The Ultimate AP® Computer Science Principles Big Idea 1 Cheat Sheet

1.1 Computing Innovations & Collaboration

Computing Innovations

Computing innovation: includes a **program** as an integral part of its function. Can be physical, software, or a concept.

Type	Example	Program does
Physical	Self-driving car	Reads sensors
Software	Photo editor	Alters pixels
Concept	E-commerce	Runs checkout
Physical	Fitness tracker	Counts steps

Purpose: why it exists → solve a problem OR pursue an interest via creative expression (art, music, games).
Program integral: remove the program → innovation no longer works as designed. A plain bike ≠ one.
Effects: innovations can have beneficial + harmful effects, often both at once (Big Idea 5 goes deeper).

Why Collaborate?

Diverse perspectives (backgrounds, skills, users) → better ideas, more use cases found, fewer blind spots.
Bias: one person's assumptions get built into code. Varied team → bias spotted + cut **before** release.
Users as collaborators: get feedback from the people who will use it (surveys, interviews, testing).
Outcome: collab → innovation reflects more people's needs → more useful, more accessible + more fair.
Solo devs still collab: user feedback, forums, code reviews.

MCQ Patterns

"Best benefit of collab?" → diverse perspectives / reduce bias. Not "less code" or "faster run time".
"Is X a computing innovation?" → yes only if a program is integral to how it works (not just a tool used nearby).

⚠ A concept (e.g., social media, e-commerce) still counts as a computing innovation if a program is integral to it.

✓ Team answer pick: choose the option that adds new viewpoints or users, not the one that just adds speed.

1.1 Collaboration Skills & Tools

How Teams Build Software

Pair programming: 2 people, 1 computer. **Driver** types; **navigator** reviews + plans. Swap roles often.
Consult users/experts: talk to people outside the team to learn their needs + test ideas early in the process.
Online tools: shared docs, version control, code-sharing, chat → collab w/o being in same room.
Divide work: split into modules/procedures → each person owns a part; integrate + test together.
Code review: teammates read each other's code → catch errors, share fixes, keep style consistent.

Interpersonal Skills (CRD-1.C)

Skill	Looks like
Communication	Clear updates, listen
Consensus building	Agree on a plan as team
Conflict resolution	Fix disputes w/ respect
Negotiation	Trade-offs on features

Effective collab: every voice heard, roles clear, decisions recorded, feedback both given + taken openly.
Bad collab: 1 person decides everything, others' ideas get dismissed → narrower, weaker, more biased program.

Create PT & Collaboration

Create PT: may collab on program code, but the **written responses** + video must be your own work.
Credit: code written w/ a partner or from any other source must be acknowledged (author/source).
Gen AI: allowed as a tool, but treat its code like any outside source → acknowledge it + understand it.
Own it: you must be able to explain every part of your program, incl. any code you got elsewhere.

⚠ Unacknowledged code from others or from AI = plagiarism → Create PT can score 0 for all of it.

✓ Pair programming MCQ: both roles are active; the navigator catches errors + thinks ahead.

1.2 Program Purpose & Behavior

Core Vocabulary

Program (software): collection of program statements that does a specific task when run by a computer.
Code segment: collection of program statements that is **part** of a program (e.g., 1 loop or 1 procedure).
Program purpose: the problem it solves or the interest it serves. **Function**: what it does when it runs.
Behavior: how a program functions during execution; often described by how a user interacts with it.
Different inputs → program can run differently → different behavior + outputs on each run of the same code.
Software: a program or set of programs (app, game, OS).
Describe behavior: what the user does → what the program does in response → what the user sees.

Explain a Code Segment

Trace it: list each variable, step line by line, update values, note what is shown (DISPLAY) at each step.
Describe it right: say what the segment accomplishes (e.g., "finds max"), not just what each line does.
Example: x ← 3; REPEAT 2 TIMES { x ← x * 2 }; DISPLAY(x) → x = 6 after pass 1, 12 after pass 2 → shows **12**.
Selection: IF (score ≥ 90) { DISPLAY("A") } ELSE { DISPLAY("Retry") }: score 95 → A; 80 → Retry.
Same code, new input: different branch runs → different output. Explain WHICH branch + WHY.

Purpose vs. Function

Purpose = WHY ("help students track study time"). **Function** = WHAT ("logs minutes, shows weekly total").
Create PT: your stated purpose + function must match what the video + code actually show running.

⚠ Don't describe purpose as a feature list. Purpose names the problem or interest; function names the behavior.

✓ Written response: name the input, what the program does with it, and the output it produces from it.

1.2 Inputs, Events & Outputs

Input

Program input: data sent to a computer for processing by a program. From a **user** or **other programs**.

Form	Input e.g.	Output e.g.
Tactile	Tap, click, key	Phone vibrates
Audio	Voice command	Alarm sound
Visual	Camera, scan	Screen, image
Text	Typed name	Printed msg

Events

Event: tied to an action (click, key press, timer, data arriving) and supplies input data to a program.
Event-driven programming: statements run when **triggered** by an event, not in one top-to-bottom flow.
Example: onEvent("btn", "click", scoreUp) → scoreUp() runs on each click. Nothing runs until a click happens.
Common events: button click, key press, mouse move, timer firing, page load, sensor reading, data arriving.
Sequential vs event-driven: sequential runs line 1 → end; event-driven waits + reacts to each event.

Output

Program output: data sent from a program to a device. Can be visual, audio, tactile, or text-based.
Output is based on the program's input OR its prior state (e.g., a stored score, a saved list of items).
Input → process → output: always link all 3 when you explain a program's behavior in a written response.
Prior state e.g.: game shows "High score!" only if new score > saved best. Same input, diff output.
Output goes to: screen, speaker, motor, LED, printer, file, or another program, which uses it as input.

⚠ Input can also come from other programs (APIs, files, sensors), not only from a person typing it in.

✓ Create PT: program must take input that affects its output. Show both clearly in the video, in order.

1.3 Development Process

Phases (order can repeat)

Investigating + reflecting → **designing** → **prototyping** → **testing**, then loop back to any phase as needed.
Development process: can be ordered + intentional, or exploratory. Either way it's planned work.
Prototyping: build a rough working version to test ideas early → cheap to change before the full build.

Iterative vs. Incremental

	Iterative	Incremental
Idea	Refine + revise	Build in pieces
Driven by	Feedback, tests	Small parts
Goes back?	Yes, revisits	Adds next part
Check	After each loop	Each piece works

Iterative: repeated cycles; feedback, testing + reflection → revisit earlier phases to refine + revise.
Incremental: break problem into smaller pieces; make sure each works before adding to the whole.
Both together: most real teams build in increments AND iterate on each increment as feedback comes in.

Investigation Methods

Find user needs: surveys, user testing, interviews, direct observation of users doing the real task.
Investigation reveals: the constraints, concerns, + interests of the people who will use the program.
Requirements: how a program should behave → used to judge if it works + to pick its test inputs.
Program specification: defines the requirements → the target the finished program must meet.

⚠ Iterative ≠ "repeating code in a loop." In Big Idea 1 it means repeating the development cycle.

✓ MCQ cue: "based on user feedback" → iterative. "One part at a time, each tested" → incremental.

1.3 Design, Documentation & Credit

Design Phase

Design steps: brainstorm, plan + **storyboard**, split program into **modules**, diagram the UI, write a test plan.
User interface (UI): what users see + interact with (buttons, screens, inputs). Design it for real users.
Modules: separate parts (often procedures) w/ one job each → easier to build, test, debug + share work.

Documentation

Program documentation: written description of how a code segment, event, procedure, or program works.
Comments: documentation written **in** the code for people to read. They do NOT affect how the program runs.
When: document **throughout** development, not just at the end → helps teammates + future you later.
Good comment: says WHY or what a block does ("// find highest score"), not every single line.
Why: aids maintenance, debugging, collab. Some environments don't support comments → use other docs.

Acknowledging Sources

Source	Must credit?
Your own new code	No
Partner/team code	Yes: name + part
Library / API	Yes: name + source
AI-generated code	Yes: tool + part

How: in documentation/comments, give the original author's name + the source of code you didn't write.
Libraries/APIs: reusing pre-written code saves time, but credit it + read its docs to use it right.

⚠ Comments never change output. Any MCQ answer saying a comment alters behavior is wrong.

✓ Documentation answer pick: it helps others understand, maintain, debug + reuse the code later.

1.4 Error Types & Debugging

4 Error Types

Error	What happens	E.g.
Syntax	Rules broken	Missing (or }
Logic	Runs, wrong result	> vs ≥
Run-time	Fails mid-run	Index too big
Overflow	# out of range	Too big for bits

Syntax error: language rules not followed (e.g., a missing bracket or quote) → program won't run at all.
Logic error: mistake in the algorithm → incorrect or unexpected behavior; no crash, so hardest to spot.
Run-time error: occurs during execution (e.g., divide by 0, list index out of bounds). Language defines these.
Overflow error: computer tries to handle a number outside its defined range of values (not enough bits).
Find it fast: won't run → syntax. Crashes partway → run-time. Wrong answer, no crash → logic.

Debugging Tools

Test cases: run w/ chosen inputs, then compare the actual output vs the expected output for each.
Hand tracing: track variables on paper, step by step.
Visualizations + **debuggers**: step through code 1 line at a time + watch variable values change.
Extra output statements: DISPLAY variable values mid-program to see exactly where it goes wrong.

Spot the Logic Error

Off-by-one: REPEAT UNTIL (i > n) vs (i ≥ n) skips or adds a pass. AP lists start at **index 1**, not 0 as in Java/Python.
Wrong operator: AND vs OR, > vs ≥, or = vs ≠ in a condition → wrong branch runs for some inputs.
Wrong init: max ← 0 fails if all values are negative → start max at the list's 1st element instead of at 0.

⚠ A program that runs w/o crashing can still have a logic error. "No error message" ≠ correct.

1.4 Testing & Create PT Moves

Testing for Correctness

Defined inputs: pick inputs, predict expected outputs/behavior first, then run + compare.
Requirements drive test choice: test every behavior the program is supposed to have, one by one.
Test data: typical values, **boundary/extreme** values (edges of ranges), + invalid values it should reject.

Test for ages 13–18	Input	Expect
Typical	15	Accept
Low boundary	13	Accept
Just outside	12, 19	Reject
Invalid	"abc"	Error msg

Passing tests ≠ proof of no bugs; it only shows correct output for the specific inputs you actually tried.
Hand trace e.g.: input 13 w/ IF (age > 13) → rejects a valid age → logic error; fix the condition to age ≥ 13.

Create PT Written Response

Program design: name a user group, the input, + exactly how that input changes the program's output.
Development: describe a specific point where you iterated or fixed something, + why you made that change.
Errors + testing: name an error you found, how you found it (test case, trace), + exactly how you fixed it.
Calls: describe 2 calls to your procedure w/ different arguments → different code paths → different results.
Conditions: say what input makes a branch or loop run, + what happens when that condition is false.
PPR: Personalized Project Reference = your code screenshots (procedure + list). No comments allowed in it.
Video: ≤1 min, shows input, program functionality, + output. No voice narration (captions OK); no ID info.

✓ Answer w/ YOUR code: quote names of your procedure, list, + variables. Generic answers lose points.