**14‑day logic boot plan (no mercy)**

each day = ~60–90 min. do it **in this order**. no skipping.

**Day 1 – Decompose or die**  
Take any problem and write 5–10 bullet “micro‑steps” before code. Force it.  
Example for “sum of evens in list”: validate input → filter evens → accumulate sum → return.

**Day 2 – Dry‑run discipline**  
For 3 tiny problems, walk through 2–3 inputs on paper: track variables line‑by‑line. This is where logic clicks.

**Day 3 – Patterns day**  
Write 5 patterns from scratch:

* counter pattern
* running min/max
* frequency dict
* two‑pointer scan
* stack use (balanced parentheses)

**Day 4 – Data modeling**  
Given a problem, list data structures first (list/dict/set/tuple). Why that choice? Code two variants swapping DS.

**Day 5 – Function contracts**  
For each function, write: inputs, outputs, side‑effects, raises. Then code. (You’ll stop “guess‑coding”.)

**Day 6 – Refactor pass**  
Take yesterday’s code and refactor: split functions, name better, remove duplication.

**Day 7 – Micro‑project**  
CLI calculator with argparse. Features: add/sub/mul/div, handles divide‑by‑zero, logs operations to log.txt.

**Day 8 – Strings day**  
5 katas: isogram, palindrome (ignore spaces/case), run‑length encoding, word frequency, title‑case with edge cases.

**Day 9 – Lists day**  
5 katas: chunk list, dedupe while preserving order, rotate k, merge two sorted lists, sliding‑window max (size k).

**Day 10 – Dicts/Sets day**  
Top‑k frequent items, anagram groups, diff of two lists, membership benchmark set vs list (tiny perf taste).

**Day 11 – File I/O day**  
Read CSV → compute summary (min/max/avg), handle missing lines, invalid numbers, nonexistent file.

**Day 12 – OOP day**  
Design Timer class (start/stop/elapsed/format). Add \_\_str\_\_, and a context manager (with Timer():).

**Day 13 – Blackjack prep**  
Write **only**: Card, Deck, Hand (with Ace logic). No UI. Green tests or it doesn’t count.

**Day 14 – Stitching logic**  
Add Player, Dealer, minimal Game loop (deal, hits, stands, resolve). Keep it CLI; no bells.

**2) testing that actually makes you dangerous**

**Edge‑case checklist (run this every time):**

* empty input / None
* wrong types (str where int expected)
* boundaries (0, 1, max, negatives)
* duplicates / ties
* ordering assumptions
* I/O errors (missing file, permission)
* weird unicode / spaces / casing (for strings)

**Unit test habits:**

* Name tests like test\_palindrome\_ignores\_spaces().
* Use 3+ cases per function: happy, boundary, nasty.
* Add a tiny **fuzz**: loop 50 random cases where possible.
* For functions with failure conditions, assert the **exception** class (with self.assertRaises(ValueError):).

**Minimal test runner ritual:**

1. python -m venv .venv && .venv/Scripts/activate (or source .venv/bin/activate)
2. pip install pytest pylint
3. Commands you run **every time**:
   * pytest -q
   * pylint src tests --score=y

**3) a real debugging framework (stop flailing)**

1. **Reproduce** in the smallest input. If you can’t reliably trigger it, you can’t kill it.
2. **Slice the code**: comment out halves (binary search the bug). Find the exact line where truth flips.
3. **Assert ruthlessly**: add assert condition, "why it should hold" near the suspected spot.
4. **Log > print**:
5. import logging; logging.basicConfig(level=logging.INFO)
6. logging.info("hand=%s, value=%d", hand, value)
7. **pdb breakpoint**: import pdb; pdb.set\_trace() → step variables, inspect state.
8. **Write a test that fails** reproducing the bug. Fix. Keep the test forever.

Start a “**bug graveyard**” file: cause → fix → lesson. 10 bugs later you’ll see your patterns.

**4) blackjack architecture that won’t collapse**

**Card**: rank, suit, value() (J/Q/King=10, Ace=11 initially).  
**Deck**: 52 cards, shuffle(), deal\_one().  
**Hand**: holds cards, computes value with **Ace adjustment**:

def value(self):

total = sum(min(10, r) if isinstance(r, int) else 10 for r in self.ranks\_without\_aces)

aces = self.ace\_count

total += 11 \* aces

while total > 21 and aces:

total -= 10 # flip one Ace from 11 to 1

aces -= 1

return total

**Player/Dealer**: both have a Hand. Dealer has should\_hit() (hit < 17).  
**Game**: state machine

* init → deal (2 each, dealer one face‑down)
* player turn (hit/stand)
* if bust → dealer wins
* else dealer turn → resolve comparisons  
  Write tests for:
* Hand.value with 0/1/2 aces (e.g., A+K, A+8+A, A+9+A+9)
* deck deals unique 52 cards
* dealer hits on 16, stands on 17

**5) daily cadence (so you don’t melt)**

* **30 min review** (notes + 1 old kata)
* **50 min build** (today’s target)
* **20 min tests/refactor** (non‑negotiable)  
  When tired: **do less, not zero**.

**6) non‑negotiables (from now on)**

* virtualenv per project
* black/pylint clean before you call it “done”
* README.md with “how to run + example”
* one screenshot or demo GIF for every mini‑project (portfolio fuel)