# CSc 120
## Introduction to Computer Programming II

Problem Decomposition and Program Development

# Your first consulting project!

Client:

"I want a program to compute student GPAs from their grades."

You:

"I'll write it tonight and be back tomorrow!"

Client:

"Great!"

# The next day...

You're back at 8am sharp and ready to show them their new program!

| | |
|---|---|
| They ask, | "What format will the file of students be in?" |
| You say, | "I thought it was for one student at a time." |
| | |
| They ask, | "How do I specify the number of units for a course?" |
| You say, | "Aren't all courses worth three units?" |
| | |
| They ask, | "How are pass/fail grades handled?" |
| You say, | "Uh..." |

# Lots of software development methods!

There are lots of software development methods!  A few:

- Waterfall
- Extreme Programming (XP)
- Test Driven Development
- Feature Driven Development
- SCRUM
  - https://www.scrumalliance.org/learn-about-scrum
- Rational Unified Process (RUP)

Some say "methodology" to mean a single method.

The process shown in the following slides is a *top-down* method, with a somewhat of a *waterfall*-ish flavor.

# Let's start again!

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

# 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 the program 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?
- Work with "the customer" to resolve those questions.
  - Beware: Customers often don't know what they really want/need!

- What's the most common reason for a software system ultimately being considered to be a failure?
  - The system didn't meet the user's needs!

# Example: cont'd

Problem statement:

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

- Exercise: What are the questions to clarify the input?
  - read from a file, or from the keyboard?
  - what is the format?
  - how many students?
  - do we accept queries for getting the GPAs?

# Example: cont'd

## Problem statement:

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

- What are the questions to clarify the output:
  - to a file, or to the screen?
  - what is the format?
  - compute GPA for all students, or only specific students?
  - ...more...

# Example: cont'd

## Problem statement:

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

- What are the questions to clarify the computation?
  - how is a GPA computed?
    - How are the grades represented in input (A, B, C, etc. or numerical)
    - Are there pass/fall grades?
    - Do classes have different units?

# 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?
  - users, product manager, domain experts and others often involved

# 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 line per student:*

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

    *all students in the input file*

# Example: cont'd

## Problem statement:

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

- Computation:
  - what grades are expected?

    *A, B, C, D, E  which map to 4, 3, 2, 1, 0*
  - do courses have different units?

    *yes*

    *a grade is weighted by the number of units (weighted average)*

# 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 # of units for each course
- translate letter grades to numbers (e.g., A = 4, B = 3, …)

# Example: cont'd (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

What is the GPA computation?

# Example: cont'd (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

GPA = (Total UxG) / (Total U) = 26/8 = 3.25

# Reality: Specifications change

Academic programming assignments rarely have a significant change in the specifications.

Elsewhere it is a simple fact of software development that specifications are extremely likely to change during the course of a project.

What are some reasons for change?

- What the customer thought would be great isn't.
- The customer's understanding of their needs was incomplete.
- A competitor comes out with features the customer wants to match or exceed.
- Business rules change.

*Agile software development* is an approach that recognizes the likelihood of changes in specifications and provides ways to minimum their impact.

# 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
  - You need a roadmap

repeat as needed

- 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 $A$

<u>Before you start writing code to solve a problem, make sure you know how to solve the problem yourself.</u>

# 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

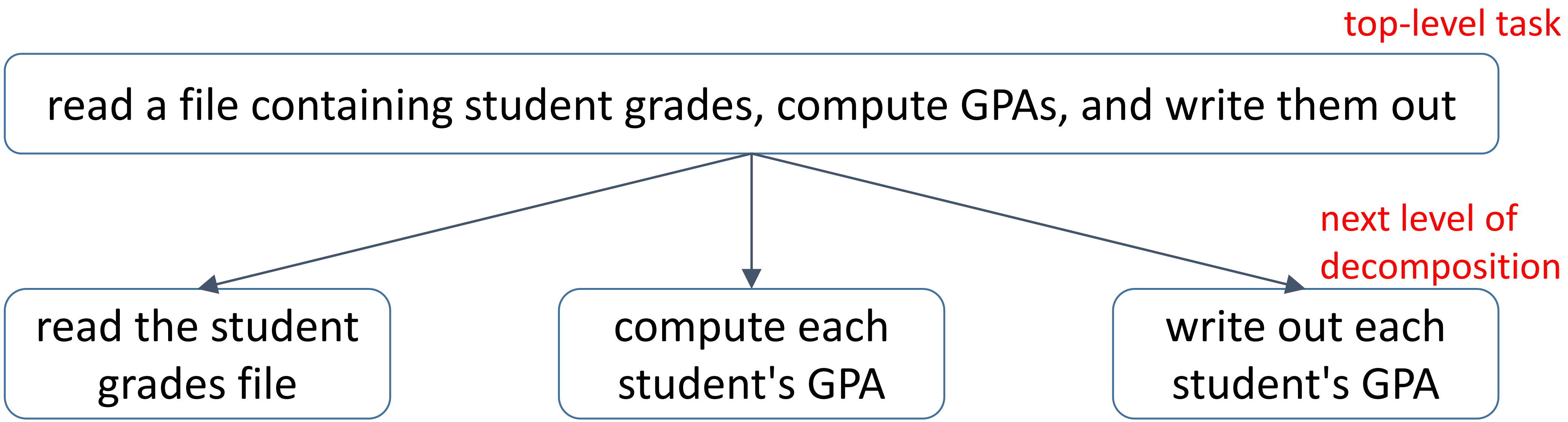
- Decomposing a task into sub-tasks $\Rightarrow$ fleshing out the code for a stub
  - repeat until no more stubs to flesh out

# Example: GPA computation (conceptual)

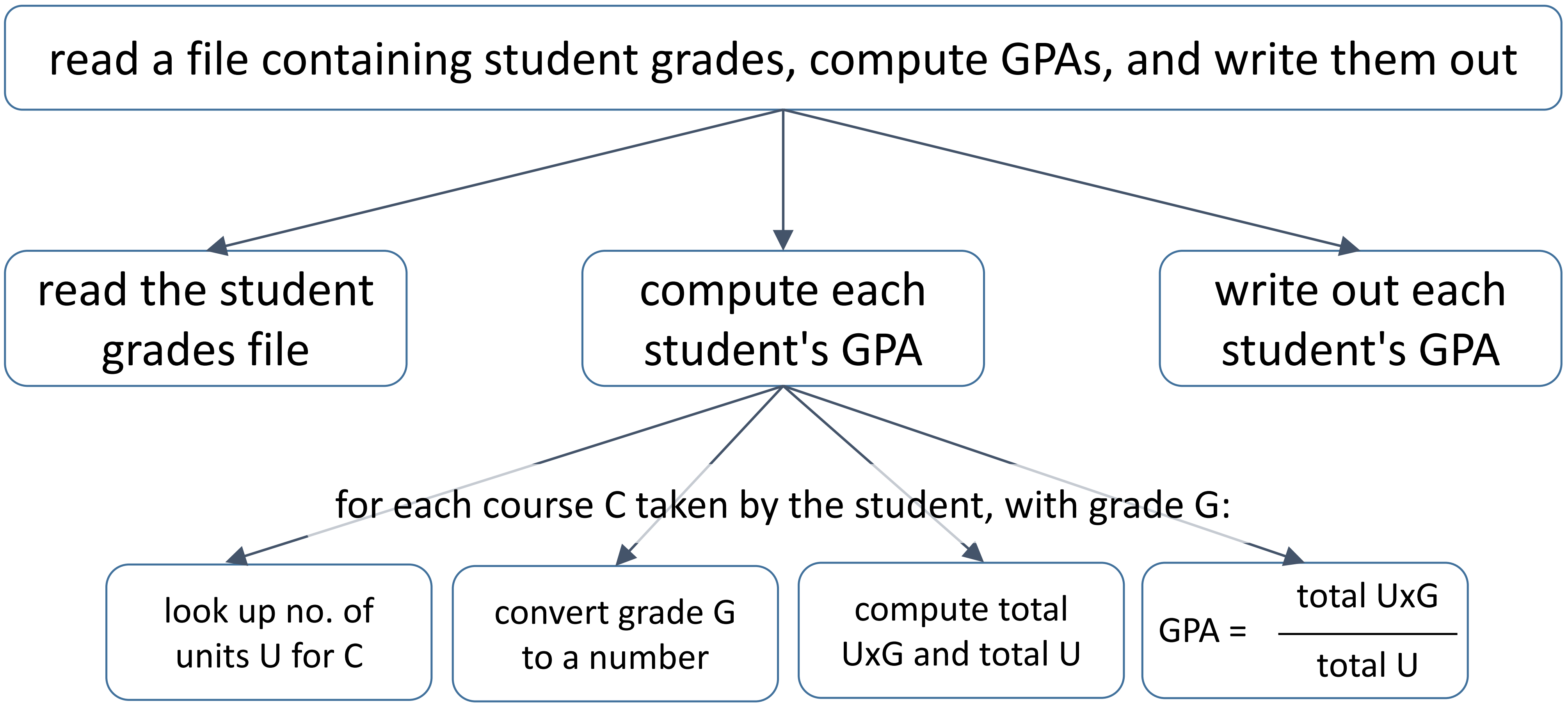
top-level task

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

# Example: GPA computation (conceptual)



# Example: GPA computation (conceptual)



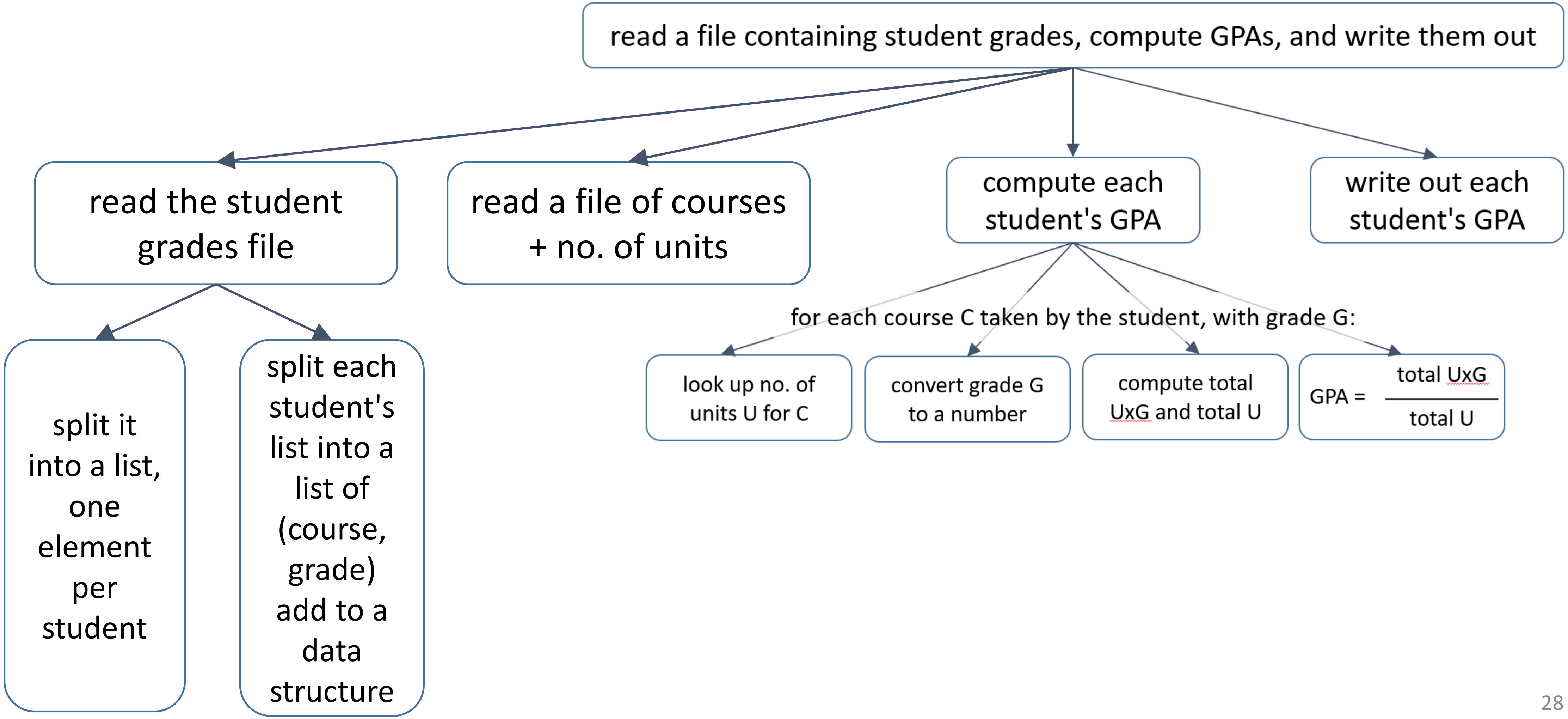
# Example: GPA computation (conceptual)

- read a file containing student grades, compute GPAs, and write them out
  - read the student grades file
  - read a file of courses + no. of units
  - compute each student's GPA
    - for each course C taken by the student, with grade G:
      - look up no. of units U for C
      - convert grade G to a number
      - compute total UxG and total U
      - GPA = $\frac{\text{total UxG}}{\text{total U}}$
  - write out each student's GPA

**Exercise!**

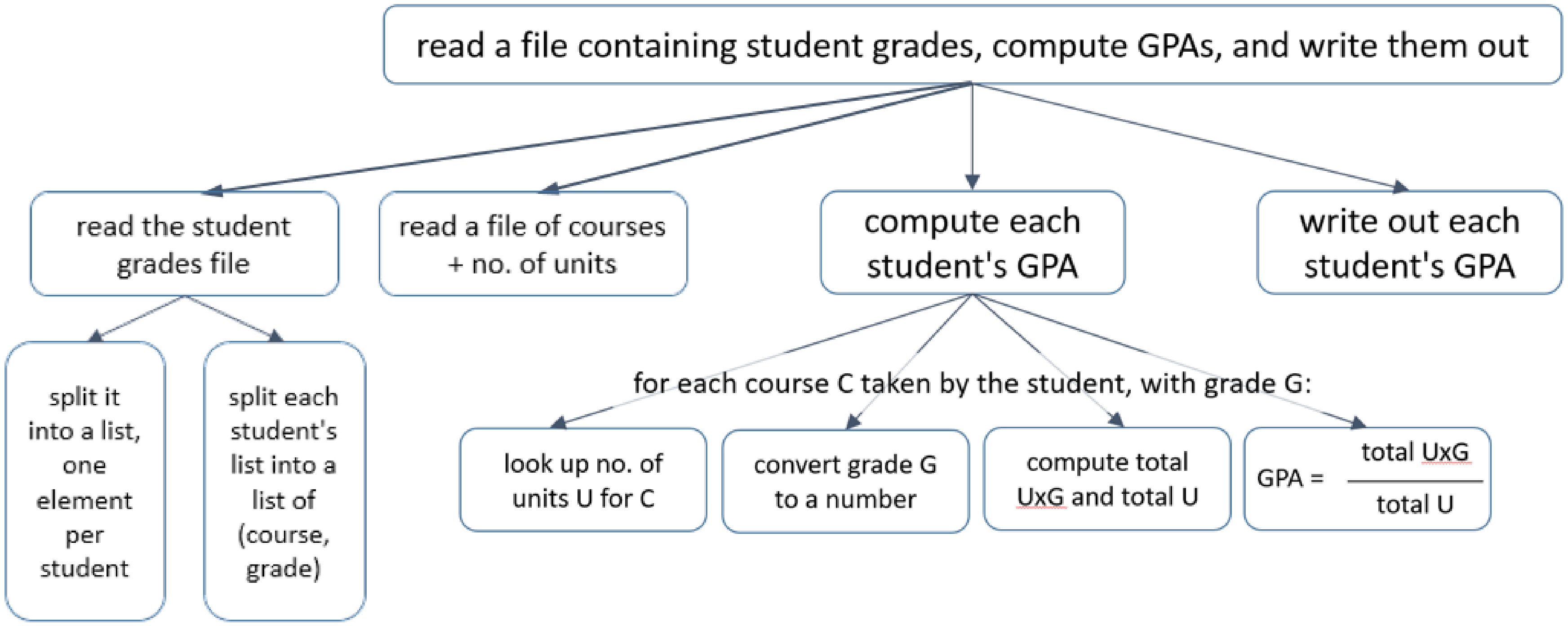
**(1 in handout)**

# Example: GPA computation (conceptual)



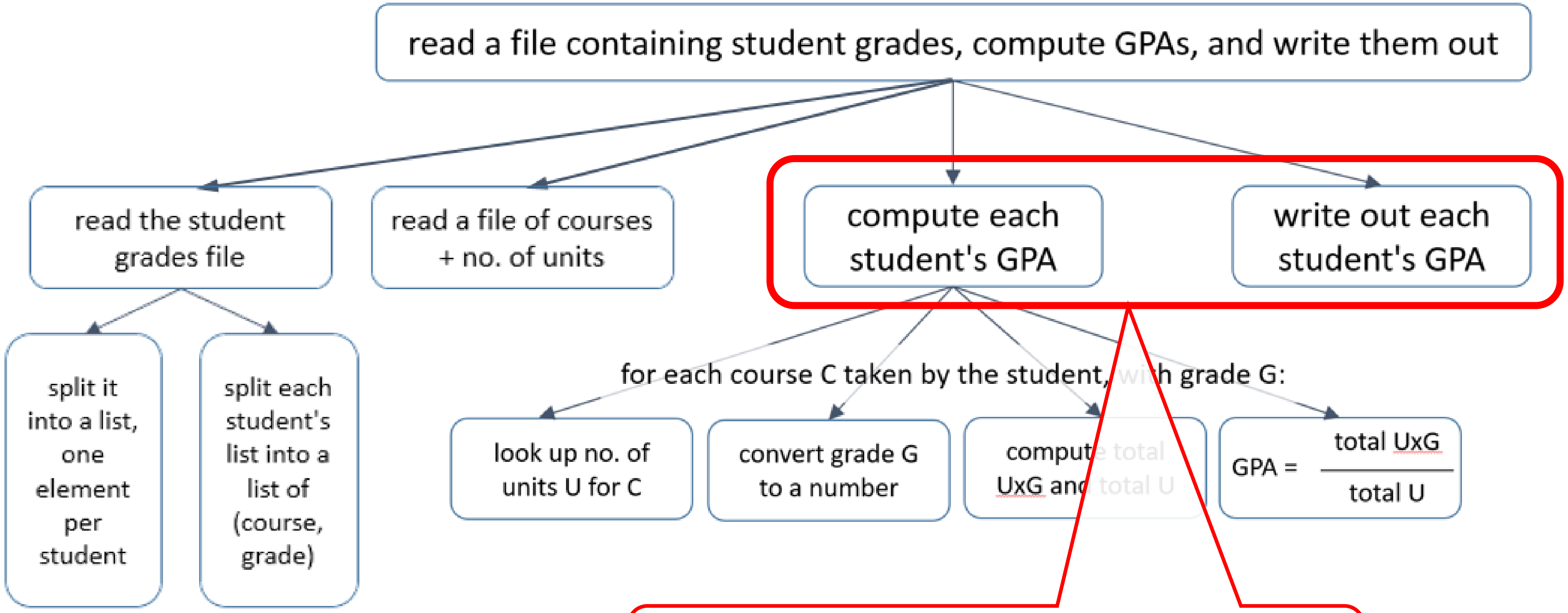
# Example: GPA computation (conceptual)

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



# Example: GPA computation (conceptual)

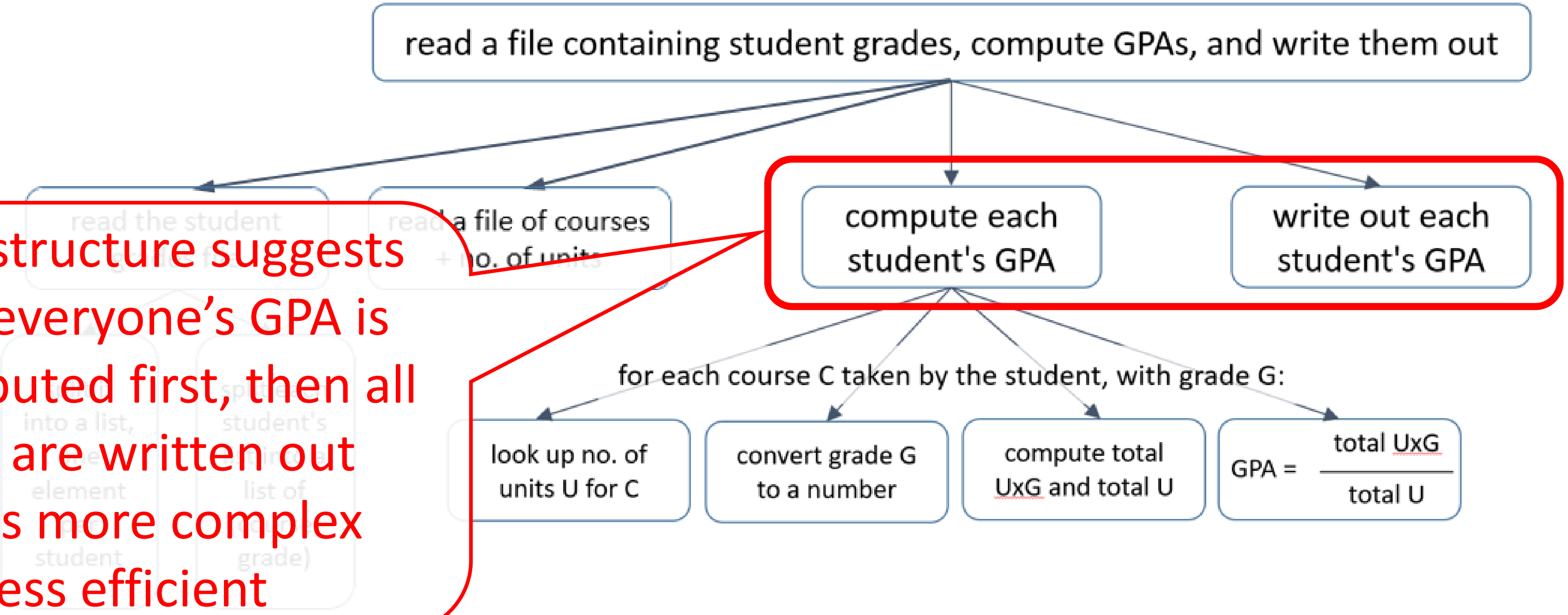
As you decompose the problem, ask whether it is a "good" (simple, efficient) decomposition




- What does this suggest?

# Example: GPA computation (conceptual)

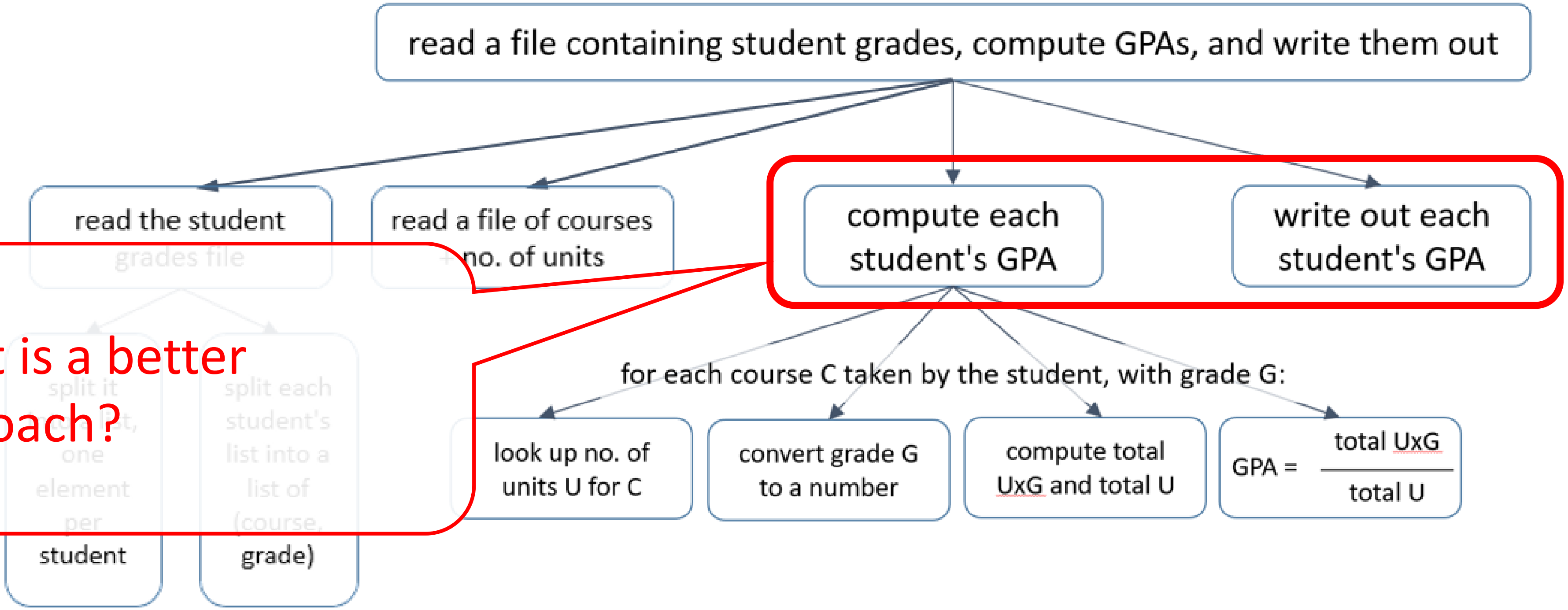
As you decompose the problem, ask whether it is a “good” (simple, efficient) decomposition




- 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)

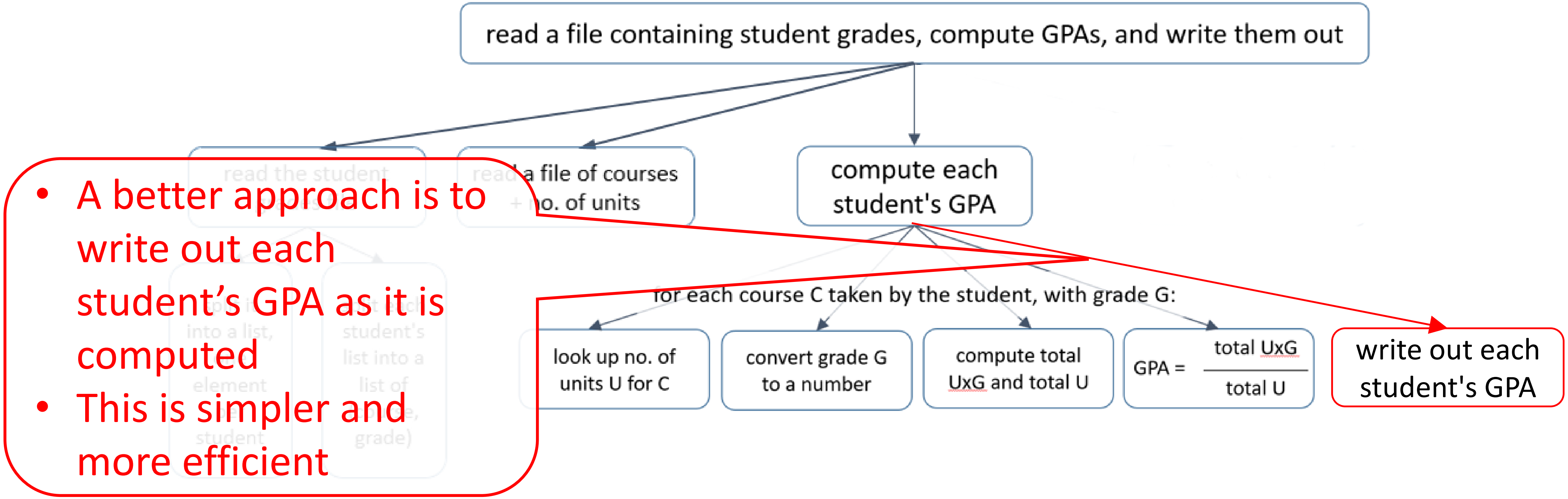
As you decompose the problem, ask whether it is a “good” (simple, efficient) decomposition




- What is a better approach?

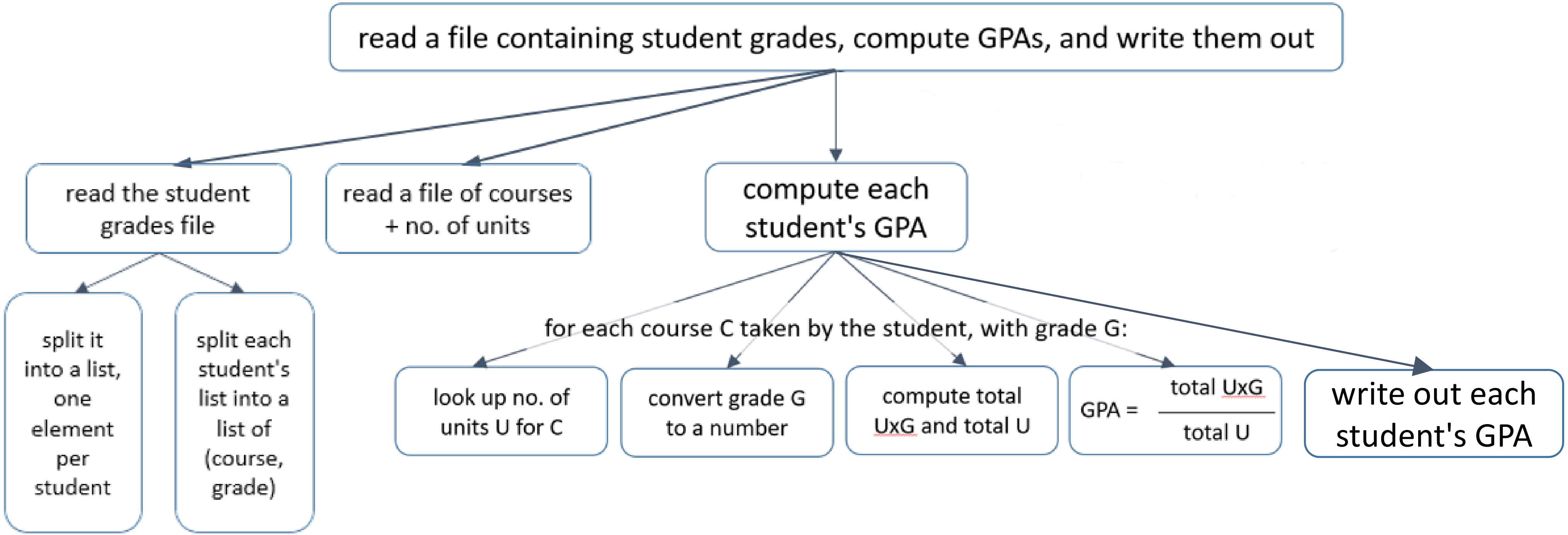
# Example: GPA computation (conceptual)

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 (programming)

**Conceptual decomposition**

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

**pass** : a placeholder statement

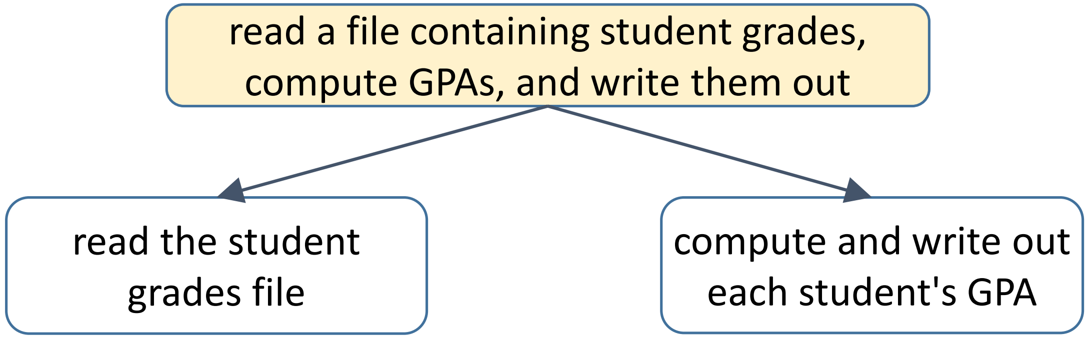
- does nothing
- useful for parts of the code that have not yet been fleshed out

**Incremental Program Development**

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

main()
```

# Example: GPA computation (programming)

## Conceptual decomposition

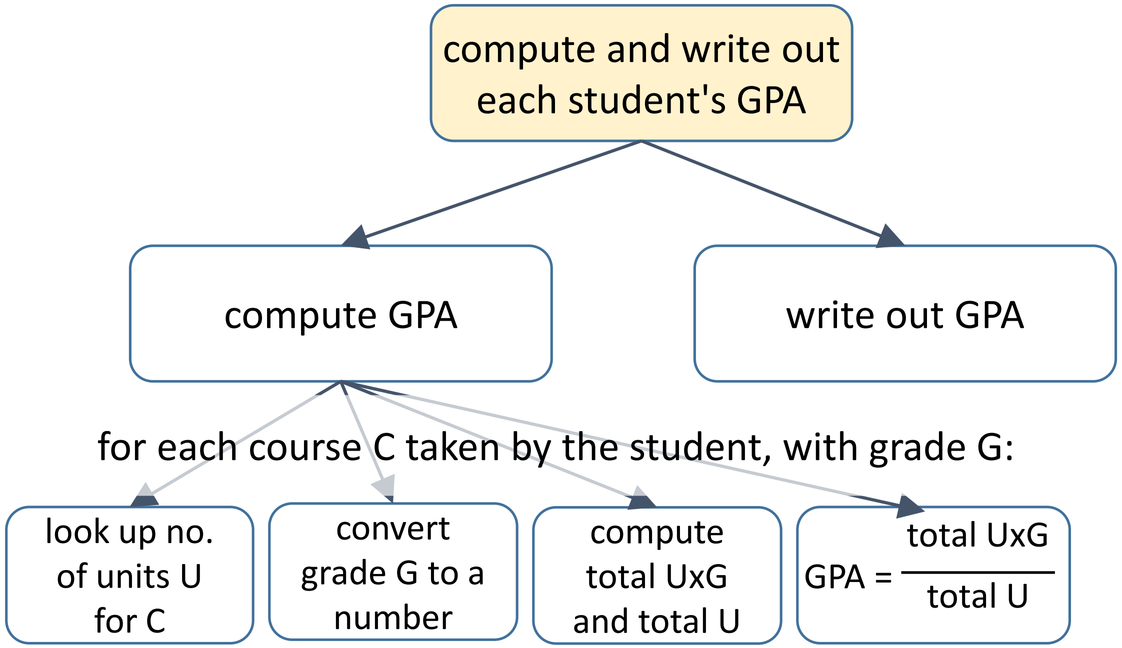



## Incremental Program Development

```
# 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

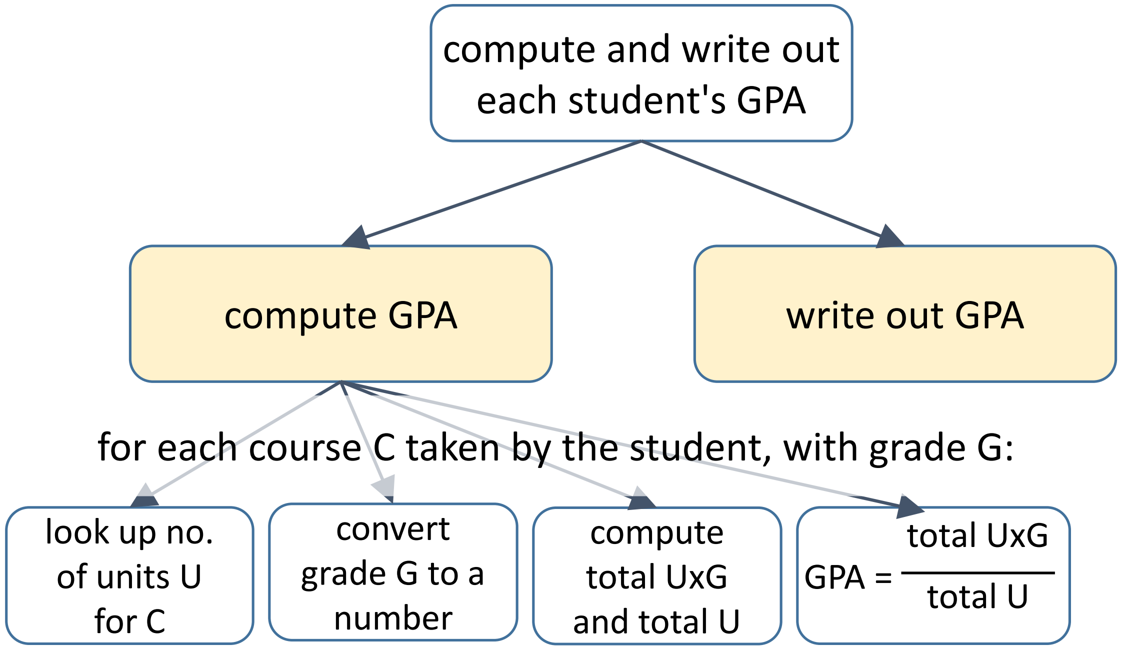



## Incremental Program Development

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

# Example: GPA computation (programming)

**Conceptual decomposition**

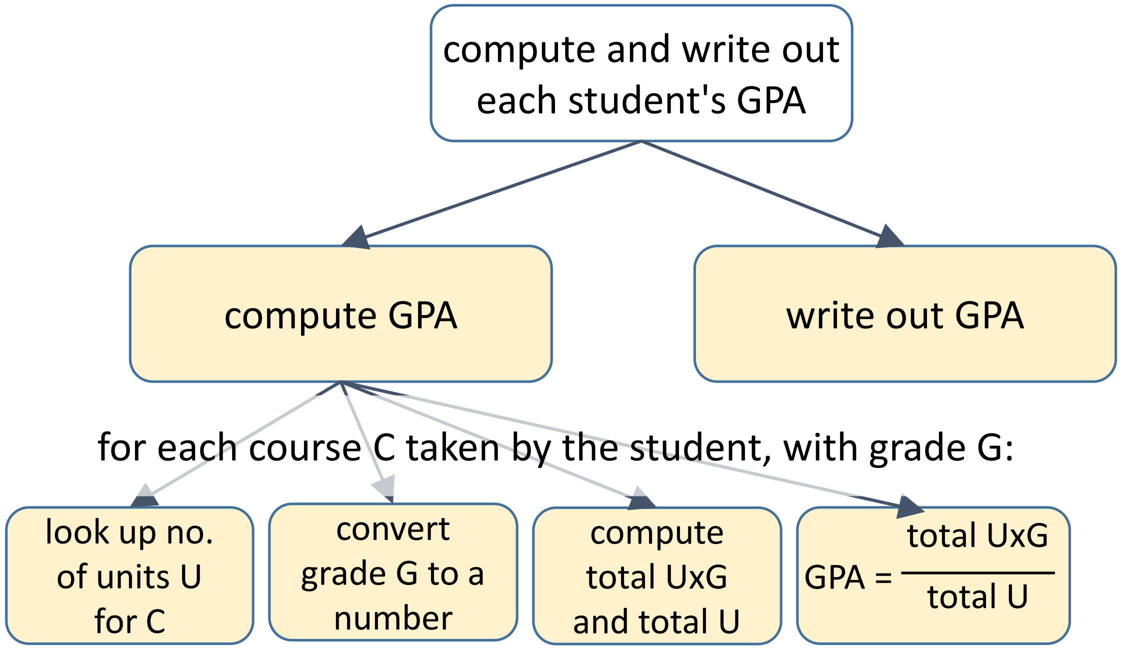



**Incremental Program Development**

```
# compute_student_gpa(student_grades): 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

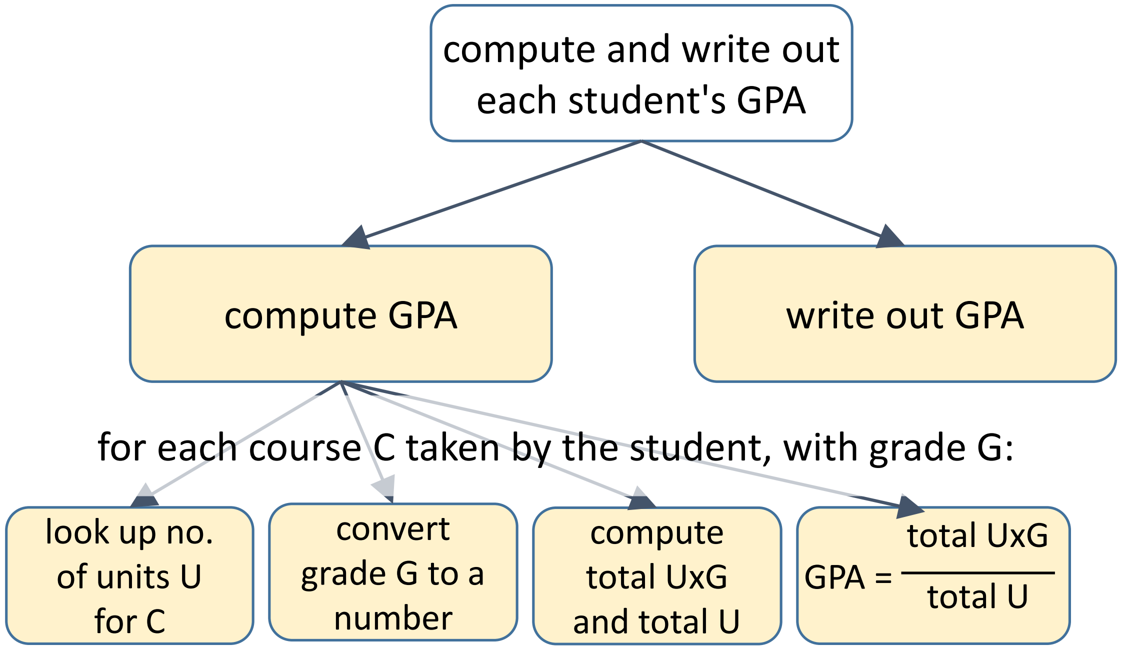



## Incremental Program Development

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



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

# Example: GPA computation (programming)

## Conceptual decomposition




## Incremental Program Development

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

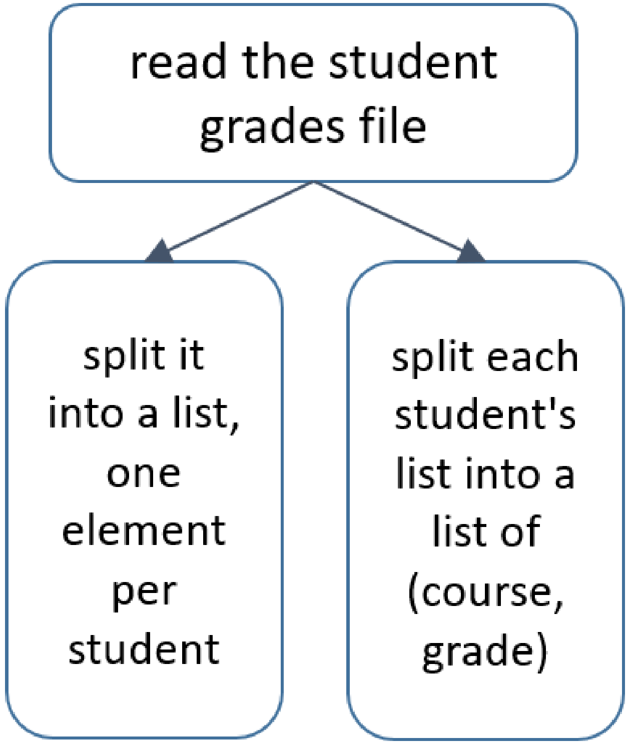
# Exercise (2. in handout)

WWYN and see if you can go from the Conceptual Decomposition on the left to code on the right. Don't go as far as code that builds lists or tuples. (Posted solution has four functions.)

A line from grades.csv: (in NOTES)

Jan Verisof, CSC 110:B, CSC 120:B, CSC 245:A, CSC 337:B

**Conceptual decomposition**

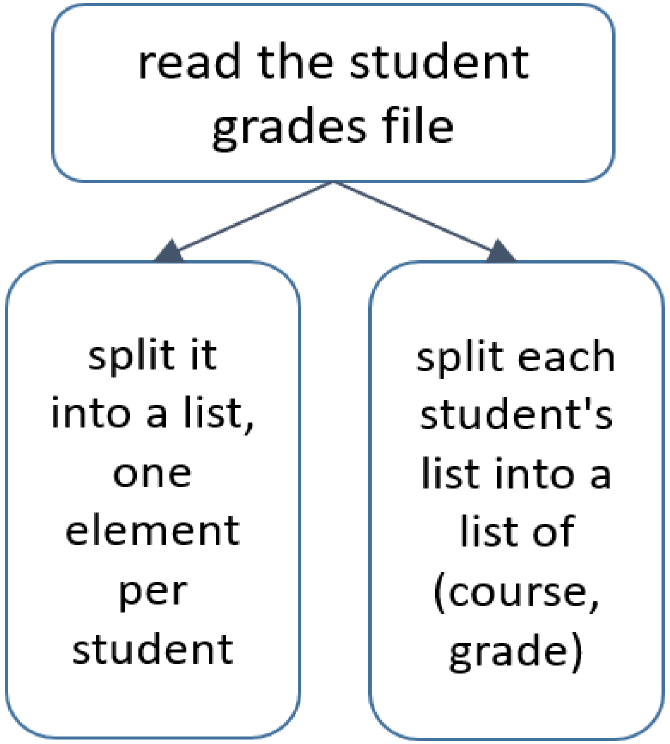



**Incremental Program Development**

```
# read a grade file and return a list of students with grades
def read_grades():
    pass

def ...


def ...


def ...
```

# Solution

Conceptual decomposition




```
def read_grades():
    """read grades.csv and return a list of students with grades"""
    lines = get_lines("grades.csv")
    student_list = parse_lines(lines)
    return student_list

def get_lines(fname):
    """read fname and return its contents as a list of lines"""
    pass

def parse_lines(lines):
    students = []
    for line in lines:
        students.append(build_student(line))
    return students

def build_student(line):
    """Given "Pirenne Hardin, CSC 110:A, CSC 120:B"
        return ('Pirenne Hardin', [('CSC 110', 'A'), ('CSC 120', 'B')]) """
    return (...name..., ...list of course/grade lists...)
```

See [https://www2.cs.arizona.edu/classes/cs120/spring19/NOTES/grades.py](https://www2.cs.arizona.edu/classes/cs120/spring19/NOTES/grades.py) for full code

# 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 or complete code)

# Steps 2a+2b. Problem decomposition (note)

Roadmap promotes clarity:

- Can prevent getting committed to a lesser, complicated design

Test and verify in small chunks:

- Look for small, testable pieces to build depth-first
- (e.g., reading the student grades file can be tested fully)

# Analogy: An outline for a paper

- Top-down design is similar to developing an outline for a paper.
- Instead of bubbles and arrows, we could just use text:

```
Compute student GPAs
    Read the grades file
         Split into list, one element per student
         Form a student tuple from each list element
    Read file with courses and units
    Compute each student's GPA/write it out
         for each course
              Look up units
              Convert grade to number
              ...
```

# Let's discuss!

- What do you like about top-down design?
- What's not so great about top-down design?

Work together and see if you can come up with three of each

# 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 *<u>only</u>* 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"*

# Invariants and assertions

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

    **`assert`** *E*

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

- assert:
  - E evaluates to True or False
  - If E evaluates to True, program execution continues
  - otherwise, the error message is printed and execution halts with an AssertionError

# Invariants and assertions

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

- Python syntax:

  **`assert`** *E*

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

Example:

```
def  sum_evens(nums):
    assert len(nums) > 0, ”nums is empty”
    ...
```

# EXERCISE

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

```
import math
def my_sqrt(n):
    return math.sqrt(n)
```

# Solution

*The function my_sqrt(n)  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)
```

# EXERCISE

*Do exercise 3. on assert statements in the handout.*

# 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)
      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)
        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

*What can we assert about units and gval?*

# 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)
        assert  units > 0, "data error"

        gval = grade_value(grade)
        assert 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

this **`assert`** states 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)
        assert  units > 0, "data error"

        gval = grade_value(grade)
        assert 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)
```

- *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.items():
        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):
    num_value = { 'A': 4,  'B': 3,  'C': 2,  'D': 1,  'E': 0 }
    assert grade in num_value, “grade_value: unknown grade”
    return num_value[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

# Using asserts

- Some invariants are complex:
  - `numlist` has at least one even number
  - `arglist` consists of strings that contain at least one vowel
- Write your own functions that can be used in assert statements

# EXERCISE

*Do exercise 4. on assert statements in the handout.*

# 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
    - identify components that can completed (depth-first)
- Program defensively
  - figure out invariants that must hold in the program
  - use **`assert`**s to express invariants in the code

[end]