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

1.1 Computing Innovations & Collaboration

Computing Innovations

Computing innovation: has a **program** as an integral (needed) part. Can be physical, software, or a concept (e.g., e-commerce, social media).

Type	Example
Physical	Self-driving car
Software	Photo editor app
Concept	E-commerce

Purpose: solve a problem OR pursue an interest via creative expression (art, games).
Program integral: take the program away → it no longer works. A plain bike ≠ one.

Why Collaborate?

Diverse perspectives (different backgrounds + skills) → better ideas, fewer blind spots.
Bias: one person's assumptions get built into code. Varied team → bias caught early.
Users: get feedback from the people who will actually use it, early + often in development.

MCQ Pattern

"Best benefit of collab?" → diverse perspectives / less bias. Not "faster code" or "less code."
"Is X a computing innovation?" → only if a program is integral to how it works.

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

⚠ A concept like e-commerce still counts as a computing innovation if a program is integral to it.

1.1 Collaboration Skills & Tools

How Teams Build Software

Pair programming: 2 people, 1 computer. **Driver** types; **navigator** reviews + plans. Swap often.
Consult users/experts: ask people outside the team about their needs, ideas + concerns.
Online tools: shared docs, chat, code-sharing → work together from anywhere, anytime.
Divide work: each person owns one module (procedure), then the team tests it all together.
Code review: teammates read each other's code → catch errors early + share fixes.

Interpersonal Skills

Skill	Looks like
Communication	Clear updates, listen
Consensus	Team agrees on plan
Conflict resolution	Fix disputes kindly
Negotiation	Trade off features

Create PT Rules

Written responses + video must be your own work, even if a partner helped w/ code.
Credit: name the author/source of any code you didn't write, incl. AI-generated code.
Own it: be able to explain every part of your code, even code you got from elsewhere.

✓ Pair programming: both roles are active. The navigator catches errors + plans ahead.

⚠ Using others' code or AI code w/o credit = plagiarism → Create PT score of 0.

1.2 Program Purpose & Behavior

Core Vocabulary

Program (software): set of statements that does a specific task when run by a computer.
Code segment: a group of statements that is **part** of a program (e.g., one loop or one IF).
Behavior: how a program acts while running, often described by how a user interacts with it.
Different inputs → different behavior + different outputs each time the same code runs.
Describe behavior: user action → program response → what the user sees.

Explain a Code Segment

Trace it: track each variable line by line in a table; note what each DISPLAY shows as you go.
Example: x ← 3; REPEAT 2 TIMES { x ← x * 2 }; DISPLAY(x) → 6, then 12 → shows 12.
Selection: IF (score ≥ 90) { DISPLAY("A") } ELSE { DISPLAY("Retry") } → output depends on input.
Big picture: say what the code does overall (e.g., "finds the max"), not what each line does.

Purpose vs. Function

Purpose = WHY the program exists (problem or interest). **Function** = WHAT it does when it runs.
Example: purpose = "help students track study time"; function = "logs minutes."

⚠ Purpose is not a feature list. It names the problem or interest the program serves.

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

1.2 Inputs, Events & Outputs

Input & Output

Input: data sent to a program to process, from a **user** or from **another program** (or a sensor).

Form	Input	Output
Tactile	Tap, click	Vibration
Audio	Voice	Sound
Visual	Camera	Screen
Text	Typed name	Message

Output: data sent from a program to a device; based on input OR prior state (e.g., saved score).
Prior state e.g.: game shows "High score!" only if new score beats the saved best.
Input → process → output: link all 3 whenever you explain how a program works in an answer.

Events

Event: an action (click, key press, timer) that supplies input data to a program as it runs.
Event-driven programming: code runs when an event **triggers** it, not top to bottom.
Example: click a button → scoreUp() runs.
Common events: button click, key press, mouse move, timer, page load, sensor reading.
Sequential vs event-driven: sequential runs top to bottom; event-driven waits + reacts.

⚠ Input can come from other programs or sensors, not only a person typing.

✓ Create PT: your program needs input that affects output. Show both in the video.

1.3 Development Process

Phases

Investigate → **design** → **prototype** → **test**, then loop back to any phase as needed.
Prototype: a rough early version to test ideas cheaply before the full build begins.

Iterative vs. Incremental

	Iterative	Incremental
Idea	Refine	Small pieces
Driven by	Feedback	Parts
Goes back?	Yes	Adds next

Iterative: repeat the cycle; use feedback, testing + reflection to revise + improve it.
Incremental: build in small pieces; make sure each piece works before adding the next one.
Both together: most teams build in pieces AND iterate on each piece as they go.

Investigation

Find user needs: surveys, user testing, interviews, and watching users do the real task.
Investigation shows: users' constraints, concerns, + interests, so you know them before you design.
Requirements: how the program should behave → used to judge if it works + to pick tests.
Specification: the full list of requirements the finished program must meet to count as done.

⚠ Here, iterative means repeating the development cycle, not a loop in code.

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

1.3 Design, Documentation & Credit

Design Phase

Design steps: brainstorm, **storyboard**, split into **modules**, sketch the UI, plan tests.
User interface (UI): what users see + click (buttons, screens, text boxes, menus).
Modules: separate parts w/ one job each → easier to build, test, debug + share among the team.

Documentation

Documentation: written description of what code does + how it was developed over time.
Comments: notes in the code for people to read. They do NOT change how the program runs.
When: document **throughout** development, not just at the very end, so nothing gets forgotten.
Why document: helps teammates + future you fix, update, + reuse the code much later on.

Give Credit

Source	Credit it?
Your own code	No
Partner's code	Yes
Library / API	Yes
AI-generated code	Yes

How: in comments or docs, give the original author's name + where the code came from.

⚠ Comments never change output or behavior. Any MCQ answer saying they do is wrong.

✓ Documentation answer: it helps others understand, maintain + reuse code.

1.4 Error Types & Debugging

4 Error Types

Error	Result
Syntax	Won't run (rules)
Logic	Runs, wrong result
Run-time	Crashes mid-run
Overflow	Number too big

Logic error: mistake in the algorithm → wrong behavior, no crash. Hardest to spot.
Syntax error: language rules broken, e.g., a missing (or } → program won't run.
Run-time error: happens while running, e.g., divide by 0 or a list index that doesn't exist.
Overflow error: a number outside the range the computer can store (too few bits).
Quick ID: won't run → syntax. Crashes mid-run → run-time. Runs but wrong answer → logic.

Debugging Tools

Test cases: compare actual vs expected output.
Hand tracing: track variables on paper.
Debuggers + extra DISPLAYs: watch variable values change while the program runs.

Common Logic Errors

Off-by-one: loop runs 1 time too many or too few (i > n vs i ≥ n). AP lists start at index 1.
Wrong operator: AND vs OR, or > vs ≥ → the wrong branch runs for some inputs.

⚠ No error message ≠ correct. A program can run fine and still have a logic error.

1.4 Testing & Create PT Moves

Testing

Plan tests: pick inputs, predict the expected outputs first, then run + compare the results.
Requirements: test every behavior the program is supposed to have, one at a time.
Test data: typical, **boundary** (edge of the range), + invalid values the program should reject.

Ages 13–18	Input	Expect
Typical	15	Accept
Boundary	13	Accept
Outside	12	Reject
Invalid	"abc"	Error

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

Create PT Written Response

Design: name a user group, the program's input + exactly how that input changes the output.
Errors: name a bug you hit, how you found it (test case, hand trace) + exactly how you fixed it.
Calls: 2 calls w/ different arguments → different code paths run → different results returned.
PPR: screenshots of your code (procedure + its call, list + its use). No comments.
Video: ≤1 min, shows input, what the program does, + output. No voice; captions OK.

✓ Use YOUR names: your procedure, list, + variables. Generic answers lose points.