialbd.ttf   Arial             Bold              700           0.0
ariali.ttf    Arial             Italic            400         -12.0
ariblk.ttf    Arial Black       Regular           900           0.0
times.ttf     Times New Roman   Regular           400           0.0
timesi.ttf    Times New Roman   Italic            400         -16.3
georgia.ttf   Georgia           Regular           400           0.0
georgiai.ttf  Georgia           Italic            400         -13.0
```

Four things stand out. First, the weight numbers match CSS: Arial Bold is `700`. Second, **Arial Black is a separate family**: it reports weight 900 but its family name is "Arial Black", with style "Regular". Font families are not always tidy, which is why `font-weight: 900` on Arial may not reach it. Third, the italic angles are negative because slanting to the right is measured as a clockwise lean (check); the angle differs between typefaces (12, 16.3 and 13 degrees here), so there is no single "italic slant". Fourth, the weight stays at 400 for the italics: italic is a *style*, not a different weight.

Italic Versus Oblique

A true **italic** is a separately drawn companion typeface. The letters change shape: an italic `a` usually becomes a single-storey `a`, and the whole face often becomes narrower and more cursive. An **oblique** is simply the upright letters tilted, with the shapes left alone. Sans-serif families often use obliques; serif families usually have true italics (check). If a family has no italic at all, a browser will slant the regular letters for you, which is called **faux (synthesised) italic**, and type designers do not love the result. The same goes for faux bold, where the browser thickens strokes artificially.

UPRIGHT

a pleasant gate

TRUE ITALIC

a pleasant gate

Look at the shape of the *a* and *g*, not just the slant.

SLANTED UPRIGHT (OBLIQUE)

a pleasant gate

Browsers may draw this as a tilt of the regular letters (check).

The CSS keeps the two ideas apart, though in practice a browser often treats `oblique` as a request for the font's italic if no oblique exists:

```
/* A family with the usual style axes, as plain CSS. */
.quote {
  font-family: Georgia, "Times New Roman", serif;
  font-style: italic;            /* the true italic, if the font has one */
}

.label {
  font-family: Arial, Helvetica, sans-serif;
  font-weight: 600;              /* semibold, if available; else the nearest weight */
  font-stretch: 85%;             /* a slightly condensed width, only if the font has one
}

.no-fakes {
  font-synthesis: none;          /* never fake bold or italic: show what the font really
}
```

Weights are asked for, not guaranteed. If you request `600` and the family only has 400 and 700, the browser applies a fixed matching rule and picks one of them (check the exact rule on MDN). That is why a design may show "bold" in two places that were meant to look different.

READ BEFORE YOU RELY ON IT

The browser support for `font-stretch` percentages and for the newer `oblique` angle form varies by font and browser (check). If a family does not offer a width axis, the property will do nothing. Test on

the fonts you actually ship.

A Short History: From Metal to Files

Why do these classes exist? Because each one is a trace of how letters were made at the time. A very short timeline (dates are approximate and worth checking before you quote them):

PERIOD	WHAT CHANGED	EFFECT ON LETTERS
Mid-1400s (check)	Movable metal type arrives in Europe, associated with Johannes Gutenberg in Mainz (check)	The first printed letters imitate the hand-written "blackletter" of manuscripts
Late 1400s to 1500s (check)	Printers in Italy and France develop rounder, lighter "roman" letters; italic appears as a space-saving companion	The old-style serif class takes shape
1700s (check)	Better paper, ink and printing let designers use finer strokes	Transitional, then didone letterforms with more and more contrast
Early 1800s (check)	Printing for advertising demands bigger, bolder letters; the first sans-serifs and slab serifs appear (check)	Display faces, heavy grotesques, slabs
1900s (check)	Industrial typesetting machines; the geometric and neo-grotesque sans-serifs of the mid-century arrive (check)	The clean, uniform "modern" look
1980s and 1990s (check)	Desktop publishing; fonts become software; outline formats (such as TrueType and OpenType) spread	Anyone can make, mix and misuse fonts
2000s to now (check)	Web fonts and, later, variable fonts that hold a whole range of weights and widths in one file	Typography becomes a thing you load on every page (Typography Fundamentals 6)

The pattern to notice: **technology and fashion keep moving the letterforms.** Metal punches limited how thin a serif could be; smoother paper allowed hairlines; sans-serif sheds serifs because they were hard to see at distance on signs and posters; monospace exists because typewriters moved the paper a fixed distance for every key; screens now ask for generous x-

heights and open counters. When someone says a typeface "feels old-fashioned" or "feels modern", they are reading these histories whether they know it or not.

Hands-On Exercises

Three exercises: a code task that measures your own fonts, a written design task that classifies typefaces by description, and a CSS task that puts the families and styles to work. Each has a solution file that opens in a new tab.

EXERCISE 1

Write a script that scans every `.ttf` file in `C:/Windows/Fonts` and uses the `post` table's fixed-pitch flag to list the family names of the monospaced fonts it finds, each only once (a family has several files). Then count how many different families the folder contains, and what fraction are monospaced. Skip any file that cannot be read, and say how many you skipped.

View solution

EXERCISE 2

Sort these six descriptions into a family or class (serif, sans-serif, slab, script, display, monospace; and for serifs, old-style, transitional or didone), and say which single detail of each description decided it. (a) Hairline thins, thick vertical stems, perfectly vertical stress, used on a fashion magazine title. (b) Letters built from near-perfect circles, no serifs, used on a minimalist poster. (c) A heavy, blocky face with thick rectangular serifs on a farm-machinery logo. (d) A narrow, tilted-stress book face with bracketed, slanted serifs. (e) A bank statement's account numbers where every digit is the same width so columns line up. (f) A wedding invitation written to look like pen strokes.

View solution

EXERCISE 3

Build a one-page **type family sampler** in HTML and CSS. Show one sentence in each of the six everyday families, each with a sensible fallback stack ending in the right generic name. Add a row that shows one serif family in weights 400 and 700 and in normal and italic style, and a final row that sets `font-synthesis: none` on a family that has no italic to see what the browser does. Write one sentence, as a comment, about what you notice.

View solution

CHAPTER 2 QUICK REFERENCE

- **Everyday families:** serif, sans-serif, slab serif, script, display, monospace. They are working labels: they overlap and describe shape, style or job
- **CSS generic names:** `serif`, `sans-serif`, `monospace`, `cursive`, `fantasy`: always the last item in a `font-family` list, and never a precise design choice
- **Monospace versus proportional:** in a monospace font every glyph has the same advance width (Arial's `m` is 3.75 times its `i`; in Consolas it is 1.00)
- **Serif classes:** old-style (tilted stress, low contrast), transitional (nearly vertical, moderate contrast), didone (vertical, very high contrast)
- **Weight, width, style** are the three axes of a family: `font-weight` 100 to 900, `font-stretch`, `font-style`
- **Italic versus oblique:** a true italic is a redrawn companion; an oblique is the upright letters tilted; browsers may fake either (`font-synthesis`)
- **History explains the classes:** each style is a trace of the tools and paper of its time

Typography Fundamentals

Course 1 · Chapter 3 · Measuring and Setting Text

In Typography Fundamentals 2 you learned to sort typefaces into families and classes. Choosing a typeface is only half the job, though: the same font can look elegant or exhausting depending on how it is **set**. Setting text means deciding how big the letters are, how long each line runs, how far apart the lines sit, and how tightly the letters are packed. Those four dials are what make a paragraph feel comfortable. This chapter gives you the units to measure them, the names of the dials, and sensible starting values for body text.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> The four Python scripts in this chapter were run on Windows with Python 3.14.0 and **fontTools 4.63.0**, reading the fonts that come with Windows. The output shown is exactly what they printed. The wrapping script uses a simple greedy line-breaker, not a real browser layout engine, so treat its line counts as estimates. The CSS snippets and live samples were **not** measured across several browsers.

The Units: Points, Picas, Pixels, Ems and Rems

Type has been measured since long before screens existed, so you meet two families of units. **Print units** come from metal type: the **point** (pt) and the **pica** (pc), where 1 pica is 12 points. **Screen units** are the CSS ones: the **pixel** (px), and the two relative units **em** and **rem** that scale with a font size rather than being fixed.

CSS defines these so that they fit together: 1 inch is 96px, and also 72pt, so 1pt is 1.333px and 1pc is 16px (check on MDN for the precise wording). Note that a CSS "pixel" is a reference unit and not always one physical dot on the screen, because high-density displays squeeze several device dots into one CSS pixel (check). So "16px" means a consistent, readable size across devices rather than a number of dots.

UNIT	KIND	WHAT IT IS RELATIVE TO	TYPICAL USE
pt	Absolute	1/72 inch (CSS: 1.333px)	Print stylesheets, word processors
pc	Absolute	12pt (CSS: 16px)	Page layout, column widths in print

UNIT	KIND	WHAT IT IS RELATIVE TO	TYPICAL USE
px	Absolute (reference)	The CSS reference pixel	Borders, hairlines, fine details
em	Relative	The font size of the *element itself* (and, for `font-size`, the parent's)	Spacing that should scale with the text
rem	Relative	The font size of the *root* element	Font sizes and layout widths that follow the user's setting
ch	Relative	The width of the digit `0` in the element's font	Line length (the measure)
%	Relative	The parent's value	Fluid widths; `line-height` and `font-size` percentages

A small script prints the same sizes in several units, and shows the one trap with `em`: it **compounds**. If each nested element says `font-size: 1.2em`, every level is 20 percent bigger than its parent, so the sizes grow quickly.

```
"""CSS absolute units side by side, and rem/em against a root size.

    py units.py
"""
PX_PER = {"in": 96, "pc": 16, "pt": 96 / 72, "px": 1}   # CSS fixes these ratios
ROOT_PX = 16                                             # browser default for 1rem

print(f"{'size':<10}{'px':>8}{'pt':>8}{'pc':>8}{'rem':>8}")
for px in (12, 14, 16, 18, 20, 24, 32, 48):
    pt = px / PX_PER["pt"]
    pc = px / PX_PER["pc"]
    rem = px / ROOT_PX
    print(f"{str(px) + 'px':<10}{px:>8}{pt:>8.1f}{pc:>8.2f}{rem:>8.3f}")

print()
print("Nested em: each level multiplies the parent's size")
size = float(ROOT_PX)
for level in range(1, 5):
    size *= 1.2
    print(f"  level {level}: font-size 1.2em -> {size:.2f}px")
```

```
size            px      pt      pc     rem
12px            12     9.0    0.75   0.750
14px            14    10.5    0.88   0.875
16px            16    12.0    1.00   1.000
18px            18    13.5    1.12   1.125
20px            20    15.0    1.25   1.250
24px            24    18.0    1.50   1.500
32px            32    24.0    2.00   2.000
48px            48    36.0    3.00   3.000

Nested em: each level multiplies the parent's size
  level 1: font-size 1.2em -> 19.20px
  level 2: font-size 1.2em -> 23.04px
  level 3: font-size 1.2em -> 27.65px
  level 4: font-size 1.2em -> 33.18px
```

A browser's default root size is 16px (check), so `1rem` is 16px unless the user has changed it. That last point is the reason to prefer `rem` for font sizes: if a reader raises their browser's default size to 20px because their eyes tire, `1rem` becomes 20px and your whole page follows. A `px` font size ignores that preference. Compare the two:

```
/* Follows the reader's settings */
body      { font-size: 1rem; }          /* 16px by default, bigger if the user asked */
small     { font-size: 0.875rem; }      /* 14px by default, still scales */
h1        { font-size: 2.25rem; }       /* 36px by default */

/* Ignores the reader's settings: avoid for text */
.fixed    { font-size: 14px; }
```

COMMON MISTAKE

Setting `html { font-size: 62.5%; }` so that `1rem` equals 10px for easy sums. It is a popular trick, but it works against a reader who has chosen a larger default, and it makes every rem value that is not set explicitly look tiny (check). Keep the root at 100% and do the arithmetic once, in a stylesheet comment.

Dial 1: Font Size

There is no universal "right" size, because **apparent size** depends on the typeface. Remember the x-height from Typography Fundamentals 1: two fonts set at 16px can look quite different because one has a taller x-height and so looks bigger. Georgia and Times New Roman at the same size are a good example; Georgia looks larger. For web body text, 16px to 20px is a common comfortable range (check), and print body text is often set between 9pt and 12pt (check). Treat these as starting points, then judge by eye, at the real reading distance, on the real device.

TIMES NEW ROMAN, 18PX

Hamburgefonstiv: the same size on paper, a different size to the eye.

GEORGIA, 18PX

Hamburgefonstiv: the same size on paper, a different size to the eye.

ARIAL, 18PX

Hamburgefonstiv: the same size on paper, a different size to the eye.

Dial 2: Line Length (the Measure)

The **measure** is the length of a line of text. If it is too long, the eye struggles to find the start of the next line after finishing one; too short, and the reader is forced to jump from line to line constantly, breaking the rhythm. A widely quoted guideline is about **45 to 75 characters per line**, with around 66 as the ideal for single-column body text (check). The guideline is in *characters*, not pixels, because the number of pixels needed for 66 characters depends on the font and size.

The next script measures this for four fonts. It adds up the real advance widths (the `hmtx` table from Typography Fundamentals 2) for a sample sentence, divides by the number of characters to get the **average letter width in ems**, and works out how many pixels 66 characters need at 18px, and what that would be in `ch` units.

```
"""How wide is a comfortable line? Average letter width measured from real fonts.

    py measure.py
"""
from pathlib import Path
```

```
from fontTools.ttLib import TTFont

TEXT = ("Good typography is mostly invisible: readers notice the story, "
        "not the letters that carry it.")
FONTS = {"Times New Roman": "times.ttf", "Georgia": "georgia.ttf",
         "Arial": "arial.ttf", "Consolas": "consola.ttf"}
FONT_DIR = Path("C:/Windows/Fonts")
SIZE_PX = 18
TARGET_CHARS = 66          # a common rule of thumb for body text (check)

print(f"{'font':<16}{'avg em/char':>12}{'chars/540px':>13}{'px for 66':>11}{'~ch for 66'
for name, filename in FONTS.items():
    font = TTFont(FONT_DIR / filename)
    cmap = font.getBestCmap()
    hmtx = font["hmtx"]
    upm = font["head"].unitsPerEm
    total = sum(hmtx[cmap[ord(c)]][0] for c in TEXT)
    avg_em = total / len(TEXT) / upm
    zero_em = hmtx[cmap[ord("0")]][0] / upm          # CSS "ch" is the width of 0
    chars_in_540 = 540 / (avg_em * SIZE_PX)
    px_for_target = TARGET_CHARS * avg_em * SIZE_PX
    ch_for_target = px_for_target / (zero_em * SIZE_PX)
    print(f"{name:<16}{avg_em:>12.3f}{chars_in_540:>13.1f}{px_for_target:>11.0f}{ch_for_
```

```
font             avg em/char  chars/540px  px for 66  ~ch for 66
Times New Roman        0.382         78.6        453        50.4
Georgia                0.420         71.5        499        45.1
Arial                  0.417         71.9        496        49.5
Consolas               0.550         54.6        653        66.0
```

Read the table row by row. A letter in Times New Roman averages 0.382 em, so 66 characters need only about 453px at 18px; Georgia, with its wider letters, needs about 499px. Consolas, a monospace font with a fixed 0.550 em per character, needs 653px. The lesson: **the same character count takes different pixel widths in different fonts**, so a fixed pixel width is a poor way to control the measure.

One more detail from the last column: `ch` is the width of the digit `0`, and in Georgia that digit is unusually narrow because Georgia uses old-style figures (check), which is why the Georgia row says 45 `ch` and not 66. In a font where `0` is about as wide as an average letter, `66ch` is a

good rough starting point; in Georgia the digit is narrow and so `66ch` would give noticeably fewer than 66 characters. This is another case where you should test with real text.

```
/* Let the text decide the width, not a magic pixel number */
.article {
  max-width: 66ch;          /* around 66 characters per line, on wide screens */
  margin-inline: auto;       /* centre the column */
  padding-inline: 1rem;      /* breathing room on narrow screens */
}
```

`max-width` rather than `width` matters: on a phone the column simply fills the screen, and the line length falls to whatever fits (often around 35 to 45 characters, which is acceptable (check)). We return to small screens in Typography Fundamentals 7.

Dial 3: Leading (Line-Height)

Leading (pronounced "ledding") is the vertical distance from one baseline to the next. The name comes from the strips of lead that compositors placed between lines of metal type (check). In CSS it is the `line-height` property. The distance is not the gap between lines but the whole step, so a 16px font with `line-height: 1.5` advances 24px per line; the extra 8px is split above and below each line.

LINE-HEIGHT	16PX TEXT STEPS BY	FEEL	WHERE IT FITS
1.0 to 1.2	16 to 19px	Tight, compact	Large headings, short labels
1.4 to 1.6	22 to 26px	Comfortable	Body paragraphs
1.8 and up	29px and up	Loose, airy	Large pull-quotes; can make a paragraph fall apart

Two rules of thumb help. First, **longer lines need more leading**, because the eye has further to travel back to the next line start. Second, **bigger type needs less**, because the gaps grow in proportion with the letters. A heading at 40px with 1.5 looks detached; at 1.15 it holds together.

> **USE A UNITLESS NUMBER**
>
> Write `line-height: 1.5`, not `1.5em` or `150%`. A unitless value is inherited as a *multiplier*, so a child element with a different font size calculates its own line height. An `em` or percentage value is

computed once on the parent and inherited as a fixed length, which can squash larger child text (check).

Does changing these dials really change the shape of a page? Here is a script that wraps a paragraph with real Georgia glyph widths, using a simple greedy rule (put each word on the current line if it fits, otherwise start a new line), and reports how many lines and how much height results for five different settings.

```
"""Estimate how tall a paragraph will be: greedy line wrap with real glyph widths.

    py wrap.py
"""
from pathlib import Path
from fontTools.ttLib import TTFont

PARAGRAPH = ("Typography is the craft of arranging type so that written language is "
             "legible, readable and pleasing when displayed. The size of the letters, "
             "the length of each line and the space between the lines all work together,
             "and changing one of them changes the feel of the others.")
FONT = TTFont(Path("C:/Windows/Fonts/georgia.ttf"))
CMAP = FONT.getBestCmap()
HMTX = FONT["hmtx"]
UPM = FONT["head"].unitsPerEm

def text_width(text, size):
    return sum(HMTX[CMAP[ord(c)]][0] for c in text) / UPM * size

def wrap(text, size, measure):
    lines, current = [], ""
    for word in text.split():
        trial = (current + " " + word).strip()
        if text_width(trial, size) <= measure:
            current = trial
        else:
            lines.append(current)
            current = word
    lines.append(current)
    return lines

print(f"{'size':>5}{'measure':>9}{'line-h':>8}{'lines':>7}{'avg chars':>11}{'height px':
for size, measure, lh in [(16, 640, 1.2), (16, 640, 1.5), (16, 400, 1.5), (16, 280, 1.5)
    lines = wrap(PARAGRAPH, size, measure)
    avg = sum(len(l) for l in lines) / len(lines)
```

```
    height = len(lines) * size * lh
    print(f"{size:>5}{measure:>9}{lh:>8}{len(lines):>7}{avg:>11.1f}{height:>11.0f}")
```

```
 size  measure  line-h  lines  avg chars  height px
   16      640     1.2      4       67.5         77
   16      640     1.5      4       67.5         96
   16      400     1.5      5       53.8        120
   16      280     1.5      8       33.2        192
   20      400     1.6      7       38.1        224
```

Compare the rows. Going from line-height 1.2 to 1.5 on the same 640px measure leaves the paragraph at four lines but grows its height from 77px to 96px. Narrowing the measure from 640px to 280px at the same 16px size turns four lines into eight, and the paragraph doubles in height (96px to 192px), with only 33 characters per line, below the guideline. The last row, bigger text at 20px, shows how size, measure and leading are linked: raising one changes how much room the others need. You will rarely change just one dial.

Dial 4: Tracking and Kerning

The last two dials work on the horizontal space between letters, and they are often confused.

	TRACKING (LETTER-SPACING)	KERNING
What it changes	The space between *all* letters in a run of text, evenly	The space between *one specific pair* of letters
Who sets it	You, in the stylesheet	The font designer, stored in the font file; the browser applies it
CSS	`letter-spacing`	`font-kerning` (and `font-feature-settings: "kern"`)
Example pairs	Not applicable: it affects everything	AV, To, Te, LT (the shapes leave a hole that looks

Compare a few settings on live text. Notice how pairs such as "AV" and "To" fit together when kerning is on, and how the headline in capitals reads better with a little added tracking:

KERNING OFF

AVATAR Toy

KERNING ON (THE DEFAULT, USUALLY)

AVATAR Toy

CAPITALS, TRACKING 0.12EM

QUIET HARBOUR

Guidelines that hold up well: keep kerning on and leave it alone; use **no tracking change** on lowercase body text; **add a little tracking** (around 0.05em to 0.15em (check)) to all-capitals and to very small text; and avoid negative tracking on small text because the letters begin to touch. Specify tracking in `em` so it scales with the size. Never track out lowercase text to make it "look designed": it breaks the word shapes that readers recognise (check).

Does your font even have kerning data? A font file can hold kerning in the older `kern` table or, more commonly now, as a `kern` feature inside the `GPOS` table. This script checks four installed fonts.

```
"""Does this font file carry kerning data, and which tracking-style features does it off

    py kern.py
"""
from pathlib import Path
from fontTools.ttLib import TTFont

FILES = ["times.ttf", "arial.ttf", "georgia.ttf", "consola.ttf"]
FONT_DIR = Path("C:/Windows/Fonts")

print(f"{'file':<14}{'old kern table':>16}{'GPOS kern feature':>20}{'GPOS features (firs
for filename in FILES:
    font = TTFont(FONT_DIR / filename, lazy=True)
    has_kern_table = "kern" in font
    features = []
    if "GPOS" in font:
        features = sorted({r.FeatureTag for r in font["GPOS"].table.FeatureList.FeatureR
    print(f"{filename:<14}{str(has_kern_table):>16}{str('kern' in features):>20}   {', '
```

```
file            old kern table   GPOS kern feature   GPOS features (first 6)
times.ttf                 True               True   cpsp, kern, mark, mkmk
arial.ttf                 True               True   cpsp, kern, mark, mkmk
georgia.ttf              False               True   cpsp, kern
consola.ttf              False              False   mark, mkmk
```

Times New Roman and Arial carry both; Georgia keeps its kerning only as a GPOS feature; and Consolas has none at all. That is sensible: in a monospace font every letter sits in an identical box, so there is nothing to adjust between pairs. It also shows kerning is a property of the font, not something the page adds. (The feature list also includes `cpsp`, capital spacing, which tells the browser it may add space automatically around capitals (check).)

Putting the Dials Together: A Body-Text Recipe

Here is a sensible starting stylesheet for a column of body text that applies everything so far. It is a *starting* point: change one value at a time and look.

```
body {
  font-family: Georgia, "Times New Roman", serif;
  font-size: 1.125rem;          /* 18px by default: follows the reader's setting */
  line-height: 1.55;            /* unitless multiplier */
  font-kerning: normal;         /* keep the font's kerning on */
}

.article {
  max-width: 62ch;              /* about 60 to 66 characters per line (test it) */
  margin-inline: auto;
  padding-inline: 1rem;
}

h1, h2 {
  line-height: 1.2;             /* headings are bigger, so tighter */
}

.kicker {                       /* small capitals label above a heading */
  font-size: 0.8rem;
  letter-spacing: 0.1em;        /* capitals and small text want a little tracking */
  text-transform: uppercase;
}
```

CHANGE ONE DIAL AT A TIME

If a paragraph "feels wrong", the cause is usually one dial out of balance rather than a bad typeface. Check in this order: measure (too wide?), leading (too tight for that measure?), size (too small to read comfortably?), then tracking. Fix the first problem, then look again before touching the next.

Hands-On Exercises

Three exercises: a code task that judges text settings, a written design task that diagnoses three problem paragraphs, and a CSS task that builds a complete text style. Each has a solution file that opens in a new tab.

EXERCISE 1

Write a script that takes a paragraph and a list of text settings (font, size in px, measure in px, line-height multiplier), wraps the paragraph with real glyph widths from Windows fonts (as the wrapping script above does), and prints for each setting the number of lines, the average characters per line, and a verdict: is the line length within 40 to 80 characters on average, and is the line-height between 1.3 and 1.8? Try it on three settings: one too wide and tight, one balanced, one too narrow and loose.

View solution

EXERCISE 2

Diagnose three paragraphs, and say which dial to turn and in which direction. (a) A news article set at 14px across a 1100px column with `line-height: 1.1`; readers lose their place at the end of each line. (b) A museum caption set at 22px in a narrow 200px column with `line-height: 2`, so each line holds only a few words. (c) A heading in capitals, 12px, with `letter-spacing: -0.05em`, on a leaflet; the letters seem to collide. Also convert two sizes by hand: 9pt to px, and 1.25rem to px with the default root.

View solution

EXERCISE 3

Write a stylesheet for a long-form article page. Use `rem` for font sizes, a measure in `ch` with `max-width`, a unitless `line-height` for body text and a tighter one for headings, a small-capitals label with added tracking, and kerning left on. Add a comment beside each line that says which of the four dials it

controls and why you chose that value. Then add a media query that raises the body font size a little on very wide screens and say in a comment what else must change with it.

View solution

CHAPTER 3 QUICK REFERENCE

- **Units:** 1in = 96px = 72pt; 1pc = 12pt = 16px; `em` is relative to the element's font size, `rem` to the root, `ch` to the width of the digit 0
- **Font sizes in** `rem` so they follow the reader's browser setting; `em` compounds when nested
- **Size:** apparent size depends on x-height; roughly 16 to 20px for web body text (check)
- **Measure:** aim for about 45 to 75 characters per line (about 66 ideal (check)); control it with `max-width` in `ch`, and test with real text
- **Leading:** `line-height` as a unitless number; about 1.4 to 1.6 for body, 1.0 to 1.2 for big headings; longer lines need more
- **Tracking** (`letter-spacing`) changes all letters evenly: none for lowercase body text, a little for capitals and small text
- **Kerning** adjusts specific pairs and lives in the font file: leave it on
- **The dials are linked:** change one at a time and look again

Typography Fundamentals

Course 1 · Chapter 4 · Hierarchy and Scale

In Typography Fundamentals 3 you learned the four dials that make a single paragraph comfortable: size, measure, leading and tracking. A real page is never one paragraph, though. It has a title, section headings, captions, lists and notes, and the reader needs to know at a glance which is which. That job is called **typographic hierarchy**: using visible differences to show which text is most important, which belongs to which, and in what order to read it. This chapter shows how to build a hierarchy deliberately, using a **modular scale** to choose sizes and four kinds of contrast to separate the levels.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> The two Python scripts in this chapter were run on Windows with Python 3.14.0, using only the standard library. The output shown is exactly what they printed. The specimens on the page are live text styled in your browser; the CSS snippets were **not** tested across several browsers.

What a Hierarchy Is For

Imagine a page where every line is 16px regular text with the same gap between lines. Nothing tells you where to start, what is a heading and what is a footnote. You are forced to read it all to find out. Now imagine a menu, a timetable or a newspaper: you glance, find the big thing, then the next biggest, and decide where to dive in. A hierarchy is that glance made possible. A good one answers three questions without the reader noticing they were asked:

- **What is this page about?** The title is the loudest thing.
- **How is it organised?** Headings and subheads show the sections and their parts.
- **What is secondary?** Captions, notes and metadata are visibly quieter than the main text.

Here is the same content set twice. Cover the right-hand sample and try to answer the three questions from the left one:

FLAT: ONE SIZE, ONE WEIGHT

Harbour Bread Club

Meet on the first Saturday of the month

RANKED: SIZE, WEIGHT, COLOUR, SPACE

Harbour Bread Club

What we bake

Sourdough, rye and a seasonal loaf. Bring a starter if you have one.

Photo: last month's table

Meet on the first Saturday of the month

What we bake

Sourdough, rye and a seasonal loaf. Bring a starter if you have one.

Photo: last month's table

The right-hand sample uses no new typeface and no decoration. It simply made the title big and bold, pushed the tagline back with a softer colour, gave the section heading its own size and a gap above it, and shrank the caption. That is hierarchy.

The Four Tools of Contrast

To make one level stand apart from its neighbour, you vary something. There are four main things to vary, and a good designer picks one or two per level rather than all four at once.

TOOL	HOW IT SIGNALS RANK	CSS YOU WOULD TOUCH	WATCH OUT FOR
Size	Bigger reads as more important	`font-size`	Neighbouring levels only 1 to 2px apart look like a mistake, not a choice
Weight	Bolder reads as more important or as a label	`font-weight`	Bold everywhere stops meaning anything; remember the weights from Typography Fundamentals 2
Colour and tone	Stronger tone leads; softer tone recedes	`color`	Grey text can become too faint to read: contrast is measured in Typography Fundamentals 7
Space	Space around text isolates it and groups what is near	`margin`, `padding`, `line-height`	Equal space above and below a heading leaves it floating

A fifth tool exists, **style** (italic, small capitals, all-caps with tracking, as you saw in Typography Fundamentals 3), and a sixth, **position** (top of the page, left of the column). They are useful, but size, weight, colour and space do most of the work.

SPACE IS THE QUIET TOOL

Beginners reach for size and bold first. Experienced designers often reach for space first. A heading with a generous gap above it and a small gap below it needs hardly any extra size to be clear, because the gaps show that it belongs to the paragraph that follows. Use space to **group**, and size and weight to **rank**.

Proximity: Space That Groups

Our eyes treat things that sit close together as related. This is called **proximity**, and it is why the gap above a heading should be larger than the gap below it. Compare the two headings below. In the first the gaps are equal; in the second the heading is pulled towards its own paragraph:

EQUAL GAPS: THE HEADING FLOATS

The oven should be hot before the loaf goes in, and the steam tray filled.

Scoring the loaf

A single curved cut lets the bread expand where you choose.

BIGGER GAP ABOVE: THE HEADING BELONGS

The oven should be hot before the loaf goes in, and the steam tray filled.

Scoring the loaf

A single curved cut lets the bread expand where you choose.

A handy rule of thumb is a gap above a heading of about **1.5 to 3 times** the gap below it (check), but trust your eye over the number.

The Levels of a Page

Most pages need only four to six levels. Giving each a name and a job keeps the design honest:

LEVEL	JOB	HTML	TYPICAL TREATMENT
Title	Names the page; only one per page	`h1`	Largest, bold, tight leading
Lead or standfirst	One or two sentences that summarise	`p` with a class	A step above body, softer tone

LEVEL	JOB	HTML	TYPICAL TREATMENT
Heading	Opens a main section	`h2`	Clearly bigger than body, bold, large gap above
Subhead	Splits a section into parts	`h3`	Smaller than a heading; often only bold at body size, or a little larger
Body	The main reading text	`p`, `li`	The baseline against which everything else is judged
Caption or note	Describes a figure or adds a side remark	`figcaption`, `small`	Smaller and quieter than body

CHOOSE THE TAG FOR MEANING, THE STYLE FOR LOOKS

Use `h1` to `h3` because of a heading's *place in the outline*, so screen readers and search engines understand the page, not because you like the default size. Never skip from `h1` straight to `h4` just to get a smaller heading: change the size in CSS instead.

Modular Scales: Picking Sizes by Ratio

How big should a heading be? "A bit bigger" is vague, and ad-hoc numbers (22px here, 27px there) tend to produce sizes that do not feel related. A **modular scale** removes the guesswork. You choose a **base size** (usually the body text, say 16px) and a **ratio**, then every other size is the base multiplied by the ratio a number of times. Going up is multiplying, going down is dividing:

```
size at step n  =  base × ratio^n

step 0   = the base (body text)
step 1   = base × ratio       (a lead paragraph, or a subhead)
step 2   = base × ratio × ratio
step -1  = base ÷ ratio       (a caption)
```

This is borrowed from music, where each note on a scale stands in a fixed ratio to the next (check), which is why the ratios carry musical names. The reason it works is the same: sizes that share a ratio feel like they belong to one family. Some common ratios:

RATIO	NAME	CHARACTER
1.125	Major second	Subtle: good for dense interfaces and long documents
1.2	Minor third	Gentle and flexible: a safe default for articles
1.25	Major third	Clear steps without shouting: a popular default
1.333	Perfect fourth	Bold: good for marketing pages and posters
1.5	Perfect fifth	Dramatic: few levels, big contrast
1.618	Golden ratio	Very steep: sizes explode after a few steps

Rather than trust the table, let us compute. This script prints the sizes five different ratios produce from a 16px base, then turns the minor-third scale into rem values for named roles.

```
"""Modular scales: one base size, one ratio, every other size follows.

    py scale.py
"""
BASE_PX = 16
RATIOS = {"major second": 1.125, "minor third": 1.2, "major third": 1.25,
          "perfect fourth": 1.333, "golden ratio": 1.618}

print(f"{'step':>5}" + "".join(f"{name:>16}" for name in RATIOS))
for step in range(-2, 7):
    row = f"{step:>5}"
    for ratio in RATIOS.values():
        row += f"{BASE_PX * ratio ** step:>14.1f}px"
    print(row)

print()
print("Minor third in rem (base 16px):")
for step, role in [(-1, "caption"), (0, "body"), (1, "lead"), (2, "h3"), (3, "h2"), (4,
    px = BASE_PX * 1.2 ** step
    print(f"  step {step:>2}  {role:<8} {px / BASE_PX:.3f}rem  ({px:.1f}px)")
```

```
 step    major second     minor third     major third  perfect fourth    golden ratio
   -2          12.6px          11.1px          10.2px           9.0px           6.1px
   -1          14.2px          13.3px          12.8px          12.0px           9.9px
    0          16.0px          16.0px          16.0px          16.0px          16.0px
```

```
    1          18.0px          19.2px          20.0px          21.3px          25.9px
    2          20.2px          23.0px          25.0px          28.4px          41.9px
    3          22.8px          27.6px          31.2px          37.9px          67.8px
    4          25.6px          33.2px          39.1px          50.5px         109.7px
    5          28.8px          39.8px          48.8px          67.3px         177.4px
    6          32.4px          47.8px          61.0px          89.8px         287.1px

  Minor third in rem (base 16px):
    step -1  caption  0.833rem  (13.3px)
    step  0  body     1.000rem  (16.0px)
    step  1  lead     1.200rem  (19.2px)
    step  2  h3       1.440rem  (23.0px)
    step  3  h2       1.728rem  (27.6px)
    step  4  h1       2.074rem  (33.2px)
```

Read down the columns. The major second grows gently: even step 6 is only 32.4px, which is why it suits long documents. The golden ratio reaches 109.7px at step 4 and 287.1px at step 6, far too large for a normal page, while its step -2 is a useless 6.1px. The lesson is that a bigger ratio gives you **fewer usable steps**. For most text-heavy pages, a ratio between 1.125 and 1.333 and only four to six steps is plenty (check).

Notice also that the small side of the scale shrinks quickly: with the perfect fourth, step -2 is 9.0px, below a comfortable reading size. A scale does not have to be used in both directions. Many designs use the base for body text and only one step down for captions.

> **COMMON MISTAKE**
>
> Following the scale blindly. A modular scale gives you good candidates, not commandments. If a computed size looks wrong next to a real typeface (a heading at 27.6px that is a hair too small, say), nudge it or pick a different step. Also, a scale tells you nothing about weight, colour or space, so it solves only one fifth of a hierarchy.

Putting a Scale into CSS

Custom properties (CSS variables) are the natural home for a scale: define the sizes once, then use them by name everywhere. Changing the ratio later means editing a few lines. Sizes are in `rem` so they follow the reader's browser setting (Typography Fundamentals 3).

```css
:root {
  /* Scale: base 1rem, ratio 1.25 (major third) */
  --size-caption: 0.8rem;
  --size-body:    1rem;
  --size-lead:    1.25rem;
  --size-h3:      1.563rem;
  --size-h2:      1.953rem;
  --size-h1:      2.441rem;
}

body       { font-size: var(--size-body); line-height: 1.55; }
h1         { font-size: var(--size-h1);  line-height: 1.15; margin: 0 0 0.75rem; }
h2         { font-size: var(--size-h2);  line-height: 1.2;  margin: 2.5rem 0 0.75rem; }
h3         { font-size: var(--size-h3);  line-height: 1.25; margin: 2rem 0 0.5rem; }
.lead      { font-size: var(--size-lead); color: #55555e; }
figcaption { font-size: var(--size-caption); color: #55555e; }
```

Notice how this stylesheet uses all four tools: size from the scale, a tighter `line-height` as the text grows (Typography Fundamentals 3), colour on the lead and caption, and margins that are larger above a heading than below it.

Auditing a Hierarchy

You can check a hierarchy by looking, and you should. But a quick script catches the commonest fault, **levels that are too close to tell apart**. This audit takes a list of levels with size and weight, computes the size ratio between each level and the next, and flags pairs that are too close in size and not rescued by a big difference in weight. The 1.15 threshold is a rule of thumb, not a law (check).

```python
"""Hierarchy audit: do neighbouring levels differ enough to be told apart?

    py audit.py
"""
LEVELS = [  # role, size in px, weight
    ("h1", 22, 700), ("h2", 20, 700), ("h3", 18, 700), ("body", 16, 400), ("caption", 15
]
MIN_RATIO = 1.15   # a rule of thumb, not a law (check)

print(f"{'role':<9}{'size':>6}{'weight':>8}{'ratio to next':>15}  verdict")
```

```
    for i, (role, size, weight) in enumerate(LEVELS):
        if i + 1 < len(LEVELS):
            nxt_size, nxt_weight = LEVELS[i + 1][1], LEVELS[i + 1][2]
            ratio = size / nxt_size
            bolder = weight - nxt_weight >= 300
            if ratio >= MIN_RATIO:
                verdict = "clear"
            elif bolder:
                verdict = "close in size, saved by weight"
            else:
                verdict = "too close: add size, weight or space"
            print(f"{role:<9}{size:>6}{weight:>8}{ratio:>15.2f}  {verdict}")
        else:
            print(f"{role:<9}{size:>6}{weight:>8}{'-':>15}  (last level)")
```

```
role       size  weight  ratio to next  verdict
h1           22     700           1.10  too close: add size, weight or space
h2           20     700           1.11  too close: add size, weight or space
h3           18     700           1.12  close in size, saved by weight
body         16     400           1.07  too close: add size, weight or space
caption      15     400              -  (last level)
```

The audit says the title and the headings are nearly the same size and the same weight: the page would read as a pile of similar-looking headings. The h3 is saved by being bold against regular body text, but the body and caption are only 1px apart with the same weight, so a caption would look like a typing slip, not a different level. The cure is a scale: moving this page to a 1.25 ratio would put the title at about 39px, the headings at about 31px and the subheads at 25px, and the audit would pass them all.

> **WHAT THE NUMBERS CANNOT TELL YOU**
>
> The script only compares sizes and weights. It cannot tell you whether a grey is too pale, whether the gaps group the right things, or whether the page *feels* organised. Use it as a smoke alarm, then squint at the page: if blurring your eyes still shows the same order of importance, the hierarchy is working.

Building a Readable Page Structure

Here is a short recipe for putting it all together, in the order a designer would work:

1. **Start with the body.** Choose its typeface, size, measure and leading (Typography Fundamentals 3). Everything else is measured from it.
2. **List the levels you really need.** Title, lead, heading, subhead, caption is usually enough. Fewer levels, more clearly separated, beats many levels that blur together.
3. **Pick a scale** and assign steps to levels. Nudge by eye.
4. **Add weight and tone** where size alone is not enough, for instance bold subheads at body size, or a softer colour for captions.
5. **Set the space** so headings sit closer to what follows than to what precedes them.
6. **Squint test.** Blur your eyes or shrink the page: the order should still be obvious.

IDEA IN THIS CHAPTER	AN IDEA YOU MAY ALREADY KNOW
Modular scale	Spacing tokens or a colour palette in a design system: a few related values reused everywhere
Hierarchy levels	The headings outline of an HTML document: `h1` to `h3` nested by meaning
Proximity	A well-grouped menu or a form with fieldsets: things near each other belong together
Squint test	Reading code at a glance by its indentation, not its words

Hands-On Exercises

Three exercises: a code task that turns a scale into CSS, a written design task that gives a flat page its hierarchy, and a CSS task that builds a complete stylesheet from a scale. Each has a solution file that opens in a new tab.

EXERCISE 1

Write a script that takes a base size and a ratio from the command line and prints a `:root` block of CSS custom properties for six named roles (caption, body, lead, h3, h2, h1) in `rem`, with a comment showing the pixel size beside each. Make it print a warning when the caption comes out below 12px or the h1 above 64px. Try it with 1.25 and with 1.618 and describe in a sentence what changes.

View solution

EXERCISE 2

A community recipe page is set entirely in 16px regular text, one colour, with equal gaps between every line. It has a recipe title, a one-sentence intro, the headings "Ingredients" and "Method", a list of ingredients, numbered method steps, a "chef's tip" and a photo caption. On paper, design its hierarchy: for each element give a size (in rem), weight, colour tone and the space above and below it. For each level say which of the four tools (size, weight, colour, space) you chose and why, and explain how your spacing shows that "Method" belongs to the steps beneath it and not to the ingredients above it.

View solution

EXERCISE 3

Write a stylesheet for a long-form article that defines its type sizes as custom properties from a modular scale (state your base and ratio in a comment). Include a small kicker label above the title, a lead paragraph, `h1` to `h3`, body text and a figure caption. Use all four tools of contrast, with a comment on each rule saying which tool carries the rank. Give headings more space above than below, and add a media query for wide screens that changes only the root font size, with a comment explaining why that is enough.

View solution

CHAPTER 4 QUICK REFERENCE

- **Hierarchy** shows what is most important, how the page is organised and what is secondary; four to six levels is usually enough
- **Four tools of contrast:** size, weight, colour and tone, space; use one or two per level, not all four at once
- **Proximity:** more space above a heading than below it, so it groups with what follows (about 1.5 to 3 times (check))
- **Modular scale:** size = base × ratio^{step}; ratios between 1.125 and 1.333 suit most text pages (check); bigger ratios give fewer usable steps
- **In CSS:** store the scale in custom properties, in `rem`; use a tighter `line-height` on bigger text
- **Tags for meaning, CSS for looks:** do not skip heading levels to get a smaller size
- **Squint test:** blur the page; the order of importance should still show

Typography Fundamentals

Course 1 · Chapter 5 · Pairing Typefaces

In Typography Fundamentals 4 you built a hierarchy from one typeface, using size, weight, colour and space to rank the levels. That is a strong foundation, and plenty of excellent pages stop there. But sooner or later you will want a second voice: a headline face with more personality, or a calm body face next to a lively title. Choosing that second typeface is called **pairing**. It is where beginners most often go wrong, because two fonts that are each lovely can still look dreadful together. This chapter gives you a few simple tests so that you are choosing for reasons and not by luck.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> The one Python script in this chapter was run on Windows with Python 3.14.0, using only the standard library. The output shown is exactly what it printed. The x-height numbers in it are rough, illustrative values typed in by hand, not measurements of the real fonts (check). The specimens on the page are live text in fonts that ship with most computers, so they may look slightly different on your system. The CSS snippets were **not** tested across several browsers.

Why Pairings Work or Clash

Think of two people at a dinner table. If they are almost the same, you wonder why both were invited. If they are completely unrelated, the conversation never starts. The ones who get on are different enough to be interesting and share something underneath. Typefaces behave the same way, and the rule that follows is easy to remember:

- **Clear difference** on at least one obvious axis (serif against sans-serif, light against heavy, plain against decorative), so the pairing looks *chosen*.
- **Quiet similarity** on another axis (similar x-height, similar proportions, similar mood or era), so the two look like they *belong*.

The failure is the middle ground: two typefaces that are **nearly** alike. A reader cannot tell whether the difference is a design decision or an accident, and the page looks slightly out of tune. Here are the three situations side by side. Same sentence, same sizes; only the face of the heading changes:

CLASH: TWO NEAR-TWIN SERIFS

Winter Allotment Notes

Lift the last of the leeks before the ground freezes hard, and cover the bare beds with compost.

TOO FAR APART: A HEAVY DISPLAY FACE WITH A MONO BODY

Winter Allotment Notes

Lift the last of the leeks before the ground freezes hard, and cover the bare beds with compost.

WORKS: SERIF HEADING, SANS BODY, MATCHED SIZE

Winter Allotment Notes

Lift the last of the leeks before the ground freezes hard, and cover the bare beds with compost.

Notice that the third sample sets the sans-serif body slightly smaller than the others. Verdana is a wide face with a tall x-height, so at the same nominal size it looks bigger than Georgia (Typography Fundamentals 1 introduced x-height). Matching the *apparent* size of two faces, not their numbers, is half the craft of pairing.

Contrast or Harmony: Two Strategies

There are two honest ways to combine type, and both are valid. Choose one on purpose.

STRATEGY	THE IDEA	EXAMPLE	RISK
Contrast	Two different families, each with a clear job	A serif for headings, a sans-serif for body	Mismatched size or mood makes them fight
Harmony	One family, or a family and its relatives, varied by weight, width and style	One sans-serif in light, regular and bold	Can feel flat if the weights are too close (see Typography Fundamentals 4)

A good way to build a contrast pairing is to change **one big thing** and keep the rest similar. If the classification differs (serif against sans-serif, from Typography Fundamentals 2), keep the proportions and the era roughly alike. If you want an expressive display face for a headline, let the body face be as plain and neutral as possible.

THE ROLES RULE

Give every typeface a **job** before you choose it: "headings", "body", "code", "captions". If you cannot say what a face is for, it does not belong on the page. A typeface with no job is decoration, and decoration is the first thing to cut.

Superfamilies: Pairing Made Easy

The safest pairing is one the designer already made for you. A **superfamily** is a set of related typefaces, often a serif, a sans-serif and sometimes a mono or slab, drawn together to share proportions and details, so they pair well by construction. Several well-known sets exist (check): for instance, some open-source families publish a matching serif and sans, and some include a monospace for code. When you are unsure, a superfamily lets you get contrast without having to judge x-heights yourself.

Even a single family gives you a lot of room. As you saw in Typography Fundamentals 2, one family may offer many weights, a condensed and an extended width, and an italic. A page that uses only one family in four or five carefully chosen styles can be both rich and perfectly consistent. Many professional designs never use a second family.

OPTION	EFFORT	VARIETY	SAFETY
One family, several styles	Lowest	Moderate	Almost unbreakable
Superfamily	Low	High	High: designed to match
Two unrelated families	Highest: you do the matching	High	Depends on your eye

The Two-Family Limit

A useful starting rule is: **no more than two families** on one page or in one small design system (check: some designers allow three if one is a mono for code). Each extra family adds visual noise and, on the web, extra files to download (Typography Fundamentals 6 returns to that cost). Two families, each used consistently for its role, plus a few weights and sizes, is enough for nearly everything.

A *consistent* use matters more than the number. If headings are serif on one page and sans-serif on the next, the site looks like several different sites stitched together, even though only two

families are involved.

Worked Examples

Three small pairings, each with a verdict. Read the reasons, not just the looks.

A. A NEWSLETTER: GEORGIA + VERDANA

The Tuesday Bake

Warm, readable and friendly. Serif and sans give clear contrast; the matched sizes keep them in step.

B. A TECH BLOG: VERDANA + COURIER NEW

Reading Log Files

Body in sans, code samples in mono. Mono has a clear job, so it earns its place.

C. A POSTER: IMPACT + IMPACT

SUMMER FAIR SATURDAY

Gates open at ten, music from noon, raffle at four.

Sample A and B are good: each pairs two different classifications with a clear role for each. Sample C fails for a different reason. It uses **one** loud face for everything, so there is no hierarchy and the small text is tiring to read; a display face is built for a few big words, not a paragraph. The cure is to keep Impact for the headline and set the details in a quiet sans-serif.

> **COMMON MISTAKES IN PAIRING**
>
> Pairing two similar-but-different faces (two serifs, two geometric sans-serifs); mixing three or four families "for variety"; choosing a display face for body text; matching sizes by number instead of by eye; and changing the pairing page by page.

A Crude Pairing Checker

You cannot automate taste, but you can automate a sanity check on the two things this chapter keeps repeating: are the classes different, and are the x-heights close? This script holds a few made-up x-height values (as fractions of the em, rough and for illustration only (check)) and gives each pair a verdict.

```
"""Pairing check: do two typefaces differ enough to look deliberate?

    py pair.py
"""
# name: (class, x-height as a fraction of the em, weight of the regular style)
FACES = {
    "Georgia":         ("serif",      0.48, 400),
    "Times New Roman": ("serif",      0.45, 400),
    "Verdana":         ("sans-serif", 0.55, 400),
    "Arial":           ("sans-serif", 0.52, 400),
    "Courier New":     ("monospace",  0.42, 400),
    "Rockwell":        ("slab",       0.48, 400),
}
PAIRS = [("Georgia", "Verdana"), ("Georgia", "Times New Roman"),
         ("Arial", "Verdana"), ("Times New Roman", "Courier New"),
         ("Rockwell", "Georgia")]

print(f"{'pair':<34}{'classes':<24}{'x-height gap':>13}  verdict")
for a, b in PAIRS:
    ca, xa, _ = FACES[a]
    cb, xb, _ = FACES[b]
    gap = abs(xa - xb)
    if ca == cb and gap < 0.05:
        verdict = "clash: near twins, looks like a mistake"
    elif ca == cb:
        verdict = "weak: same class, mismatched size"
    elif gap > 0.08:
        verdict = "contrast, but fix the sizes to match"
    else:
        verdict = "good: clear contrast, similar size"
    print(f"{a + ' + ' + b:<34}{ca + '/' + cb:<24}{gap:>13.2f}  {verdict}")
```

```
pair                            classes                 x-height gap  verdict
Georgia + Verdana               serif/sans-serif                0.07  good: clear con
Georgia + Times New Roman       serif/serif                     0.03  clash: near twi
Arial + Verdana                 sans-serif/sans-serif           0.03  clash: near twi
Times New Roman + Courier New   serif/monospace                 0.03  good: clear con
Rockwell + Georgia              slab/serif                      0.00  good: clear con
```

The script agrees with your eye on the first three rows: serif with sans is fine, and two near-identical serifs or two near-identical sans-serifs clash. It is also **wrong** in an instructive way. It

calls Rockwell with Georgia "good" simply because the classes differ, yet a slab and a bookish serif are both warm, sturdy and fairly close in feel, so many designers would find that pairing too similar. A script that only knows classes and x-heights cannot see mood, and that is why the final check is always your own eye on real text.

Putting a Pairing into CSS

The pairing belongs in two custom properties, one per role, so that the whole site follows from two lines and a change of mind costs you two lines (the same idea as the size scale in Typography Fundamentals 4). Always add a generic fallback at the end of each stack.

```
:root {
  /* Two roles, two families, each with a generic fallback */
  --font-heading: Georgia, "Times New Roman", serif;
  --font-body:    Verdana, Geneva, sans-serif;
  --font-code:    "Courier New", monospace;
}

body        { font-family: var(--font-body); font-size: 0.9375rem; line-height: 1.6; }
h1, h2, h3  { font-family: var(--font-heading); font-weight: 700; }
code, pre   { font-family: var(--font-code); font-size: 0.9em; }
```

Two details are worth noting. The body is set slightly below `1rem` because Verdana looks large; that is the apparent-size matching from earlier, done in code. And the `code, pre` rule sets `0.9em` because monospace faces tend to look larger than the text around them (check).

IDEA IN THIS CHAPTER	AN IDEA YOU MAY ALREADY KNOW
Contrast with shared ground	A colour scheme: a bold accent colour works because the rest is neutral
Roles for typefaces	Variable names in code: each has one clear purpose
Superfamily	A component library: parts built together fit together
Two-family limit	Keeping a project's dependencies down: each extra one has a cost

Hands-On Exercises

Three exercises: a written design task that picks and defends a pairing, a code task that extends the checker, and a CSS task that builds a pairing into a page. Each has a solution file that opens in a new tab.

EXERCISE 1

A small independent bookshop wants a new look for its website, with three pieces of text: the shop name, the book descriptions and the opening-hours notice. On paper, choose a type plan using no more than two families (you may use well-known system fonts such as Georgia, Verdana, Arial, Courier New or Times New Roman). Give each family a role, say whether you chose contrast or harmony and why, state one thing the two faces have in common, and name one pairing you rejected and the specific reason it would clash.

View solution

EXERCISE 2

Extend the pairing checker so that each face also has a **mood** tag (for example "formal", "friendly", "technical", "loud") and a **role** list (for example "heading", "body", "code"). Add two new rules: warn when two faces have the same mood and the same class, and warn when a face is used for a role it is not suited to (such as a loud display face for body text). Add at least two new faces, run it on at least six pairs and print a summary line at the end that says how many pairs passed without a warning.

View solution

EXERCISE 3

Write a stylesheet for a short recipe page that sets up a pairing with custom properties for a heading family, a body family and a code or quantities family. Give each a generic fallback, adjust the body size so that its apparent size matches the heading face, and style a small label (such as "Serves 4") in the heading family in small capitals with tracking. Add a comment on each rule saying which role the family plays, and a final comment explaining how you would change the whole page to a different pairing by editing only two lines.

View solution

CHAPTER 5 QUICK REFERENCE

- **A pairing needs clear difference plus quiet similarity;** the failure is two faces that are nearly alike
- **Two strategies:** contrast (two families with clear jobs) or harmony (one family, varied by weight, width and style)
- **Roles first:** give each typeface a job (heading, body, code) before choosing it
- **Superfamilies** are designed to match; one family in several styles is almost unbreakable
- **Two-family limit** as a starting rule (check); be consistent about which family does which job
- **Match apparent size,** not the font-size number: wide, tall-x-height faces need a smaller size
- **In CSS:** store each family in a custom property, with a generic fallback; a pairing then changes in two lines
- **Display faces** are for a few big words, not for paragraphs

Typography Fundamentals

Course 1 · Chapter 6 · Type on Screen

In Typography Fundamentals 5 you chose two typefaces and gave each a job. Choosing a typeface is one thing; getting it onto a reader's screen is another. A font designed with curves and fine details has to be turned into a few thousand coloured squares, and on the web the font file also has to travel across a network before anyone can see it. This chapter covers both halves: how a screen draws a letter, and how a web page loads and falls back when the font you wanted is late or missing.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> The one Python script in this chapter was run on Windows with Python 3.14.0, using only the standard library; the output shown is exactly what it printed. The font file sizes mentioned in the text are rough ballpark figures (check). The CSS snippets were **not** tested across several browsers, and the font names and file paths in them are invented placeholders.

How a Screen Draws a Letter

On paper, a letter is ink: a printer can place a dot of ink almost anywhere, and modern printers lay down hundreds of dots per inch (check). A screen is a grid of **pixels**, and each pixel is one fixed square of light. At body-text sizes, a letter such as an "e" may be only seven or eight pixels tall. The thin diagonal and the curve of the bowl simply do not fit neatly into a grid that coarse, so the screen has to decide, pixel by pixel, what to do with the edge of every stroke. Three techniques do that work.

TECHNIQUE	WHAT IT DOES	WHAT YOU SEE
Rasterising	Converts the outline of each glyph into a grid of pixels at the requested size	The basic shape of the letter
Anti-aliasing	Colours edge pixels in shades between the text and background colours, according to how much of the pixel the stroke covers	Smooth edges instead of jagged "stairs"
Hinting	Instructions inside the font that nudge outlines so stems and heights land on whole pixels	Crisper, more even letters at small sizes

Anti-Aliasing, Made Visible

The easiest way to understand anti-aliasing is to do it yourself. This script draws a slanted stroke (a stand-in for the diagonal of a letter) on a tiny grid, twice. The first drawing switches each pixel fully on or off. The second measures, for every pixel, how much of it the stroke covers by checking a 4 by 4 set of sample points inside it, and picks a darker or lighter character accordingly.

```
"""Anti-aliasing demo: draw a slanted stroke on a tiny pixel grid, two ways.

    py aa.py
"""
W, H = 16, 6          # grid size in pixels
SUB = 4               # sub-samples per pixel in each direction
SHADES = " .:-=+*#%@"  # light to dark

def inside(x, y):
    """The 'letter': a slanted stroke, about 2.2 pixels wide."""
    centre = 3 + y * 1.6
    return abs(x - centre) < 1.1

def coverage(px, py):
    hits = 0
    for sy in range(SUB):
        for sx in range(SUB):
            x = px + (sx + 0.5) / SUB
            y = py + (sy + 0.5) / SUB
            hits += inside(x, y)
    return hits / (SUB * SUB)

print("Without anti-aliasing (each pixel fully on or off):")
for py in range(H):
    print("".join("@" if coverage(px, py) >= 0.5 else " " for px in range(W)))

print()
print("With anti-aliasing (pixel darkness = how much the stroke covers it):")
for py in range(H):
    row = ""
    for px in range(W):
        c = coverage(px, py)
        row += SHADES[round(c * (len(SHADES) - 1))]
    print(row)
```

```
Without anti-aliasing (each pixel fully on or off):
   @@
    @@@
      @@
       @@@
         @@
           @@

With anti-aliasing (pixel darkness = how much the stroke covers it):
  -%#.
   .*@=
     :%%:
       =@*.
        .#%-
          -%#.
```

Look at the first drawing: the stroke zig-zags, and its width wobbles between two and three pixels from row to row. That is the jagged look of text with no anti-aliasing. In the second drawing the same stroke fades in at its edges, so the eye reads one smooth diagonal. Real systems do this with shades of grey, or, with a technique called **subpixel rendering**, by using the separate red, green and blue parts of each pixel on an LCD screen to triple the horizontal detail (check which systems use it by default; it varies, and it is turned off on some displays).

> **HINTING IN ONE SENTENCE**
>
> Anti-aliasing softens the edges you cannot avoid; **hinting** reduces how many edges land badly in the first place. A hinted font carries instructions such as "make these two stems exactly one pixel wide" or "round this x-height to a whole pixel". How much hinting matters depends on the operating system and the screen: it was very important on the low-resolution screens of the past, and matters less on today's high-resolution displays (check).

Screen Versus Print

The same typeface can be a delight on paper and tiring on screen, or the other way round. The differences you can control are worth knowing.

QUALITY	PRINT	SCREEN
Resolution	Very fine; thin strokes and small serifs survive	Coarser, though high-density ("retina") screens narrow the gap (check)
Light	Reflected: ink on paper	Emitted: the screen glows, so white backgrounds can dazzle
Reader's control	None: the page is fixed	Reader can zoom, resize the window and change the system font size
Unit of size	Points and picas (Typography Fundamentals 3)	CSS pixels, `rem` and `em`
Fonts guaranteed?	Yes: the printer uses what you supply	No: the reader's device may lack your font

Two practical consequences follow. First, a delicate hairline-weight typeface that looks elegant in a printed brochure can fade away at 14 pixels on a phone, so pick a sturdier weight for body text on screen. Second, because the reader controls the screen and you do not, **design for flexibility**: size text in `rem`, so it respects the reader's settings, and never block zooming. A `px` unit in CSS is not a physical pixel but a reference unit that the browser scales to the display (check), which is why the same CSS looks similar on a standard and a high-density screen.

LIGHT WEIGHT, SMALL SIZE

The thin strokes of a light weight lose contrast at small sizes, especially on dim or low-resolution screens.

REGULAR WEIGHT, SAME SIZE

The thin strokes of a regular weight hold up better at small sizes, especially on dim or low-resolution screens.

Live text: the difference may be subtle on a high-density screen, and your system may substitute another font if it lacks this one.

Web Fonts: Sending a Typeface to the Reader

Until web fonts arrived, a website could only use the fonts already installed on each visitor's device, which is why the pairings in Typography Fundamentals 5 used familiar names such as Georgia and Verdana. A **web font** is a font file your server sends along with the page, so everyone sees the same typeface. You tell the browser about it with the `@font-face` rule.

```css
@font-face {
  font-family: "Orchard Serif";           /* the name you will use in CSS */
  src: url("/fonts/orchard-serif-regular.woff2") format("woff2"),
       url("/fonts/orchard-serif-regular.woff") format("woff");
  font-weight: 400;
  font-style: normal;
  font-display: swap;                      /* show fallback text immediately */
}

body {
  font-family: "Orchard Serif", Georgia, "Times New Roman", serif;
}
```

Read the rule as a small catalogue entry. `font-family` gives the font a name. `src` lists where to get it, best format first, and the browser downloads the first one it understands. `font-weight` and `font-style` say which style this particular file is, so that bold text uses the bold file and not a smeared fake. And `font-display` decides what the reader sees while the file is on its way, which we come back to below.

Font Formats

FORMAT	WHAT IT IS	USE IT?
WOFF2	A compressed font format made for the web, usually the smallest (check)	Yes: the default choice, supported by all current major browsers (check)
WOFF	The earlier compressed web format	Only as a fallback for old browsers (check whether you still need it)
TTF / OTF	The ordinary desktop font formats; uncompressed	Rarely on the web: larger files, no advantage

CHECK THE LICENCE FIRST

A font on your own computer is not automatically free to publish on a website. Many fonts have a separate web licence, and some forbid embedding altogether. Open-source and free-to-use font libraries exist (check the licence of each font before using it), and a licence mistake is a legal problem no amount of good typography fixes.

Variable Fonts: Many Styles in One File

A traditional family ships a separate file for each style: regular, italic, bold, bold italic, and so on. Four styles can mean four downloads. A **variable font** packs a whole range of styles into one file by describing the outlines with **axes**, sliders that the browser can set to any value. The standard axes are weight (`wght`), width (`wdth`), italic (`ital`), slant (`slnt`) and optical size (`opsz`), and a font designer may add custom ones (check the current list).

```
@font-face {
  font-family: "Orchard Sans";
  src: url("/fonts/orchard-sans-var.woff2") format("woff2");
  font-weight: 300 800;                /* the range this one file covers */
  font-display: swap;
}

h1     { font-weight: 800; }
h2     { font-weight: 650; }          /* any value in the range works */
.note  { font-variation-settings: "wdth" 85; }  /* a lower-level way to set one axis */
```

The big gain is flexibility: a weight of 650 is as easy as 600 or 700, and a smooth transition between weights becomes possible. The saving in bytes is real when you need several styles, though a variable file is bigger than any single static one, so if you only need regular and bold, two static WOFF2 files can still be smaller (check by comparing the actual files). Not every font is available as a variable font, and the axes it offers differ from one font to the next.

Fallback Stacks and the Loading Moment

A font file can arrive late, or never. The `font-family` list is called a **fallback stack**: the browser uses the first family that is available and moves down the list. You met the idea in Typography Fundamentals 5; this is why every stack ends with a generic family such as `serif` or `sans-serif` , so that some sensible font always appears.

While a web font downloads, the browser has to choose what to show. `font-display` controls that choice:

VALUE	WHILE WAITING	BEST FOR
swap	Shows fallback text at once, then swaps to the web font	Body text: readable immediately, but the text may visibly change
block	Hides the text for a short period, then swaps	Rarely: invisible text is a poor experience
fallback	A very short hidden period, then fallback, with a limited time to swap	A compromise for long text
optional	Uses the web font only if it is available almost instantly (check the exact timing)	Non-essential decorative fonts

The visible jolt when text swaps fonts is called a **layout shift**: the fallback and the web font have different widths, so lines re-wrap and the page jumps. You can soften it by choosing a fallback with similar proportions, and CSS offers descriptors such as `size-adjust` and the `ascent-override` group that let you tune a local font to match the web font more closely (check browser support).

> **THE TYPOGRAPHY OF WAITING**
>
> Most readers never see your web font at its best; they see the fallback for a moment, or for ever, on a slow connection. A design that only works with the exact font is fragile. Test your page with the web font **switched off** and make sure it still looks reasonable.

Performance: Fonts Are Files

Each font file costs download time, and text cannot settle into its final form until the files arrive. A few habits keep that cost down:

- **Use fewer files.** Every weight and style is another request. Does the page really need regular, italic, bold and bold italic? (This is the two-family rule from Typography Fundamentals 5 turned into a performance rule.)
- **Prefer WOFF2.** It is usually the smallest format (check), so serve it first.
- **Subset.** A font may hold thousands of glyphs for dozens of languages. If your page is in English, a Latin-only subset can be much smaller (check; Typography Fundamentals 7 explains why you must not subset away characters your readers actually need).

- **Preload the critical file.** A `<link rel="preload" as="font" crossorigin>` tag tells the browser to fetch the main heading or body font early (check the exact attributes before using it).
- **Consider system fonts for body text.** The system font stack, such as `system-ui`, costs no download at all and looks native on each device.

```
:root {
  /* A system stack: zero downloads, a native look on each device */
  --font-system: system-ui, -apple-system, "Segoe UI", Roboto, Arial, sans-serif;
  /* A web font for personality, with the system stack behind it */
  --font-display: "Orchard Serif", Georgia, "Times New Roman", serif;
}

body { font-family: var(--font-system); }
h1, h2 { font-family: var(--font-display); }
```

This pattern is a popular compromise: the body text, which is most of the page, appears instantly in a system font, and only the headings pay the cost of a web font. If the web font fails, the headings fall back to Georgia and nothing breaks.

IDEA IN THIS CHAPTER	AN IDEA YOU MAY ALREADY KNOW
Anti-aliasing	Resizing a photo: smooth blended pixels instead of hard blocks
Fallback stack	A try/except chain, or an `||` chain of default values
Subsetting a font	Tree-shaking a JavaScript bundle: ship only what is used
Variable font	A function with parameters, in place of many near-identical copies
Font preload	Warming a cache before it is needed

COMMON MISTAKES WITH TYPE ON SCREEN

Setting body text in a hairline weight; sizing text in `px` so readers' settings are ignored; loading six font files for one page; forgetting a generic family at the end of the stack; leaving out `font-display` so text stays invisible while loading; and only ever testing on your own fast connection with the font already cached.

Hands-On Exercises

Three exercises: a written plan for loading fonts on a real-sounding site, a code task that checks a font budget, and a CSS task that sets up a variable font with a safe fallback. Each has a solution file that opens in a new tab.

EXERCISE 1

A small bakery wants a website that feels warm and handmade, but many of its visitors will be on phones with slow connections. On paper, write a font-loading plan. Say which text uses a web font and which uses a system font, how many font files you will ship and in which format, which `font-display` value you choose and why, what your fallback stacks are, and how you will reduce the visual jump when the web font arrives. Name one thing you decided *not* to load and give the reason.

View solution

EXERCISE 2

Write a Python script that holds a list of font files, each with a name, a format (woff2, woff or ttf) and a size in kilobytes. Where the same font is offered in more than one format, keep only the best format (woff2, then woff, then ttf). Print a table of the chosen files with a total, compare the total with a budget, and say how far under or over you are. When over budget, name the largest file as the first one to drop or subset. Run it and show the output.

View solution

EXERCISE 3

Write the CSS for a page that uses one variable font file with a weight range of 300 to 800. Include an `@font-face` rule with `font-display`, a second `@font-face` that tunes a local fallback font to match the space of the web font, a custom property holding the full fallback stack, and headings that use at least one in-between weight (such as 650). Add a comment on each part explaining its job, and a short print rule that adjusts the body weight.

View solution

CHAPTER 6 QUICK REFERENCE

- **Screens draw letters on a pixel grid:** rasterising makes the shape, anti-aliasing smooths the edges, hinting nudges outlines onto the grid
- **Screen differs from print:** coarser resolution, emitted light, and a reader who can zoom and resize; choose sturdy weights and size in `rem`
- **`@font-face`** names a web font and lists its files in `src`, best format first; set `font-weight` and `font-style` for each file
- **WOFF2** is the default format for the web (check); TTF and OTF are rarely needed there
- **Variable fonts** hold many styles in one file through axes such as `wght` and `wdth`; declare the range in `font-weight`
- **Fallback stacks** end with a generic family; `font-display: swap` shows readable text while the file loads
- **Reduce layout shift** with a similar fallback and the `size-adjust` and override descriptors (check support)
- **Performance:** fewer files, WOFF2, subsetting, preload the critical file, or use a system stack for body text
- **Check the licence** before putting any font on a website, and test with the web font switched off

Typography Fundamentals

Course 1 · Chapter 7 · Readability and Language

In Typography Fundamentals 6 you got a typeface onto the reader's screen, with a sensible fallback if the file arrived late. But "it appears" is not the same as "it can be read". Some of your readers have tired eyes, small phones, dim rooms, dyslexia or low vision. Others read in a language your first font never planned for. This chapter is about making type work for the widest range of people: enough contrast, sizes that respect the reader, and fonts that actually contain the letters a text needs.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> The two Python scripts in this chapter were run on Windows with Python 3.14.0, using only the standard library; the output shown is exactly what each printed. The contrast thresholds quoted here follow the WCAG guidelines as I understand them (check the current version before relying on them for a compliance claim). The CSS snippets were **not** tested across several browsers, and the font names in them are invented placeholders.

Contrast: Can the Text Be Seen?

The most common readability failure on the web is not an odd typeface; it is pale grey text on a white background. **Contrast** is the difference in brightness between text and its background, and the Web Content Accessibility Guidelines (WCAG) turn it into a number: the **contrast ratio**, written like `4.5:1`. It runs from 1:1 (identical colours, invisible text) to 21:1 (black on white).

TEXT KIND	MINIMUM RATIO (LEVEL AA)	HIGHER TARGET (LEVEL AAA)
Body text	4.5:1	7:1
Large text (roughly 24px, or 19px bold; check the exact definition)	3:1	4.5:1
Icons and interface outlines	3:1	(no separate target; check)

Calculating a Ratio Yourself

The formula looks scary but is short: convert each colour's red, green and blue values to a "relative luminance" (how bright the eye perceives it, with green counting most), then divide the brighter by the darker, each with a small offset added. This script does exactly that for a handful of colour pairs.

```
    """Contrast ratio checker (WCAG formula).

        py contrast.py
    """
    def channel(v):
        v = v / 255
        return v / 12.92 if v <= 0.03928 else ((v + 0.055) / 1.055) ** 2.4

    def luminance(hexcol):
        r, g, b = (int(hexcol[i:i + 2], 16) for i in (1, 3, 5))
        return 0.2126 * channel(r) + 0.7152 * channel(g) + 0.0722 * channel(b)

    def ratio(fg, bg):
        a, b = sorted((luminance(fg), luminance(bg)), reverse=True)
        return (a + 0.05) / (b + 0.05)

    pairs = [
        ("Black on white", "#000000", "#ffffff"),
        ("Dark grey on white", "#595959", "#ffffff"),
        ("Mid grey on white", "#999999", "#ffffff"),
        ("Light grey on white", "#cccccc", "#ffffff"),
        ("Soft text on dark page", "#c9d1d9", "#0f1117"),
    ]
    print(f"{'Pair':<24}{'Ratio':>8}  Body (4.5)  Large (3.0)")
    for name, fg, bg in pairs:
        r = ratio(fg, bg)
        print(f"{name:<24}{r:>6.2f}:1  {'pass' if r >= 4.5 else 'FAIL':<10}  {'pass' if r >=
```

```
Pair                      Ratio  Body (4.5)  Large (3.0)
Black on white          21.00:1  pass        pass
Dark grey on white       7.00:1  pass        pass
Mid grey on white        2.85:1  FAIL        FAIL
Light grey on white      1.61:1  FAIL        FAIL
Soft text on dark page  12.23:1  pass        pass
```

Notice that `#999999`, a grey many designers reach for because it "looks subtle", fails even the large-text threshold. And notice the dark-theme row: this course's own page colours pass comfortably. Light-on-dark text can be perfectly readable, though very bright white on pure black can shimmer for some readers, which is why many dark themes use a soft off-white (check).

PASSES: 7:1

Dark grey on white reads comfortably, even in a bright room.

FAILS: 2.85:1

Mid grey on white looks tidy but tires the eye, and vanishes in sunlight.

Live text with fixed colours, so you can judge the difference on your own screen.

CONTRAST IS MORE THAN THE RATIO

A passing ratio does not guarantee good reading: a hairline-weight font can fail in practice even at a good ratio, and text over a photograph may change contrast from one word to the next. Never use colour alone to carry meaning (a red word with no other cue is invisible to some colour-blind readers), and test with real devices and real people where you can.

Choices That Help Readers with Dyslexia or Low Vision

People read differently, and no single setting suits everyone. Research on "dyslexia-friendly" fonts has been mixed, and claims that one special typeface fixes reading difficulty should be treated with caution (check current studies). What does have wide agreement is that a handful of ordinary choices help many readers at no cost to anyone else:

CHOICE	WHY IT HELPS	IN PRACTICE
Clear letter shapes	Letters like `I`, `l` and `1`, or `b` and `d`, are easily confused if the design makes them near-identical	Look at a sample of your typeface with those characters side by side
Generous spacing	Crowded letters and lines blur together	Line-height around 1.5 and a little extra word spacing for long text (check ranges)
Left-aligned, ragged-right text	Justified text can open uneven "rivers" of white space	`text-align: left` for body copy

CHOICE	WHY IT HELPS	IN PRACTICE
Sturdy weight	Hairlines fade for low-vision readers	Regular or medium for body text
Avoid long italic and all-caps passages	Both slow down reading	Keep them to short emphasis (Typography Fundamentals 4)
Short, plain paragraphs	Gives the eye regular places to rest	Break long blocks up with subheads

LET THE READER DECIDE

The strongest accessibility feature is flexibility. Readers who need larger text, a different font or custom colours already set them in their browser or operating system. Your job is mostly to **not get in the way**: do not force pixel sizes, do not lock colours with hard-coded overrides where a preference would do, and do not disable zoom.

Size and Zoom

Typography Fundamentals 6 recommended `rem` for text. Here is the accessibility reason. If the reader raises their browser's default font size, `rem` and `em` values grow with it, while fixed `px` text may ignore the setting (check how your target browsers behave). Page zoom is the other tool: pressing Ctrl and plus should enlarge everything, up to 200% and beyond, without hiding or cutting off any content.

```
<!-- Good: lets readers pinch-zoom -->
<meta name="viewport" content="width=device-width, initial-scale=1">

<!-- Bad: blocks zoom for people who need it -->
<meta name="viewport" content="width=device-width, user-scalable=no, maximum-scale=1">

/* Good: scales with the reader's settings */
body { font-size: 1rem; line-height: 1.5; }
small { font-size: 0.875rem; }   /* still readable: not smaller than about 14px (check) */
```

A useful test: zoom your page to 200%, then resize the window to phone width. Does any line need sideways scrolling? Is anything overlapping? If so, a fixed-width layout is working against the reader.

Line Length on Small Screens

Typography Fundamentals 3 gave you a comfortable measure of roughly 45 to 75 characters. On a phone the screen is narrow, so lines can become *too short*: with large text at 200% zoom you may see only 20 or 25 characters per line, which makes the eye jump back constantly and leaves words hyphenated or broken awkwardly. The remedy is not to shrink the type but to let the column fill the width with sensible margins, and use a `max-width` in `ch` units so that on *wide* screens the lines do not become too long.

```
.article {
  max-width: 65ch;                /* cap the measure on wide screens */
  margin-inline: auto;            /* centre the column */
  padding-inline: 1rem;           /* breathing room on phones */
  hyphens: auto;                  /* needs a lang attribute on the page */
  overflow-wrap: break-word;      /* never let a long word break the layout */
}
```

Diacritics: The Marks Above and Below

So far the chapter has assumed readers can see your letters. The other half is making sure they can see the *right* letters. Many languages add marks, called **diacritics**, to a base letter: the acute in *é*, the tilde in *ñ*, the cedilla in *ç*, the double acute in the Hungarian *ő*. These marks change the meaning and the pronunciation, so they are not decoration: dropping them can turn one word into another.

Good typography treats a letter with a mark as a designed character, not a letter with a mark stuck on. Marks need room, which affects line-height (tall capitals with accents can collide with the line above if leading is too tight, as seen in Typography Fundamentals 3), and the mark's position should be refined for each base letter. Inside the computer, text is stored as numbered **code points** from the Unicode standard, and a character with a mark can be stored in two ways. This script shows both.

```
"""Look inside text: code points, names and normalisation.

    py chars.py
"""
import unicodedata
```

```
samples = ["é", "ő", "ñ", "ß", "Ж", "ㅁ", "あ", "한"]
for ch in samples:
    print(f"U+{ord(ch):04X}  {unicodedata.name(ch)}")

print()
composed = "\u00e9"        # one code point
decomposed = "e\u0301"     # e plus a combining acute accent
print("composed length:  ", len(composed))
print("decomposed length:", len(decomposed))
print("same after NFC:   ", unicodedata.normalize("NFC", decomposed) == composed)
```

```
U+00E9  LATIN SMALL LETTER E WITH ACUTE
U+0151  LATIN SMALL LETTER O WITH DOUBLE ACUTE
U+00F1  LATIN SMALL LETTER N WITH TILDE
U+00DF  LATIN SMALL LETTER SHARP S
U+0416  CYRILLIC CAPITAL LETTER ZHE
U+0643  ARABIC LETTER KAF
U+3042  HIRAGANA LETTER A
U+D55C  HANGUL SYLLABLE HAN

composed length:   1
decomposed length: 2
same after NFC:    True
```

A LESSON FROM RUNNING THIS SCRIPT

The first version of this script printed the characters themselves next to their code points. On the Windows console used here, the default text encoding could not display most of them and the script crashed with a `UnicodeEncodeError`. The fix was to print only the code point and the name. The takeaway applies to the web too: a character is only as visible as the whole chain behind it (file encoding, console or page encoding, and font) allows. On a web page, declare UTF-8 with `<meta charset="utf-8">` so the browser reads the characters correctly.

Non-Latin Scripts and Why Coverage Matters

A single font file covers only the characters its designer drew. The set of characters a font includes is its **coverage**, and it is not the same for every font. Many popular fonts cover the basic Latin alphabet plus the accents of Western European languages, then stop. Central

European languages, Greek, Cyrillic, Arabic, Hebrew, the scripts of India, and Chinese, Japanese and Korean each need their own glyphs, and each script has its own typographic traditions.

SCRIPT FAMILY	A TYPOGRAPHIC POINT WORTH KNOWING
Latin with diacritics	Needs accent-aware spacing; check that a font includes the extended Latin range for languages such as Polish, Hungarian and Turkish
Cyrillic and Greek	Have their own capitals, lower-case forms and italic traditions; not every Latin font includes them
Arabic and Hebrew	Read right to left; Arabic letters change shape depending on their position in a word, and the text flows in the opposite direction to the page's usual layout
Chinese, Japanese, Korean	Thousands of characters; fonts are very large, so subsetting and web delivery need special care (check)
Indic scripts (Devanagari and others)	Letters combine into joined forms, so the font and the browser together must apply the right substitutions (check specifics per script)

When a font lacks a character, the browser does not stop. It quietly borrows the character from the next font in the fallback stack, and, failing that, from another font on the system, or shows an empty box (often called "tofu"). The text is technically displayed, but with mismatched shapes and sizes in the middle of a word. This is why Typography Fundamentals 6 stressed fallback stacks and why subsetting is risky: if you strip a font down to English-only glyphs and a reader's name contains an accented letter, that one character will look wrong. This script checks coverage the way you might audit a font before choosing it.

```
"""Font coverage check: which characters in a text would fall back to another font?

    py coverage.py
"""
import unicodedata

# A pretend font covering Basic Latin and Latin-1 Supplement (U+0020-U+00FF).
FONT_RANGES = [(0x0020, 0x007E), (0x00A0, 0x00FF)]

def covered(ch):
    return any(lo <= ord(ch) <= hi for lo, hi in FONT_RANGES)

texts = {
    "English":    "Good morning, everyone",
    "French":     "Ça va très bien, merci",
```

```
    "Hungarian":  "Köszönöm szépen, tűzoltó",
    "Polish":     "Dzień dobry, zażółć gęślą jaźń",
    "Greek":      "Καλημέρα",
}

for lang, text in texts.items():
    missing = sorted({ch for ch in text if not covered(ch)})
    if not missing:
        print(f"{lang:<10} OK: all {len(set(text))} distinct characters covered")
    else:
        names = ", ".join(f"U+{ord(c):04X}" for c in missing)
        print(f"{lang:<10} MISSING {len(missing)}: {names}")
        for c in missing[:2]:
            print(f"           e.g. {unicodedata.name(c)}")
```

```
English    OK: all 13 distinct characters covered
French     OK: all 15 distinct characters covered
Hungarian  MISSING 1: U+0171
           e.g. LATIN SMALL LETTER U WITH DOUBLE ACUTE
Polish     MISSING 8: U+0105, U+0107, U+0119, U+0142, U+0144, U+015B, U+017A, U+017C
           e.g. LATIN SMALL LETTER A WITH OGONEK
           e.g. LATIN SMALL LETTER C WITH ACUTE
Greek      MISSING 7: U+039A, U+03AD, U+03B1, U+03B7, U+03BB, U+03BC, U+03C1
           e.g. GREEK CAPITAL LETTER KAPPA
           e.g. GREEK SMALL LETTER EPSILON WITH TONOS
```

The Hungarian line is the sneaky one: the sample sentence is mostly fine, because *ö* and *ó* live in the Western European range, but *ű* is missing. A designer who tested a font only on French and Spanish would never notice. Always test a font with the real text of every language you serve.

Telling the Browser the Language

Browsers and assistive software need to know which language a page, or a stretch of it, is written in. The `lang` attribute does this, and it affects hyphenation, quotation marks, which regional variant of a character is drawn (check; some scripts share characters between languages but draw them differently) and what a screen reader pronounces.

```
<!-- The page language -->
<html lang="en">

<!-- A passage in another language -->
```

```
<p>The waiter said, <span lang="fr">Bon appétit</span>.</p>

<!-- A right-to-left passage -->
<p lang="ar" dir="rtl">...</p>

/* CSS can then pick fonts and quotes per language */
:lang(ja) { font-family: "Example Gothic JP", sans-serif; }
:lang(fr) { quotes: "«\00a0" "\00a0»"; }
```

IDEA IN THIS CHAPTER	AN IDEA YOU MAY ALREADY KNOW
Contrast ratio	A signal-to-noise ratio: how far the text stands above the background
Relative units (`rem`)	Responsive design: values that adapt instead of being fixed
Unicode normalisation	Comparing strings after trimming and lower-casing them
Font coverage	A library that supports only some of the locales your users need
The `lang` attribute	A locale setting: it changes behaviour without changing the text

COMMON MISTAKES WITH READABILITY AND LANGUAGE

Pale grey body text; colour as the only signal; `user-scalable=no`; a pixel-locked layout that breaks at 200% zoom; justified text in narrow columns; subsetting a font until names with accents break; testing only with English; forgetting `lang` and `charset`; and assuming a "dyslexia font" replaces good spacing, sizing and contrast.

Hands-On Exercises

Three exercises: a written accessibility review of a flawed page, a code task that audits a stylesheet's colours, and a CSS task that builds a flexible, language-aware text style. Each has a solution file that opens in a new tab.

EXERCISE 1

A local library's event page uses light grey (#aaaaaa) 12px text on white, justified paragraphs about 120 characters wide, all-caps italic for every event description, blocks pinch-zoom with a viewport setting, and shows a red word for "sold out" with no other cue. On paper, write a review: list each problem, say

who it affects and why, and give a specific fix for each (with numbers where you can). Finish by naming the three fixes you would make first and the reason for that order.

View solution

EXERCISE 2

Write a Python script that holds a list of text and background colour pairs for a website (at least six, including headings, body text, links and a footer), calculates the WCAG contrast ratio for each, and prints a table showing the ratio, a pass/fail for body text (4.5:1) and for large text (3:1). For each failing pair, suggest a darker text colour by darkening the text step by step until it passes, and print the colour that works. Run it and show the output.

View solution

EXERCISE 3

Write the CSS and the small HTML skeleton for a multilingual article page. Include the language declarations, a body style in `rem` that respects the reader's settings, a column capped in `ch` units that still fits a phone at 200% zoom, a fallback stack for Latin text, a different font stack for Japanese and for Arabic passages chosen with `:lang()`, right-to-left handling, and a `prefers-reduced-motion` rule. Add a comment on each part explaining its job.

View solution

CHAPTER 7 QUICK REFERENCE

- **Contrast ratio** runs from 1:1 to 21:1; aim for at least 4.5:1 for body text and 3:1 for large text (check the current guidelines)
- **Mid grey on white fails:** #999999 on white measures 2.85:1; #595959 measures 7:1
- **Helpful for many readers:** clear letter shapes, generous spacing, left-aligned body text, sturdy weights, short paragraphs; no single "dyslexia font" is a cure (check studies)
- **Respect the reader:** size in `rem`, never block zoom, test at 200% and at phone width
- **Cap the measure** with `max-width` in `ch`, with padding for small screens
- **Diacritics are letters,** not decoration; Unicode can store them as one code point or as a base plus a combining mark, and normalisation makes them comparable

- **Coverage:** a font draws only the characters it includes; missing ones fall back to another font or show as boxes; test with real text in every language you serve
- **Declare language and encoding:** `lang`, `dir` and `<meta charset="utf-8">`

COMING NEXT

In Chapter 8 (Typography Fundamentals 8, the final chapter) you put the whole course together: take a poor page and redesign its type step by step, using a checklist built from every chapter so far.

Typography Fundamentals

Course 1 · Chapter 8 · Putting It Together

In Typography Fundamentals 7 you learned to make type work for readers with different eyes, screens and languages. Now comes the payoff. Seven chapters of vocabulary, measurements and rules only become skill when you apply them to something real, so this final chapter is one long worked example: take a deliberately bad page and fix it, one decision at a time, naming the chapter each fix comes from.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> The one Python script in this chapter was run on Windows with Python 3.14.0, using only the standard library; the output shown is exactly what it printed. The CSS snippets were **not** tested across several browsers, the font names are real system fonts chosen only for illustration, and the numeric thresholds are rules of thumb rather than laws (check them against the guidelines you work to).

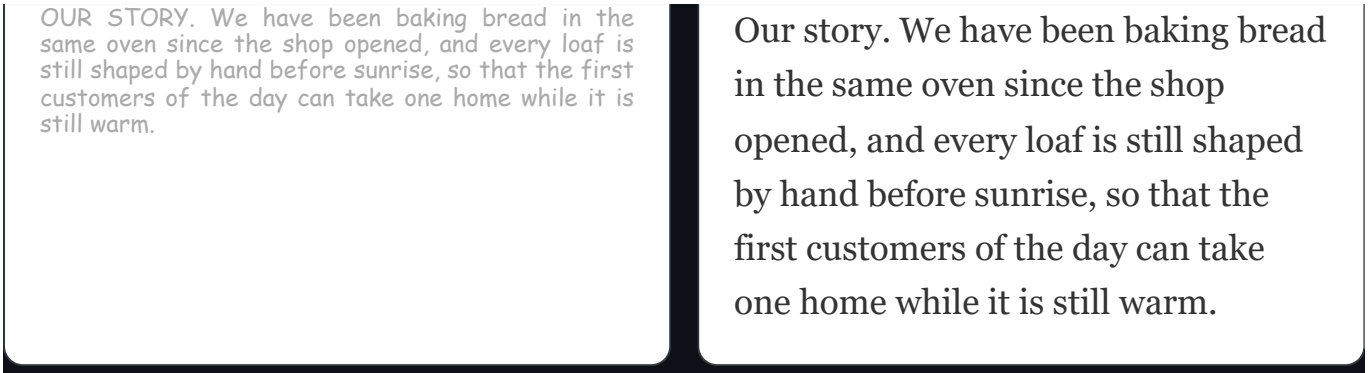
The Patient: A Poor Page

Imagine a small bakery's "About Us" page. The owner built it quickly and it shows. Here is its stylesheet, with a comment on each choice. Read it as a doctor reads a chart: how many problems can you spot before the next section tells you?

```
body {
  font-family: "Comic Sans MS", "Papyrus", sans-serif;
  font-size: 12px;               /* small */
  line-height: 1.1;               /* tight */
  color: #aaaaaa;                 /* "subtle" grey */
  background: #ffffff;
  text-align: justify;
}
.page  { width: 1100px; }          /* one very wide column */
h1     { font-family: Impact; font-size: 14px; text-transform: uppercase; }
h2     { font-family: "Times New Roman"; font-size: 13px; }
.note  { font-size: 9px; font-style: italic; text-transform: uppercase; }
```

BEFORE

AFTER



The same paragraph, live text with fixed colours: judge the difference on your own screen.

DIAGNOSIS BEFORE TREATMENT

A good redesign starts with a list, not with a font. Count the problems first, rank them by how many readers they affect, and fix them in that order. Changing the typeface before fixing size, spacing and contrast is the classic beginner's mistake, because the typeface is the most visible change and the least important one.

Step-by-Step Treatment

Each step below names the chapter it comes from, so you can see the whole course working as a single toolkit.

STEP	PROBLEM FOUND	FIX	FROM
1. Size	12px body text, 9px notes	`1.125rem` body, notes no smaller than `0.875rem`	Typography Fundamentals 3 and 7
2. Contrast	#aaaaaa on white (about 2.3:1)	#333333 on white (about 12.6:1; the script below checks it)	Typography Fundamentals 7
3. Leading	line-height 1.1	line-height 1.55	Typography Fundamentals 3
4. Measure	1100px column, about 130 characters per line	`max-width: 65ch` plus padding	Typography Fundamentals 3 and 7
5. Alignment	Justified text with rivers	`text-align: left`	Typography Fundamentals 7
6. Families	Four unrelated families	One serif for text, one sans-serif for headings, with fallback stacks	Typography Fundamentals 5 and 6

STEP	PROBLEM FOUND	FIX	FROM
7. Hierarchy	Headings nearly the size of the body	A modular scale with a ratio of 1.25	Typography Fundamentals 4
8. Emphasis	Small all-caps italic notes	Sentence case, upright, a little muted but still above 4.5:1	Typography Fundamentals 7
9. Delivery	No fallback, no language declared	System-font stacks, `lang="en"`, `charset`	Typography Fundamentals 6 and 7

Build the Scale First

Steps 1 to 5 are single-property corrections. Step 7 deserves a moment because it replaces guesswork with arithmetic. With a body size of 1.125rem and a ratio of 1.25, each heading level multiplies the one below it. The sizes in the finished stylesheet below are those multiples, rounded to tidy numbers. The script that follows then runs this chapter's checklist as an audit on the before and after settings.

```
"""Type audit: score a page's text settings against a simple checklist.

    py audit.py
"""
def channel(v):
    v = v / 255
    return v / 12.92 if v <= 0.03928 else ((v + 0.055) / 1.055) ** 2.4

def luminance(hexcol):
    r, g, b = (int(hexcol[i:i + 2], 16) for i in (1, 3, 5))
    return 0.2126 * channel(r) + 0.7152 * channel(g) + 0.0722 * channel(b)

def ratio(fg, bg):
    a, b = sorted((luminance(fg), luminance(bg)), reverse=True)
    return (a + 0.05) / (b + 0.05)

def audit(page):
    results = []
    results.append(("Body size at least 16px", page["body_px"] >= 16))
    results.append(("Line height 1.4 to 1.7", 1.4 <= page["line_height"] <= 1.7))
    results.append(("Measure 45 to 75 characters", 45 <= page["measure"] <= 75))
    results.append(("Body contrast 4.5:1 or more", ratio(page["text"], page["bg"]) >= 4.
    results.append(("Two families or fewer", len(page["families"]) <= 2))
    results.append(("Heading 1.7x body size or more", page["h1_px"] / page["body_px"] >=
```

```
    results.append(("Left-aligned body text", page["align"] == "left"))
    return results

before = {"body_px": 12, "line_height": 1.1, "measure": 130, "text": "#aaaaaa",
          "bg": "#ffffff", "families": ["Comic Sans", "Papyrus", "Impact", "Arial"],
          "h1_px": 14, "align": "justify"}
after = {"body_px": 18, "line_height": 1.55, "measure": 66, "text": "#333333",
         "bg": "#ffffff", "families": ["Georgia", "Verdana"],
         "h1_px": 36, "align": "left"}

for label, page in (("BEFORE", before), ("AFTER", after)):
    res = audit(page)
    print(f"{label}: {sum(ok for _, ok in res)}/{len(res)} checks passed "
          f"(contrast {ratio(page['text'], page['bg']):.2f}:1)")
    for name, ok in res:
        print(f"  [{'x' if ok else ' '}] {name}")
    print()
```

```
BEFORE: 0/7 checks passed (contrast 2.32:1)
  [ ] Body size at least 16px
  [ ] Line height 1.4 to 1.7
  [ ] Measure 45 to 75 characters
  [ ] Body contrast 4.5:1 or more
  [ ] Two families or fewer
  [ ] Heading 1.7x body size or more
  [ ] Left-aligned body text

AFTER: 7/7 checks passed (contrast 12.63:1)
  [x] Body size at least 16px
  [x] Line height 1.4 to 1.7
  [x] Measure 45 to 75 characters
  [x] Body contrast 4.5:1 or more
  [x] Two families or fewer
  [x] Heading 1.7x body size or more
  [x] Left-aligned body text
```

WHAT THIS AUDIT CANNOT TELL YOU

Seven passing checks do not make a beautiful page. The script measures numbers, not taste: it cannot see whether the two chosen families actually suit each other, whether the heading has real personality, [illegible] replacement for them. (The first draft of this script labelled the heading check as a "1.2x step", which

was wrong, since the code tests three steps of 1.2, about 1.7 times. A label that disagrees with its code is a small bug; reading your own output aloud is a quick way to catch it.)

The Finished Stylesheet

Here are all nine fixes combined. Notice how much shorter the thinking is when the rules are on your side: every value has a reason.

```
:root {
  --text: #333333;
  --muted: #595959;                    /* 7:1 on white */
  --step-0: 1.125rem;                  /* body */
  --step-1: 1.4rem;                    /* 1.125 x 1.25, rounded */
  --step-2: 1.75rem;
  --step-3: 2.2rem;
}

body {
  font-family: Georgia, "Times New Roman", serif;
  font-size: var(--step-0);
  line-height: 1.55;
  color: var(--text);
  background: #ffffff;
}

.page {
  max-width: 65ch;                     /* a comfortable measure */
  margin-inline: auto;
  padding-inline: 1rem;                /* room on phones */
}

h1, h2 {
  font-family: "Trebuchet MS", Verdana, sans-serif;
  line-height: 1.2;                    /* big type needs less leading */
  margin: 2rem 0 0.5rem;
}
h1 { font-size: var(--step-3); }
h2 { font-size: var(--step-2); }

.note {
  font-size: 0.875rem;
```

```
    color: var(--muted);
  }
```

KEEP WHAT YOU WERE GIVEN

The finished version still respects the reader: sizes are in `rem`, nothing blocks zoom, and the column is capped in `ch` rather than fixed in pixels. A redesign that looks better but takes control away from the reader has only moved the problem.

The Final Checklist

Here is the whole course on one page, in the order you would work. Keep it beside you the next time you open an empty stylesheet.

ASK YOURSELF	RULE OF THUMB (CHECK FOR YOUR PROJECT)	CHAPTER
Is the text a designed object, not a default?	Choose deliberately; read it as a reader, not a designer	Typography Fundamentals 1
Does the typeface suit the job?	Pick a family type that fits the tone; use weights and italics with purpose	Typography Fundamentals 2
Are size, measure and leading set for reading?	About 16px or more; 45 to 75 characters; line-height about 1.4 to 1.6	Typography Fundamentals 3
Can a reader see the structure at a glance?	A scale, a few clear levels, contrast through size, weight, colour and space	Typography Fundamentals 4
Do the typefaces belong together?	Two families at most; contrast or harmony, not a near miss	Typography Fundamentals 5
Does it arrive and render well on screen?	WOFF2, fallback stacks, `font-display`, subset with care	Typography Fundamentals 6
Can everyone read it, in any language?	4.5:1 contrast, zoom works, `lang` and `charset` declared, coverage tested	Typography Fundamentals 7

Course Complete

You started with the parts of a single letter and finished by repairing an entire page. Along the way you learned to name what you see (baseline, x-height, counter), to tell the families apart, to measure and set text, to build a hierarchy, to pair typefaces, to deliver fonts to a screen and to make sure every reader can use the result.

WHERE TO GO FROM HERE

The best next step is practice on real material: restyle a page you use every day, or set a favourite poem or recipe as carefully as you can. After that, courses on page layout and grids, on colour, or on front-end styling in general would build naturally on what you now know, and a deeper look at type design (drawing letters yourself) is a rewarding direction if you fall in love with the craft. Keep a small notebook of typography you admire, and write down why it works. Your eye will improve faster than you expect.

Hands-On Exercises

The last three exercises are capstones: a written redesign brief, a code task that builds an audit tool, and a CSS task that restyles a page from scratch. Each has a solution file that opens in a new tab.

EXERCISE 1

Choose a real page you know (a school or club website, a recipe page, a news article) and write a one-page redesign brief for its type. Describe the current state in the words from this course, list at least six problems, and for each give a fix with a number or a named choice. Rank the fixes in the order you would make them and explain why. Finish by naming two things the page already does well that you would keep.

View solution

EXERCISE 2

Extend the audit script from this chapter. Add three new checks of your own devising (for example: a maximum number of font weights, a minimum size for small print, a check that the page declares a language). Add a third page called `unfinished` that passes some checks and fails others, and make the script print a short "fix first" line naming the failed check you consider most important. Run it and show the output.

View solution

EXERCISE 3

Write the HTML and CSS for a small one-page article, such as a recipe or a short biography, from a blank file. Use a modular scale built from custom properties, a capped measure in `ch`, two font stacks with fallbacks, accessible contrast, a caption style, a pull quote and a `lang` declaration. Add a comment above each rule saying which chapter of this course it comes from.

View solution

CHAPTER 8 QUICK REFERENCE

- **Diagnose first:** list every problem, rank by how many readers it affects, then fix in that order
- **Typeface last:** size, contrast, leading, measure and alignment matter more than the font name
- **The poor page:** 12px, #aaaaaa, line-height 1.1, about 130 characters per line, justified, four families
- **The fixed page:** 1.125rem, #333333 (about 12.6:1), line-height 1.55, `65ch`, left-aligned, two families, a 1.25 scale
- **Audits help, eyes decide:** numbers catch the obvious failures, not bad taste
- **Respect the reader:** `rem`, zoom, `lang`, `charset` and tested font coverage stay in every redesign

COURSE COMPLETE

You have finished Typography Fundamentals. You can now describe type precisely, choose and pair it with reasons, set it for reading on paper and on screen, and check that it works for everyone. Well done.