

Unit 1: Creative Development

Unit 1 is about how programs get built, not how code is written. It covers the creative development process, what programs are for, how people build them together, how errors are found and fixed, and the Create Performance Task that counts toward your AP score.

AP Computer Science Principles

Creative Development

About 10 minutes to read

How to use this guide

Read it in order the first time. Unit 1 is mostly vocabulary and scenarios: the process names the phases of building, errors name what goes wrong, and debugging names how you find it. The exam tests these as situations, not definitions, so read each example and picture what you would do in that spot.

After the first read, use the trap boxes and the tables to review the distinctions that exam questions test most often. Finish with the practice questions, then complete the recall check on the last page out loud and note any items you cannot explain yet.

What this unit is worth. Creative Development is about 10 to 13 percent of the AP Computer Science Principles exam. That is the smallest slice of the exam, but the vocabulary here shows up in every other unit: program, input, output, statement, code segment. This unit also introduces the Create Performance Task, which counts for 30 percent of your AP score together with the written responses you answer on exam day. Learn the process language now and the rest of the course gets easier to read.

1.1 The Creative Development Process

A **computing innovation** is a new or improved product or process that uses computing. A scheduling app, a better checkout flow, a game nobody had thought of: all of them started as ideas that someone developed. The **creative development process** is the set of steps programmers follow to turn an idea into a working program, and it is **iterative**, which means you repeat the steps and improve the program each time through.

Most descriptions of the process use four phases. **Investigate.** Define the problem and gather information. Talk to the people who might use the program, study existing solutions, and decide what the program needs to do. **Design.** Plan the program before writing much code. Decide on the inputs and outputs, sketch the user interface, and choose the approach the program will take. **Prototype.** Build an early, incomplete working version. A prototype exists to get feedback, not to be finished. **Test.** Run the prototype, check it against its goals, and collect feedback from users.

Iteration is what ties the phases together. Feedback from testing usually sends you back to an earlier phase: redesign a screen, rebuild a feature, retest. Each pass through the loop improves the program. A prototype that fails a test is not a wasted effort. It is the loop working the way it should.

Trap. A prototype is built to be changed, so do not confuse it with a rough draft of the final product. Testing only once at the very end throws away the main benefit of the process, which is catching problems while they are still cheap to fix. On exam scenarios, if a programmer gathers user feedback and then revises the program, that is iteration, whichever phase they return to.

1.2 Program Purpose, Inputs, and Outputs

The **purpose** of a program is what it does and why it exists: the problem it solves or the function it performs. A grade calculator exists so teachers can compute final grades quickly. Purpose answers why someone built the program.

Inputs are the data a program receives. Common sources are the user (typed text, clicks, taps), files, sensors (a microphone, a GPS receiver), and events (a button press, a message arriving). **Outputs** are what the program produces in response: text on a screen, graphics, sounds, or actions taken by a device, like a robot turning. One program's output often becomes another program's input.

A **program statement** is a single instruction. A **code segment** is a sequence of statements that does one part of the program's job. Programs are built out of code segments. Two statements from the exam reference sheet that you will see all year:

```
name ← INPUT()
DISPLAY("Hello, " + name)
```

INPUT() reads data from the user, ← stores a value in a variable, and DISPLAY shows output on the screen.

Trap. Purpose and output are different questions. The purpose of a weather app is helping people plan around the forecast. Its output is the forecast display on the screen. When a question asks for an input, name the data going in, like the user's location, not what the program does with it.

1.3 Collaboration

Programs are usually built by teams, and the exam treats collaboration as a skill, not just a nice habit. Two benefits show up again and again. **Diverse perspectives** lead to better programs, because people with different backgrounds notice different problems and suggest different fixes. **Division of labor** shortens development time, because tasks can be split up and worked on in parallel.

Pair programming is a specific collaboration method: two programmers share one computer. The **driver** operates the computer and writes the code. The **navigator** reviews each line as it is written, thinks about the overall design, and catches errors. The partners switch roles regularly, and both share responsibility for the decisions.

Collaboration also means giving credit. When you use someone else's code or ideas, a tutorial snippet, an open-source library, a partner's algorithm, say so. Using another person's work as your own without credit is **plagiarism**. Crediting sources is part of honest development, and the Create Performance Task expects you to cite the code and ideas that are not yours.

Trap. The driver is not the leader and the navigator is not the assistant. Both programmers make decisions; the roles describe who is typing and who is reviewing, and they rotate. Also, collaboration has limits on the Create task. Developing the program code with a partner is allowed. The video, the written responses, and your project reference must be your own work.

1.4 Program Errors

Every program has bugs at some point. The exam names four kinds of errors. The difference is when the error shows up and what the program does about it.

ERROR TYPE	WHEN IT APPEARS	EXAMPLE
Syntax error	Before the program runs. The code breaks the rules of the language, so it cannot be translated.	A misspelled keyword or a missing parenthesis. The program never starts.
Runtime error	While the program is running. The program starts, then stops or misbehaves.	Dividing by zero, or reading past the end of a list. The program crashes partway through.
Logic error	After the program finishes. It runs to completion but produces the wrong result.	A tax program that multiplies by 0.8 instead of 0.08. Nothing crashes, but every answer is wrong.
Overflow error	During a computation, when a value is too large for its data type to hold.	A score that grows past the maximum value a fixed-size integer can store.

Logic errors are the hardest to find, because nothing visibly goes wrong. Syntax errors are the easiest to notice, because the program refuses to run at all.

Trap. A program that runs without crashing is not necessarily correct. That is exactly where logic errors live. Match the error to the symptom: refuses to start, points to syntax. Crashes partway, points to runtime. Finishes with the wrong answer, points to logic.

1.5 Debugging Strategies

Debugging is finding and fixing errors. **Testing** comes first: run the program with known inputs and compare the actual output to the expected output. A **test case** is one specific input plus the output you expect. Good testing includes **edge cases**, inputs at the extremes, like an empty entry, the smallest allowed value, or the largest.

Three strategies help when a test fails. **Hand tracing.** Walk through the code line by line on paper, writing down the value of each variable as it changes. Slow, but it shows exactly what the program does instead of what you assumed it does. **DISPLAY statements.** Add temporary DISPLAY statements that print variable values at key points, then run the program and read where the values stop matching your expectations. Remove the extra statements when the bug is fixed. For example:

```
total ← 0
FOR EACH price IN prices
{
    total ← total + price
    DISPLAY(total)
}
```

If the final total is wrong, the printed values show which addition went wrong. The `←` and `DISPLAY` spellings match the exam reference sheet. One reference-sheet habit to build now: list positions start at 1, so the first item in `prices` is `prices[1]`, not `prices[0]`.

Rubber duck debugging. Explain your code out loud, line by line, to someone else, or to an actual rubber duck. Putting each step into words forces you to confront the assumptions you skipped while writing.

Fix one error at a time and rerun your test cases after each change. Fixing two things at once makes it impossible to tell which change worked.

Trap. Testing finds errors; it never proves a program correct. Passing every test case you wrote only means those tests found nothing. New inputs, especially edge cases, can still break it. On the exam, if a question asks what testing guarantees, the answer is that it reveals the presence of errors, never their absence.

1.6 Documentation and Comments

A **comment** is a note written inside the code that the computer ignores. Comments exist for people: they explain *why* a section of code works the way it does, which is the part the code itself cannot say. This line is noise:

```
count ← count + 1 // adds 1 to count
```

The code already says that. A useful comment says something the code does not, like why the count starts at 0 or what the count is used for later. **Documentation** is the broader record around a program: how to install and use it, what each part does, and the design decisions behind it. Both make programs easier to read, understand, and modify, which matters the moment someone else, or future you, has to change the code.

Trap. Comments explain the why, not the what. If a comment just repeats the code in English, it adds nothing. The exam may show a comment and ask whether it is useful. Ask whether it tells you something the code does not.

1.7 The Create Performance Task

The **Create Performance Task** is the part of the AP exam you build yourself. You design and write a program of your choice, then submit the program code, a video, and a Personalized Project Reference. Together with the written responses you answer on exam day, the task counts for 30 percent of your AP score.

The video is one minute or less. It shows your program running and must include input, at least one aspect of the program's functionality, and output. It has no narration and no identifying information. The Personalized Project Reference is your own document of key code segments from your program. You keep it in front of you during the exam.

On exam day you get 60 minutes to answer two written-response questions, four prompts in all, about your own program. They ask about your program's design and purpose, the algorithm you implemented, how you tested the program and fixed the errors you found, and the abstractions you used. Everything in this unit, purpose, process, errors, debugging, documentation, is the vocabulary those prompts expect.

You get at least 9 hours of class time to build the program, record the video, and prepare your reference. Working with a partner on ideas, code, and testing is allowed, but the video, the reference, and the written responses must be entirely your own work.

Trap. The students who struggle with the Create task usually picked a program too big to finish or started too late. Pick something small enough to complete and leave time to test it. A finished simple program with clean written responses beats an unfinished ambitious one.

Confusions That Cost Points

PAIR	HOW TO KEEP THEM STRAIGHT
Investigate, design, prototype, test	Investigate defines the problem and gathers information. Design plans the program. Prototype builds an early version for feedback. Test evaluates it against its goals. Feedback sends the process back to an earlier phase.
Purpose vs. output	Purpose is why the program exists. Output is what the program produces. A weather app exists to help people plan; its output is the forecast display.
Input vs. what the program does	Input is data going in. "It calculates averages" describes the program's behavior, not its input. The input is the numbers it was given.
Driver vs. navigator	The driver operates the computer and types the code. The navigator reviews each line and thinks about design. Both make decisions, and the roles rotate.
Syntax vs. runtime vs. logic vs. overflow	Refuses to start: syntax. Crashes while running: runtime. Finishes with the wrong answer: logic. Value too large for the data type: overflow.
Testing vs. debugging	Testing finds errors by comparing actual output to expected output. Debugging fixes them. Testing never proves a program has no errors.
Comment vs. documentation	A comment is a note inside the code, ignored by the computer. Documentation is the broader record: user guides, design notes, how-to-use records.
Passing tests vs. proven correct	Passing tests means those particular tests found nothing. Edge cases and untested inputs can still fail.

Practice Questions

Original questions written for this guide in the style of the AP exam. Answers and explanations are on the next page, so complete the questions before checking them.

1. In pair programming, the navigator's main responsibility is to

1. write the code while the driver watches
2. review each line of code as it is written and think about the overall design
3. test the finished program after development is complete
4. write the documentation while the driver codes

2. A program asks the user for two numbers and divides the first by the second. It runs normally until the user enters 0 as the second number, at which point the program stops unexpectedly. This is an example of a

1. runtime error
2. syntax error
3. logic error
4. overflow error

3. Lena has an idea for a study app. Before writing the full program, she builds a simple working version with only the flashcard screen so she can show it to classmates and gather feedback. This step is

1. investigating
2. designing
3. prototyping
4. testing

4. A program reads a list of daily temperatures from a file and displays the average. Which of the following is an input to this program?

1. The average temperature it displays
2. The list of temperatures read from the file
3. The formula it uses to compute the average
4. The purpose of tracking weather trends

5. A student's program passes every test case she wrote, including several edge cases. Which conclusion is valid?

1. The program is proven to contain no errors.
2. The program's logic is definitely correct.
3. No errors were found by these test cases.
4. No further testing is needed.

6. Marcus suspects a variable holds the wrong value midway through his program. He reads his code aloud line by line, explaining what each line does, and spots the mistake while explaining. This strategy is

1. hand tracing
2. writing a test case
3. adding documentation
4. rubber duck debugging

7. Two students building a program together notice different problems: one spots a confusing screen layout, the other spots a miscalculation. Their program ends up better than either would have built alone. This best illustrates which benefit of collaboration?

1. Division of labor
2. Pair programming
3. Iteration
4. Diverse perspectives

8. For the Create Performance Task, which of the following must be completed without collaboration?

1. The video and the written responses
2. Developing the program code
3. Testing the program for errors
4. Planning the program's design

Answer Key

- 1. B.** The navigator reviews each line as it is written and thinks about the overall design, catching errors the driver misses. A reverses the roles; the driver is the one who writes the code. C describes testing, which both partners do throughout development, not the navigator's specific role. D invents a division of work; documentation is shared, not assigned to one role.
- 2. A.** The program ran and then stopped unexpectedly during execution, which is the definition of a runtime error. B is wrong because a syntax error prevents the program from running at all, and this program ran fine until the bad input. C is wrong because a logic error finishes with a wrong result; here nothing finished. D is wrong because overflow is about a value too large for its data type, not division by zero.
- 3. C.** She built an early, incomplete working version specifically to gather feedback, which is a prototype. A is wrong because investigating is gathering information and defining the problem, which happened before this step. B is wrong because designing is planning, not building. D is wrong because testing is evaluating a built version against its goals; the version she built for feedback is the prototype itself.
- 4. B.** The temperatures read from the file are the data going into the program, which makes them inputs. A is the output, what the program displays. C is part of the program's behavior, the instructions it follows, not data it receives. D is the purpose of the program, which is why it exists, not an input.
- 5. C.** Passing tests only shows that those particular test cases found nothing. A is the classic trap: testing can reveal the presence of errors but can never prove their absence. B overreaches for the same reason; the tests checked specific inputs, not every possible behavior. D is dangerous confidence; untested inputs may still fail, so more testing is always possible.
- 6. D.** Explaining code out loud line by line to find a mistake is rubber duck debugging. A is wrong because hand tracing is done on paper by tracking variable values, not by explaining aloud. B is wrong because a test case is a specific input paired with an expected output, not a read-aloud. C is wrong because documentation is a record for users and future developers, not a debugging strategy.
- 7. D.** The two students noticed different problems because they see things differently, and the program improved as a result; that is the diverse-perspectives benefit. A is wrong because division of labor is about splitting tasks to save time, not about noticing different problems. B is wrong because pair programming names a specific method, while the question asks for the benefit it produced. C is wrong because iteration is repeating the development phases, not a benefit of collaboration.
- 8. A.** The video, the Personalized Project Reference, and the written responses must be entirely your own work. B, C, and D are all allowed with a partner: developing the program code, testing it for errors, and planning its design are all collaborative parts of the task.

When you check your answers, note which distinction each miss came from. Make a flashcard for that distinction and drill it spaced out over the next few days instead of rereading the whole section. If you missed one of these questions, the same distinction is worth practicing again in Rycal, where the Creative Development deck has flashcards for it and more practice questions use the same kinds of traps.

One-Page Recall Check

Say each answer out loud before you look back, and mark the ones you cannot finish. Anything you cannot say out loud yet belongs in your flashcard deck. In Rycal, add those items to the Creative Development deck and let spaced review bring them back over the next few days.

- Define a computing innovation and give an example.
- List the four phases of the creative development process and what happens in each.
- Explain what iteration means and why it improves a program.
- State the purpose of a program in your own words.
- Give two examples of program inputs and two of program outputs.
- Explain the difference between a program statement and a code segment.
- Name the two benefits of collaboration the exam tests.
- Describe the driver and navigator roles in pair programming.
- Define syntax, runtime, logic, and overflow errors, and give an example of each.
- Explain the difference between testing and debugging.
- Describe how a test case is built and what an edge case is.
- Explain how hand tracing, DISPLAY statements, and rubber duck debugging each help find a bug.
- State what makes a code comment useful.
- Describe what the Create Performance Task requires you to submit.
- Name what must be your own work on the Create task.

Where to go next. Turn every missed item above into flashcards and drill them spaced out over several days rather than in one sitting. In Rycal, open the Creative Development deck under AP Computer Science Principles. The deck covers the terms in this guide, and its practice questions target the same traps named here. If you have a test date, add it in the Test Planner. You can also start your next review with a Brain Dump, then check what you missed against this guide.

Key terms for this unit

Computing innovation, Creative development process, Iteration, Investigate, Design, Prototype, Test, Purpose, Input, Output, Program statement, Code segment, Collaboration, Pair programming, Driver, Navigator, Syntax error, Runtime error, Logic error, Overflow error, Debugging, Test case, Expected output, Actual output, Edge case, Hand tracing, Rubber duck debugging, Comment, Documentation, Plagiarism, Create Performance Task, Personalized Project Reference.

About this guide. Written for Rycal and aligned to the College Board AP Computer Science Principles course framework, Unit 1. All questions and explanations are original Rycal writing. Rycal is independent and is not affiliated with or endorsed by the College Board.