CSc 120
Introduction to Computer Programming II

Adapted from slides by
Dr. Saumya Debray

02: Problem Decomposition and Program Development

A common student lament

"I have this big programming assignment.
I don't know where to start."

Steps in writing a program

1. Understand what tasks the program needs to perform

2a. Figure out how to do those tasks

2b. Write the code

3. Make sure the program works correctly

An example

Problem statement:

"Write a program to compute student GPAs from their grades."

Steps in writing a program

1. Understand what tasks the program needs to perform

2a. Figure out how to do those tasks

2b. Write the code

3. Make sure the program works correctly

Step 1. Problem specification

- Before you start writing code, make sure you understand exactly what your code needs to do.
  - what is the input?
  - what is the output?
  - what is the computation to be performed?
  - how can we tell that the program is working correctly?
- If necessary, ask questions to clarify these points.
- Time spent doing this is an investment, not a waste.

Example: cont'd

Problem statement:

"Write a program to compute student GPAs from their grades."

Example: cont'd

Problem statement:

"Write a program to compute student GPAs from their grades."

- Input:
  - read from a file, or from the keyboard?
  - what is the format?
  - how many students?
  - …

Example: cont'd

Problem statement:

"Write a program to compute student GPAs from their grades."

- Output:
  - to a file, or to the screen?
  - what is the format?
  - compute GPA for all students, or only specific students?
  - …

Example: cont'd

Problem statement:

"Write a program to compute student GPAs from their grades."

- Computation:
  - how is a GPA computed?
    - what information do we need?

Example: cont'd

Problem statement:

"Write a program to compute student GPAs from their grades."

- Testing:
  - how can we tell whether the program is working correctly?
    - how should we test it?
    - how can we tell whether all the pieces of the program are working properly?

Example: cont'd

Problem statement:

"Write a program to compute student GPAs from their grades."

- Input:
  - read from a file, or from the keyboard?

    from a file
  - what is the format?

    one student per line

    format of each line: student name, $course_1$: $grade_1$, …, $course_n$: $grade_n$

    different students may take different numbers of courses
  - how many students?

    not fixed ahead of time

Example: cont'd

Problem statement:

"Write a program to compute student GPAs from their grades."

- Output:
  - to a file, or to the screen?

    to the screen
  - what is the format?

    one student per line

    student name **:** *GPA*
  - compute GPA for all students, or only specific students?

    all students in the input file

Example: cont'd (digression: computing GPAs)

Suppose a student has the following grades:

Course	No. of units (U)	Grade (G)	U x G*
CSc 110	4	A	4 x 4 = 16
CSc 352	3	C	3 x 2 = 6
CSc 391	1	A	1 x 4 = 4
TOTAL:	4 + 3 + 1 = 8		16 + 6 + 4 = 26

* A = 4
B = 3
C = 2
D = 1
E = 0

The student's GPA = (Total UxG) / (Total U) = 26/8 = 3.25

Example: cont'd

Problem statement:

"Write a program to compute student GPAs from their grades."

- what is the input?
- what is the output?
- what is the computation to be performed?
- how can we tell that the program is working correctly?

There may be more than one way to do these

Need to:

- figure out the no. of units for each course
- translate letter grades to numbers (e.g., A = 4, B = 3, …)

Steps in writing a program

1. Understand what tasks the program needs to perform

2a. Figure out how to do those tasks

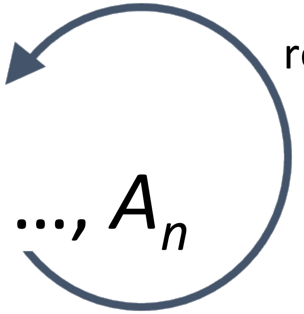
2b. Write the code

3. Make sure the program works correctly

Step 2a. Problem decomposition (conceptual)

- Write down the task(s) the program needs to perform

- pick a task A
- break A down into a set of simpler tasks $A_1, \ldots, A_n$
  - $A_1, \ldots, A_n$ together accomplish A



repeat as needed

before you start writing code to solve a problem, make sure you know how to solve the problem yourself.

Steps in writing a program

1. Understand what tasks the program needs to perform

2a. Figure out how to do those tasks

2b. Write the code

3. Make sure the program works correctly

Step 2b. Problem decomposition (programming)

- Write a piece of code for each task that has to be performed
  - initially the code will contain *stubs*, i.e., parts that have not yet been fleshed out
  - write down the task to be performed as a comment

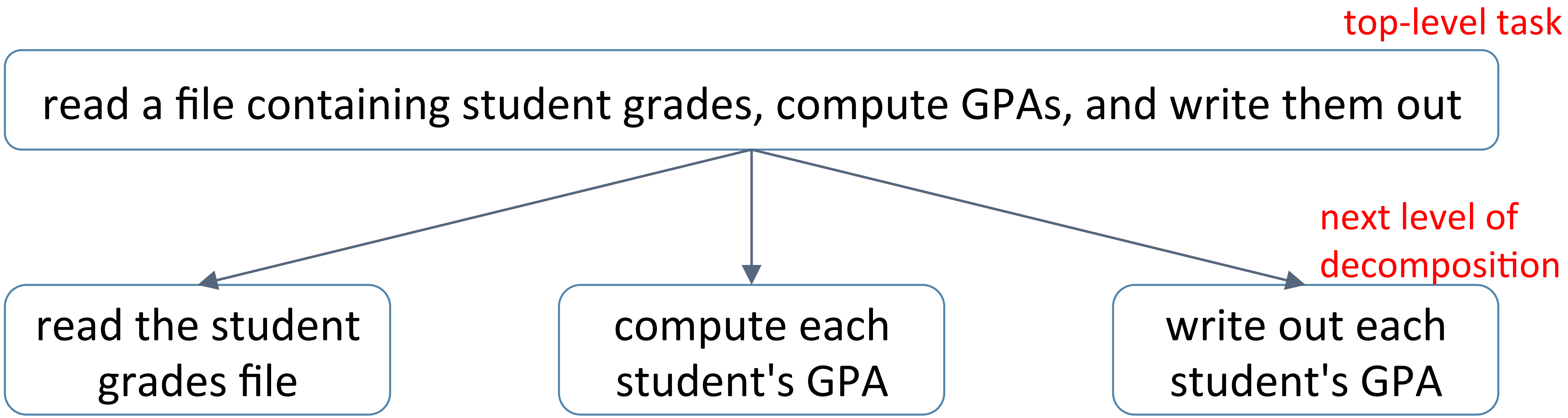
- Decomposing a task into sub-tasks $\Rightarrow$ fleshing out the code for a stub
  - repeat until no more stubs to flesh out

Example: GPA computation (conceptual)

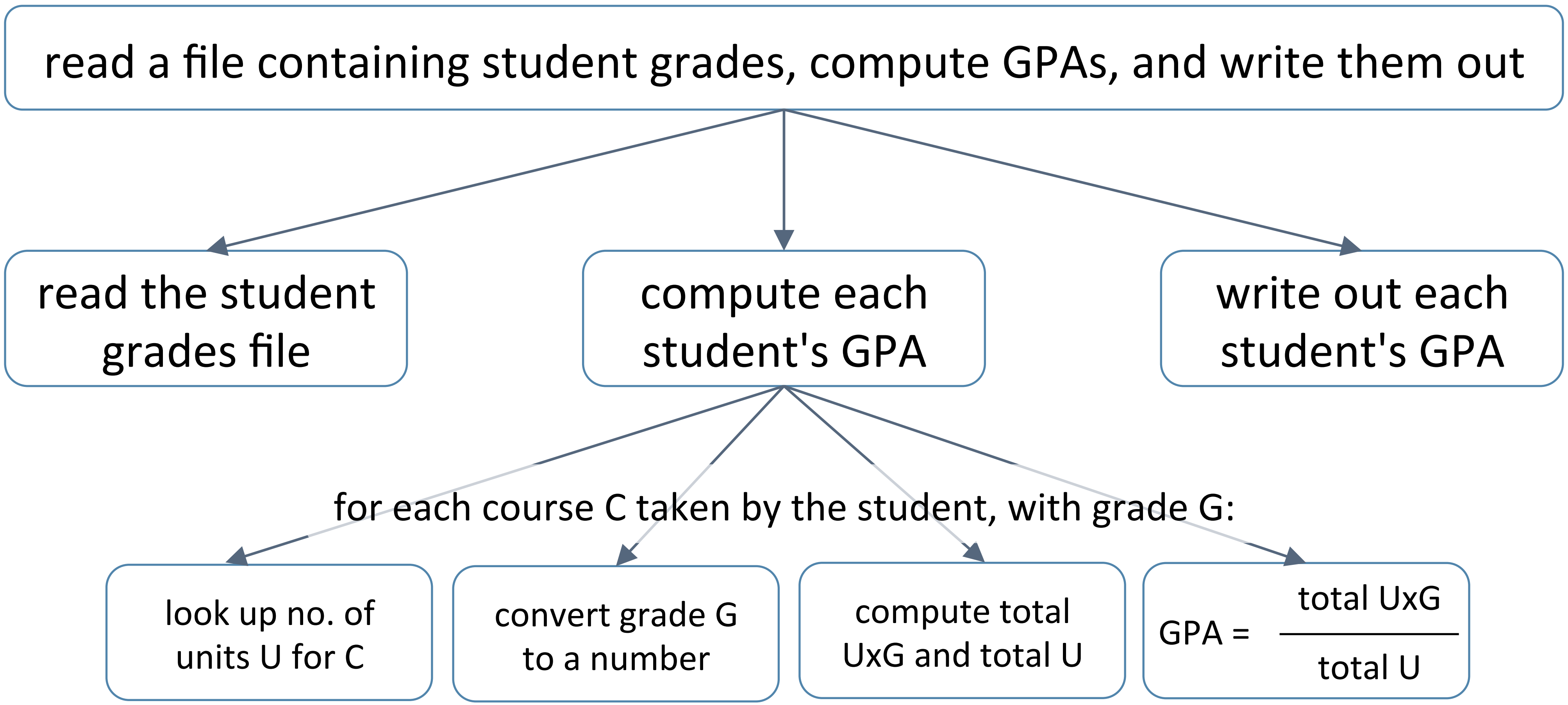
top-level task

read a file containing student grades, compute GPAs, and write them out

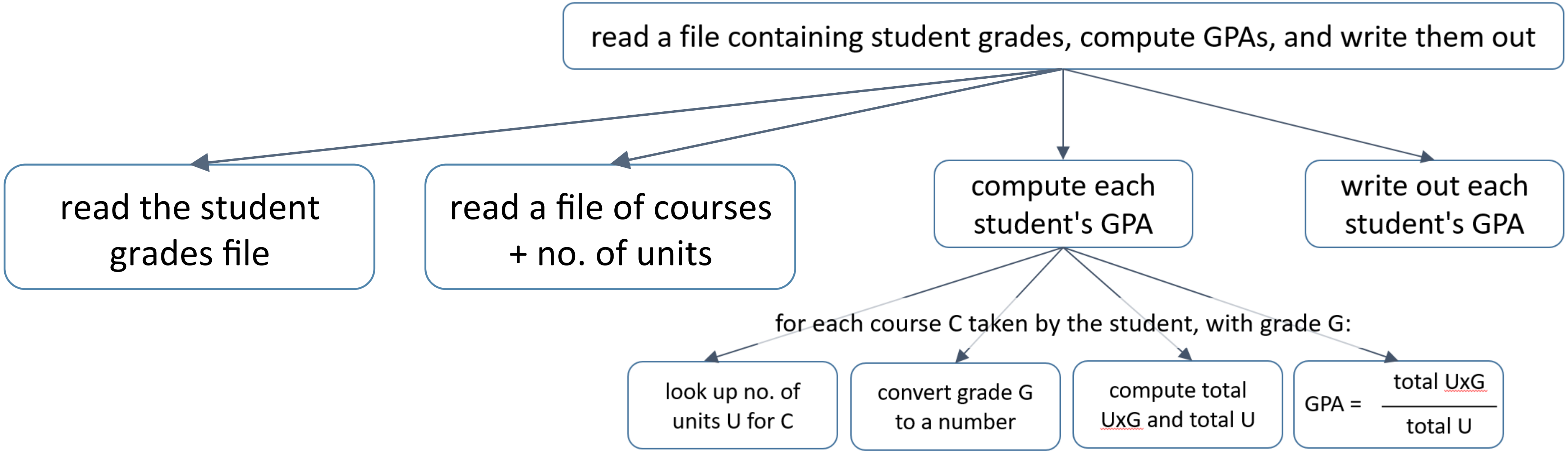
Example: GPA computation (conceptual)



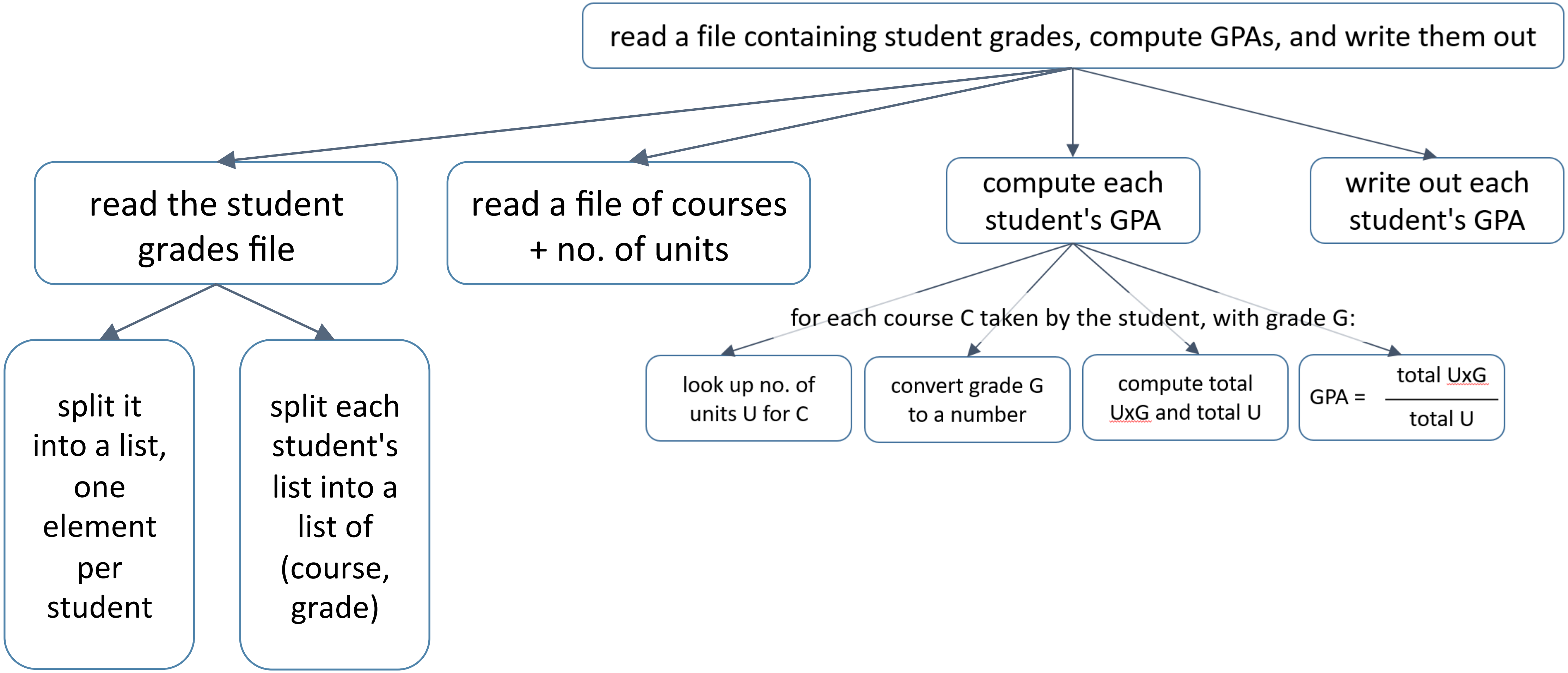
Example: GPA computation (conceptual)



Example: GPA computation (conceptual)

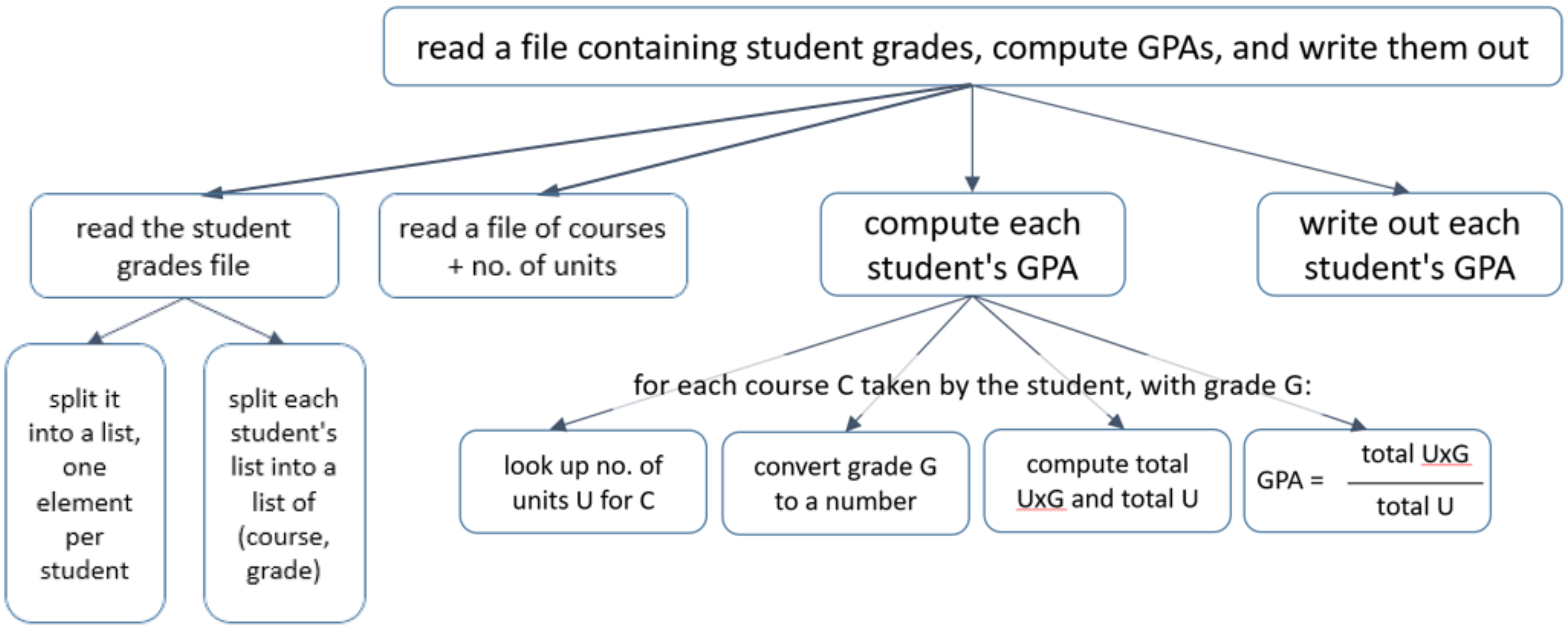


Example: GPA computation (conceptual)



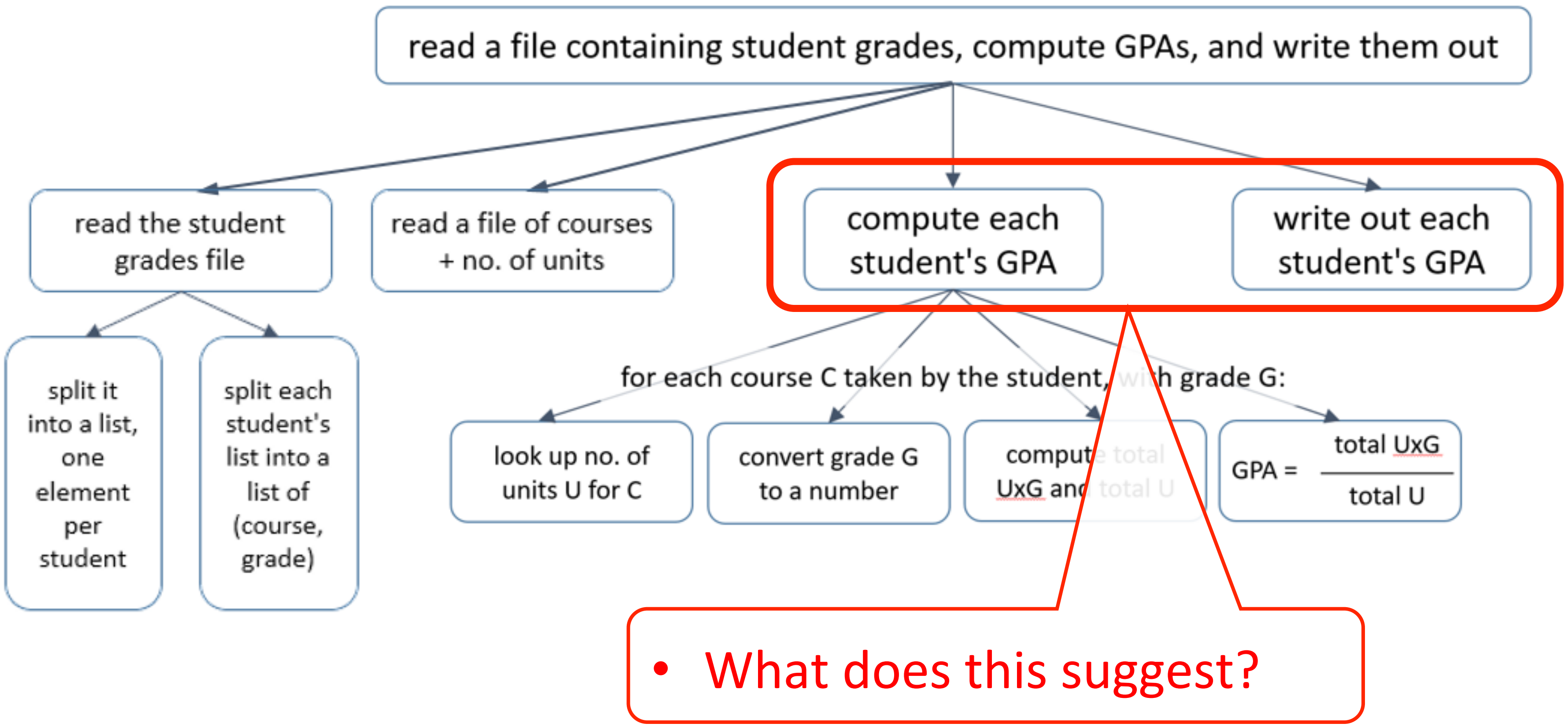
Example: GPA computation (conceptual)

As you decompose the problem, ask whether it is a "good" (simple, efficient) decomposition



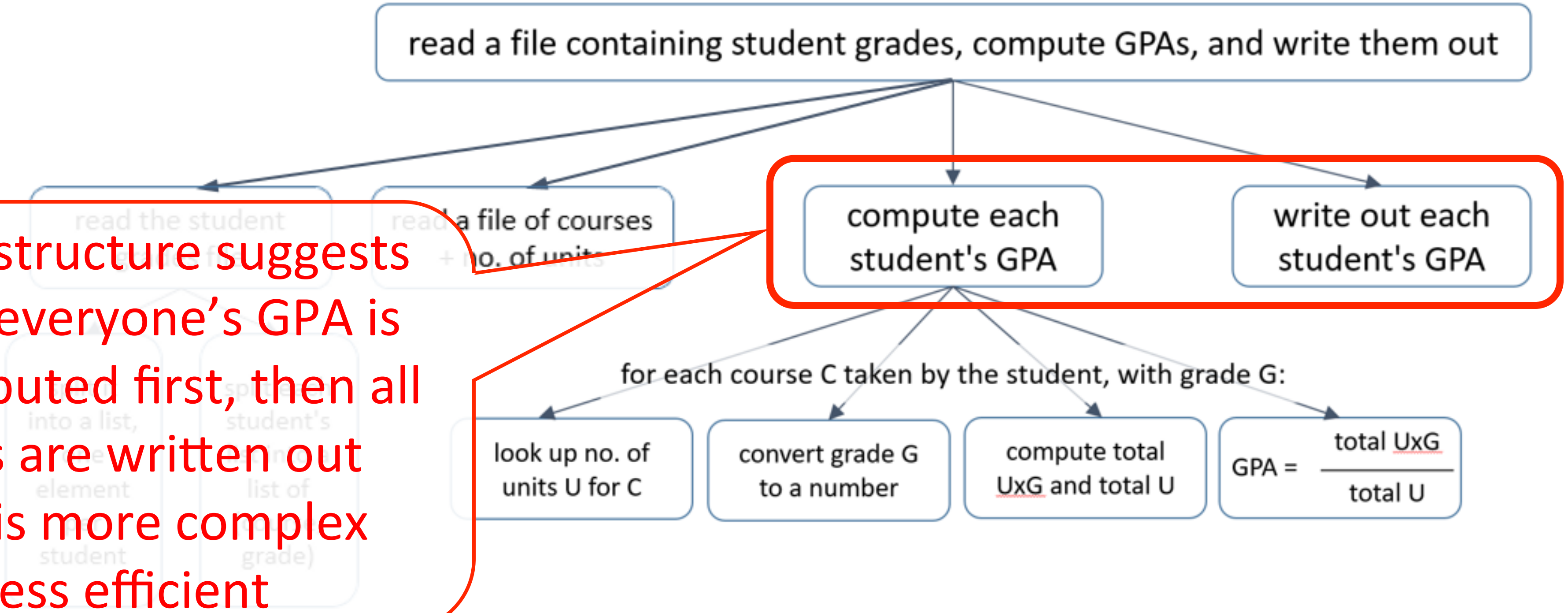
Example: GPA computation (conceptual)

As you decompose the problem, ask whether it is a “good” (simple, efficient) decomposition



Example: GPA computation (conceptual)

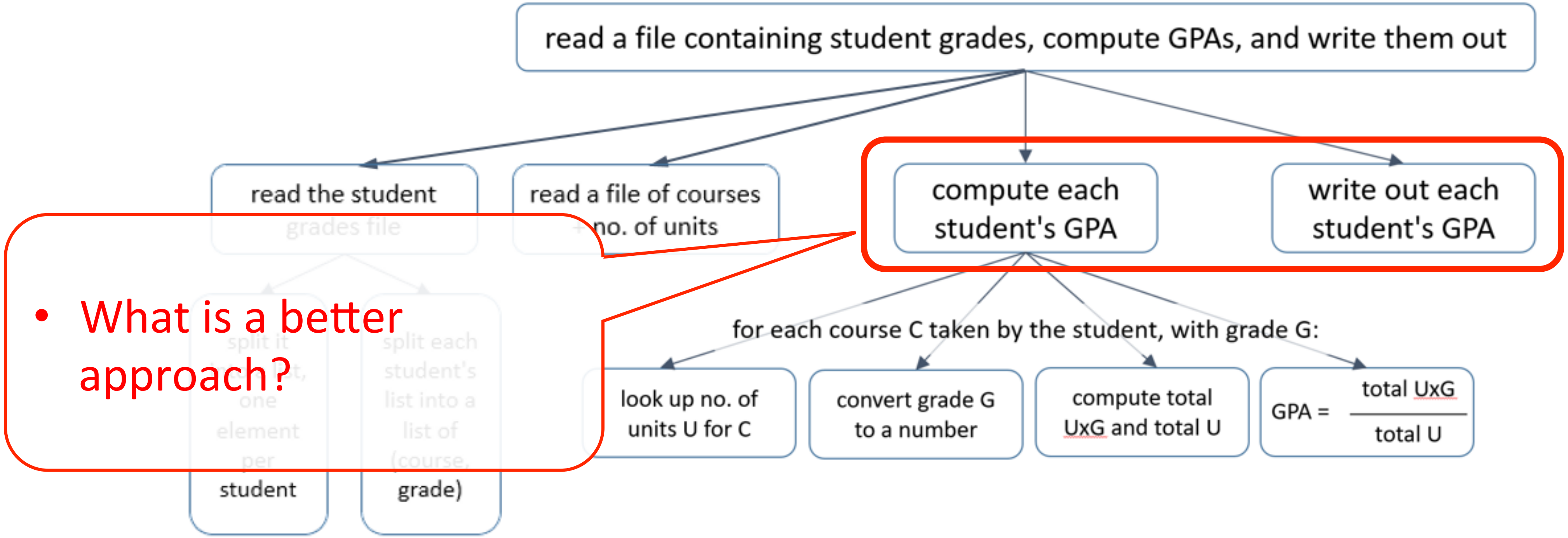
As you decompose the problem, ask whether it is a “good” (simple, efficient) decomposition



- This structure suggests that everyone’s GPA is computed first, then all GPAs are written out
- This is more complex and less efficient

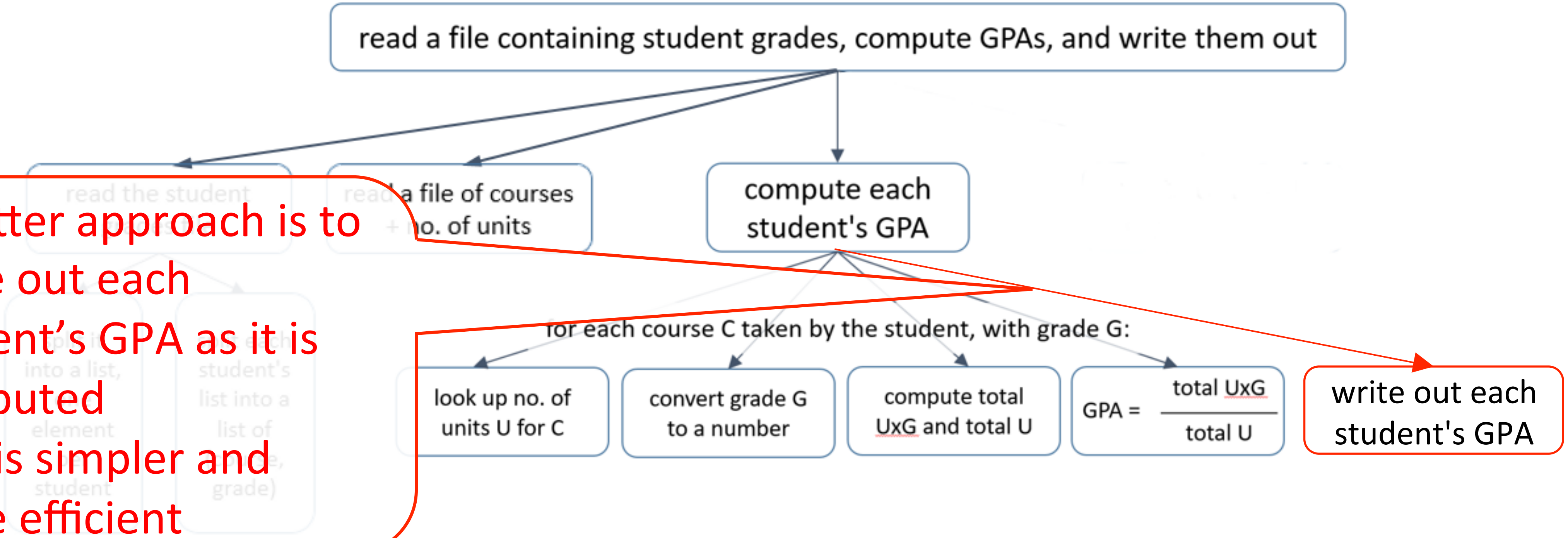
Example: GPA computation (conceptual)

As you decompose the problem, ask whether it is a “good” (simple, efficient) decomposition



Example: GPA computation (conceptual)

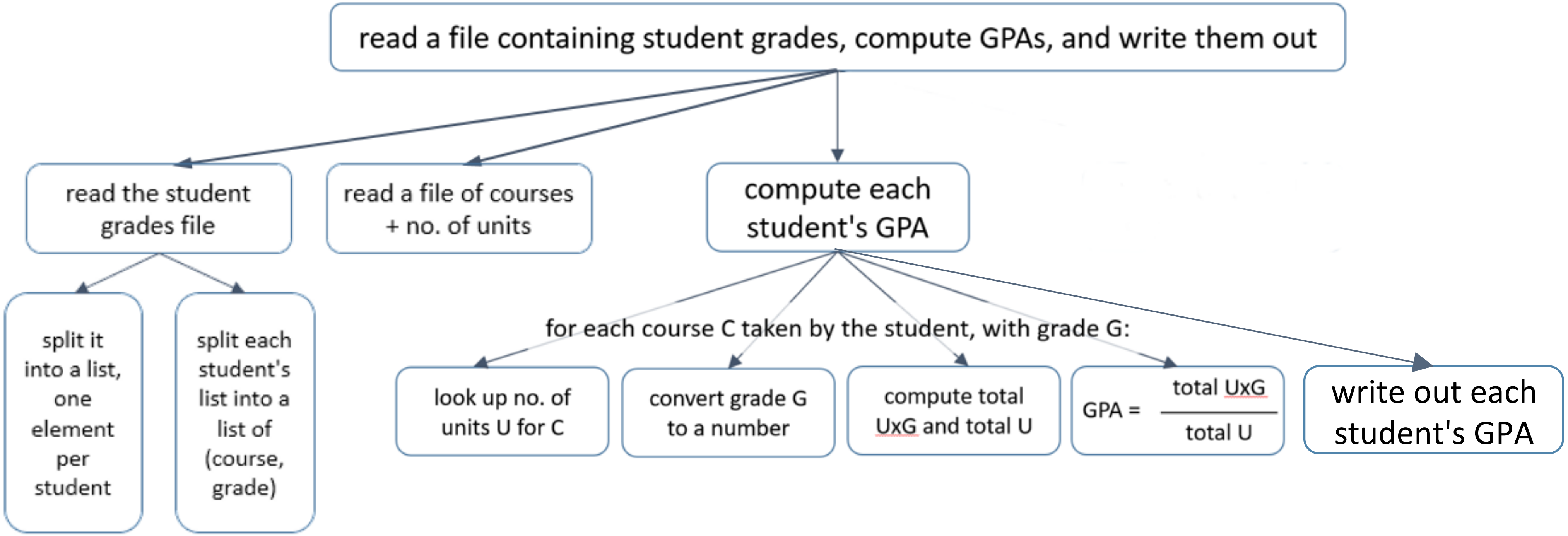
As you decompose the problem, ask whether it is a “good” (simple, efficient) decomposition



- A better approach is to write out each student’s GPA as it is computed
- This is simpler and more efficient

Example: GPA computation (conceptual)

As you decompose the problem, ask whether it is a “good” (simple, efficient) decomposition



Example: GPA computation (programming)

Conceptual decomposition

read a file containing student grades, compute GPAs, and write them out

pass : a placeholder statement

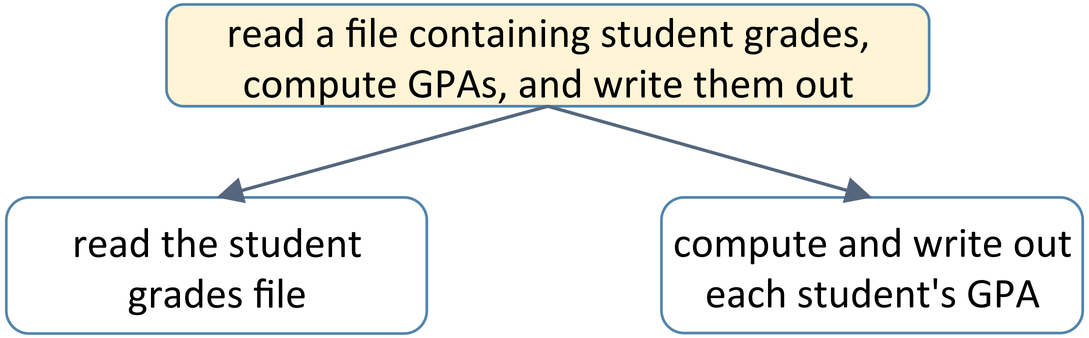
- does nothing
- useful for parts of the code that have not yet been fleshed out

Incremental Program Development

```
# main(): read student grades file, compute GPAs,
# write them out
def main():
    pass

main()
```

Example: GPA computation (programming)

Conceptual decomposition

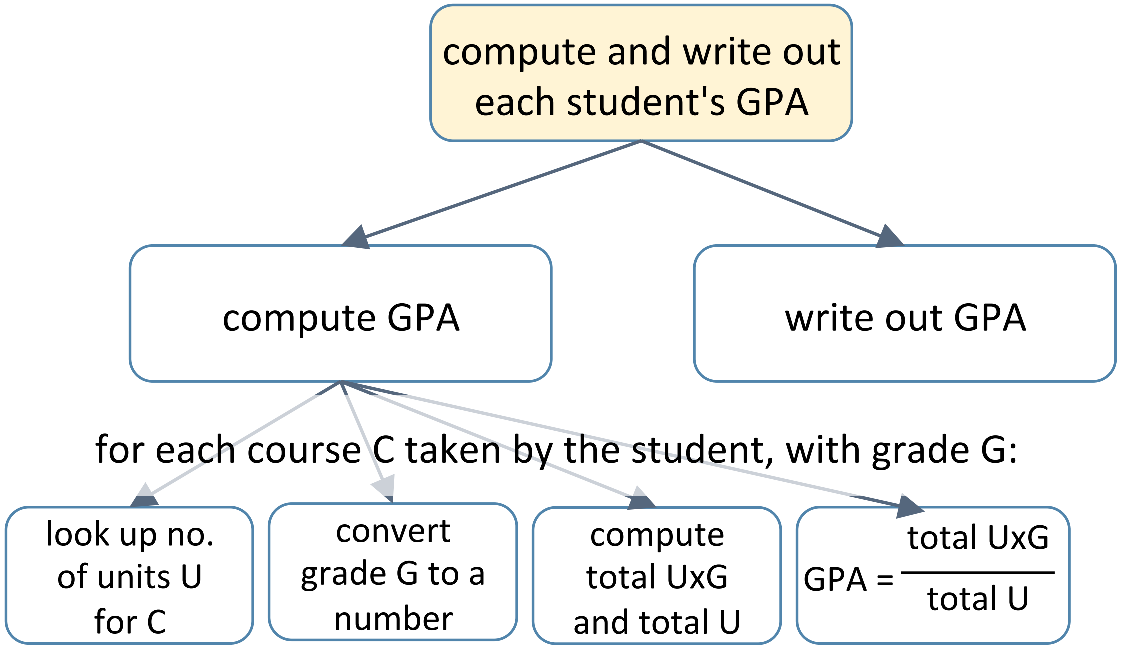


Incremental Program Development

```
# main(): read student grades file, compute GPAs,
# write them out
def main():
    grades = read_grades()
    compute_gpas(grades)

# read_grades() : read a grade file into a list of each
student’s grades
def read_grades():
    pass

# compute_gpas(grades) : compute and write out
the GPA for each student
def compute_gpas(grades):
    pass

main()
```

Example: GPA computation (programming)

Conceptual decomposition

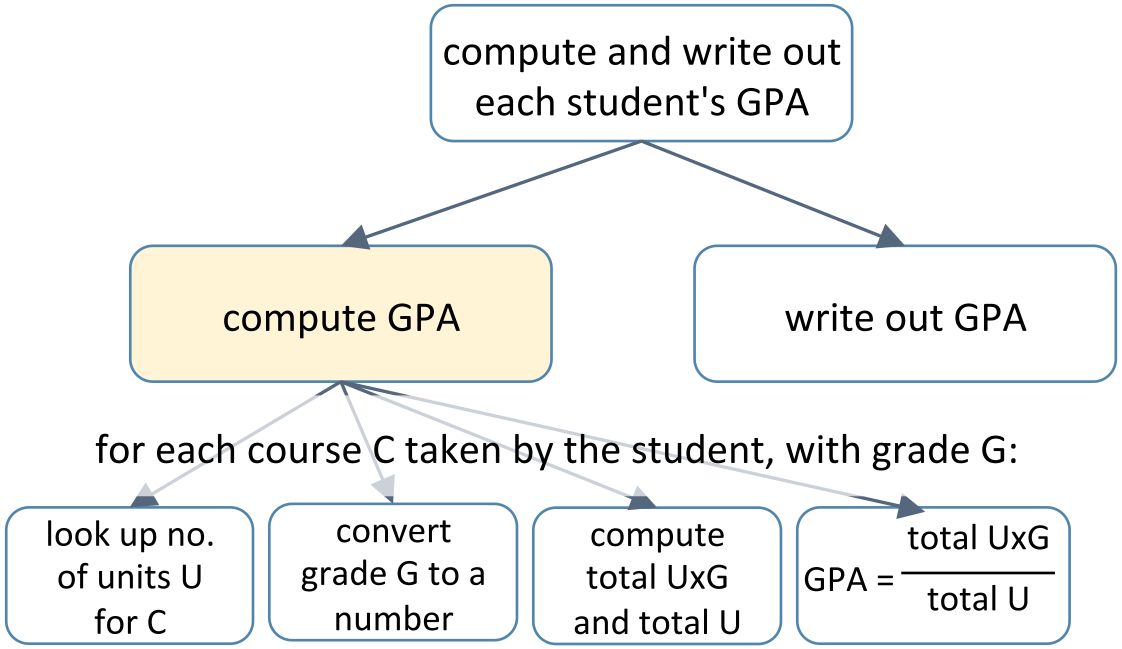


Incremental Program Development

```
# compute_gpas(grades) : compute and write out the GPA for each student
def compute_gpas(grades):
    for student_grades in grades:
        compute_student_gpa(student_grades)

# compute_student_gpa(student_data): compute
# and write out an individual student's GPA
def compute_student_gpa(student_grades):
    pass
```

Example: GPA computation (programming)

Conceptual decomposition

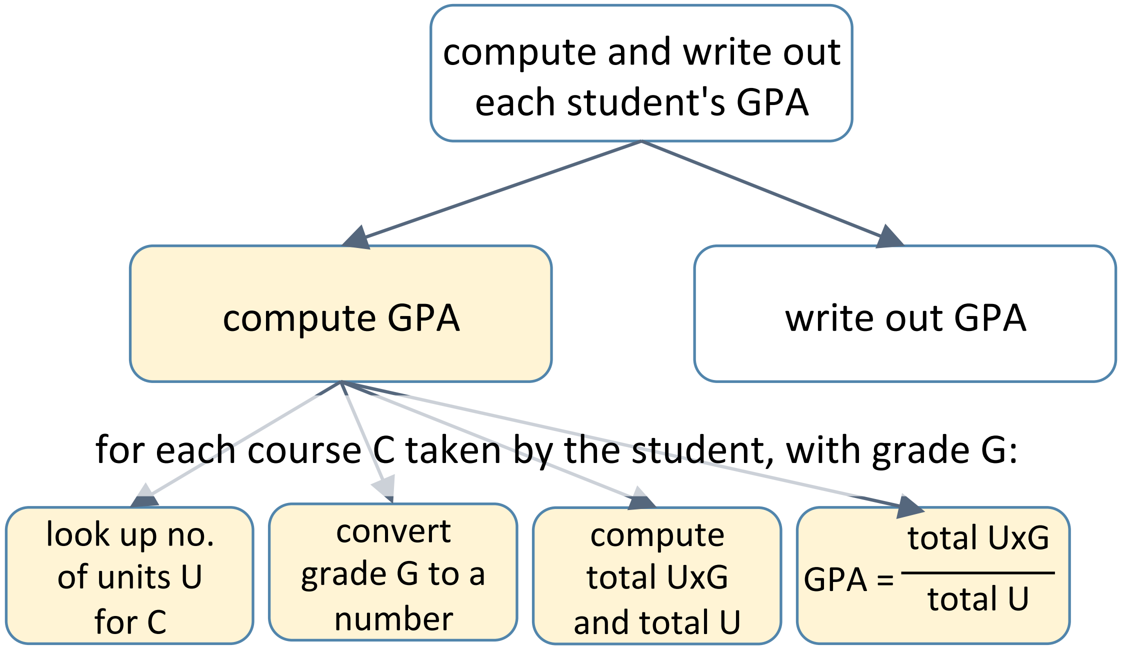


Incremental Program Development

```
# compute_student_gpa(student_data): compute
# and write out an individual student's GPA
def compute_student_gpa(student_grades):
    for [course,grade] in student_grades:
        # compute the gpa
        pass

    write_gpa()
```

Example: GPA computation (programming)

Conceptual decomposition

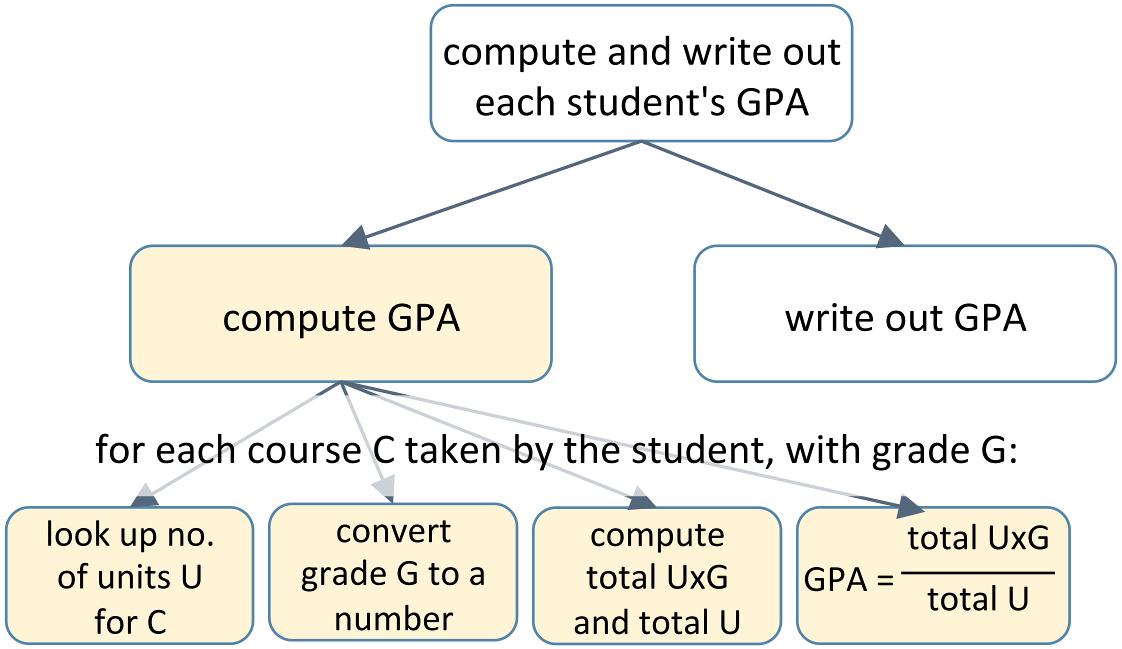


Incremental Program Development

```
# compute_student_gpa(student_data): compute
# and write out an individual student's GPA
def compute_student_gpa(student_grades):
    for [course,grade] in student_data:




def lookup_units(course):
    pass
...
```

Example: GPA computation (programming)

Conceptual decomposition

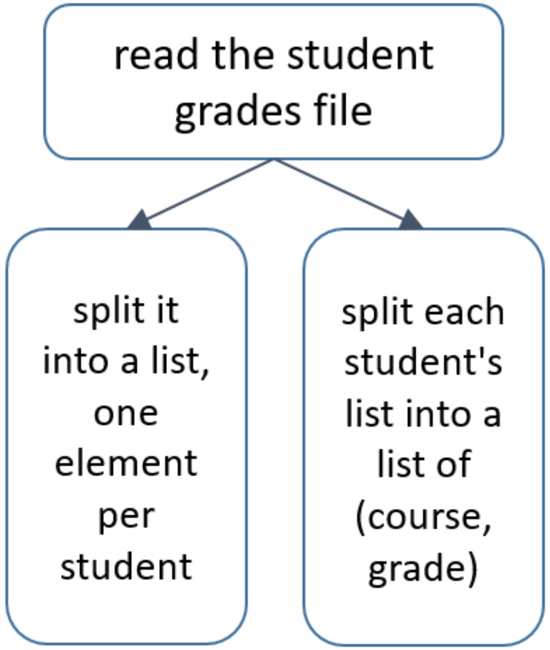


Incremental Program Development

```
# compute_student_gpa(student_data): compute
# and write out an individual student's GPA
def compute_student_gpa(student_grades):
    for [course,grade] in student_data:
        units = lookup_units(course)
        gval = grade_value(grade)
        weighted_gval += units * gval
        total_units += units

    gpa = weighted_gval / total_units
    student_name = lookup_name(student_grades)
    write_gpa(student_name, gpa)

def lookup_units(course):
    pass
...
```

EXERCISE

Conceptual decomposition



Incremental Program Development

?

Steps 2a+2b. Problem decomposition (summary)

- Begin:
  - identify the task(s) the program needs to do
  - define a stub function for each task

conceptual step
programming step

- while not done:
  - pick a task *A* and break it down into simpler tasks $A_1, \ldots, A_n$
  - flesh out the stub for *A* to execute the code for $A_1, \ldots, A_n$ (these may themselves be stubs)

Steps in writing a program

1. Understand what tasks the program needs to perform

2a. Figure out how to do those tasks

2b. Write the code

3. Make sure the program works correctly

Step 3. Ensuring correctness

- Goals:
  – the program produces the expected outputs for all (selected) inputs

- very often, this is the ***only*** thing that programmers check
- In general this is not enough
  – a program can produce the expected output "accidentally"

Passing test cases "accidentally"

- Problem spec:
  - *"Write a function grid_is_square(arglist) that returns True if arglist is a square grid, i.e., its no. of rows equals its no. of columns."*

- Submitted "solution":

```
def grid_is_square(arglist):
    return True
```

Passes half the test cases ...

... but is wrong!

Step 3. Ensuring correctness

- Goals:
  - the program produces the expected outputs for all (selected) inputs
  - each piece of the program behaves the way it's supposed to
  - each piece is used the way it's supposed to be used
    - any assumptions made by the code are satisfied
- Approach:
  - add *assertions* in the code to pinpoint problems
  - *test* the code to ensure that there are no problems

Invariants and assertions

- *Invariant*: an expression at a program point that *always* evaluates to True when execution reaches that point

- *Assertion*: a statement that some expression *E* is an invariant at some point in a program
  - Python syntax:

    `assert` *E*

    `assert` *E*, *"error message"*

EXERCISE

Write a function my_sqrt(n) that returns the square root of n. Use an assert statement to enforce that n must not be negative.

```
import math
def my_sqrt(n):
```

EXERCISE

Write a function my_sqrt(n) that returns the square root of n. Use an assert statement to enforce that n must not be negative.

```
import math
def my_sqrt(n):
    assert n >= 0, "negative argument to my_sqrt"
    return math.sqrt(n)
```

Example

```
# compute_student_gpa(student_grades): compute
# and write out an individual student's GPA
def compute_student_gpa(student_grades):
    weighted_gval = 0
    total_units = 0
    for [course,grade] in student_grades:
        units = lookup_units(course)
        gval = grade_value(grade)

        assert  units > 0 and gval >= 0, "data error"

        weighted_gval += units * gval
        total_units += units
    gpa = weighted_gval / total_units
    student_name = lookup_name(student_grades)
    write_gpa(student_name, gpa)
```

Example

```
# compute_student_gpa(student_grades): compute
# and write out an individual student's GPA
def compute_student_gpa(student_grades):
    weighted_gval = 0
    total_units = 0
    for [course,grade] in student_grades:
        units = lookup_units(course)
        gval = grade_value(grade)

        assert units > 0 and gval >= 0, "data error"

        weighted_gval += units * gval
        total_units += units

    gpa = weighted_gval / total_units
    student_name = lookup_name(student_grades)
    write_gpa(student_name, gpa)
```

lookup_units() returns the number of units for a course

- e.g., lookup_units('CSc 120') → 4

grade_value() returns the numerical value of a grade

- e.g., grade_value("C") → 2

Example

```
# compute_student_gpa(student_grades): compute
# and write out an individual student's GPA
def compute_student_gpa(student_grades):
    weighted_gval = 0
    total_units = 0
    for [course,grade] in student_grades:
        units = lookup_units(course)
        gval = grade_value(grade)

        assert  units > 0 and gval >= 0, "data error"

        weighted_gval += units * gval
        total_units += units

    gpa = weighted_gval / total_units
    student_name = lookup_name(student_grades)
    write_gpa(student_name, gpa)
```

this **`assert`** states that all courses must have nonzero units and that a grade value cannot be negative

- *guards against data entry errors*

Example

```
# compute_student_gpa(student_grades): compute
# and write out an individual student's GPA
def compute_student_gpa(student_grades):
    weighted_gval = 0
    total_units = 0
    for [course,grade] in student_grades:
        units = lookup_units(course)
        gval = grade_value(grade)

        assert  units > 0 and gval >= 0, "data error"

        weighted_gval += units * gval
        total_units += units
    gpa = weighted_gval / total_units
    student_name = lookup_name(student_grades)
    write_gpa(student_name, gpa)
```

this **`assert`** states that all courses must have nonzero units and that a grade value cannot be negative

- *guards against data entry errors*

- *It's better to catch errors early*
- *It's better to catch bad values close to where they are computed*

So it would be to better to push these asserts into the functions that compute these values

Example

```
# lookup_units(course, course_units) : looks up the
# no. of units for a course
def lookup_units(course, course_units):
    for crs, units in course_units:
        if course == crs:
            assert units > 0, “lookup_units: grade error”
            return units

    assert False, “lookup_units: course not found”
```

```
# grade_value(grade) : returns the numerical value
# for a letter grade
def grade_value(grade):
    if grade == ‘A’ :
        return 4
    elif grade == ‘B’:
        return 3
    elif grade == ‘C’:
        return 2
    elif grade == ‘D’:
        return 1
    elif grade == ‘E’:
        return 0
    else:
        assert False, “grade_value: unknown grade”
```

Using asserts

- checking arguments to functions
  - e.g., if an argument's value has to be positive
- checking data structure invariants
  - e.g., i >= 0 and i < len(name)
- checking "can't happen" situations
  - this also serves as documentation that the situation can't happen
- after calling a function, to make sure its return value is reasonable

Steps in writing a program: summary

- Understand what the program needs to do before you start coding
- Develop the program logic incrementally
  – top-down problem decomposition
  – incremental program development
    - use stubs for as-yet-undeveloped parts of the program
- Program defensively
  – figure out invariants that must hold in the program
  – use `assert`s to express invariants in the code