Python Basics

From Your First Line of Code to AI Coding

Islam Hamama

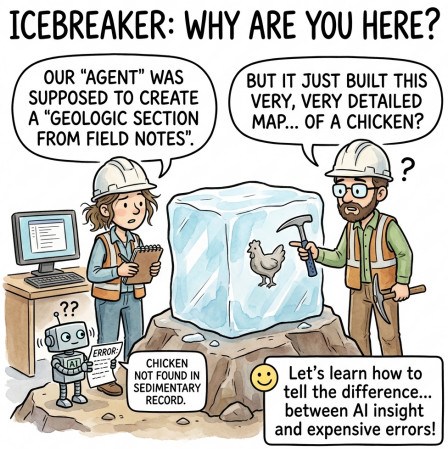
National Research Institute of Astronomy and Geophysics (NRIAG), Egypt
islam.hamama@nriag.sci.eg - i-hamama.com

August 6, 2026

Bring a laptop (or just a browser). We will be **live in Google Colab** all day you type, you run, you break things, you fix them.

Try it live in Colab

Before We Start



Today, *"AI writes code"* becomes *"I can read what AI wrote."*

We are **not** teaching "Python basics, then AI tools" as two separate halves. Today is **one story** with one goal:

The person you are becoming

Not the person who **types** code
the person who **reviews** code.

- AI writes fast, fluent Python. It is also confidently wrong sometimes.
- Your job is to **read**, **check**, **fix**, and **own** what it produces.
- You cannot review code you cannot read so we build your reading, one small step at a time.

The Question We Will Not Answer Yet

The question

"AI writes good Python. Why are you here?"

I will **not** answer this. I am collecting your answers:

- Write down your reason.
- Keep it. We come back to it at **minute 150**.

Today's promise

By minute 150 you will answer this question *yourself* by catching a real, live agent failure with your own eyes and your own basics.

Try it live in Colab

Lecture Goals Through the Reviewer's Eyes

By the end of today you will be able to:

1. Write, run, and debug Python programs with confidence
2. Use the core building blocks: variables, conditions, loops, data structures, functions
3. Read and write files, handle errors, and use scientific libraries
4. Work with **geoscience data**: earthquake catalogs, well logs, rock samples
5. Call an LLM from Python with a well-crafted prompt
6. Review an **OpenCode** agent's work, *safely and effectively*

The fair question (we asked before)

"Why learn the basics in the AI era?" because AI writes code and **you** review it. You cannot review what you cannot read. Keep your minute-3 answer; we test it at minute 150.

Roadmap

Why Python for Geoscientists?

- Reads almost like English $\rightarrow$ fastest path to *thinking like a programmer*
- The language of **data science and AI** (NumPy, pandas, PyTorch)
- Answers questions that pre-packaged software cannot
- Free, runs everywhere, huge community

The geoscience ecosystem

ObsPy seismology · **segyio** SEG-Y · **lasio** well logs · **GemPy** 3-D geomodeling · **SimPEG** geophysical inversion · **PyGMT** maps · **matplotlib** figures

The limitation of programming is only your imagination

A few lines of Python can process a whole earthquake catalog or a whole seismic survey.

Our Two Workbenches

Google Colab (zero setup)

- Browser-based notebooks, free
- Nothing to install; files saved to Drive
- Most science libraries preinstalled
- `https://colab.research.google.com`

Your own device

- Python 3 (Anaconda or `https://python.org`)
- VS Code (or any editor) + terminal
- Needed later for **OpenCode** (it runs locally on your projects)

Today: **Colab for learning** $\rightarrow$ **local machine for AI agents.**

How to Follow Along

1. Open `https://colab.research.google.com` → **New notebook**
2. Type code in a cell, press **Shift+Enter** to run it
3. Better: upload the companion notebook `Python_Basics_Workshop.ipynb` (File → Upload notebook) every slide example is inside

On your own device (later)

```
python3 --version          # should say 3.10+
python3 my_script.py       # run a file from the terminal
```

Type every example yourself. Watching code is like watching the gym.

Roadmap

Your First Program

```python
# This is a comment - Python ignores it
print("Hello, Geoscientist!")
print("Earthquake catalog loaded.")
```

Output:

```
Hello, Geoscientist!
Earthquake catalog loaded.
```

- `print()` displays values; it adds a newline at the end
- Comments start with # write them for your future self

Try it live in Colab

print() Has Superpowers: sep and end

```
print("EQ-001", "HLG", "M4.3")
print("EQ-001", "HLG", "M4.3", sep=" | ")
print("depth", end="=")
print(12.5, end=" km")
```

Output:

```
EQ-001 HLG M4.3
EQ-001 | HLG | M4.3
depth=12.5 km
```

- sep: what goes *between* items (default: a space)
- end: what goes *after* the last item (default: newline)

Variables: Names That Point at Values

```python
event_id = "EQ-2026-001"    # str   (text)
magnitude = 4.3             # float (decimal)
depth_km = 12               # int   (whole number)
felt_report = True          # bool  (True/False)

print(event_id, magnitude, depth_km, felt_report)
```

Output:

```
EQ-2026-001 4.3 12 True
```

Naming rules

Letters, digits, underscore; cannot start with a digit; case-sensitive.
Convention: `snake_case` for variables and functions.

Python is dynamically typed

A name can later point to a different type: `x = 10` then `x = "ten"` is legal.

Know Your Types with type()

```
values = [12, 4.3, "HLG", True, None]
for v in values:
    print(repr(v), "->", type(v).__name__)
```

Output:

```
12 -> int
4.3 -> float
'HLG' -> str
True -> bool
None -> NoneType
```

- None means "no value yet" e.g. a station that reported nothing
- When confused: `print(type(x))` is your best debugging friend
- Mutable (changeable): `list`, `dict`, `set` Immutable: `str`, `int`, `float`, `bool`, `tuple`

Numbers and Operators (Doing Real Geophysics)

```python
depth_m, v = 2000, 3000
twt = 2 * depth_m / v      # two-way time
print(twt)                 # always a float
print(4500 // 1000)        # 4   (km part)
print(4500 % 1000)         # 500 (m part)
E = 10 ** (4.8 + 1.5 * 4.3)
print(f"{E:.2e} J")        # energy, M=4.3

samples = 0
samples += 100             # samples =
    samples + 100
```

Output:

```
1.3333333333333333
4
500
1.78e+11 J
```

Watch out

/ **always** returns a float · // floor division · % remainder · ** power · augmented: += -= *= /=

Strings and f-Strings (the Modern Way)

```python
network = "EG"
station = "HLG"
print(network + "-" + station)    # concatenation

mag, depth = 4.3, 12.5
print(f"Station {station}: M{mag}")
print(f"Depth: {depth:.1f} km")      # 1 decimal
print(f"Energy: {10 ** (4.8 + 1.5 * mag):.2e} J")
```

Output:

```
EG-HLG
Station HLG: M4.3
Depth: 12.5 km
Energy: 1.78e+11 J
```

- f-strings: prefix with `f`, embed values in `{...}` use them everywhere

Talking to the User: `input()` and Casting

```
station = input("Station code? ")
mag_text = input("Magnitude? ")
mag = float(mag_text)      # str -> float
print(f"Station {station} recorded M{mag}")
```

Classic bug #1

`input()` **always returns a string**.
`"4.3" + 1` → TypeError
Fix: `int()`, `float()`, `str()`

- `float("4.3")` → 4.3; `int("12")` → 12
- `float("M4.3")` → `ValueError` (we will catch it later)

Your Turn #1

Exercise Acoustic Impedance (5 min)

Density ρ (g/cm^3) = 2.4 and velocity v (m/s) =2600. Compute `Z = rho * v` and print it.
please also list the types of the variables you used!

Try it live in Colab starter cells are in the companion notebook

Roadmap

if / elif / else: Classifying Earthquakes

```python
mag = 4.3

if mag < 2:
    print("micro")
elif mag < 4:
    print("minor")
elif mag < 6:
    print("light to moderate")
else:
    print("strong - damaging")
```

Output:

```
light to moderate
```

Indentation IS the syntax

4 spaces per level, always consistent. Other languages use { }; Python uses indentation. The colon : starts a block. (Improper indentation = the most popular error!)

Comparisons and Truthiness

```
mag = 4.3
print(mag >= 4.0)           # True
print(2.0 <= mag <= 6.0)    # chaining!

station = ""
if station:
    print("Station:", station)
else:
    print("Station code missing")
```

Output:

```
True
True
Station code missing
```

Falsy values (memorize this short list)

`False`, `0`, `0.0`, `""`, `[]`, `{}`, `set()`, `None`
Everything else is **truthy**. Combine with `and`, `or`, `not`.

- Compare identity with `is None`, equality with ==

Your Turn #2

Exercise Reservoir Quality Classifier (8 min)

Ask for a porosity value (in %) and classify the reservoir rock:
`>= 20 -> excellent`, `>= 10 -> good`, `>= 5 -> poor`, else `non-reservoir`.
Bonus: reject values outside 0–100 with an error message.

Hint

Order your conditions from highest to lowest the first true branch wins.

Try it live in Colab

Roadmap

for Loops and range()

```python
for d in range(0, 31, 10):   # depth slices
    print(f"{d} km", end="   ")
print()

for st in ["HLG", "KOT", "SUT"]:
    print("Processing station", st)

for ch in "shale":         # strings too
    print(ch, end="-")
```

Output:

```
0 km   10 km   20 km   30 km
Processing station HLG
Processing station KOT
Processing station SUT
s-h-a-l-e-
```

- `range(stop)`, `range(start, stop)`, `range(start, stop, step)`
- stop is excluded the most common off-by-one bug

while Loops, break and continue

```python
depth, td = 0, 3000
while depth < td:
    depth += 1000       # keep drilling!
    print(f"Drilling... {depth} m")
print("TD reached!")

for m in [2.1, None, 3.4, 6.5, 2.0]:
    if m is None:
        continue        # skip bad reading
    if m >= 6:
        print("Major event - alert!"); break
    print("ok:", m)
```

Output:

```
Drilling... 1000 m
Drilling... 2000 m
Drilling... 3000 m
TD reached!
ok: 2.1
ok: 3.4
Major event - alert!
```

Infinite loops

If the condition never becomes false, the loop never ends. In Colab: **stop button** (or Ctrl+C in a terminal).

Your Turn #3

Exercise Catalog Sweep

Loop over `[1.5, 2.3, 4.1, 5.9, 3.0]` and print each magnitude with its class (micro / minor / light to moderate / strong) using `if`/`elif` inside the loop.

Try it live in Colab

Roadmap

Lists: Ordered, Changeable Collections

```
mags = [2.1, 3.4, 4.0, 2.8]
print(mags[0])       # first:  2.1
print(mags[-1])      # last:   2.8
print(mags[1:3])     # slice:  [3.4, 4.0]
print(mags[::-1])    # reversed
print(len(mags))     # 4
```

Output:

```
2.1
2.8
[3.4, 4.0]
[2.8, 4.0, 3.4, 2.1]
4
```

- Indexing starts at **0**; negative indices count from the end
- Slicing `[start:stop:step]` stop excluded (same rule as `range`)

List Methods You Will Use Daily

```python
mags = [2.1, 4.0, 3.4]
mags.append(2.8)      # add at the end
mags.insert(0, 5.1)   # insert at index 0
mags.remove(5.1)      # remove first match
last = mags.pop()     # remove+return last
mags.sort()           # sort IN PLACE
print(mags)
print(sorted(mags, reverse=True))  # NEW list
print(4.0 in mags)    # membership: True
```

Output:

```
[2.1, 3.4, 4.0]
[4.0, 3.4, 2.1]
True
```

sort() vs sorted()

`list.sort()` changes the list and returns `None`; `sorted(x)` returns a new list. A classic trap: `mags = mags.sort()` → mags is now None!

Classic Bug #2: Assignment Does NOT Copy

```python
a = [10.0, 12.5, 8.0]    # depths
b = a            # same list, two names!
b.append(20.0)
print("a:", a)    # 4 values! surprise!

c = a.copy()      # a real copy
c.append(25.0)
print("a:", a)    # unchanged
print("c:", c)
```

Output:

```
a: [10.0, 12.5, 8.0, 20.0]
a: [10.0, 12.5, 8.0, 20.0]
c: [10.0, 12.5, 8.0, 20.0, 25.0]
```

- Variables are **labels on objects**. `b = a` adds a label, not a copy.
- Use `.copy()` or `list(a)` or `a[:]` for an independent copy.

Tuples and Sets

```
hlg = (27.3, 33.8)    # (lat, lon): immutable
lat, lon = hlg        # unpacking
print(lat, lon)

a = {"quartz", "feldspar", "mica"}  # sample A
b = {"quartz", "calcite"}           # sample B
print(a & b)          # in BOTH samples
print(a | b)          # in either sample
print(set("sandshale"))             # unique
```

Output:

```
27.3 33.8
{'quartz'}
{'mica', 'quartz', 'calcite', 'feldspar'}
{'a', 'd', 'e', 'h', 'l', 'n', 's'}
```

- **Tuple**: fixed record, e.g. coordinates safe from accidental change
- **Set**: unordered, unique items, fast membership perfect for mineral lists

Dictionaries: Named Data (You Will Use These Constantly)

```python
event = {"id": "EQ-001", "mag": 4.3, "depth": 12.5}
print(event["mag"])                # access
event["region"] = "Red Sea"        # add
event["mag"] = 4.5                 # update (revised!)
print(event.get("felt", "unknown"))
print("depth" in event)            # True
print(list(event.keys()))
```

Output:

```
4.3
unknown
True
['id', 'mag', 'depth', 'region']
```

- Key → value lookup; keys are unique (str, int, tuple...)
- `d[key]` raises `KeyError` if missing; `d.get(key, default)` is the safe way

Looping Dictionaries + Nested Data

```python
best = {"Red Sea": 4.1, "Cairo": 2.8, "Med": 5.2}

for region, mag in best.items():
    print(f"{region}: M{mag}")

top = max(best, key=best.get)   # strongest region
print("Strongest:", top)

catalog = [                     # like USGS GeoJSON!
    {"id": "EQ-01", "mag": 3.2},
    {"id": "EQ-02", "mag": 4.5},
]
print(catalog[1]["mag"])        # 4.5
```

Output:

```
Red Sea: M4.1
Cairo: M2.8
Med: M5.2
Strongest: Med
4.5
```

- `.items()` gives (key, value) pairs the pattern you will use most
- A list of dicts = a tiny "database table"; this is exactly how data centers answer

Choosing the Right Structure

Structure	Ordered?	Changeable?	Use it for...
`list`	yes	yes	measurements, catalogs, anything you loop over
`tuple`	yes	no	coordinates, fixed records, dict keys
`set`	no	yes	minerals, uniqueness, fast membership
`dict`	yes*	yes	event/station records, JSON-like data

* insertion-ordered since Python 3.7

Rule of thumb

Same kind of things → `list`. Different named fields → `dict`.
Many records with the same fields → **list of dicts** (your catalog!).

Comprehensions: Loops in One Line

```
mags = [2.1, 3.4, 4.0, 1.9, 5.2]
strong = [m for m in mags if m >= 4.0]

depths = [1000, 2000, 3000]
twt = [round(2 * d / 3000, 2) for d in
    depths]

counts = {"HLG": 12, "KOT": 8}
labels = {s: "busy" if c > 10 else "quiet"
          for s, c in counts.items()}
print(strong, twt, labels)
```

Output:

```
[4.0, 5.2] [0.67, 1.33, 2.0]
{'HLG': 'busy', 'KOT': 'quiet'}
```

- Pattern: `[expression for item in iterable if condition]`
- Powerful but if it does not fit on one line, write a normal loop. **Readability wins.**

Roadmap

String Methods You Will Use Daily

```
raw = "  hlg  "
print(raw.strip().upper())      # 'HLG'

print("2026-08-04".split("-"))    # -> list
print("-".join(["2026", "08", "04"]))

print("EQ-2026-001".startswith("EQ"))  # True
print("sandstone".replace("sand", "lime"))
print("shale".find("al"))       # index or -1
```

Output:

```
HLG
['2026', '08', '04']
2026-08-04
True
limestone
2
```

Strings are immutable

Methods **return a new string**; the original never changes: `raw.strip()` does not change `raw` assign it: `raw = raw.strip()`.

Slicing Deep Dive + a Tiny Challenge

```python
code = "EQ-2026-0042"
print(code[:2])      # 'EQ'
print(code[3:7])     # '2026' (the year)
print(code[-4:])     # '0042' (last 4 chars)
print(code[::-1])    # reversed!

def is_palindrome(word):
    return word == word[::-1]

print(is_palindrome("level"))    # True
print(is_palindrome("shale"))    # False
```

Output:

```
EQ
2026
0042
2400-6202-QE
True
False
```

- `"level"` a palindrome *and* a surveying instrument!

Roadmap

Defining and Calling Functions

```python
def two_way_time(depth_m, velocity=2000):
    """Two-way travel time in seconds."""
    return 2 * depth_m / velocity

t = two_way_time(1500)    # capture result
print(t)

def acoustic_impedance(rho, v):
    return rho * v

print(acoustic_impedance(2.65, 3000))
```

Output:

```
1.5
7950.0
```

`return` ≠ `print` the most important slide of the day

`print` *displays* text. `return` *hands back a value* you can store, reuse, and test. Functions that return values compose; functions that print do not.

No `return` → returns `None`. DRY: **D**on't **R**epeat **Y**ourself.

Arguments: Positional, Keyword, Defaults

```python
def gardner_density(v, a=0.31, b=0.25):
    """Gardner's relation: velocity -> density."""
    return a * v ** b

print(gardner_density(3000))       # defaults
print(gardner_density(3000, 0.23))
print(gardner_density(b=0.25, v=3000, a=0.31))
```

Output:

```
2.2987...
1.6977...
2.2987...
```

Roadmap

Exceptions: Expect the Unexpected

```python
def to_float(text):
    try:
        return float(text)
    except ValueError:
        print(f"'{text}' is not a number")
        return None

print(to_float("4.3"))      # 4.3
print(to_float("M4.3"))     # friendly message
```

Output:

```
4.3
'M4.3' is not a number
None
```

Common exceptions (read these names in tracebacks!)

`ValueError` bad value · `TypeError` wrong type · `KeyError` missing dict key · `IndexError` bad list index · `ZeroDivisionError` · `FileNotFoundError`

Rules

Catch **specific** exceptions, never bare `except:`. Python style: "ask forgiveness, not permission" (EAFP).

Files: Reading and Writing Safely

```python
# write a mini catalog (creates/overwrites)
with open("catalog.txt", "w") as f:
    f.write("EQ-001,4.3,12.5\n")
    f.write("EQ-002,3.1,8.0\n")

# read and parse it back
with open("catalog.txt") as f:
    for line in f:
        eid, mag, dep = line.strip().split(
            ",")
        print(eid, float(mag), float(dep))
```

Output:

```
EQ-001 4.3 12.5
EQ-002 3.1 8.0
```

- `with` **always closes the file**, even on errors never skip it
- Modes: `"r"` read, `"w"` overwrite, `"a"` append
- Real geoscience formats: **miniSEED** & **SEG-Y** (ObsPy / segyio), **LAS** well logs (lasio)

Roadmap

Modules and the Standard Library

```python
import math, random, datetime, json

print(math.sqrt(16), round(math.pi, 4))
print(random.randint(5, 30))    # depth, km
print(datetime.date.today())

event = {"id": "EQ-001", "mag": 4.3}
text = json.dumps(event)              # -> JSON string
print(text)
print(json.loads(text)["mag"])        # back to dict
```

Output:

```
4.0 3.1416
17
2026-08-04
{"id": "EQ-001", "mag": 4.3}
4.3
```

Batteries included explore these

os/pathlib files · re regex · datetime · json · collections.Counter (count lithologies in one line!) · statistics · random

Installing Packages: pip (+ the Geoscience Shelf)

In Colab:

```
%pip install obspy
```

On your machine:

```
python3 -m venv .venv          # isolated
    env
source .venv/bin/activate      # macOS/
    Linux
pip install obspy
pip freeze > requirements.txt
```

The geoscience shelf (install as needed)

obspy waveforms, catalogs, miniSEED, SEG-Y
segyio fast SEG-Y reading **lasio** LAS well logs
gempy 3-D geological modeling **simpeg** geophysical inversion
pygmt beautiful maps **scikit-learn** machine learning

Why virtual environments?

Each project gets its own package versions no conflicts, reproducible setups.

A 5-Minute Taste of NumPy and pandas

NumPy fast numeric arrays:

```
import numpy as np
t = np.array([0.5, 1.0, 1.5])  # TWT, s
v = 3000                       # m/s
print(v * t / 2)   # depths, vectorized!

mags = np.array([2.1, 3.4, 4.0, 2.8])
print(mags.mean(), mags.max())
```

```
[ 750. 1500. 2250.]
3.075 4.0
```

pandas data tables (your catalog!):

```
import pandas as pd
df = pd.DataFrame({
  "id": ["EQ-01","EQ-02","EQ-03"],
  "mag": [3.2, 4.5, 2.8],
  "region": ["Red Sea","Cairo","Red Sea"]})
print(df[df["mag"] >= 3.0])
print(df.groupby("region")["mag"].max())
```

```
     id  mag   region
0  EQ-01  3.2  Red Sea
1  EQ-02  4.5    Cairo
region
Cairo      4.5
Red Sea    3.2
```

Bonus: Your First Seismic Trace (30 Seconds of Code)

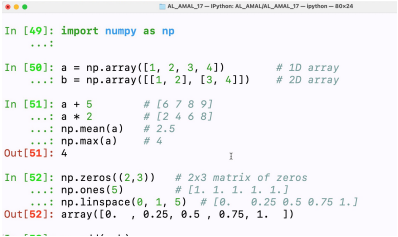
```python
import numpy as np
t = np.arange(0, 1, 0.001)            # 1 s, 1 ms
trace = np.sin(2 * np.pi * 25 * t) * np.exp(-8 * t)

print(trace.max(), trace.argmax())

# In Colab, uncomment for the magic:
# import matplotlib.pyplot as plt
# plt.plot(t, trace)
# plt.xlabel("Time (s)"); plt.ylabel("Amplitude")
```

What just happened?

A damped 25 Hz sine = a toy seismic wavelet. Real work: load real traces with **ObsPy** and plot with **matplotlib**.



Try it live in Colab it is waiting in the notebook

Roadmap

What Is an LLM?

- A neural network trained on a **huge** amount of text to do one thing: *predict the next token* (piece of a word)
- From that simple objective emerge: translation, summarization, coding, reasoning...
- Trained on code too $\rightarrow$ they read and write Python fluently

What this implies

- They are **fluent**, not always **right** they can *hallucinate* facts, formulas, and APIs
- They have a **knowledge cutoff** and a limited **context window**
- They predict likely text so **your job is to verify** (remember our tests?)

Three Ways to Use an LLM

1. **Chat apps** (Claude, ChatGPT, Gemini) great for learning and quick questions
2. **API from Python** build LLMs into *your* programs ← today's focus
3. **Agentic tools** (**OpenCode**, Claude Code, Codex) the LLM uses tools: reads your project, edits files, runs tests ← next section

API anatomy (all providers are similar)

Your code → HTTPS request with an **API key** + **model name** + a list of **messages**
Roles: `system` (instructions), `user` (you), `assistant` (model's replies).
You pay per **token** (roughly per word piece) short prompts are cheap.

Fast-moving field

Model names and SDK details change every few months. **Always check the official docs.**

First Call: Anthropic Claude API

```python
# %pip install anthropic
import os, getpass
if "ANTHROPIC_API_KEY" not in os.environ:
    os.environ["ANTHROPIC_API_KEY"] = getpass.getpass("API key: ")

from anthropic import Anthropic
client = Anthropic()          # reads ANTHROPIC_API_KEY

msg = client.messages.create(
    model="claude-sonnet-4-5",  # check docs for current names
    max_tokens=300,
    messages=[{"role": "user",
               "content": "Explain P-waves vs S-waves in 3 bullets."}],
)
print(msg.content[0].text)
```

- Get a key at `https://console.anthropic.com` never paste it into code you share

Same Idea: OpenAI API

```python
# %pip install openai
import os, getpass
if "OPENAI_API_KEY" not in os.environ:
    os.environ["OPENAI_API_KEY"] = getpass.getpass("API key: ")

from openai import OpenAI
client = OpenAI()              # reads OPENAI_API_KEY

resp = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[{"role": "user",
               "content": "Explain P-waves vs S-waves in 3 bullets."}],
)
print(resp.choices[0].message.content)
```

- Notice: the *shape* is the same key, model, messages, text out. Learning one SDK teaches you them all.

Prompt Engineering: Six Principles That Actually Work

1. **Role:** "You are a geophysics tutor..."
2. **Task:** one clear verb explain, fix, rewrite, extract
3. **Context:** who is it for, what data is involved
4. **Examples:** show 1–2 input/output pairs (few-shot)
5. **Format:** bullets? JSON? under 100 words?
6. **Iterate:** first prompt is a draft, not a spell

Weak prompt

```
"Tell me about seismic inversion"
```

Strong prompt

"You are a geophysics tutor for geology graduates. Explain seismic inversion using one real-life analogy, then one Python example of at most 8 lines. End with one practice question. Under 120 words."

Get structured data back (very powerful for geoscience)

"Extract the minerals from this rock description as JSON" → `json.loads(text)` → real data structures your program can use.

Turn Prompts into Reusable Functions

```python
SYSTEM = "You are a patient geophysics tutor for beginners."

def ask(question, style="short"):
    prompt = f"""{SYSTEM}

Answer the question below.
Style: {style} (short = max 80 words, detailed = include code).
Question: {question}"""
    # msg = client.messages.create(model="claude-sonnet-4-5",
    #     max_tokens=500,
    #     messages=[{"role": "user", "content": prompt}])
    # return msg.content[0].text
    return prompt       # preview mode: show the prompt text

print(ask("What is seismic tomography?", style="detailed"))
```

Your Turn #9 (10 min)

With your own key: build `explain_like_im_five("seismic tomography")` and `quiz_me("structural geology", 3)`. Compare a weak vs. strong prompt discuss the difference with your neighbor.

Keys, Costs, and Safety

- Keys are **passwords**: environment variables or `getpass`, never hardcoded
- On your machine: a `.env` file + `python-dotenv`, and put `.env` in `.gitignore`
- Set spending limits in the provider console
- Never send personal data, unpublished data, or secrets without permission

Real-world rule

If a leaked key would be a disaster, it is not stored safely. Treat API keys like credit cards.

Good habits

```
# .gitignore (on your machine)
.env
__pycache__/
.venv/
```

Roadmap

From Chatbot to Agent

A chatbot answers. An agent acts.

1. **Understand:** reads your files, tests, git history
2. **Plan:** proposes steps before touching code
3. **Act:** edits files, runs commands, writes tests
4. **Observe:** reads errors and output, then retries

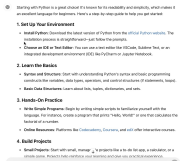
. . . in a loop, **with your approval** at each step.

You are the senior scientist

The agent is a fast junior: tireless, knowledgeable, but needs clear tasks, code review, and tests to stay on track.

This is why we learned basics first

You cannot review code you cannot read.


ChatGPT 4o mini

Starting with Python is a great choice! It's known for its readability and simplicity, which makes it an excellent language for beginners. Here's a step-by-step guide to help you get started:

1. Set Up Your Environment

- **Install Python:** Download the latest version of Python from the official Python website. The installation process is straightforward—just follow the prompts.
- **Choose an IDE or Text Editor:** You can use a text editor like VSCode, Sublime Text, or an integrated development environment (IDE) like PyCharm or Jupyter Notebook.

2. Learn the Basics

- **Syntax and Structure:** Start with understanding Python's syntax and basic programming constructs like variables, data types, operations, and control structures (if statements, loops).
- **Basic Data Structures:** Learn about lists, tuples, dictionaries, and sets.

3. Hands-On Practice

- **Write Simple Programs:** Begin by writing simple scripts to familiarize yourself with the language. For instance, create a program that prints "Hello, World!" or one that calculates the factorial of a number.
- **Online Resources:** Platforms like Codecademy, Coursera, and edX offer interactive courses.

4. Build Projects

- **Small Projects:** Start with small, manage_ts projects like a to-do list app, a calculator, or a simple game. Projects help reinforce your learning and give you confidence.

The 2026 Landscape (and Our Pick)

- **OpenCode** free, open-source agent; works with *any* model ← **today's focus**
- Claude Code (Anthropic), OpenAI Codex excellent, tied to one vendor
- GitHub Copilot, Cursor, Gemini CLI, Aider...

Why OpenCode for teaching?

Free to install, free models included, and you can plug in *whatever you already have* ChatGPT Plus/Pro, GitHub Copilot, or any API key. Same skills transfer to every other tool.

Disclaimer

Install commands and model names evolve monthly **check the official docs**:
`https://opencode.ai/docs`

What Is OpenCode?

- **Free & open source** AI coding agent (160k+ GitHub stars, millions of users)
- Runs where you work: **terminal**, desktop app, IDE extension
- **Any model**: 75+ providers via Models.dev Claude, GPT, Gemini, or local models
- **Free models included** start at zero cost
- Privacy-first: your code is **not stored** by OpenCode

Handy features

- Multi-session: several agents in parallel
- Share links for sessions (great for debugging with colleagues)
- Log in with your existing **ChatGPT Plus/Pro** or **GitHub Copilot** account

OpenCode: Install and First Run

```
# Option A: install script (macOS/Linux)
curl -fsSL https://opencode.ai/install | bash

# Option B: npm / Homebrew
npm install -g opencode-ai
brew install sst/tap/opencode

# Then, inside ANY project folder:
cd my_project
opencode          # start the agent
```

Inside OpenCode

- `/connect` add a provider or log in
- `/models` pick a model (free ones marked)
- `/init` write an `AGENTS.md` rulebook for the project
- **Tab** switch between *build* and *plan* agents

- First prompts: "Explain this codebase" · "Add input validation to `catalog.py`, then run the tests"

Choosing a Model: Strongest vs. Free

	Frontier models (paid)	**Free / open-weight models**
Examples	Claude Opus 4.5 / Sonnet 4.5 GPT-5.2 / GPT-5.1 Codex Gemini 3 Pro	DeepSeek V3.x / R1 · Qwen3 Coder GLM-4.x · Kimi K2 · MiniMax M2 Gemma / Llama (local: Ollama, LM Studio)
Best for	hard debugging, big refactors, subtle geophysics code	learning, small tasks, drafts, offline work, private data
Cost	per-token (or subscription)	zero (local: your hardware)

Practical strategy

Learn and prototype with **free** models; switch to a **frontier** model when the task gets hard or the code gets serious. In OpenCode: `/models` to switch anytime.

Moving target

New models every week. Check leaderboards before believing anyone (including me): LMArena, SWE-bench, `https://models.dev`

What Does It Cost? OpenCode Subscriptions and Free Routes

The OpenCode app itself is free. You only ever pay for *models* and even that can be $0:

Free routes (start here!)

- Free models included in OpenCode (`/models`)
- Log in with **ChatGPT Plus/Pro** or **GitHub Copilot** use a subscription you already have
- Free API tiers (e.g. Google AI Studio)
- **Local models** via Ollama / LM Studio free, offline, private

OpenCode Zen (optional, pay-as-you-go)

- Add $20 balance, **pay per request, zero markup**
- Models **tested & benchmarked** by the OpenCode team for coding agents
- Monthly spend limits, auto top-up, cancel anytime
- Zero data retention; works with *any* agent, not just OpenCode

Or bring your own API keys (Anthropic, OpenAI, . . .) and pay the provider directly.

Resources

Python & geoscience

- Official tutorial: `https://docs.python.org/3/tutorial/`
- CS50P (Harvard, free) · *Using Python for Research* (edX)
- ObsPy gallery: `https://docs.obspy.org`
- agile-geoscience notebooks (SEG-Y, well ties. . .)
- Google Colab: `https://colab.research.google.com`

LLMs & OpenCode

- OpenCode: `https://opencode.ai` · docs: `https://opencode.ai/docs`
- Model catalog: `https://models.dev`
- Anthropic / OpenAI docs for API details
- Leaderboards: LMArena, SWE-bench
- Companion notebook: `Python_Basics_Workshop.ipynb`

Minute 150: Time to Answer Our Own Question

"AI writes good Python. Why were you here?"

Go back to what you wrote at minute 3. Read it out if you want.

1. **You cannot review what you cannot read.** An agent's output is a diff and in science you sign the result, not the tool.
2. **Specifying is the new bottleneck.** *"Make it better"* gives noise; *"phi is a fraction, raise ValueError outside 0–1, add asserts"* gives reviewable work. The vocabulary *is* the skill.
3. **The dangerous failure is plausibility, not errors.** Python catches syntax; nothing catches fraction-vs-percent, m-vs-ft, or ergs-vs-joules. Those run clean and land in your thesis.
4. **Someone has to break the loop.** When the agent re-applies a broken fix, reading the traceback yourself is the only exit.
5. **Trust scales with cheap verification.** You wrote `assert` before you ever opened an agent tests are how you delegate safely.

The one sentence to remember

AI changed who writes the first draft. It did not change who is responsible for the final one.

Thank You!

Questions?

Islam Hamama · islam.hamama@nriag.sci.eg · NRIAG, Egypt