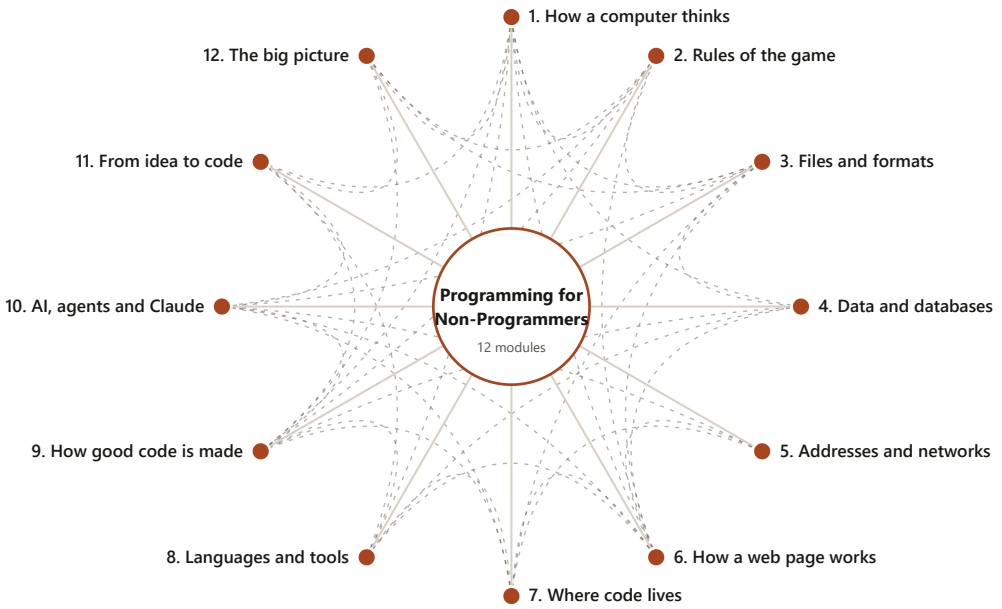
Programming for Non-Programmers

What you need to understand to build with AI agents

The goal is not to turn you into a programmer. The goal is that you understand the words your AI agent uses, and that you can break a problem down far enough for it to be solved.



Contents

1. How a computer thinks

What a program is · Variable · Data types · List · Object · Condition · For loop · While loop · Function · Comment · Terminal

2. Rules of the game

Garbage in, garbage out · What a bug is · Error or crash · How to read an error message · Debugging · Code is predictable, AI is not

3. Files and formats

File, extension, path · Text or binary · Plain text and Markdown · CSV — a table as text · JSON — structure as text · PDF · Word and Excel files · Picture or drawing · HTML · Which formats are AI-friendly · OCR — from picture to text

4. Data and databases

The spreadsheet as a starting point · Same data, four shapes · What a database is · Relational databases and keys · SQL · Document databases · Which one, when

5. Addresses and networks

Anatomy of an address · Domains and DNS · IP address · localhost · The local network · Ports · Request and response · Status codes

6. How a web page works

The browser is the engine · HTML — the structure · CSS — the appearance · JavaScript — the behaviour · Frontend and backend · The API call · The journey of one click · Static and dynamic

7. Where code lives

Rung one: my own machine · Rung two: the office · Rung three: a tunnel · Rung four: a rented server · Rung five: static hosting · Rung six: the cloud · Web technology without the web · What may go on a public address

8. Languages and tools

Why there are so many languages · Compiled and interpreted · C and C++ · Python · JavaScript and TypeScript · Java and C# · SQL · Bash and PowerShell · Runtimes and Node.js · Libraries · Library or your own code · Dependencies and versions

9. How good code is made

Breaking a problem down · Main and modules · Naming · What a test is · Kinds of tests · Why tests matter more with AI · Git and commits · The architecture.md file

10. AI, agents and Claude

What a language model is · Tokens · What tokens cost · Prompt and context · The context window · Summarise and start fresh · Spending fewer tokens · Hallucinations · What must not go into AI · What an agent is · Claude: chat · Claude: Cowork · Claude: Code · Artifacts · Connectors · Skills · MCP — how an agent sees the world · Sub-agents · Building agents well

11. From idea to code

The grilling skill · The handoff skill · The improve skill · Where to get skills · Recipe 1: from idea to architecture · Recipe 2: from architecture to code · The implementer never sees the tests

12. The big picture

How it all connects · Where to go next

13. Glossary

MODULE 1

How a computer thinks

The nine building blocks every program on earth is made of — and the black window where it all happens.

What a program is `program / code`

A program is a sequence of instructions the computer carries out one after another, top to bottom.

The instructions are written in a language a human can read, which the machine translates into its own as it goes.

The only real difference from a recipe is the reader: a cook reads a recipe, a machine reads a program — which is why it has to be far more precise.

count.py

```
# Count the lines in a file.
path = "guests.txt"

handle = open(path, encoding="utf-8")
lines = handle.readlines()
handle.close()

print("Number of lines:", len(lines))
```

OUTPUT

```
Number of lines: 248
```

Six lines, executed first to last. That is an entire program.

> You can tell a cook *salt to taste* and lunch will be fine. Tell a computer the same and it either stops or adds nothing at all. That one difference — a computer never guesses — explains almost everything a beginner finds illogical. When you later write instructions for an AI agent, you are writing a recipe for something in between: clever enough to fill in the gaps, and dangerous for exactly that reason when it fills them in wrongly.
>
> MORE ON THIS
>
> - Automate the Boring Stuff — Python Basics (free book) — https://automatetheboringstuff.com/2e/chapter1/
> - Python: the official tutorial — https://docs.python.org/3/tutorial/introduction.html

Variable `variable`

A variable is a named place holding one value.

Left of the equals sign is the `name`, right of it is the `value`.

Change the value in one place and everything that uses it follows.

vat.py

```
vat_rate = 22          # change it only here
net_price = 100

vat = net_price * vat_rate / 100
gross_price = net_price + vat

print(vat, gross_price)
```

OUTPUT

```
22.0 122.0
```

If the rate changes to 9.5 you edit one line, not seventeen.

> Why this matters when working with AI: if the VAT rate is written out in seventeen places, changing it is risky and the agent will almost certainly miss one. If it lives in a single variable, the change is one line. When you tell an agent *change the tax rate*, what you are really asking for is exactly this. A value written straight into the code has a name of its own: a `hard-coded value`.
>
> MORE ON THIS
>
> - Real Python — Variables in Python — https://realpython.com/python-variables/

Data types `data types`

A computer draws a hard line between a number `int`, text `str`, a yes/no value `bool` and a date.

The number `22` and the text `"22"` are not the same thing: you can add the first, you can only glue the second.

The quotation marks decide it — whatever sits between them is text, even if it is all digits.

A surprisingly large share of all bugs are values that look like numbers but are quietly text.

types.py

```python
quantity = 22          # int   - a number
quantity_txt = "22"    # str   - text
is_paid = True         # bool  - yes / no

print(quantity + 1)
print(quantity_txt + "1")
print(type(quantity), type(quantity_txt))
```

OUTPUT

```
23
221
<class 'int'> <class 'str'>
```

Line 5 adds, line 6 glues. Same key on the keyboard, different meaning.

> The classic office case: a column of amounts where one cell reads `1,200.00 €`. To a human that is twelve hundred euros; to the computer it is a string of characters containing a comma and a currency symbol. The total silently comes out wrong, or the program stops. When your agent reports `TypeError: expected number, got string`, this is precisely what happened — and the fix is to convert the text into a number, not to patch the arithmetic.
>
> MORE ON THIS
>
> - Real Python — Basic Data Types in Python — https://realpython.com/python-data-types/
> - Python: numbers, strings and lists — https://docs.python.org/3/tutorial/introduction.html

List `list / array`

A list is a numbered sequence of values held under one single name.

You reach an individual `item` by its number, its `index`, and Python starts counting at zero.

Reach for a list whenever you have *more of the same thing*: rows in a table, files in a folder, people signed up.

list.py

```
guests = ["Ana", "Bor", "Cvet", "Dan"]

print(guests[0])        # the first
print(guests[3])        # the last
print(len(guests))      # how many

guests.append("Eva")    # add to the end
print(guests)
```

OUTPUT

```
Ana
Dan
4
['Ana', 'Bor', 'Cvet', 'Dan', 'Eva']
```

guests[4] would raise IndexError - no such slot exists.

> Counting from zero is not a programmer's joke; it follows from how a list is stored in memory — the number says how many slots past the start the item sits. In practice, remember only this: the first item is number 0 and the last is *length − 1*. An `IndexError: list index out of range` means you reached for a slot that does not exist — almost always because you counted from one.
>
> MORE ON THIS
>
> - Real Python — Lists and Tuples — https://realpython.com/python-lists-tuples/
> - Automate the Boring Stuff — Lists — https://automatetheboringstuff.com/2e/chapter4/

Object `object / dict / key-value`

An object is a filled-in form: every field has a name, its `key`, and a `value`.

You reach a value by the field's name rather than by a number.

This is the shape of nearly all the data you will meet in `.json` files and in answers from web services.

object.py

```
customer = {
    "name": "Novak",
    "email": "novak@example.com",
    "amount": 120,
    "paid": False,
}

print(customer["email"])
print(customer["amount"] * 1.22)
```

OUTPUT

```
novak@example.com
146.39999999999998
```

A list of objects is a table: items are rows, field names are columns.

> Lists and objects are the two blocks almost everything else is built from. A list of objects — *several filled-in forms* — is exactly what a spreadsheet is: rows are the items of the list, columns are the fields of the object. By the way, that `146.39999999999998` is not a broken program but a consequence of how computers store decimals `floating point`; with money you avoid it by rounding or by counting whole cents.
>
> MORE ON THIS
>
> - Real Python — Dictionaries in Python — https://realpython.com/python-dicts/
> - Automate the Boring Stuff — Dictionaries and Structuring Data — https://automatetheboringstuff.com/2e/chapter5/

Condition `if / else`

A condition is a railway switch: if something holds, the program takes one track, otherwise the other.

It is written almost exactly as you would say it — *if the amount is over a hundred, give a discount, otherwise charge full price.*

Any business rule you can state using the word *if* can be written as code.

discount.py

```python
amount = 120

if amount > 100:
    discount = 0.10
elif amount > 50:
    discount = 0.05
else:
    discount = 0.0

print("Discount:", discount * 100, "%")
```

OUTPUT

```
Discount: 10.0 %
```

In Python the indentation is what decides which lines belong to which branch.

> Most misunderstandings come from rules we say only half of. *Regular customers get a discount* says neither what counts as regular nor what happens to everyone else. A computer needs every `branch`. That is why a good agent — and the `grilling` skill in Module 11 — will push you to say the second half of the sentence.

MORE ON THIS

- Real Python — Conditional Statements — https://realpython.com/python-conditional-statements/
- Automate the Boring Stuff — Flow Control — https://automatetheboringstuff.com/2e/chapter2/

For loop `for loop`

A loop repeats the same steps for every item in a list.

Use `for` when you already know what you are going through: three hundred rows, twelve months, every file in a folder.

Instead of three hundred repetitions by hand, you write the steps once and say what to run them over.

loop.py

```
1  invoices = [120, 45, 310]
2  total = 0
3
4  for amount in invoices:
5      total = total + amount
6      print("Added", amount, "-> total", total)
7
8  print("Done:", total)
```

OUTPUT

```
Added 120 -> total 120
Added 45 -> total 165
Added 310 -> total 475
Done: 475
```

The indented lines 5 and 6 run three times; the last line runs once.

> Loops are the reason programming pays off at all. Work that takes three hours by hand takes three seconds in a loop — and more importantly, it takes three seconds again next month. Whenever you catch yourself repeating something in a spreadsheet row by row, you are looking at a loop. One pass through a loop is called an `iteration`.
>
> MORE ON THIS
>
> - Real Python — For Loops — https://realpython.com/python-for-loop/
> - Python: control flow — https://docs.python.org/3/tutorial/controlflow.html

While loop `while loop`

`while` repeats as long as some condition holds — you do not know in advance how many rounds that will be.

If the condition never stops holding, the program spins forever — an `infinite loop` — and you have to stop it by hand.

while.py

```
1  stock = 3
2
3  while stock > 0:
4      print("Sold one. Left:", stock)
5      stock = stock - 1
6
7  print("Out of stock.")
```

OUTPUT

```
Sold one. Left: 3
Sold one. Left: 2
Sold one. Left: 1
Out of stock.
```

Delete line 5 and the condition never changes - an infinite loop, Ctrl + C.

> The difference is simple: `for` means *four times*, `while` means *until*. An infinite loop is one of the rare cases where a computer looks crashed while diligently doing exactly what it was told.
>
> MORE ON THIS
>
> - Real Python — While Loops — https://realpython.com/python-while-loop/

Function `function`

A function is a named piece of code: data goes in as `arguments`, a `return value` comes out.

You write it once and then call it by name as often as you like.

A good function does one thing, and its name says which thing.

function.py

```
1  def with_vat(price, rate=22):
2      return price * (1 + rate / 100)
3
4  print(with_vat(100))
5  print(with_vat(250, 9.5))
6  print(with_vat(80))
```

OUTPUT

```
122.0
273.75
97.6
```

def = define. return = what comes back. rate=22 is a default value.

> A function is the smallest unit that can be checked on its own by a test (Module 9). That is exactly why it matters when working with an agent: an agent that hands you one function with a clear input and output is verifiable. An agent that hands you three hundred lines in one block is not. When you say *break this into smaller pieces*, you are asking for functions.
>
> **MORE ON THIS**
>
> - Real Python — Defining Your Own Python Function — https://realpython.com/defining-your-own-python-function/
> - Automate the Boring Stuff — Functions — https://automatetheboringstuff.com/2e/chapter3/

Comment `comment`

A comment is a line the computer ignores entirely and a human reads.

Its job is to explain *why* something was done this way — what the code does is visible in the code itself.

With AI they earn their keep twice over: an agent reads comments as instructions, not just as remarks.

comment.py

```python
import math


# Finance requires rounding UP (agreed 2024).
# That is why this is ceil() and not round().
amount = math.ceil(12.31)


print(amount)
```

OUTPUT

```
13
```

A bad comment here would be "round the amount". The code already says that.

> A bad comment restates the code in worse English. A good one carries what the code cannot show: that this exception exists because finance asked for it, that this order must not be changed, that it used to work differently and why that failed.

MORE ON THIS

- Real Python — Writing Comments in Python — https://realpython.com/python-comments-guide/

Terminal `terminal / command line / cmd`

A terminal is a window where you type commands instead of clicking buttons.

One command is one line: the name of a program, followed by its settings, its `arguments`.

The `prompt` on the left tells you which folder you are in — that is where the commands will act.

Almost anything you can do with a mouse can be done here — only faster, and repeatably.

Command Prompt (cmd)

```
C:\Users\Klemen> cd Documents\project



C:\Users\Klemen\Documents\project> dir

  count.py

  guests.txt



C:\...\project> python count.py

Number of lines: 248



C:\...\project> _
```

The prompt changes as soon as cd moves you into another folder.

`cd folder_name`	go into a folder
`cd ..`	go back up one folder
`cd /d D:\project`	jump to another drive and folder
`dir`	list what is in the folder (`ls` on Mac and Linux)
`type guests.txt`	print a file's contents (`cat` elsewhere)
`python count.py`	run a program
`cls`	clear the screen
`Tab`	complete a half-typed filename
`↑ ↓`	recall previous commands
`Ctrl + C`	stop whatever is running

> The black window that opens when you double-click `zazeni-lokalno.bat` is a terminal. Everything printed in it is a program talking to you. Two things are worth trusting: `Ctrl + C` stops whatever is running, and the last line of an error is almost always the useful one. Windows ships two of these — the older `cmd` and the newer `PowerShell`; the commands are mostly the same and the differences only start to matter for harder tasks. Agents like Claude Code live here — which is why it pays not to be scared of the window.
>
> **MORE ON THIS**
>
> - Microsoft — Windows commands reference — https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/windows-commands
> - Microsoft — the cd command — https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/cd

MODULE 2

Rules of the game

Why things break, how to read an error, and the one way code differs fundamentally from AI.

Garbage in, garbage out `garbage in, garbage out`

A computer never checks whether your data makes sense — only whether it can process it.

If one amount among many is stored as text, the sum does not come out wrong; it stops altogether.

Most *program errors* are not errors in the program at all, but errors in the data that went into it.

gigo.py

```
1  amounts = [120, 45, "310"]   # the last one is text!
2  total = 0
3
4  for a in amounts:
5      total = total + a
6
7  print(total)
```

OUTPUT

```
TypeError: unsupported operand type(s)
for +: 'int' and 'str'
```

The program is not broken. The input is.

> The same holds for AI agents, except the consequence differs: a program stops, an agent carries on. Give it a table with vague column names and half the values missing and it will hand you an answer — confident and wrong. That is why most of the work in building with AI is not in writing instructions but in preparing the input.

MORE ON THIS

- Wikipedia — Garbage in, garbage out — https://en.wikipedia.org/wiki/Garbage_in,_garbage_out

What a bug is bug

A bug is the gap between what you thought you asked for and what you actually asked for.

It is almost never the computer making a mistake — it did exactly what the code says.

The most common bug is the case nobody thought about: an empty list, a negative number, a missing field.

bug.py

```
def average(numbers):
    return sum(numbers) / len(numbers)

print(average([2, 4, 6]))
print(average([]))        # what if the list is empty?
```

OUTPUT

```
4.0
ZeroDivisionError: division by zero
```

The function is correct for every case except the one nobody considered.

> The word comes from 1947, when actual moths flew into the Mark II computer and had to be picked out of a relay. What follows from the story matters more than the story: finding a bug is not finding someone to blame, it is finding the case the instructions did not cover. That is exactly why tests (Module 9) are such a powerful tool — they are the list of cases you did think of, written so that it checks itself.
>
> MORE ON THIS
>
> - Wikipedia — Software bug — https://en.wikipedia.org/wiki/Software_bug

Error or crash `error / exception`

When a program meets something it cannot handle it raises an `exception` — and stops.

The very same situation can be anticipated and handled instead of crashing.

The difference between *it crashed* and *it told me a value was missing* is only whether someone thought ahead.

exception.py

```
1  entry = "twelve"
2
3  try:
4      number = int(entry)
5  except ValueError:
6      number = 0
7      print("Not a number, using 0.")
8
9  print("Result:", number)
```

OUTPUT

```
Not a number, using 0.
Result: 0
```

Without try/except the program would stop on line 4 with a ValueError.

> The trap is that `try/except` can also sweep an error under the carpet: replace every failure with a silent zero and the program keeps running while quietly computing nonsense. That is worse than a crash, because a crash you notice. A good rule: only handle the errors you know what to do about.
>
> **MORE ON THIS**
>
> - Python — errors and exceptions — https://docs.python.org/3/tutorial/errors.html

How to read an error message `traceback / stack trace`

An error message is not a punishment; it is the most precise description of the problem you will get.

Read it *from the bottom up: the last line says what is wrong, the lines above say where*.

Paste the whole message to your agent instead of saying *it doesn't work* and the fix arrives far faster.

Error output

```
Traceback (most recent call last):
  File "invoices.py", line 5, in <module>
    total = total + a
            ~~~~~~^~~
TypeError: unsupported operand type(s)
for +: 'int' and 'str'
```

The same case as 2.1, this time in full.

`TypeError: ...`	last line: what went wrong — read this first
`File "invoices.py", line 5`	which file and which line
`total = total + a`	the line of code that raised it
`~~~~~~^~~`	the arrow points at the exact spot
`Traceback ...`	the path the program took to get there

> In larger programs a traceback runs ten lines and they all look equally important. They are not: you want the last line with the error name, and the last mention of *your own* file. Everything in between belongs to library files you never wrote. When working with an agent, paste the whole message — it can read more out of it than you can, but only if it gets all of it.

MORE ON THIS

- Real Python — Understanding Tracebacks — https://realpython.com/python-traceback/

Debugging `debugging`

Debugging is checking assumptions: what do I think is in this variable, and what is actually in it?

The oldest method is still among the best — print the value and look.

When something *doesn't work*, one of your assumptions is almost always false.

check.py

```
1  amounts = [120, 45, "310"]
2
3  for a in amounts:
4      print(repr(a), type(a))   # temporary: what is really in there?
```

OUTPUT

```
120 <class 'int'>
45 <class 'int'>
'310' <class 'str'>
```

repr() shows the quotation marks - the text value is visible at a glance.

> Professional tools (a *debugger*) can pause a program mid-run and show every value at once, but printing is plenty to start with. The habit matters more than the tool: instead of guessing where the fault is, cut the program in half — check whether the values are still right in the middle. A few rounds of that narrow the search from three hundred lines to three.
>
> MORE ON THIS
>
> - Real Python — Python Debugging With pdb — https://realpython.com/python-debugging-pdb/

Code is predictable, AI is not `deterministic / non-deterministic`

The same code with the same input gives *_the same_* result every single time — that is `determinism`.

A language model asked the same question twice need not answer the same way; it works in probabilities, not rules.

This is not a defect of the model but its nature — and it is why code written by AI always gets checked.

Hence the rule: let AI write the code, let the code do the computing. Never the other way round.

determinism.py

```python
def with_vat(price):
    return price * 1.22

print(with_vat(100))
print(with_vat(100))
print(with_vat(100))
```

OUTPUT

```
122.0
122.0
122.0
```

Same question three times, same answer three times. Not so with AI.

> One practical rule follows, and it will save you more trouble than any other: do not ask an AI agent to *add up* your three hundred invoices — ask it to write a program that adds them up. The first gives you a number you cannot trust and cannot reproduce. The second gives you a tool you verify once and use a hundred times. That same difference separates *I asked an AI* from *I built myself an application*.
>
> MORE ON THIS
>
> - Anthropic — how Claude works (documentation) — https://docs.claude.com/en/docs/about-claude/models/overview

MODULE 3

Files and formats

Which files an AI can read and which it cannot — and why that single difference matters most.

File, extension, path `file, extension, path`

A file is a lump of data with a name; the `extension` after the dot is only a promise about what is inside.

A `path` is the file's address on disk — folders separated by slashes, with the name at the end.

When you tell an agent *this file*, what it actually needs is the path; without it there is nothing to open.

paths.py

```python
from pathlib import Path

path = Path("D:/project/invoices/january.pdf")

print(path.name)      # name with extension
print(path.suffix)    # just the extension
print(path.parent)    # the folder it sits in
print(path.exists())  # does it even exist
```

OUTPUT

```
january.pdf
.pdf
D:\project\invoices
False
```

Windows writes \, the web and Python write /. Python understands both.

> Renaming `data.txt` to `data.xlsx` does not turn it into a spreadsheet — it only changes the promise. Windows hides extensions by default, which causes a surprising amount of confusion; turn them on in Explorer under *View → Show → File name extensions*. When working with an agent, always give the full path or put the file in the project folder.
>
> MORE ON THIS
>
> - Python — pathlib (working with paths) — https://docs.python.org/3/library/pathlib.html

Text or binary `text vs. binary`

Every file is ultimately a sequence of numbers; the only question is whether those numbers stand for letters.

A text file opens in Notepad and makes sense — a binary file opens too, but you get garbage.

This is the single most important division in the whole course: what is text, an AI reads directly; everything else has to be converted first.

text_or_binary.py

```python
# a text file - readable
print(open("guests.txt", encoding="utf-8").read(22))

# an image - same call, very different result
print(open("logo.png", "rb").read(12))
```

OUTPUT

```
Ana Novak
Bor Kovac

b'\x89PNG\r\n\x1a\n\x00\x00\x00\r'
```

Both are files. The agent reads the first; the second must be converted first.

> So the question worth asking before any AI task is: *is my input text?* If it is, almost everything goes smoothly. If it is not — a photo, a scanned PDF, a screenshot of a table — the first step is conversion, not a better prompt. A large share of disappointment with AI comes from a binary input meeting a text-shaped expectation.
>
> MORE ON THIS
>
> - Wikipedia — Optical character recognition — https://en.wikipedia.org/wiki/Optical_character_recognition

Plain text and Markdown `.txt / .md`

`.txt` is bare content with no formatting — the most innocent format there is.

`.md` (Markdown) is that same plain text with a few agreed characters for headings, lists and bold.

Markdown is the language you talk to agents in, and the language of the `architecture.md` file from Module 9.

markdown.py

```
content = """# Meeting notes

## Decisions
- Deadline: 30 September
- Owner: **Klemen**

Details are in the [report](report.pdf).
"""

open("notes.md", "w", encoding="utf-8").write(content)
```

OUTPUT

```
(notes.md is written)
```

The characters #, - and ** are all the Markdown you need 90 % of the time.

> Markdown's advantage is that it reads well in both directions: a human understands it without a renderer, and a computer turns it into a formatted document, a web page or a PDF. That is why it is the default language of developer documentation and the default output of language models.

MORE ON THIS

- Markdown Guide — basic syntax — https://www.markdownguide.org/basic-syntax/

CSV — a table as text `.csv`

CSV is a table written as plain text: one line per row, commas between columns.

It has no formulas, no colours, no sheets, and no idea what a date is — only rows and columns.

Which is exactly why it is the most reliable way to move spreadsheet data into a program or an AI agent.

read_csv.py

```python
# guests.csv is plain text:
#   name,email,amount
#   Novak,novak@example.com,120
#   Kovac,kovac@example.com,45

import csv

with open("guests.csv", encoding="utf-8") as f:
    for row in csv.DictReader(f):
        print(row["name"], "->", row["amount"])
```

OUTPUT

```
Novak -> 120
Kovac -> 45
```

The first line holds the column names. Each next line is one object from 1.5.

> Two traps bite in Europe: Excel often separates columns with a *semicolon rather than a comma, and writes decimals with a comma. If a program reads a CSV wrongly, that is almost always why. The second trap is encoding: if accented letters turn into `Ĺˇ`, the file was saved in a different character set — save it as* CSV UTF-8*.
>
> MORE ON THIS
>
> - Wikipedia — Comma-separated values — https://en.wikipedia.org/wiki/Comma-separated_values
> - Python — the csv module — https://docs.python.org/3/library/csv.html

JSON — structure as text `.json`

JSON writes the objects and lists from Module 1 as plain text.

Curly braces hold objects, square brackets hold lists, quotation marks mark field names.

Almost everything two machines say to each other on the internet is written in JSON.

json_example.py

```python
import json

text = '{"name": "Novak", "amount": 120, "paid": false}'

customer = json.loads(text)      # text -> object
print(customer["name"], customer["amount"])

print(json.dumps(customer, indent=2))   # object -> text
```

OUTPUT

```
Novak 120
{
  "name": "Novak",
  "amount": 120,
  "paid": false
}
```

loads = read, dumps = write. Same data, two shapes.

> Unlike CSV, JSON can nest: a customer holds a list of orders, each order holds a list of items. That is why it is the default language of web services (Module 6) and also how an agent describes the tools it knows how to use (MCP, Module 10).

MORE ON THIS

- json.org — the format itself — https://www.json.org/json-en.html
- Python — the json module — https://docs.python.org/3/library/json.html

PDF `.pdf`

PDF exists so that a page looks identical everywhere — not so that data can be taken out of it.

Two completely different things hide behind the same extension: real text, or a picture of a page.

Which of the two it is decides whether an agent reads the file in a second or cannot read it at all.

read_pdf.py

```python
# pip install pypdf
from pypdf import PdfReader

page = PdfReader("invoice.pdf").pages[0]
text = page.extract_text()

print(len(text), "characters")
print(text[:40])
```

OUTPUT

```
0 characters
```

Zero characters means the PDF came from a scan - it needs OCR (3.11).

> A test that needs no programming: open the PDF and try to select the text with your mouse. If it highlights, there is real text inside and an agent will read it. If nothing selects, you are looking at a picture. The third and most annoying case is tables: the text extracts but the column layout is lost, so always check tables taken out of PDFs.
>
> MORE ON THIS
>
> - pypdf — documentation — https://pypdf.readthedocs.io/en/stable/
> - Anthropic — PDF support — https://docs.claude.com/en/docs/build-with-claude/pdf-support

Word and Excel files `.docx / .xlsx`

`.docx` and `.xlsx` are really compressed folders (`.zip`) full of XML files.

The content is therefore text, merely packed — which is why programs and agents read them reliably.

Rename `report.docx` to `report.zip`, open it, and see for yourself.

docx_is_zip.py

```python
import zipfile


with zipfile.ZipFile("report.docx") as z:
    for name in z.namelist()[:4]:
        print(name)
```

OUTPUT

```
[Content_Types].xml
_rels/.rels
word/document.xml
word/styles.xml
```

word/document.xml holds the entire text of the document.

> This explains why an agent reads a Word document effortlessly but struggles with a photo of the same page: the first is packed text, the second is dots. It works the other way too — when you ask an agent for a Word document, it assembles exactly these XML parts.
>
> MORE ON THIS
>
> - Wikipedia — Office Open XML — https://en.wikipedia.org/wiki/Office_Open_XML

Picture or drawing `raster vs. vector (.png / .svg)`

`.jpeg` and `.png` are grids of dots — enlarge them and they go soft.

`.svg` is a drawing instruction written as text: *a line from here to there, a circle of this radius*.

An agent can read and change an SVG; a grid of dots it can only look at.

svg_is_text.py

```
1  svg = open("assets/img/logo.svg", encoding="utf-8").read()
2
3  print(svg[:90])
```

OUTPUT

```
<svg role="img" aria-label="Zlata ovca" version="1.1"
 xmlns="http://www.w3.org/2000/svg" viewBox=
```

That is this deck's logo. Being text is why it recolours with the theme.

> The practical consequence: keep logos, icons, diagrams and charts as SVG, and photographs as JPEG. An SVG scales to any size without losing quality, takes little space, and can be recoloured at will — including by an agent you simply ask to change the colour. With a photograph that is impossible, because it contains no shapes, only dots.
>
> **MORE ON THIS**
>
> - MDN — SVG — https://developer.mozilla.org/en-US/docs/Web/SVG

HTML `.html`

HTML is text with tags that say what is a heading, what is a paragraph and what is a link.

A browser does not show it as text; it *runs* it and draws a page.

An `.html` file stands on its own — double-click and it opens, with no internet at all.

make_page.py

```
html = """<!doctype html>
<html lang="en">
  <body>
    <h1>Hello</h1>
    <p>This is a <b>web page</b>.</p>
  </body>
</html>"""

open("page.html", "w", encoding="utf-8").write(html)
```

OUTPUT

```
(double-click page.html to open a browser)
```

The material you are reading is one such file. More in Module 6.

> That very property — a page is just a file — is what Module 7 is built on: web technology can power a purely local application that needs neither internet nor installation. On a work machine where you may not install software, that is often the only route to a tool of your own.

MORE ON THIS

- MDN — HTML — https://developer.mozilla.org/en-US/docs/Web/HTML

Which formats are AI-friendly `AI-friendly formats`

The rule is simple: the closer a format is to plain text, the less trouble you will have.

If the source of the data is available, use it — export a CSV instead of sending a screenshot of a table.

Conversion is always the first step, not a better prompt.

`.txt .md`	excellent — plain text, no conversion
`.csv`	excellent — a table as text
`.json`	excellent — structure as text
`.html .svg`	excellent — text with tags
`.docx .xlsx`	good — packed text, reads reliably
`.pdf from text`	fair — text extracts, table layout is lost
`.pdf from a scan`	poor — OCR first
`.jpeg .png`	poor — the model sees a picture, not data
`screenshot of a table`	worst — a source file almost always exists

> The order of the table is also the order in which you should hunt for your data. Before you photograph a screen, ask whether an export exists. Before you send a PDF, ask whether the spreadsheet it came from still exists. Every step back towards the source saves you one conversion and one place where data can go missing.
>
> **MORE ON THIS**
>
> - Anthropic — working with files — https://docs.claude.com/en/docs/build-with-claude/files

OCR — from picture to text OCR

OCR `optical character recognition` turns the letters in a picture back into plain text.

It works in steps: straighten the image, clean it, find the lines and letter shapes, then guess each character.

Because it guesses, it makes mistakes — most often confusing `0` with `O` and `1` with `l`.

So always check OCR output, especially amounts and reference numbers.

ocr.py

```python
# pip install pytesseract pillow  (+ the Tesseract program)
import pytesseract
from PIL import Image

image = Image.open("invoice_scan.png")
text = pytesseract.image_to_string(image, lang="eng")

print(text[:60])
```

OUTPUT

```
INVOICE no. 2026-0148
Date: 12/09/2026
Amount: 1,2O0.00 EUR
```

Look at the last line: 1,2O0 - OCR read the letter O instead of a zero.

> Modern language models can read an image directly, without a separate OCR program, and are often better with awkward layouts. The weakness is identical though — a guess remains a guess. A good office rule: use OCR for searching and reviewing, but never copy an amount from OCR into the accounts without looking at the original.
>
> MORE ON THIS
>
> - Tesseract OCR (open source) — https://github.com/tesseract-ocr/tesseract
> - Anthropic — reading images — https://docs.claude.com/en/docs/build-with-claude/vision

MODULE 4

Data and databases

From a spreadsheet to a real database, in four steps and without a leap.

The spreadsheet as a starting point `spreadsheet`

A spreadsheet is already almost a database: named columns, rows of values.

A formula is the same thing as a line of code — except you have to drag it down the column every time.

A program does the same sum, but tomorrow you can run it on a different file without touching anything.

total.py

```python
# In a spreadsheet you would write:  =SUM(C2:C4)
# In Python:

import csv

with open("guests.csv", encoding="utf-8") as f:
    amounts = [int(r["amount"]) for r in csv.DictReader(f)]

print(sum(amounts))
```

OUTPUT

```
475
```

Same result. The difference is that tomorrow you run this over 300 files.

> A spreadsheet hits its limits in three places: when the data no longer fits one table, when several people need it at once, and when the same routine has to be repeated every week. A database solves the first two, a program the third. Until one of those three hurts, a spreadsheet is a perfectly respectable choice — and no developer will tell you that.

MORE ON THIS

- Python — the csv module — https://docs.python.org/3/library/csv.html

Same data, four shapes `same data, four shapes`

One single row of data gets written four ways, and you will meet all four everywhere.

The content is identical in all of them — only the notation and the reader differ.

Once you see this, CSV, JSON and SQL stop being three separate mysteries.

four_shapes.py

```python
# 1) Spreadsheet - rows and columns
#      name   | amount
#      Novak  |    120

# 2) CSV - a table as text
csv_form = "name,amount\nNovak,120"

# 3) JSON - an object as text
json_form = '[{"name": "Novak", "amount": 120}]'

# 4) SQL - an instruction to a database
sql_form = "INSERT INTO customers (name, amount) VALUES ('Novak', 120);"
```

Four notations, one single fact: Novak, 120.

> Choosing between them is a question of purpose, not taste. CSV is for exchange. JSON is for programs talking to each other. SQL is for storing and finding. A spreadsheet is for a human who wants to look and fix. Almost all data work is really moving between these four shapes.

MORE ON THIS

- Wikipedia — Comma-separated values — https://en.wikipedia.org/wiki/Comma-separated_values

What a database is `database`

A database is a program that stores data and answers questions about it quickly.

It differs from a file in three ways: it enforces order, it searches millions of rows, and it survives many users at once.

The simplest database is a single file — SQLite ships inside Python, nothing to install.

database.py

```python
import sqlite3

db = sqlite3.connect("company.db")       # the file appears

db.execute("CREATE TABLE IF NOT EXISTS customers ("
           "id INTEGER PRIMARY KEY, name TEXT, amount INTEGER)")
db.execute("INSERT INTO customers (name, amount) VALUES ('Novak', 120)")
db.commit()

print(db.execute("SELECT * FROM customers").fetchall())
```

OUTPUT

```
[(1, 'Novak', 120)]
```

The whole database is one file. Your phone stores its messages exactly this way.

> The big databases (PostgreSQL, MySQL, SQL Server) are the same idea running as a separate server that many programs connect to at once. For a personal tool or a prototype, SQLite is almost always the right pick: nothing to install, nothing to configure, and you copy the entire database by copying one file.

MORE ON THIS

- Python — the sqlite3 module — https://docs.python.org/3/library/sqlite3.html
- SQLite — appropriate uses — https://www.sqlite.org/whentouse.html

Relational databases and keys `relational database, primary/foreign key`

A relational database keeps data in several tables that reference each other.

Every row has its own unique number, the `primary key`.

Another table points at it using that same number, a `foreign key`.

So a customer's details are written once even if the customer has fifty orders.

keys.py

```
db.execute("CREATE TABLE orders ("
           "id INTEGER PRIMARY KEY,"
           "customer_id INTEGER,"     # foreign key -> customers.id
           "amount INTEGER)")

db.execute("INSERT INTO orders (customer_id, amount) VALUES (1, 120)")
db.execute("INSERT INTO orders (customer_id, amount) VALUES (1, 45)")
db.commit()

rows = db.execute(
    "SELECT customers.name, orders.amount "
    "FROM orders JOIN customers ON customers.id = orders.customer_id"
).fetchall()

print(rows)
```

OUTPUT

```
[('Novak', 120), ('Novak', 45)]
```

The name 'Novak' is stored exactly once, yet appears in both rows.

> That property is what makes relational databases powerful: when a customer changes their email you fix it in one place and it is right everywhere. In a spreadsheet you would fix it in all fifty rows — and one would certainly stay stale. It is the same principle as the variable in 1.2, applied to data.

MORE ON THIS

- Wikipedia — Relational database — https://en.wikipedia.org/wiki/Relational_database

SQL `SQL`

SQL is the language you talk to a database in, and it reads remarkably like an English sentence.

You never say *how* the database should find the data — only *what* you want.

Four words cover most of it: `SELECT`, `FROM`, `WHERE`, `ORDER BY`.

query.sql

```sql
SELECT customers.name,
       SUM(orders.amount) AS total
FROM orders
JOIN customers ON customers.id = orders.customer_id
WHERE orders.amount > 50
GROUP BY customers.name
ORDER BY total DESC
LIMIT 10;
```

OUTPUT

```
Novak | 120
```

In English: take orders over 50, total them per customer, sort high to low, give me ten.

SQL is worth knowing even if you never program: when you ask an AI agent for a report out of a database it will write SQL, and you need to be able to check that it picked the right rows. The most common mistake is not syntax but a forgotten `WHERE`, which quietly lets cancelled and test records into the report.

MORE ON THIS

- SQLite — the SQL language — https://www.sqlite.org/lang.html
- Wikipedia — SQL — https://en.wikipedia.org/wiki/SQL

Document databases `document database / NoSQL`

A document database stores whole objects rather than rows — exactly the shape you saw in JSON.

Every record may carry different fields; no shape is agreed in advance.

The gain is freedom; the price is that keeping order becomes your job rather than the database's.

document.py

```python
# One document - in a relational database this would be two tables
customer = {
    "name": "Novak",
    "email": "novak@example.com",
    "orders": [
        {"date": "2026-09-01", "amount": 120},
        {"date": "2026-09-14", "amount": 45},
    ],
}

print(customer["orders"][0]["amount"])
```

OUTPUT

```
120
```

MongoDB, Firestore and their kin store precisely documents like this.

> The danger of document databases is quiet: with no agreed shape, a year later the same collection holds records with `email`, `e-mail` and `mail`. The database reports nothing, and the report silently drops a third of the customers. A relational database would simply have refused the entry.
>
> MORE ON THIS
>
> - Wikipedia — Document-oriented database — https://en.wikipedia.org/wiki/Document-oriented_database

Which one, when `choosing a store`

The choice almost always comes down to one question: is the data the same shape everywhere?

If it is, take a relational database — it will help you keep order.

If it is not and the shape is still moving, a document database gets you to a working prototype faster.

`Spreadsheet / CSV`	up to a few thousand rows, one user, manual work
`SQLite`	personal tool, prototype, single application — no install
`PostgreSQL / MySQL`	many users at once, data that matters, reporting
`MongoDB / Firestore`	record shape still changing, lots of nesting
`A JSON file`	settings and small lists — not a record system

> A practical note for working with an agent: tell it how many rows you expect, how many people will use the tool, and whether the data shape is still moving. Those three answers drive the decision — without them the agent picks whatever it has seen most often, which is usually overkill for your case.
>
> **MORE ON THIS**
>
> - SQLite — appropriate uses — https://www.sqlite.org/whentouse.html

MODULE 5

Addresses and networks

What a web address actually is, and how one computer finds another.

Anatomy of an address URL

A web address is not one word but six separate parts, each with its own job.

Once you can see where one part ends and the next begins, addresses stop being mysterious.

The part after the question mark is the `query` — in this very course it carries the language choice.

address.py

```python
from urllib.parse import urlparse

u = urlparse("https://www.zlataovca.si:443/course/index.html?lang=en#module3")

print(u.scheme)     # protocol
print(u.hostname)   # host (domain)
print(u.port)       # port
print(u.path)       # path to the file
print(u.query)      # query
print(u.fragment)   # section on the page
```

OUTPUT

```
https
www.zlataovca.si
443
/course/index.html
lang=en
module3
```

Look at this page's address in your browser - ?lang=en is the very same part.

> Two practical consequences. First: everything after the question mark is visible to anyone along the way and gets written into server logs, so passwords and personal data never belong there. Second: the `#section` is never sent to the server at all — it is an instruction to the browser about where to scroll on a page it already has.
>
> MORE ON THIS
>
> - MDN — what is a URL — https://developer.mozilla.org/en-US/docs/Web/API/URL
> - Python — urllib.parse — https://docs.python.org/3/library/urllib.parse.html

Domains and DNS `domain, DNS`

Computers do not call each other by name; they call each other by number.

DNS is the directory that turns `example.com` into that number.

You rent a domain from a registrar for a few tens of euros a year — that rents the name, not a server.

dns.py

```python
import socket


print(socket.gethostbyname("example.com"))
print(socket.gethostbyname("www.python.org"))
```

OUTPUT

```
172.66.147.243
151.101.64.223
```

The same domain may return a different number next week - the directory changes.

> This is why moving a website takes hours to reach everyone: the directory does not update everywhere at once, because intermediate servers remember answers for a while. The same property explains why a site sometimes works for you but not for a colleague — their machine still holds the old answer.
>
> MORE ON THIS
>
> - Wikipedia — Domain Name System — https://en.wikipedia.org/wiki/Domain_Name_System

IP address `IP address`

An IP address is a computer's house number on a network, written as four numbers from 0 to 255.

Some ranges are reserved for home and office networks and do not exist on the internet at all.

That is precisely why your machine cannot be reached from outside without help.

`127.0.0.1`	this computer, always and everywhere (`localhost`)
`192.168.x.x`	a home or office network
`10.x.x.x`	larger corporate networks
`172.16–31.x.x`	also private
`anything else`	a public address, reachable from the internet

> Because the private ranges are the same everywhere, your machine at home and a machine in the office can share the address `192.168.1.10` — which is fine, as they sit on separate networks. The consequence is that someone on the internet cannot simply type your address and open your page. The tunnel in 7.3 is what solves that.
>
> **MORE ON THIS**
>
> - Wikipedia — Private network — https://en.wikipedia.org/wiki/Private_network

localhost `localhost / 127.0.0.1`

`localhost` means *this computer* — an address that never leaves your machine.

When you run `zazeni-lokalno.bat` the server runs right here and nobody else can see it.

It is the safest place to try things: the internet knows nothing about it.

server.py

```
import http.server, socketserver

with socketserver.TCPServer(("127.0.0.1", 8080),
                            http.server.SimpleHTTPRequestHandler) as srv:
    print("Open http://localhost:8080")
    srv.serve_forever()
```

OUTPUT

```
Open http://localhost:8080
```

Three lines are an entire web server. This course runs on nearly this code.

> Notice the `127.0.0.1` in the code: the server listens only on that address, so only this machine can reach it. To let colleagues in the office see it, it would have to listen on `0.0.0.0`, meaning *on every network card*. That one number is the whole difference between 5.4 and 5.5.
>
> MORE ON THIS
>
> - Python — http.server — https://docs.python.org/3/library/http.server.html

The local network `LAN, 192.168.x.x`

When a server listens on every network card, everyone on the same wifi can reach it.

You then use your machine's address on that network rather than `localhost`.

It is the fastest way to show something to a colleague or open it on your own phone.

lan.py

```python
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect(("8.8.8.8", 80))      # sends nothing, just picks a card
print("http://" + s.getsockname()[0] + ":8080")
s.close()
```

OUTPUT

```
http://192.168.1.24:8080
```

This is the exact line the launcher prints - open it on your phone.

> The limitation is that the other device must be on the same network. On a phone that means wifi, not mobile data. Companies often keep the guest wifi separate from the office network, so the address works from one and not the other — which is not a bug in the program but a network setting.
>
> MORE ON THIS
>
> - Wikipedia — Private network — https://en.wikipedia.org/wiki/Private_network

Ports `port`

One computer runs many programs at once; the `port` says which of them should take the connection.

The address is the building, the port is the flat number.

If you get *port already in use*, somebody already lives in that flat — pick another number.

`80`	ordinary web pages (http)
`443`	secure web pages (https) — the default, which is why you never see it
`8080 8000 5000`	development servers you run yourself
`3306 5432`	databases (MySQL, PostgreSQL)
`22`	remote login to a server (SSH)

> This is why you rarely see a port in a web address: for `https://` the browser adds 443 itself. With your own server you must state it, because 8080 is nobody's default. To find out what currently occupies a port on Windows, `netstat -ano | findstr 8080` will tell you.
>
> **MORE ON THIS**
>
> - Wikipedia — list of TCP and UDP ports — https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

Request and response `HTTP request / response`

The web runs on a simple pattern: the browser sends a request, the server returns a response, the connection closes.

The request carries a method (`GET` or `POST`), a path, and some extra detail in `headers`.

The response carries a status code, headers and content — a file, an image or JSON.

request.py

```python
import urllib.request

with urllib.request.urlopen("http://localhost:8080/index.html") as response:
    print(response.status)
    print(response.headers["Content-Type"])
    print(len(response.read()), "bytes")
```

OUTPUT

```
200
text/html; charset=utf-8
2184 bytes
```

Exactly what a browser does when you type an address - it just draws the result too.

> An important property: between two requests the server remembers nothing — every request arrives new and empty. That is why cookies and login tokens exist: so the browser can say who you are all over again each time. The same property is why AI agents need context sent with every message — more on that in Module 10.

MORE ON THIS

- MDN — an overview of HTTP — https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

Status codes `status codes`

Every response opens with a three-digit number saying how the request went.

The first digit is all you need: 2 is fine, 3 is a redirect, 4 is your mistake, 5 is theirs.

So the famous 404 does not mean *broken*, it means *there is nothing at that address*.

`200 OK`	all good, content attached
`301 / 302`	it moved, go here instead
`400 Bad Request`	the request is malformed
`401 / 403`	not logged in / not allowed
`404 Not Found`	nothing at this address
`429 Too Many Requests`	too many calls too fast — wait
`500 Internal Server Error`	the server crashed — their fault
`503 Service Unavailable`	overloaded or under maintenance

> The 4xx/5xx split is very practical when working with agents: a 401 or 403 means a key or a permission on your side needs fixing. A 500 means no amount of prompt tweaking will help — the fault is with the service, and waiting or reporting it is the only sensible move. A 429 means you hit a rate limit, and a pause between calls is what you need.
>
> **MORE ON THIS**
>
> - MDN — HTTP status codes — https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

MODULE 6

How a web page works

What really happens between clicking a button and seeing the result.

The browser is the engine `browser / rendering engine`

A browser is not a window onto the internet; it is a program that fetches files and *runs* them on your machine.

Type an address and it downloads a handful of files, reads them, and draws a page out of them.

Everything you then see and click happens locally — not on the server.

What the browser fetches

```
GET /index.html           -> 2 KB  (content and structure)

GET /assets/css/style.css -> 12 KB  (appearance)

GET /assets/js/app.js     -> 18 KB  (behaviour)

GET /assets/img/logo.svg  -> 25 KB  (image)



Four requests, 57 KB. The page is drawn.
```

These are this course's real files. Press F12 -> Network and see for yourself.

> This is why a page can open without a server at all: if the files are already on disk there is nothing to fetch and the browser simply reads them. Double-clicking `index.html` does exactly that. The property is the foundation of Module 7, and the reason you can run your own application on a work machine where you may install nothing.
>
> MORE ON THIS
>
> - MDN — your first website — https://developer.mozilla.org/en-US/docs/Learn_web_development/Getting_started/Your_first_website

HTML — the structure HTML

HTML says *what* is on the page: a heading, a paragraph, a list, a button, an image.

Tags sit in angle brackets and almost always come in pairs — one open, one closed.

It says nothing about appearance; that is CSS's job.

index.html

```
<!doctype html>
<html lang="en">
  <body>
    <h1>Guests</h1>
    <ul id="list">
      <li>Ana Novak</li>
    </ul>
    <button id="add">Add a guest</button>
  </body>
</html>
```

Any element can carry an id - the name CSS and JavaScript call it by.

> When the browser reads HTML it turns it into a tree of elements called the DOM. That tree is what JavaScript changes — which is why a page can change without reloading. When you tell an agent *add a button*, what it really adds is one line to this structure.
>
> MORE ON THIS
>
> - MDN — HTML — https://developer.mozilla.org/en-US/docs/Web/HTML
> - MDN — what the DOM is — https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

CSS — the appearance `CSS`

CSS says *how* things should look: colours, sizes, spacing, layout.

A rule picks elements and gives them properties — nothing more than that.

The same HTML with different CSS looks like an entirely different product.

style.css

```css
:root {
  --accent: #d97757;      /* one colour, one place */
}

button {
  background: var(--accent);
  color: white;
  border: 0;
  border-radius: 8px;
  padding: 0.5rem 1rem;
}
```

This really is this course's code. Change --accent and everything recolours.

> That `--accent` is the variable from 1.2, living in CSS. It is exactly what the dark/light switch on this page is built on: not one character of HTML changes, only a handful of colour variables. If you ever tell an agent *change the colour scheme*, this is where it will work.
>
> **MORE ON THIS**
>
> - MDN — CSS — https://developer.mozilla.org/en-US/docs/Web/CSS

JavaScript — the behaviour `JavaScript / TypeScript`

JavaScript is the only programming language a browser understands directly.

It handles everything that happens *afterwards*: clicks, typing, validation, changing the page.

TypeScript is the same language with the data types from 1.3 added — mistakes surface as you type.

app.js

```
1  const button = document.getElementById("add");
2  const list = document.getElementById("list");
3
4  button.addEventListener("click", function () {
5      const row = document.createElement("li");
6      row.textContent = "New guest";
7      list.appendChild(row);
8  });
```

Clicking adds a row. The page never reloads.

> Notice that this example never calls a server — the row is added in the browser only and disappears on refresh. That is the crucial difference between *it looked like it changed* and *it was saved*. For the second you need an API call and a database, which are the next two concepts.

MORE ON THIS

- MDN — JavaScript — https://developer.mozilla.org/en-US/docs/Web/JavaScript
- TypeScript — documentation — https://www.typescriptlang.org/docs/

Frontend and backend `frontend / backend`

The frontend is everything running in the user's browser — HTML, CSS, JavaScript.

The backend is the program running on the server that holds the data and enforces the rules.

The line between them matters because the frontend is visible and editable by anyone.

where_it_runs.py

```
# BACKEND - runs on the server, the user never sees it
def may_view_invoice(user, invoice):
    return invoice.owner_id == user.id        # <- the real check

# FRONTEND - runs in the browser, anyone can read and change it
# if (user.isOwner) { showButton(); }         <- appearance only
```

Hiding a button is not security. Checking on the server is.

> This is the most common security mistake made by beginners and by AI agents alike: the rule is checked only in the browser. Anyone can open developer tools, edit the code and bring the button back. So the rule reads: the frontend handles convenience, the backend handles truth. Anything that matters must be checked where the user has no reach.

MORE ON THIS

- MDN — what an API is — https://developer.mozilla.org/en-US/docs/Glossary/API

The API call `API call`

An API is an agreed list of questions one program may ask another.

The browser sends a request to an address and the server returns JSON — no pictures, no styling.

Mobile apps, other programs and AI agents all come in through that same door.

call.js

```
async function saveGuest(name) {
    const response = await fetch("/api/guests", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ name: name })
    });

    if (!response.ok) throw new Error("Error " + response.status);
    return await response.json();
}
```

OUTPUT

```
{ "id": 42, "name": "New guest", "saved": true }
```

POST means 'store this', GET means 'give me'. The answer is JSON from 3.5.

> Now the whole picture assembles: the JSON from Module 3 is the shape of the message, the status code from Module 5 says how it went, and the database from Module 4 is where the data actually lands. MCP in Module 10 works exactly the same way — the agent sends a request to a tool and gets JSON back.

MORE ON THIS

- MDN — using fetch — https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

The journey of one click `the journey of one click`

One click sets off a chain of six steps that completes in a few hundredths of a second.

Any step can fail — and the status code tells you which one did.

Once you can see this path, *why doesn't it work* becomes a far sharper question.

`1. Browser`	the user clicks, JavaScript fires
`2. Request`	`POST /api/guests` carrying JSON
`3. Server`	checks permissions and rules (`6.5`)
`4. Database`	`INSERT INTO guests ...` — the data is written
`5. Response`	`200 OK` and JSON of the new record
`6. Browser`	receives it and updates the page

> Diagnosis by step: if the request never appears in developer tools, step 1 is at fault. A 401 or 403 means step 3. A 500 means step 3 or 4 on the server. A 200 with nothing changing on screen means step 6. That breakdown is exactly what you hand an agent, and the difference between *it doesn't work* and *the request returns 500* is the difference between an hour of searching and a minute.
>
> **MORE ON THIS**
>
> - MDN — HTTP status codes — https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Static and dynamic `static vs. dynamic`

A static page is a file that looks the same to everyone — the server just passes it along.

A dynamic page is assembled per user, because it has to look in a database first.

This course is static, which is why it works with no internet and no server.

difference.py

```
# STATIC - the server just sends the file
#   GET /index.html  ->  index.html

# DYNAMIC - the server builds the page on every request
def page_for(user):
    orders = db.execute(
        "SELECT * FROM orders WHERE customer_id = ?", (user.id,)
    ).fetchall()
    return build_html(orders)
```

Static is faster and safer. Dynamic is needed when content is personal.

> A common and very practical middle road: keep the page static and fetch the data with the API calls from 6.6. You get the speed of a static page and the freshness of a dynamic one. That is what most modern applications do — and what the course you are reading does, except that it reads its own data files instead of a server.

MORE ON THIS

- Wikipedia — Static web page — https://en.wikipedia.org/wiki/Static_web_page

MODULE 7

Where code lives

The hosting ladder from your own machine to the cloud — cost, permissions and traps at every rung.

Rung one: my own machine `localhost`

The lowest rung of hosting is your own computer: the program runs when you start it and dies when you close the window.

It costs nothing, needs nobody's permission, and nobody else sees a thing.

For learning, experimenting and personal tools that is entirely enough — and often the best choice.

the launcher

```
D:\project> python tools\server.py


  On this computer:     http://localhost:8080

  On the local network: http://192.168.1.24:8080


  Stop with:  Ctrl + C
```

While that window runs, the page exists. Close it and it is gone.

> That nothing is permanent is an advantage at the start, not a shortcoming. You cannot break anything, you cannot leak anything, and you can delete it all at any moment. Before you put anything higher up this ladder, make it work flawlessly on this rung.

MORE ON THIS

- Python — http.server — https://docs.python.org/3/library/http.server.html

Rung two: the office `LAN hosting`

If the server listens on every network card, everyone on the same wifi can see it.

It still costs nothing and still requires no installation on anyone else's device.

For showing colleagues, or a tool used by one team, that is often enough.

`Cost`	nothing
`Permissions`	none — unless IT blocks the port
`Who can see it`	everyone on the same network
`When you shut down`	unreachable
`Typical use`	a team tool, a demo in a meeting

> The limitation that bites in companies is organisational rather than technical: guest wifi is often separated from the office network, and a firewall may block everything but browsing. If the address works for you and not for a colleague, first check that you are really on the same network.

MORE ON THIS

- Wikipedia — Private network — https://en.wikipedia.org/wiki/Private_network

Rung three: a tunnel `tunnel (trycloudflare, ngrok)`

A tunnel is a program that opens a connection outward from your machine and hands you a public address.

Because you open the connection, the firewall has nothing to let in — which is why it works at the office.

The address is temporary: it lives as long as the window runs, and is different next time.

the public launcher

```
D:\project> python tools\tunnel.py


  Opening a public tunnel ...


  +---------------------------------------------+
  |  https://some-random-words.trycloudflare.com  |
  +---------------------------------------------+


  The address works while this window runs.
```

This is how the address you may be reading this on came to exist.

> A security note worth saying out loud: that address is public. Anyone who gets it sees everything the server serves — no password, no login. So never put personal data, keys or internal documents in the folder a tunnel serves. For serious use there are account-based tunnels that can require a login.

MORE ON THIS

- Cloudflare — TryCloudflare (quick tunnels) — https://developers.cloudflare.com/cloudflare-one/connections/connect-networks/do-more-with-tunnels/trycloudflare/

Rung four: a rented server VPS

A VPS is a computer in somebody else's data centre, rented for a few euros a month. It runs continuously, has a permanent public address, and you can install anything on it. The price of that freedom is that you are responsible for it: updates, security, backups.

`Cost`	roughly €5–20 a month
`Permissions`	a card, and someone to approve it
`Who can see it`	the whole internet
`When you shut down`	it keeps running
`Typical use`	a real application with a database and users

> The most common mistake at this rung is setting a server up and then forgetting it. An unmaintained server with a public address becomes a target within months, so the rule is: rent a VPS only once you know who will keep it updated. For many projects the next rung is the better answer.

MORE ON THIS

- Wikipedia — Virtual private server — https://en.wikipedia.org/wiki/Virtual_private_server

Rung five: static hosting `static hosting`

If a page is static there is nothing to run — somebody just has to serve the files.

Providers like GitHub Pages or Netlify do that for free, with a permanent address and an HTTPS certificate.

For course material, a deck, documentation or a personal site, this is almost always the right choice.

`Cost`	free for small sites
`Permissions`	just an account with the provider
`Who can see it`	the whole internet, at a permanent address
`When you shut down`	it keeps running
`Limitation`	no database and no server-side code

> This course could be hosted exactly this way: being static, it would only need uploading and would have a permanent address independent of your machine. We chose a tunnel because it is simpler for a prototype and needs no account — but if you share the material regularly, static hosting is the next sensible step.
>
> **MORE ON THIS**
>
> - GitHub Pages — https://pages.github.com/
> - Netlify — documentation — https://docs.netlify.com/

Rung six: the cloud `cloud (AWS, Azure)`

The cloud is not one computer but hundreds of services you assemble as needed and pay for by use.

The upside is that it scales itself; the downside is complexity and a bill that is hard to predict.

For a tool used by one team it is almost always overkill.

`Cost`	by usage — from cents to surprises
`Permissions`	a business account, often procurement too
`Who can see it`	whatever you configure
`Difficulty`	high — it is a profession
`Typical use`	many users, much data, uptime requirements

> Worth knowing when working with an AI agent: it will often propose a cloud solution simply because it has seen the most of them. Write into your `architecture.md` (Module 9) that this is a tool for ten people with no outside access, and it will propose something far simpler — and something you can actually maintain.

MORE ON THIS

- Wikipedia — Cloud computing — https://en.wikipedia.org/wiki/Cloud_computing

Web technology without the web `local web app, offline`

HTML, CSS and JavaScript do not need the internet — they need a browser.

A folder of files can be a fully working application you open by double-clicking.

On a work machine where you may not install software, this is often the only route to a tool of your own.

The course you are reading is exactly that: it runs with no server and no network.

Two routes to the same page

```
1) With a server:

   D:\project> python tools\server.py

   http://localhost:8080



2) Without a server:

   double-click  D:\project\index.html

   file:///D:/project/index.html



Same page. No installation, no internet.
```

Which is why this course links to no external library at all.

> There are three limits. Without a server there is no shared database (data stays in one user's browser), a browser may not read arbitrary files off the disk for security reasons, and some capabilities are only allowed over `https`. For a personal calculator, a viewer, a checklist, or material exactly like this, none of that matters.
>
> **MORE ON THIS**
>
> - MDN — progressive web apps — https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps

What may go on a public address `what to expose`

Once something has a public address, assume somebody will find it — even if you told nobody.

Search engines, automated scanners and the merely curious find addresses faster than you expect.

The rule is simple: only put on a public address what you would happily pin to a noticeboard.

`Course material, decks`	fine
`Customers' personal data`	never without a login
`Internal documents`	never
`` `.env` files, keys, passwords ``	never — not even in the project folder
`A database`	never directly; only through a server that checks
`Test data with real names`	better not — invent them

> With AI agents there is one more trap: an agent building an application will happily add a quick data-viewing page with no login, because nobody said otherwise. So write explicitly into `architecture.md` (Module 9) who may see what — otherwise the default is *everyone*. And before you point a tunnel at a folder, look at what is in it.
>
> **MORE ON THIS**
>
> - MDN — web security — https://developer.mozilla.org/en-US/docs/Web/Security
> - Wikipedia — firewall — https://en.wikipedia.org/wiki/Firewall_(computing)

MODULE 8

Languages and tools

Why there are so many programming languages, what each is for, and what libraries are.

Why there are so many languages `programming languages`

Every language can do the same things — they differ in what they make easy and what they make hard.

Some sit close to the machine and are fast; others sit close to the human and are fast to write.

Choosing a language is almost always a decision about where the program must run, not a matter of taste.

same_program.txt

```
# Python
print("Hello")

// JavaScript
console.log("Hello");

// C
printf("Hello\n");

-- SQL
SELECT 'Hello';
```

Four languages, one task. The differences are in detail, not in idea.

> The useful consequence for a non-technical reader: when an agent proposes a language, ask *why that one*. A good answer sounds like *because this has to run in a browser* or *because the best PDF library is in Python*. A poor answer is *because it is popular*.

MORE ON THIS

- Python — general FAQ — https://docs.python.org/3/faq/general.html

Compiled and interpreted `compiled vs. interpreted`

A compiled language is translated into machine code before it runs — the result is a standalone `.exe`.

An interpreted language is read line by line as it runs, so the machine needs the language installed.

Compiled is faster to execute; interpreted is faster to write and fix.

Two ways to run

```
COMPILED (C):

  gcc program.c -o program.exe    <- compile (once)

  program.exe                     <- run (many times)



INTERPRETED (Python):

  python program.py               <- compile AND run, every time
```

Which is why Python needs Python installed, and an .exe needs nothing.

> The difference also explains why this course's launchers are `.bat` files and not `.exe`: a `.bat` is plain text that Windows reads as it goes, so you can open it and check what it does. With an unsigned `.exe` you cannot — which is exactly why work machines often block them.

MORE ON THIS

- Wikipedia — Interpreter — https://en.wikipedia.org/wiki/Interpreter_(computing)

C and C++ `C / C++`

Closest to the machine; used where every thousandth of a second counts.

Operating systems, drivers, games, device controllers — and the foundation of almost every other language.

example.c

```c
#include <stdio.h>

int main(void) {
    int amount = 100;
    printf("With VAT: %.2f\n", amount * 1.22);
    return 0;
}
```

The type (int) must be written out. The compiler guesses nothing.

> For a non-programmer only one thing matters: if an agent proposes C or C++, ask whether you are really solving a speed problem. In office work the answer is almost always no — and Python will give the same result in a tenth of the lines.

MORE ON THIS

- isocpp.org — about C++ — https://isocpp.org/

Python Python

Readable, extremely widespread, and excellent for data work, automation and AI.

It has libraries for reading files, tables, PDFs and web services — for almost anything.

For a non-technical user it is nearly always the right first choice.

example.py

```python
import csv

with open("invoices.csv", encoding="utf-8") as f:
    total = sum(int(r["amount"]) for r in csv.DictReader(f))

print(f"Total with VAT: {total * 1.22:.2f}")
```

OUTPUT

```
Total with VAT: 579.50
```

Five lines do what C would need fifty for.

> Python is also the language AI agents write most reliably, simply because there is more of it in their training data. Unless you have a specific reason to choose otherwise, it is the best language to collaborate with an agent in — and it is already installed on your machine.
>
> MORE ON THIS
>
> - Python — the official tutorial — https://docs.python.org/3/tutorial/introduction.html

JavaScript and TypeScript `JavaScript / TypeScript`

JavaScript is the only language that runs directly in a browser, which makes it unavoidable on the web.

TypeScript is JavaScript with types added; mistakes surface while you write, not at the user.

With Node.js the same language also runs outside the browser, on a server.

example.ts

```ts
function withVat(price: number, rate: number = 22): number {
    return price * (1 + rate / 100);
}

console.log(withVat(100));
console.log(withVat("one hundred"));   // <- TypeScript flags this here
```

That ': number' is the whole difference between TypeScript and JavaScript.

> A practical rule: if the product lives in a browser, it will contain JavaScript or TypeScript whether you like it or not. If the job is data processing, Python is usually the shorter route. Many projects use both — which is not confusion but the normal state of affairs.
>
> **MORE ON THIS**
>
> - MDN — JavaScript — https://developer.mozilla.org/en-US/docs/Web/JavaScript
> - TypeScript — documentation — https://www.typescriptlang.org/docs/

Java and C# `Java / C#`

The languages of large business systems: banking, insurance, ERP, public administration.

Both are stricter and more verbose, which pays off when fifty people work on one codebase for ten years.

Example.java

```java
public class Example {
    public static void main(String[] args) {
        double amount = 100;
        System.out.println("With VAT: " + amount * 1.22);
    }
}
```

Six lines for the same output. Structure is the price of large teams.

> If your company has an internal system, chances are it is written in one of these two. You almost certainly do not need them for a tool of your own — but they are the reason that integrating with an internal system always requires a developer.
>
> **MORE ON THIS**
>
> - dev.java — the official site — https://dev.java/
> - Microsoft — C# — https://learn.microsoft.com/en-us/dotnet/csharp/

SQL SQL

SQL is not for writing programs but for questioning databases — which is why it turns up everywhere.

Learning to read it is the fastest-repaying investment a non-technical person can make.

query.sql

```sql
SELECT month, SUM(amount) AS total
FROM invoices
WHERE year = 2026
GROUP BY month
ORDER BY month;
```

SQL is covered properly in Module 4.

> What is special about SQL is that you describe the *result*, not the procedure. That is why it is so short — and why it is so dangerous when you forget a condition: `DELETE FROM invoices` without a `WHERE` deletes everything.
>
> **MORE ON THIS**
>
> - SQLite — the SQL language — https://www.sqlite.org/lang.html

Bash and PowerShell `Bash / PowerShell`

These are command-line languages: made for driving other programs, not for writing applications.

You use them to automate what you would otherwise click — copying, renaming, launching.

PowerShell

```powershell
# rename every PDF in the folder by its modification date

Get-ChildItem *.pdf | ForEach-Object {

    Rename-Item $_ ("invoice_" + $_.LastWriteTime.ToString("yyyy-MM-dd") + ".pdf")

}
```

Windows uses PowerShell, Mac and Linux use Bash. Same idea.

> The `.bat` files that launch this course are the simplest form of the same thing. For a non-programmer this is often the fastest win with AI: ask an agent for a script that tidies a folder of two hundred files and you have it in ten seconds. Just copy the folder first.

MORE ON THIS

- Microsoft — PowerShell — https://learn.microsoft.com/en-us/powershell/scripting/overview
- GNU — the Bash manual — https://www.gnu.org/software/bash/manual/bash.html

Runtimes and Node.js `runtime / Node.js`

A language does not run by itself — it needs a program that executes it, called a `runtime`.

For Python that is `python.exe`; for JavaScript in a web page it is the browser itself.

Node.js is that same JavaScript runtime outside the browser, so it can read files and serve pages.

When somebody says *you need Node*, this is what they mean: without a runtime, the language cannot run.

What is installed

```
D:\project> python --version

Python 3.13.5



D:\project> node --version

'node' is not recognized as an internal or external command



-> Python is installed, Node.js is not.
```

Which is why this course needs no Node.js - everything runs in the browser and in Python.

> This is also the most common reason instructions found online *do not work*: they assume a runtime your machine does not have. On a work computer installing one is often not allowed, so it is worth telling an agent up front what you actually have — otherwise it will propose something you cannot run.

MORE ON THIS

- Node.js — about — https://nodejs.org/en/about

Libraries `libraries (pip, npm)`

A library is code somebody else has already written that you use instead of writing your own.

For Python you install one with `pip install`, for JavaScript with `npm install`.

Reading PDFs, working with spreadsheets, drawing charts, sending mail — libraries exist for all of it.

Installing a library

```
D:\project> pip install pypdf

Collecting pypdf

  Downloading pypdf-6.0.0-py3-none-any.whl

Successfully installed pypdf-6.0.0



D:\project> python

>>> from pypdf import PdfReader        <- now available
```

Instead of a thousand lines to read a PDF, you write one import line.

> It is also an entry point for trouble: every library is somebody else's code running on your machine. Before installing something unfamiliar, check the name is spelled correctly — libraries with deliberately similar names exist. On a work machine, `pip install` is often exactly what IT blocks.
>
> **MORE ON THIS**
>
> - PyPI — the Python package index — https://pypi.org/
> - npm — about the registry — https://docs.npmjs.com/about-npm

Library or your own code `build or borrow`

Use a library when the problem is general and well solved: dates, PDFs, encryption, networking.

Write it yourself when the problem is yours, small and specific — your company's rules are in no library.

Never write your own encryption, date handling or password checking; there, *yourself* is almost always wrong.

`Reading PDFs, spreadsheets, images`	library — reliably
`Dates and time zones`	library — always
`Encryption, passwords`	library — never yourself
`Your business rules`	your own code — nobody else knows them
`A few dozen lines of logic`	your own code — fewer dependencies
`A library for something trivial`	better not — a dependency is never free

> Every library is a debt: somebody has to maintain it, update it, and one day it will stop working with a new version of the language. So: a big library for a big problem is a saving, a small library for a small problem is often a burden. Say this to your agent explicitly, or it will add a dependency for every trifle.
>
> **MORE ON THIS**
>
> - Python — installing packages — https://packaging.python.org/en/latest/tutorials/installing-packages/

Dependencies and versions `dependencies, versions`

A program using five libraries depends on five other people's projects and their changes.

So versions get written into a file — to make sure the program still runs the same a year from now.

A label like `6.0.0` reads major.minor.patch; the first number changes when something is no longer compatible.

requirements.txt

```
pypdf==6.0.0
pytesseract==0.3.13
pillow==11.0.0
```

'pip install -r requirements.txt' installs exactly these versions.

> Separate environments (`python -m venv .venv`) let two projects on one machine use different versions of the same library. For a non-technical user the important part is this: when something *worked yesterday*, suspect a version change first — and never delete `requirements.txt`.
>
> **MORE ON THIS**
>
> - Semantic versioning — https://semver.org/
> - Python — virtual environments — https://docs.python.org/3/library/venv.html

MODULE 9

How good code is made

Breaking problems down, modules, tests, and the file an AI agent builds from.

Breaking a problem down `decomposition`

A large problem cannot be solved; only a sequence of small ones can.

Decomposition means splitting a task into steps you can each describe on their own.

If you cannot describe a step in one sentence, it is not small enough yet.

This is the skill you need most when working with AI — more than any programming language.

decomposition.txt

```
TASK: "sort out my invoices"      <- far too big, the agent will guess

BROKEN DOWN:
  1. read every .pdf from the "inbox" folder
  2. pull the date, number and amount out of each one
  3. if extraction fails, move the file to "manual"
  4. write the rows into invoices.csv
  5. rename the file to YYYY-MM-DD_number.pdf
  6. print how many succeeded and how many did not
```

Six sentences. Each can be checked on its own - and each can be its own function.

> Notice step three: it says what should happen when something goes wrong. That single step separates an instruction that yields a usable tool from one that yields a program that crashes on the first unusual file. At every step, ask: *and what if it doesn't work?*

MORE ON THIS

- Wikipedia — separation of concerns — https://en.wikipedia.org/wiki/Separation_of_concerns

Main and modules `main + modules`

Code gets split into modules: each file does one kind of work.

The `main` file only sets the order — it does nothing difficult itself.

That way you can fix or replace one part without touching the rest.

main.py

```python
from reading import read_pdfs
from extract import extract_data
from writing import write_csv

def main():
    for document in read_pdfs("inbox"):
        data = extract_data(document)
        write_csv(data, "invoices.csv")

if __name__ == "__main__":
    main()
```

Nine lines say what the program does. The detail lives in three other files.

> This split is practically essential when working with an AI agent, for a quite different reason: the agent can fix one file without reading and rewriting the whole program. Less code to process at once means fewer tokens, fewer mistakes, and less chance of breaking something else along the way.

MORE ON THIS

- Wikipedia — separation of concerns — https://en.wikipedia.org/wiki/Separation_of_concerns

Naming `naming`

A name should say what a thing is or does — not how it was built.

A good name saves a comment; a bad one demands an explanation every single time.

names.py

```python
# bad
def proc(d, x):
    return [i for i in d if i[1] > x]

# good
def invoices_above(invoices, threshold):
    return [i for i in invoices if i.amount > threshold]
```

The second version needs no comment. The name is the comment.

> With an agent, naming is a surprisingly powerful tool: the agent infers intent from names. Call a function `invoices_above` and it writes code that filters invoices. Call it `proc` and it guesses — based on what it has seen in other projects, not in yours.

MORE ON THIS

- PEP 8 — the Python style guide — https://peps.python.org/pep-0008/

What a test is `test`

A test is a small program that runs your code and checks the result is what you expected.

You write it once, and it runs automatically on every change.

Its value is not proving correctness today but shouting when something breaks tomorrow.

test_vat.py

```
1  from vat import with_vat
2
3  def test_basic_case():
4      assert with_vat(100) == 122.0
5
6  def test_reduced_rate():
7      assert with_vat(100, 9.5) == 109.5
8
9  def test_zero():
10     assert with_vat(0) == 0
```

OUTPUT

```
3 passed in 0.01s
```

assert means 'I claim that'. If the claim fails, the test fails.

> The most useful tests are not the ordinary cases but the edges: an empty list, a negative number, a missing field, an enormous value. That is where the bug from 2.2 hides — and exactly where an AI agent most often overlooks a case.

MORE ON THIS

- pytest — documentation — https://docs.pytest.org/en/stable/
- Python — unittest — https://docs.python.org/3/library/unittest.html

Kinds of tests `unit / integration / end-to-end`

Unit tests check a single function; they are fast and you can have many.

Integration tests check that parts work together — a program and a database, say.

End-to-end tests follow a whole user journey; they are the slowest and the most brittle.

`Unit test`	one function · milliseconds · lots of them
`Integration test`	several parts together · seconds · a few dozen
`End-to-end test`	the whole application · minutes · key paths only
`Manual test`	a human at a screen · always needed for appearance

> The classic advice is a pyramid: many fast unit tests at the bottom, few slow ones on top. The reason is simple — when a unit test fails you know exactly which function is at fault; when an end-to-end test fails you only know something, somewhere along the way, is wrong.
>
> **MORE ON THIS**
>
> - Martin Fowler — the test pyramid — https://martinfowler.com/bliki/TestPyramid.html
> - Wikipedia — test automation — https://en.wikipedia.org/wiki/Test_automation

Why tests matter more with AI `tests with AI`

An agent writes code faster than you can read it — tests are the only way checking keeps pace.

Because an agent is non-deterministic (2.6), you cannot assume an answer is right just because it reads convincingly.

A test is a requirement written so that it checks itself — which makes it a better instruction than a sentence.

The best order is: tests first, then code, then verification.

Working order with agents

```
1. You:      describe the requirement

2. Agent A:  writes the TESTS (and nothing else)

3. You:      read the tests - are these really your requirements?

4. Agent B:  writes the CODE, never sees the tests

5. Computer: runs the tests



   3 passed, 1 failed  ->  agent B fixes until it is green
```

Agent B cannot see the tests, so it cannot write code tailored to them.

> Why two separate agents: if one agent writes both the tests and the code, it will quietly shape the tests around the code it just wrote — and both will share the same wrong assumption. The separation is cheap and effective: an agent that cannot see the tests has to write code that actually works rather than code that looks like a solution. Detail in 11.7.
>
> **MORE ON THIS**
>
> - Claude Code — overview — https://docs.claude.com/en/docs/claude-code/overview

Git and commits `Git, commit`

Git remembers every state of your code that you confirm — and lets you return to any of them.

A commit is a snapshot with a message: *what I changed and why*.

With an agent it is your safety net: if it wrecks the code, one line takes you back to the last working state.

Basic git

```
D:\project> git init

D:\project> git add .

D:\project> git commit -m "Working PDF invoice reader"



D:\project> git log --oneline

a3f9c21 Working PDF invoice reader



D:\project> git restore .        <- undo everything since the last commit
```

Commit BEFORE you let an agent start changing things. Always.

`git init`	start tracking changes in this folder
`git add .`	stage every change for the commit
`git commit -m "..."`	save a snapshot with a message
`git log --oneline`	show the history
`git diff`	what changed since the last commit
`git restore .`	throw away all unsaved changes

> For a non-technical user one rule matters above all: **commit before every session with an agent**. When the agent does something strange — and it will — that is the difference between `git restore .` and an afternoon of recovery. On small projects Git is not about collaboration; it is about that one undo.

MORE ON THIS

- Git — documentation — https://git-scm.com/doc

The architecture.md file `architecture.md`

This is the document describing what you are building, written before the agent writes a line of code.

It holds the goal, the inputs, the outputs, the constraints, the file layout and the decisions already made.

The agent reads it at the start of every session, so it stops revisiting questions you have answered.

It is the cheapest way to stop an agent building something you never asked for.

architecture.md

```
# Incoming invoice processing tool

## Goal
Turn PDF invoices in the "inbox" folder into a CSV, and rename
the files by date and invoice number.

## Input
- PDF files, ~200 a month, some of them scanned

## Output
- invoices.csv (date, number, amount, supplier, filename)
- failures moved to the "manual" folder

## Constraints
- Windows, Python 3.13, no Node.js
- no software installs requiring administrator rights
- data must not leave the machine

## Decisions
- pypdf library; OCR only when extract_text() comes back empty
- no database, CSV is enough
```

This file is worth more than an hour of conversation with an agent.

> The *Constraints* section is the one people leave out and the one that pays most. Without it an agent will propose something needing Node.js, a cloud account or administrator rights — none of which may be possible on a work machine. As a conversation produces decisions, write them here; this is also what preserves meaning when you run out of context window (Module 10).

MORE ON THIS

- Claude Code — project memory file — https://docs.claude.com/en/docs/claude-code/memory

MODULE 10

AI, agents and Claude

Tokens, context window, agents, skills, MCP — and which of these actually affects your work.

What a language model is LLM

A language model predicts which piece of text most likely follows what has been written so far.

It does not look things up in a database and does not understand in the human sense — it computes the most probable continuation.

Because the prediction is probabilistic, an answer can be excellent or confidently wrong with equal ease.

How an answer is produced

```
Input:  "The capital of Slovenia is"



The model scores candidate next pieces:

   " Ljubljana"   98.2 %

   " a"            0.7 %

   " known"        0.4 %

   ...



It picks the most likely, appends it, and repeats.
```

The answer is built piece by piece, not as a whole. That is why you watch it appear.

> Everything else in this module follows from that. The model has no memory between conversations, does not know what is true, and has no access to your files unless you explicitly give it. All it does is continue the text it can see. A surprising amount of useful work fits into that one ability — but only if you know its limits.

MORE ON THIS

- Anthropic — models overview — https://docs.claude.com/en/docs/about-claude/models/overview

Tokens `token`

A model reads neither letters nor words but tokens — chunks of text averaging three to four characters.

Common English words are often a single token, while rarer words and other languages split into several.

So the same content in a less-represented language costs noticeably more tokens — and more money.

Same sentence, two languages

```
EN: "The invoice was paid yesterday."

    [The][ invoice][ was][ paid][ yesterday][.]        ~6 tokens



SL: "Racun je bil vceraj placan."

    [Rac][un][ je][ bil][ vc][eraj][ pla][can][.]      ~9 tokens



Over longer texts the ratio settles around 1.5x to 2x.
```

Rule of thumb: 1 token ~ 4 characters. Only the model can count exactly.

> Two practical consequences. First, if you work with very large texts and cost matters, English is cheaper — even for Slovene content, the instructions can be in English. Second, estimates of *how many pages fit in a conversation* are more pessimistic in other languages than the English examples in documentation suggest.
>
> **MORE ON THIS**
>
> - Anthropic — counting tokens — https://docs.claude.com/en/docs/build-with-claude/token-counting

What tokens cost `pricing`

You pay for two things: the tokens you send (input) and the tokens the model writes (output).

Output costs roughly five times input, so long answers cost far more than long questions.

On a subscription you do not pay per token, but the same numbers decide when you hit a usage limit.

`Claude Opus 5`	$5 in / $25 out per million tokens · 1M context
`Claude Sonnet 5`	$2 in / $10 out · 1M context
`Claude Haiku 4.5`	$1 in / $5 out · 200K context
`One million tokens`	roughly 700,000 English words
`A typical question`	a few hundred tokens — hundredths of a cent
`A whole book in context`	a few hundred thousand tokens — euros, not cents

> Prices change; the ratios do not. A stronger model costs more, output costs more than input, and repeatedly resending the same long text is the biggest silent cost. `Prompt caching` exists for exactly that, charging a resent prefix far less. The figures above held when this course was written (September 2026) — check the official page.
>
> MORE ON THIS
>
> - Anthropic — pricing — https://docs.claude.com/en/docs/about-claude/pricing
> - Anthropic — prompt caching — https://docs.claude.com/en/docs/build-with-claude/prompt-caching

Prompt and context `prompt / context`

The prompt is what you write; the context is everything the model sees alongside it.

Context includes the whole conversation so far, attached files, and the instructions it started with.

The model knows nothing that is not in the context — so *I told you earlier* only works if earlier is still there.

What the model actually receives

```
[system instructions]      who you are, how to answer
[architecture.md]          1,200 tokens
[previous 14 messages]     8,400 tokens
[attachment invoice.pdf]   3,100 tokens
[your question]               40 tokens
                           -----------------
                           12,740 tokens sent with EVERY message
```

Which is why the second question in a long chat costs more than the first.

> It also explains behaviour that feels illogical: correct an instruction halfway through and the model still sees the original too. What is written stays written. So for longer work, a short summary of decisions saved in `architecture.md` plus a fresh conversation beats a long, repeatedly-corrected one.

MORE ON THIS

- Anthropic — context windows — https://docs.claude.com/en/docs/build-with-claude/context-windows

The context window `context window`

The context window is the ceiling on how much text the model can see at once.

For Claude Opus 5 and Sonnet 5 that is a million tokens; for Haiku 4.5, two hundred thousand.

As the window fills, conversation gets slower and dearer; once it is full, something has to go.

A million tokens sounds enormous — but a real project with code and documents eats it faster than you expect.

Filling the window

```
[##########.............................]  25 %   comfortable

[####################...................]  50 %   still fine

[##############################........]  80 %   time to summarise

[########################################] 100 %   the oldest part drops out
```

Claude Code shows you the percentage as you go.

> Why answers degrade in very long conversations: the model has to weigh more and more text, much of it irrelevant — abandoned attempts, dead ideas, errors already fixed. That is not a malfunction but a consequence. The fix is not a better prompt; it is cleaning the context.

MORE ON THIS

- Anthropic — context windows — https://docs.claude.com/en/docs/build-with-claude/context-windows

Summarise and start fresh `summarise and continue`

When the window is around eighty percent full, ask the agent for a summary of where things stand.

The summary should hold: the goal, what is already done, which decisions are settled, and the next step.

Then open a new conversation, paste the summary, and carry on — at a tenth of the cost and with better answers.

This is the most useful habit in working with AI, and the one people put off longest.

summary-prompt.txt

```
Write a handover summary of this session for a fresh conversation.
Include:
  1. the goal of the project in two sentences
  2. what is already built and working
  3. decisions made and why (e.g. why CSV and not a database)
  4. the next step
  5. known traps and what NOT to try again

Write it so that someone with no context can continue.
```

Save the summary into architecture.md - then it outlives the conversation.

> Point five is the one people leave out and the one that pays most: the list of things already tried that did not work. Without it, the new conversation will confidently propose the same dead end. Claude Code has a `handoff` skill that produces this summary to a proven pattern (Module 11).

MORE ON THIS

- Claude Code — project memory file — https://docs.claude.com/en/docs/claude-code/memory

Spending fewer tokens `saving tokens`

Cost depends not on how much you write but on how much the model re-reads each time.

The big savings are in shorter conversations, smaller files and sharper questions.

`A new conversation per topic`	the biggest saving of all
`Summarise at 80 % of the window`	a tenfold cut in context
`Smaller files (modules)`	the agent reads 200 lines, not 3,000
`A precise question`	a shorter answer, less correcting
`Send the CSV, not a screenshot`	fewer tokens and a better result
`Do not attach the whole folder`	only the files that are genuinely needed
`A weaker model for simple work`	Haiku to tidy, Opus to think
`Do not repeat context`	the model already sees the whole conversation

> The first row is worth all the rest combined. People leave one conversation running for a week and wonder at the cost — while every new question drags along everything since Monday. The rule: one conversation, one task. When the task is done, close it.
>
> **MORE ON THIS**
>
> - Anthropic — prompt caching — https://docs.claude.com/en/docs/build-with-claude/prompt-caching

Hallucinations `hallucination`

A hallucination is an answer that sounds right but is not true — an invented figure, a function that does not exist, a dead link.

It is neither a lie nor a bug in the ordinary sense: the model computed the most likely continuation and it happened to be false.

It is commonest exactly where the model has no data: precise numbers, names, dates, links and citations.

Hence the rule: check anything checkable — and ask for a source rather than a claim.

Where it fails most

HIGH RISK	LOW RISK
precise figures	explaining a concept
names and dates	rewriting text
links and citations	proposing a structure
library function names	code you can actually run
legal and tax detail	summarising text you supplied

Right column: the model works from what you gave it. Left: it is guessing.

> The best defence is not a better prompt but a checkable output. Code can be run and tested (Module 9). A summary can be compared with the original. A number out of thin air can be compared with nothing. Hence the rule from 2.6: have the AI write the program that calculates, rather than calculate itself. Every link in this course, for instance, was verified by a script rather than recalled from memory.

MORE ON THIS

- Anthropic — models overview — https://docs.claude.com/en/docs/about-claude/models/overview

What must not go into AI `privacy`

Everything you type into a conversation leaves your machine and travels to the service.

In a business setting, which plan you use and what your company's contract says both matter.

Some things do not belong in a conversation under any circumstances — not out of distrust, but because they are not needed.

`Passwords, API keys, tokens`	never — not even 'just for a test'
`Customers' personal data`	only where the company explicitly allows it
`Health and financial records`	as a rule, no
`Contracts and internal documents`	per company policy
`Code with no secrets in it`	usually fine
`Invented test data`	always fine — and usually sufficient

> One technique solves most cases: anonymise before sending. An agent does not need real customer names to write an invoice-processing program — it needs the right *shape* of data. Replace the names, cut the file to ten rows, and send that. The result is the same and the risk is zero.
>
> **MORE ON THIS**
>
> - Anthropic — help centre — https://support.claude.com/

What an agent is `agent`

An agent is a language model with tools, running in a loop: think, use a tool, look at the result, repeat.

The difference between a chat and an agent is not the model but whether it is allowed to act.

Tools might be reading a file, searching the web, running a program, or calling somebody else's service.

The loop ends when the agent judges the task done — or when you stop it.

The agent loop

```
You:    "Rename all the invoices by date."



Agent:  think  -> "I need to see what is in the folder"

        tool   -> dir inbox\

        result <- 214 files

        think  -> "read the first PDF and find the date"

        tool   -> python read.py

        result <- error: missing library

        think  -> "install it"

        tool   -> pip install pypdf

        ...

        done   -> "Renamed 209 files, 5 are in the manual folder."
```

Every step of the loop is another model call - and every one costs tokens.

> The cost is visible in that transcript: an agent taking thirty steps calls the model thirty times, each time with everything it has seen so far. Small files, clear instructions and a bounded task therefore show up directly on the bill — and that is why the `architecture.md` from 9.8 pays for itself.
>
> **MORE ON THIS**
>
> - Claude Code — overview — https://docs.claude.com/en/docs/claude-code/overview

Claude: chat `Claude chat`

Chat is a conversation window with no access to your files beyond what you attach.

It is for thinking, explaining, writing and planning — not for doing.

Which makes it the best place for the first step of any project: getting the idea straight.

`Best for`	thinking, explaining, writing, planning
`What it sees`	the conversation and whatever you attach
`What it can change`	nothing on your machine
`Typical use`	*explain this, think it through with me, draft this*

> The mistake almost everyone makes at first is jumping straight into a tool that writes code. Half an hour in an ordinary chat clarifying what you actually want saves several times that later. The recipe for doing it is in 11.5.
>
> **MORE ON THIS**
>
> - Claude — help and guides — https://support.claude.com/en/collections/4078531-claude-apps

Claude: Cowork `Claude Cowork`

Cowork is an agentic surface for work that is not programming: documents, spreadsheets, decks, research.

There Claude does not merely advise; it carries out a sequence of steps and hands back a deliverable.

It is for people who have no interest in code but every interest in the result.

`Best for`	a longer task with something to show at the end
`What it sees`	the files you give it and connected services
`What it can change`	it creates and edits files in its own workspace
`Typical use`	*go through these 40 reports and summarise them*

> The line between Cowork and Code is not difficulty but the kind of deliverable: if the result is a document, a table or an analysis, Cowork fits; if the result is code that will be run and maintained, Code does. These products move quickly, so check the current capabilities on the official pages.
>
> **MORE ON THIS**
>
> - Claude — help and guides — https://support.claude.com/en/collections/4078531-claude-apps

Claude: Code `Claude Code`

Claude Code is an agent that runs on your own machine with access to a project folder.

It reads and writes files, runs commands, tries the code and fixes what does not work.

This course was built in it — the launchers, the page and this very text.

`Best for`	building and maintaining code and tools
`What it sees`	the project folder you point it at
`What it can change`	files in that folder, and commands you approve
`Safety catch`	anything dangerous needs a human to confirm
`Typical use`	*build the tool described in architecture.md*

> Because it touches files, Git from 9.7 is indispensable here. One rule not worth breaking: commit before every session. And a second: point the agent at the project folder, not the whole disk — that bounds the damage in advance.
>
> **MORE ON THIS**
>
> - Claude Code — overview — https://docs.claude.com/en/docs/claude-code/overview
> - Claude Code — product page — https://claude.com/product/claude-code

Artifacts `artifacts`

An artifact is a self-contained thing produced during a conversation and shown beside it: a document, a page, an app, a diagram.

It is not merely text in the chat — it is something you can open, use and share.

It earns its place when the result is not an answer but something somebody will actually use.

> For a non-technical user this is the shortest path from idea to usable thing: describe what you need and get a working tool rather than instructions for building one. The same technology as Module 6 — HTML, CSS and JavaScript — with the intermediate step removed.
>
> **MORE ON THIS**
>
> - Claude — help and guides — https://support.claude.com/en/collections/4078531-claude-apps

Connectors `connectors`

A connector links Claude to a service you already use — a calendar, mail, a drive, a task system.

Once connected, Claude can read that data and, where permitted, change it.

Every such link widens what the agent can do — and equally what it can break.

> The same rule applies as for public addresses in 7.8: enable only what you need, and check what each one permits. The gap between *may read the calendar* and *may send invitations in your name* is enormous, even though both look alike when you click Connect. Technically, connectors rest on the MCP protocol from 10.17.
>
> **MORE ON THIS**
>
> - Anthropic — remote MCP servers — https://docs.claude.com/en/docs/agents-and-tools/remote-mcp-servers

Skills `skills`

A skill is a written procedure the agent reads when it needs it — a manual on a shelf, not knowledge by heart.

It is an ordinary folder with a `SKILL.md` file saying when to use it and how the task is done.

Because it loads only at the right moment, it costs no context until it is needed.

The mechanism matters less than the consequence: write a good procedure once and you have it every time.

SKILL.md

```
---
name: monthly-report
description: Use when the user asks for a monthly report on
             incoming invoices.
---

# Monthly report

1. Read invoices.csv
2. Filter to the requested month
3. Total by supplier, sort descending
4. Print the table and the grand total
5. Flag invoices with no supplier
```

Five lines of instruction replace five minutes of explaining - every time.

> The description in the header is the crucial part: it is how the agent decides *when* the skill applies at all. A vague description means a skill that exists but never fires. Skills are also the easiest way to share your working procedures with colleagues — you simply send them the folder.
>
> **MORE ON THIS**
>
> - Anthropic — Agent Skills — https://docs.claude.com/en/docs/agents-and-tools/agent-skills
> - Claude Code — skills — https://docs.claude.com/en/docs/claude-code/skills

MCP — how an agent sees the world MCP

MCP is an agreed way of telling an agent which tools exist and how to call them.

Each tool introduces itself with a name, a description and an input shape — the agent then calls it like the API in Module 6.

Because the convention is shared, the same MCP server works with different agents and different models.

Put simply: MCP is the socket you plug your own systems into.

tool.json

```json
{
  "name": "find_invoice",
  "description": "Finds an invoice by number and returns its amount and date.",
  "inputSchema": {
    "type": "object",
    "properties": {
      "number": { "type": "string" }
    },
    "required": ["number"]
  }
}
```

The agent reads the DESCRIPTION and infers from it when to use the tool.

> Now the chain from earlier modules closes: the tool description is JSON (3.5), the call is a request and a response (5.7, 6.6), and the result comes back as JSON and lands in the context (10.4). Which is why description quality matters so much — exactly like a good function name in 9.3. A badly described tool is either ignored or misused.

MORE ON THIS

- Model Context Protocol — the specification — https://modelcontextprotocol.io/
- Anthropic — MCP — https://docs.claude.com/en/docs/mcp

Sub-agents `sub-agents`

A sub-agent is a separate agent with its own context that the main agent calls for a self-contained job.

The gain is twofold: the main context stays clean, and the sub-agent sees only what it actually needs.

That narrowness is the point — a sub-agent that cannot see the tests cannot write code tailored to them.

Dividing the work

```
MAIN AGENT  (knows the whole project)
  |
  +-- sub-agent A: "write tests for the function with_vat"
  |      sees: the requirement + architecture.md
  |      does NOT see: the existing code
  |
  +-- sub-agent B: "implement the function with_vat"
  |      sees: the requirement + architecture.md
  |      does NOT see: agent A's tests
  |
  +-- runs the tests -> 3 passed, 1 failed -> B fixes
```

Separate contexts are not a limitation here; they are the purpose.

> The cost is extra tokens: every sub-agent is its own conversation. So the technique pays where accuracy matters more than price — writing tests, reviewing code, and research where the main context would otherwise fill with material nobody needs afterwards.

MORE ON THIS

- Claude Code — sub-agents — https://docs.claude.com/en/docs/claude-code/sub-agents

Building agents well `building agents well`

An agent is only as good as the precision of the task and the narrowness of its tools.

The usual cause of a poor result is not the model but a loose instruction and too large a scope.

`Write` `` `architecture.md` ``	without it the agent guesses your intent
`Bound the project folder`	less access, less damage
`One task, one conversation`	the context stays clean
`Ask for a plan before the work`	a mistake in a plan is cheaper to fix
`Tests before implementation`	and a separate sub-agent for them
`Commit before every session`	the safety net
`Check what it did`	a human reads the diff, not the agent
`Write down what failed`	or it will try the same thing again

> All of these share one denominator: they reduce uncertainty. The model is non-deterministic (2.6), so you cannot make it reliable — but you can shrink the space in which it can go wrong. A clear goal, a small scope, a checkable output and a way back are the four levers you actually hold.
>
> **MORE ON THIS**
>
> - Claude Code — overview — https://docs.claude.com/en/docs/claude-code/overview

MODULE 11

From idea to code

The whole recipe: from one sentence in a chat to verified code, step by step.

The grilling skill `grilling`

Grilling reverses the roles: instead of answering you, the agent starts questioning you.

It asks in rounds — each round opens only the questions that can be answered right now.

Every question comes with its own recommendation, so answering is quick even when you have no opinion.

The result is that an idea becomes a set of decisions before a single line of code exists.

What it looks like

```
You:   "I want a tool for processing invoices."


Agent: Q1 - Input: are the invoices PDFs, scans, or both?
          -> Recommendation: both, detected automatically.

       Q2 - What should happen when extraction fails?
          -> Recommendation: move to a "manual" folder and log it.

       Q3 - Where does the data end up: CSV or a database?
          -> Recommendation: CSV, at 200 invoices a month.
```

This course was built exactly that way - 28 questions across four rounds.

> The value is not in the questions themselves but in being made to answer things the agent would otherwise quietly guess. Every unanswered question is a place where something you did not want will appear. Use it whenever the task is bigger than a single file.

MORE ON THIS

- grilling — the skill's source — https://github.com/mattpocock/skills/tree/main/skills/productivity/grilling

The handoff skill `handoff`

Handoff prepares a hand-over: a summary of the state that a new conversation or another person can pick up.

It solves exactly the problem from 10.6 — what to do when the context window fills.

A good handoff also records what has already been tried and did not work.

handover.md

```
# Project state - 21 September 2026

## Goal
Tool for processing incoming invoices (see architecture.md).

## Done
- reading PDFs, extracting date and amount (works on 209/214)
- writing invoices.csv

## Open
- 5 scanned invoices need OCR

## DO NOT try again
- pdfminer: slower and worse results than pypdf
- regex for the amount without line context: too many false positives
```

The last section is what separates a hand-over from a wish list.

> The same file serves when you hand work to a colleague or return to the project a month later. It is really the same document as `architecture.md`, except that it describes the current state rather than the intent — many people keep both in one file under two headings.

MORE ON THIS

- handoff — the skill's source — https://github.com/mattpocock/skills/tree/main/skills/productivity/handoff

The improve skill `improve`

Improve surveys existing code like a senior advisor and produces a plan of improvements.

It never edits the code itself — its deliverable is a plan precise enough for another agent to execute.

That separation is the point: a strong model does the thinking, a cheaper one can do the typing.

What it does

```
1. Reads the project: language, structure, tests, conventions
2. Looks for opportunities: bugs, security, performance, missing tests
3. VERIFIES each finding in the code (it does not trust its first impression)
4. Presents a list ordered by impact over effort
5. You choose which findings become plans
6. Writes a self-contained plan per choice into plans/
```

Each plan stands alone: an executor understands it without this conversation.

> Step three is the one people overlook and the one that pays most: the skill checks its own findings before presenting them, because first impressions are often wrong. The same habit is worth copying by hand — before believing an agent that something is broken, ask it to show you the line.

MORE ON THIS

- improve — the skill's source — https://github.com/shadcn/improve

Where to get skills `skills.sh`

Skills are ordinary folders containing a `SKILL.md` — you can write your own or take someone else's.

A curated collection lives at *skills.sh*.

Read a skill before you use it: these are instructions your agent will carry out on your machine.

`skills.sh`	a collection to browse and install from
`` `SKILL.md` ``	the heart of every skill — read it first
`The `description` field`	decides when the skill fires at all
`Your own skill`	fastest route: write down a procedure you repeat
`Sharing`	send a colleague the folder; it works immediately

> Same rule as for libraries in 8.10: someone else's instructions are someone else's code. A skill you have not read can tell an agent to do anything — including deleting or sending data. With well-known sources the risk is small, but reading before using is a habit worth keeping.
>
> **MORE ON THIS**
>
> - skills.sh — https://skills.sh/
> - Anthropic — Agent Skills — https://docs.claude.com/en/docs/agents-and-tools/agent-skills

Recipe 1: from idea to architecture `recipe 1`

The first half happens in an ordinary chat — no code, no tools, no hurry.

The goal is one file, `architecture.md`, that the agent will later build from.

Only once that file exists is it worth opening a tool that writes code.

`1. Describe the idea`	in an ordinary chat, in your own words
`2. State the inputs`	what you have: files, tables, access
`3. State the outputs`	what you expect: a file, a report, a tool
`4. Ask to be grilled`	*grill me* — answer round by round
`5. State the constraints`	Windows, no installs, data stays local
`6. Ask for` `` `architecture.md` ``	and read it before going further
`7. Save it in the project folder`	that is where the agent will find it

> Step six is not a formality. Read the file as the customer, not as a reader: is there anything in it you never said? Is anything missing that you assumed was obvious? Every sentence you fix here is a fix you will not have to make in code later — and that is the difference between ten minutes and two days.
>
> **MORE ON THIS**
>
> - Claude Code — project memory file — https://docs.claude.com/en/docs/claude-code/memory

Recipe 2: from architecture to code `recipe 2`

The second half happens in a tool with folder access — Claude Code, for example.

The order never changes: plan, approval, tests, implementation, verification.

Never skip the approval; a mistake in a plan is a hundred times cheaper than one in code.

`1. \`git init\` and a first commit`	the safety net, before anything starts
`2. Give it the project folder`	with `architecture.md` inside
`3. If anything is unclear: grilling`	questions now beat guesses later
`4. If it is clear: improve`	have it produce an implementation plan
`5. Read the plan`	then approve or correct it
`6. Tests by a sub-agent`	a separate agent that will not write the code
`7. Implementation`	another agent that never sees the tests
`8. Run the tests`	and let the agent fix until it is green
`9. Review the diff`	the one step a human does
`10. Commit`	and only then the next task

> Step nine cannot be delegated to a machine. You do not need to understand every line — it is enough to check that the agent changed what you expected and did not quietly change anything else. Which is why small steps and frequent commits matter so much: a diff touching three files can be read; one touching thirty cannot.

MORE ON THIS

- Claude Code — overview — https://docs.claude.com/en/docs/claude-code/overview
- Git — documentation — https://git-scm.com/doc

The implementer never sees the tests `blind implementer`

Have one agent write the tests and a different one write the code, without sight of them.
If one agent writes both, it will quietly shape the tests around the code it just produced.
Then both match the same wrong assumption and the tests prove nothing.
The separation is cheap, fits in one sentence of instruction, and visibly reduces defects.

The instruction that achieves it

```
You (to the main agent):



  "For the function with_vat, first have a sub-agent write the

   tests from architecture.md. Then have a SEPARATE sub-agent

   that does NOT see the tests implement the function. Then run

   the tests and let the second sub-agent fix until all pass.

   Do not modify the tests while fixing."
```

The last sentence is the crucial one: otherwise it will fix the test, not the code.

> That last line is not excessive caution. When a test fails, editing the test is the shortest path to green — and an agent will take it unless forbidden. A test bent to fit the code is no longer a requirement; it is a description of whatever the code happens to do. If a test does turn out to be genuinely wrong, you change it — deliberately, and for a stated reason.

MORE ON THIS

- Claude Code — sub-agents — https://docs.claude.com/en/docs/claude-code/sub-agents

MODULE 12

The big picture

How all of it connects, and where to go next.

How it all connects `the big picture`

The whole course rests on four ideas that recur in every module.

One: what is text is useful to a computer and to an AI; what is not must be converted first.

Two: one truth in one place — a variable, a key in a database, `architecture.md`.

Three: code is predictable and AI is not, so the output must be checkable.

Four: a large problem is only ever solved by becoming a sequence of small ones.

One idea across modules

```
ONE TRUTH IN ONE PLACE

  1.2  a variable           one box, not seventeen

  4.4  a key in a database  the customer's name stored once

  6.3  a colour in CSS      --accent in one place

  9.8  architecture.md      decisions in one place

 10.6  a session summary    state in one place


TEXT IS USABLE, BINARY IS NOT

  3.2  text vs. binary      the fundamental split

  3.10 AI-friendly formats  the consequence

  3.11 OCR                  the fix when it is already too late
```

Once you notice the pattern, you stop memorising the details.

> If in six months you have forgotten every detail — what `git restore` does, which port PostgreSQL uses, how to spell `SELECT` — nothing is lost. All of that can be looked up or asked. The four ideas above are what let you ask the right question, and that is the difference between using tools and building with them.

MORE ON THIS

- Anthropic — Agent Skills — https://docs.claude.com/en/docs/agents-and-tools/agent-skills

Where to go next `where to go next`

Do not start with a big project; start with the chore that annoys you every week.

Make it small enough to finish in an afternoon and yours enough that you know when it is right.

Let the first win be something you will actually use — then everything in this course settles by itself.

`First project`	a chore you repeat every week
`First tool`	an ordinary chat — get the idea straight
`First file`	`architecture.md`, written through grilling
`First habit`	commit before every agent session
`Second habit`	one conversation, one task
`When you get stuck`	read the last line of the error and paste it to the agent
`When the window fills`	summarise and start fresh

> The commonest mistake after a workshop like this is not technical but the choice of first project: people reach for something large while the enthusiasm is fresh, and get stuck. A small chore you finish teaches more than a big project you abandon — and it gives you the only thing that really convinces colleagues: a working example.
>
> **MORE ON THIS**
>
> - skills.sh — a collection of skills — https://skills.sh/
> - Claude Code — overview — https://docs.claude.com/en/docs/claude-code/overview

GLOSSARY

Glossary

Every term in alphabetical order, English ↔ Slovene.

Anatomy of an address `URL`

A web address is not one word but six separate parts, each with its own job.

Artifacts `artifacts`

An artifact is a self-contained thing produced during a conversation and shown beside it: a document, a page, an app, a diagram.

Bash and PowerShell `Bash / PowerShell`

These are command-line languages: made for driving other programs, not for writing applications.

Breaking a problem down `decomposition`

A large problem cannot be solved; only a sequence of small ones can.

Building agents well `building agents well`

An agent is only as good as the precision of the task and the narrowness of its tools.

C and C++ `C / C++`

Closest to the machine; used where every thousandth of a second counts.

Claude: chat `Claude chat`

Chat is a conversation window with no access to your files beyond what you attach.

Claude: Code `Claude Code`

Claude Code is an agent that runs on your own machine with access to a project folder.

Claude: Cowork `Claude Cowork`

Cowork is an agentic surface for work that is not programming: documents, spreadsheets, decks, research.

Code is predictable, AI is not `deterministic / non-deterministic`

The same code with the same input gives the same result every single time — that is determinism.

Comment `comment`

A comment is a line the computer ignores entirely and a human reads.

Compiled and interpreted `compiled vs. interpreted`

A compiled language is translated into machine code before it runs — the result is a standalone .exe.

Condition `if / else`

A condition is a railway switch: if something holds, the program takes one track, otherwise the other.

Connectors `connectors`

A connector links Claude to a service you already use — a calendar, mail, a drive, a task system.

CSS — the appearance `CSS`

CSS says how things should look: colours, sizes, spacing, layout.

CSV — a table as text `.csv`

CSV is a table written as plain text: one line per row, commas between columns.

Data types `data types`

A computer draws a hard line between a number int, text str, a yes/no value bool and a date.

Debugging `debugging`

Debugging is checking assumptions: what do I think is in this variable, and what is actually in it?

Dependencies and versions `dependencies, versions`

A program using five libraries depends on five other people's projects and their changes.

Document databases `document database / NoSQL`

A document database stores whole objects rather than rows — exactly the shape you saw in JSON.

Domains and DNS `domain, DNS`

Computers do not call each other by name; they call each other by number.

Error or crash `error / exception`

When a program meets something it cannot handle it raises an exception — and stops.

File, extension, path `file, extension, path`

A file is a lump of data with a name; the extension after the dot is only a promise about what is inside.

For loop `for loop`

A loop repeats the same steps for every item in a list.

Frontend and backend `frontend / backend`

The frontend is everything running in the user's browser — HTML, CSS, JavaScript.

Function `function`

A function is a named piece of code: data goes in as arguments, a return value comes out.

Garbage in, garbage out `garbage in, garbage out`

A computer never checks whether your data makes sense — only whether it can process it.

Git and commits `Git, commit`

Git remembers every state of your code that you confirm — and lets you return to any of them.

Hallucinations `hallucination`

A hallucination is an answer that sounds right but is not true — an invented figure, a function that does not exist, a dead link.

How it all connects `the big picture`

The whole course rests on four ideas that recur in every module.

How to read an error message `traceback / stack trace`

An error message is not a punishment; it is the most precise description of the problem you will get.

HTML `.html`

HTML is text with tags that say what is a heading, what is a paragraph and what is a link.

HTML — the structure `HTML`

HTML says what is on the page: a heading, a paragraph, a list, a button, an image.

IP address `IP address`

An IP address is a computer's house number on a network, written as four numbers from 0 to 255.

Java and C# `Java / C#`

The languages of large business systems: banking, insurance, ERP, public administration.

JavaScript — the behaviour `JavaScript / TypeScript`

JavaScript is the only programming language a browser understands directly.

JavaScript and TypeScript `JavaScript / TypeScript`

JavaScript is the only language that runs directly in a browser, which makes it unavoidable on the web.

JSON — structure as text `.json`

JSON writes the objects and lists from Module 1 as plain text.

Kinds of tests `unit / integration / end-to-end`

Unit tests check a single function; they are fast and you can have many.

Libraries `libraries (pip, npm)`

A library is code somebody else has already written that you use instead of writing your own.

Library or your own code `build or borrow`

Use a library when the problem is general and well solved: dates, PDFs, encryption, networking.

List `list / array`

A list is a numbered sequence of values held under one single name.

localhost `localhost / 127.0.0.1`

localhost means this computer — an address that never leaves your machine.

Main and modules `main + modules`

Code gets split into modules: each file does one kind of work.

MCP — how an agent sees the world `MCP`

MCP is an agreed way of telling an agent which tools exist and how to call them.

Naming `naming`

A name should say what a thing is or does — not how it was built.

Object `object / dict / key-value`

An object is a filled-in form: every field has a name, its key, and a value.

OCR — from picture to text `OCR`

OCR optical character recognition turns the letters in a picture back into plain text.

PDF `.pdf`

PDF exists so that a page looks identical everywhere — not so that data can be taken out of it.

Picture or drawing `raster vs. vector (.png / .svg)`

.jpeg and .png are grids of dots — enlarge them and they go soft.

Plain text and Markdown `.txt / .md`

.txt is bare content with no formatting — the most innocent format there is.

Ports `port`

One computer runs many programs at once; the port says which of them should take the connection.

Prompt and context `prompt / context`

The prompt is what you write; the context is everything the model sees alongside it.

Python `Python`

Readable, extremely widespread, and excellent for data work, automation and AI.

Recipe 1: from idea to architecture `recipe 1`

The first half happens in an ordinary chat — no code, no tools, no hurry.

Recipe 2: from architecture to code `recipe 2`

The second half happens in a tool with folder access — Claude Code, for example.

Relational databases and keys `relational database, primary/foreign key`

A relational database keeps data in several tables that reference each other.

Request and response `HTTP request / response`

The web runs on a simple pattern: the browser sends a request, the server returns a response, the connection closes.

Rung five: static hosting `static hosting`

If a page is static there is nothing to run — somebody just has to serve the files.

Rung four: a rented server `VPS`

A VPS is a computer in somebody else's data centre, rented for a few euros a month.

Rung one: my own machine `localhost`

The lowest rung of hosting is your own computer: the program runs when you start it and dies when you close the window.

Rung six: the cloud `cloud (AWS, Azure)`

The cloud is not one computer but hundreds of services you assemble as needed and pay for by use.

Rung three: a tunnel `tunnel (trycloudflare, ngrok)`

A tunnel is a program that opens a connection outward from your machine and hands you a public address.

Rung two: the office `LAN hosting`

If the server listens on every network card, everyone on the same wifi can see it.

Runtimes and Node.js `runtime / Node.js`

A language does not run by itself — it needs a program that executes it, called a runtime.

Same data, four shapes `same data, four shapes`

One single row of data gets written four ways, and you will meet all four everywhere.

Skills `skills`

A skill is a written procedure the agent reads when it needs it — a manual on a shelf, not knowledge by heart.

Spending fewer tokens `saving tokens`

Cost depends not on how much you write but on how much the model re-reads each time.

SQL `SQL`

SQL is the language you talk to a database in, and it reads remarkably like an English sentence.

SQL `SQL`

SQL is not for writing programs but for questioning databases — which is why it turns up everywhere.

Static and dynamic `static vs. dynamic`

A static page is a file that looks the same to everyone — the server just passes it along.

Status codes `status codes`

Every response opens with a three-digit number saying how the request went.

Sub-agents `sub-agents`

A sub-agent is a separate agent with its own context that the main agent calls for a self-contained job.

Summarise and start fresh `summarise and continue`

When the window is around eighty percent full, ask the agent for a summary of where things stand.

Terminal `terminal / command line / cmd`

A terminal is a window where you type commands instead of clicking buttons.

Text or binary `text vs. binary`

Every file is ultimately a sequence of numbers; the only question is whether those numbers stand for letters.

The API call `API call`

An API is an agreed list of questions one program may ask another.

The architecture.md file `architecture.md`

This is the document describing what you are building, written before the agent writes a line of code.

The browser is the engine `browser / rendering engine`

A browser is not a window onto the internet; it is a program that fetches files and runs them on your machine.

The context window `context window`

The context window is the ceiling on how much text the model can see at once.

The grilling skill `grilling`

Grilling reverses the roles: instead of answering you, the agent starts questioning you.

The handoff skill `handoff`

Handoff prepares a hand-over: a summary of the state that a new conversation or another person can pick up.

The implementer never sees the tests `blind implementer`

Have one agent write the tests and a different one write the code, without sight of them.

The improve skill `improve`

Improve surveys existing code like a senior advisor and produces a plan of improvements.

The journey of one click `the journey of one click`

One click sets off a chain of six steps that completes in a few hundredths of a second.

The local network `LAN, 192.168.x.x`

When a server listens on every network card, everyone on the same wifi can reach it.

The spreadsheet as a starting point `spreadsheet`

A spreadsheet is already almost a database: named columns, rows of values.

Tokens `token`

A model reads neither letters nor words but tokens — chunks of text averaging three to four characters.

Variable `variable`

A variable is a named place holding one value.

Web technology without the web `local web app, offline`

HTML, CSS and JavaScript do not need the internet — they need a browser.

What a bug is `bug`

A bug is the gap between what you thought you asked for and what you actually asked for.

What a database is `database`

A database is a program that stores data and answers questions about it quickly.

What a language model is `LLM`

A language model predicts which piece of text most likely follows what has been written so far.

What a program is `program / code`

A program is a sequence of instructions the computer carries out one after another, top to bottom.

What a test is `test`

A test is a small program that runs your code and checks the result is what you expected.

What an agent is `agent`

An agent is a language model with tools, running in a loop: think, use a tool, look at the result, repeat.

What may go on a public address `what to expose`

Once something has a public address, assume somebody will find it — even if you told nobody.

What must not go into AI `privacy`

Everything you type into a conversation leaves your machine and travels to the service.

What tokens cost `pricing`

You pay for two things: the tokens you send (input) and the tokens the model writes (output).

Where to get skills `skills.sh`

Skills are ordinary folders containing a SKILL.md — you can write your own or take someone else's.

Where to go next `where to go next`

Do not start with a big project; start with the chore that annoys you every week.

Which formats are AI-friendly `AI-friendly formats`

The rule is simple: the closer a format is to plain text, the less trouble you will have.

Which one, when `choosing a store`

The choice almost always comes down to one question: is the data the same shape everywhere?

While loop `while loop`

while repeats as long as some condition holds — you do not know in advance how many rounds that will be.

Why tests matter more with AI `tests with AI`

An agent writes code faster than you can read it — tests are the only way checking keeps pace.

Why there are so many languages `programming languages`

Every language can do the same things — they differ in what they make easy and what they make hard.

Word and Excel files `.docx / .xlsx`

.docx and .xlsx are really compressed folders (.zip) full of XML files.
