What is programming (the "chai" example → algorithm → code)

Programming is fundamentally the art and science of instructing a computer to perform a series of tasks by writing code in a language that the machine can understand and execute. At its core, it's about converting a logical sequence of steps—often derived from real-world problem-solving—into a structured program that automates those steps. This process involves breaking down complex problems into smaller, manageable actions, handling conditions (like decisions or errors), and ensuring the output is reliable and efficient.

To make this relatable, the course uses the example of making chai (Indian tea), which is a brilliant analogy because it's a everyday task that mirrors programming logic. Let's dive deep into this example and expand on it to illustrate key programming concepts.

Programming = turning a recipe/series of steps into precise instructions a computer can run.

Chai example (high level steps):

- 1. Boil water
- 2. Add tea leaves
- 3. Add milk if available else add water
- 4. Add sugar if desired
- 5. Steep, strain, serve

Algorithmic version (pseudo):

```
if water_available:
    heat(water)
else:
    error "no water"

add(tea_leaves)
if milk_available:
    add(milk)
add(sugar_if_wanted)
steep(minutes=3)
serve()
```

Translated to Python (simple, explicit):

```
def make_tea(water=True, milk=True, sugar=False):
    if not water:
        raise RuntimeError("No water")
    print("Boil water")
    print("Add tea leaves")
    if milk:
        print("Add milk")
    if sugar:
        print("Add sugar")
    print("Steep for 3 minutes")
    print("Strain and serve")
```

Key idea: programming forces you to be explicit (what to do in each condition). That is the core of writing correct programs.

Getting started with Python (Mac & Windows) — how to run code

Two common ways to run Python:

- Interactive shell (REPL) good for quick experiments
 - mac/linux: python3 then type commands
 - o windows: py or python then type commands
- Script file (.py) write code in a file and run it:

- mac/linux: python3 script.py
- windows: py script.py or python script.py

Quick install notes (common):

- mac: brew install python@3.x or download from python.org
- windows: download installer from python.org (check "Add Python to PATH"), then use py launcher.

VS Code: install Microsoft Python extension + Pylance. Use the integrated terminal to run python file.py, or use the Run/Debug UI. Warp (Mac) is a terminal app — works same as Terminal, but optional.

Helpful tips:

- On mac/linux you may need to call python3 if python points to Python 2 or is missing.
- Shebang + executable (mac/linux): #!/usr/bin/env python3 at top, then chmod +x script.py and ./script.py.

Virtual environments — what, why, how

What: isolated Python environment with its own site-packages so project dependencies don't collide.

Why: different projects may need different versions of libraries; venv keeps them separate and portable.

Basic (built-in) approach:

```
# create
python3 -m venv venv
# activate (mac / linux - bash / zsh)
source venv/bin/activate
# activate (Windows PowerShell)
.\venv\Scripts\Activate.ps1
# activate (Windows cmd)
.\venv\Scripts\activate.bat
# install packages
pip install requests
# freeze for sharing
pip freeze > requirements.txt
# later, recreate
pip install -r requirements.txt
# deactivate
```

deactivate

Modern alternatives & when to use:

- poetry dependency management + packaging (great for apps/libraries; manages virtualenvs for you).
- pipenv older "pip + venv manager".
- pyenv manage multiple Python versions (useful if you need Python 3.8/3.11 etc).
- pipx install & run Python CLI tools globally in isolation.

Rule of thumb: for small scripts python -m venv is fine; for serious projects consider poetry.

Organizing and structuring code (namespaces, scopes, projects)

Project layout (simple, recommended):

Modules & packages

- A file utils.py is a module; a folder with __init__.py is a package.
- Import with from mypackage.utils import foo or import mypackage.utils as u.

Namespaces

• Each module has its own namespace: variables in utils.py don't collide with variables in main.py unless imported.

Scopes — the LEGB rule

- Local (L): names defined inside a function
- Enclosing (E): names in enclosing function(s) (for nested functions)
- Global (G): top-level names in the module
- Built-in (B): Python builtins like len, str

Example:

```
x = "global"

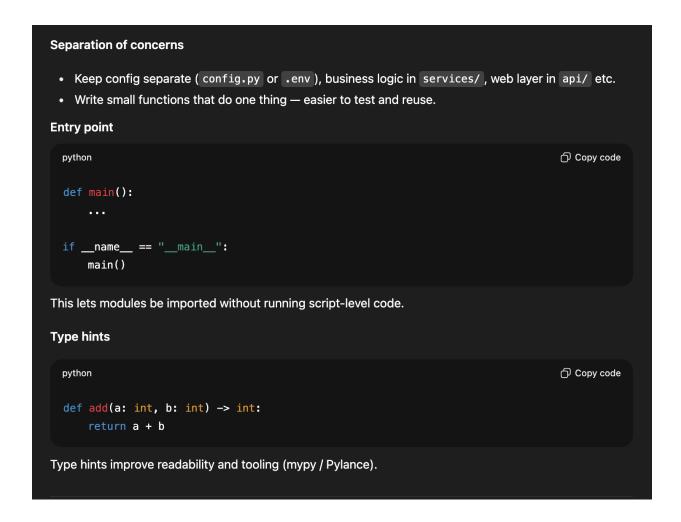
def outer():
    x = "enclosing"
    def inner():
        x = "local"  # Local
        print(x)
    inner()
    print(x)  # Enclosing

outer()
print(x)  # Global
```

Use global and nonlocal sparingly.

Separation of concerns

- Keep config separate (config.py or .env), business logic in services/, web layer in api/ etc.
- Write small functions that do one thing easier to test and reuse.



PEP8 & the Zen of Python — style and philosophy

PEP8 (practical rules)

- Indent with 4 spaces (no tabs).
- Max line length ~79 (PEP8) note that many projects use 88 (Black's default).
- Blank lines: top-level functions/classes separated by two blank lines.
- Imports: grouped and ordered standard lib, third-party, local.

- Naming: snake_case for functions/variables, PascalCase for classes, UPPER_SNAKE for constants.
- Use docstrings for modules/functions (PEP257).

Tools: black (autoformatter), isort (sort imports), ruff/flake8 (linters). Use pre-commit hooks for consistent style.

The Zen of Python (type import this in Python). Key lines and meaning:

- Beautiful is better than ugly. → prefer clear design.
- Simple is better than complex. → choose readability.
- Readability counts. → future-you will thank you.
- Explicit is better than implicit. → avoid magic behavior.
- There should be one and preferably only one obvious way to do it. → prefer conventions.

Apply these while designing APIs and modules.

Why use Python — practical strengths

- Readable & expressive short, clear code → faster development.
- Large ecosystem web (Django/Flask/FastAPI), data (pandas, numpy), ML (scikit-learn, PyTorch), automation, scripting, testing.
- Batteries included powerful standard library for files, networking, parsing, subprocesses.
- Cross-platform runs on mac/linux/windows easily.
- Great community & learning resources lots of tutorials, examples and packages.

Small example of automation (rename .txt files to .bak):

```
python

from pathlib import Path

for p in Path(".").glob("*.txt"):
    p.rename(p.with_suffix(".bak"))

One short script, real effect — that's the "chai level happiness".

Putting it together: first principles + investigative approach

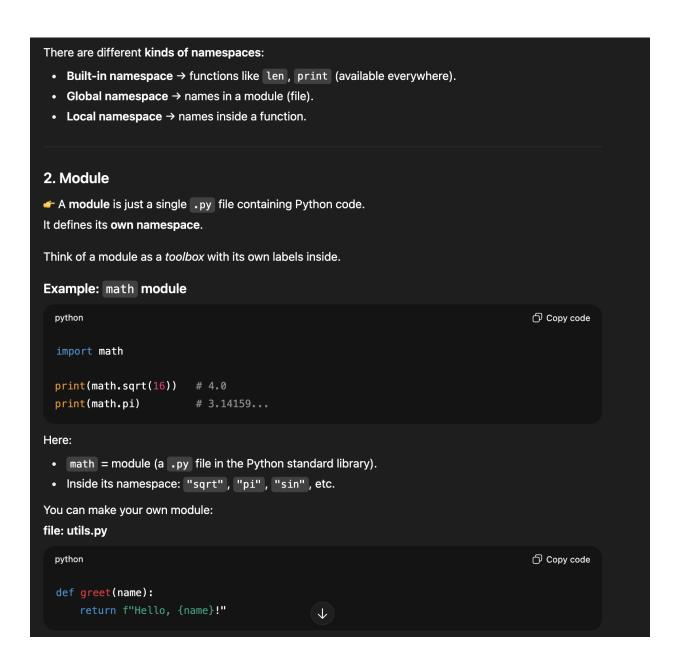
First principles: learn what python — venv actually does, what import does, how scoping works (LEGB), why strings are immutable — dig to fundamentals.

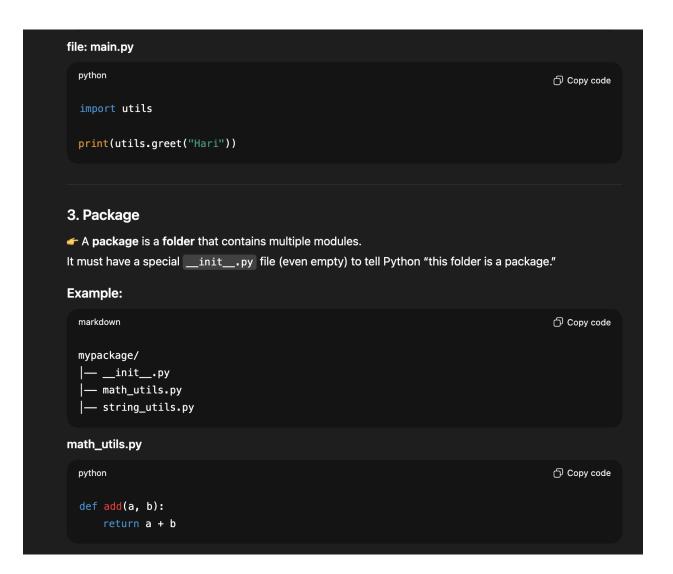
Investigative: run experiments — try text[::-1], change start/stop/step, inspect range() values, break code and debug.
```

Example workflow for learning a library:

- 1. Read the basic concept (first principles).
- 2. Write a tiny script that uses it (investigate).
- 3. Break it on purpose, inspect errors, fix and understand why.
- 4. Refactor for readability (PEP8) and add tests.

1. Namespace classes) to **objects** (values in memory). Think of it like: • A dictionary behind the scenes. • Keys = names ("x", "print", "math") • Values = actual Python objects (integer, function, module, etc.) **Example:** python Copy code x = 10y = 20def add(a, b): return a + b Namespace (like a dict): python Copy code "add": <function add at 0x...>





string_utils.py python Copy code def shout(text): return text.upper() main.py Copy code python from mypackage import math_utils, string_utils print(math_utils.add(2, 3)) # 5 print(string_utils.shout("hi")) # HI Now mypackage is like a big toolbox with smaller toolboxes inside. **☑** Big Picture • Namespace = a "map of names to objects" (like a dictionary of labels). • Module = one .py file → has its own namespace. • Package = a folder of modules (with __init__.py). Analogy: • Namespace = your kitchen labels (sugar, tea, milk).

• Module = one drawer (all tea stuff).

• Package = a whole cupboard (multiple drawers: tea, coffee, spices).