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?
    o how should we test it?
    o 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:

<table>
<thead>
<tr>
<th>Course</th>
<th>No. of units (U)</th>
<th>Grade (G)</th>
<th>U x G*</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSc 110</td>
<td>4</td>
<td>A</td>
<td>4 x 4 = 16</td>
</tr>
<tr>
<td>CSc 352</td>
<td>3</td>
<td>C</td>
<td>3 x 2 = 6</td>
</tr>
<tr>
<td>CSc 391</td>
<td>1</td>
<td>A</td>
<td>1 x 4 = 4</td>
</tr>
<tr>
<td>TOTAL:</td>
<td>4 + 3 + 1 = 8</td>
<td></td>
<td>16 + 6 + 4 = 26</td>
</tr>
</tbody>
</table>

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

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

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

There may be more than one way to do these
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 $A_1, ..., A_n$
  $-$ $A_1, ..., A_n$ together accomplish $A$

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)

read a file containing student grades, compute GPAs, and write them out
Example: GPA computation (conceptual)

- Read a file containing student grades, compute GPAs, and write them out
  - Read the student grades file
  - Compute each student's GPA
  - Write out each student's GPA
Example: GPA computation (conceptual)

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

read the student grades file

compute each student's GPA

write out 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}
Example: GPA computation (conceptual)

read the student grades file

read a file of courses + no. of units

compute each student's GPA

write out 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

\[ \text{GPA} = \frac{\text{total UxG}}{\text{total U}} \]
Example: GPA computation (conceptual)

read the student grades file
split it into a list, one element per student

read a file of courses + no. of units
split each student's list into a list of (course, grade)

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
Example: GPA computation (conceptual)

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

- Read a file containing student grades, compute GPAs, and write them out.
- Read the student grades file.
- Split it into a list, one element per student.
- Read a file of courses + no. of units.
- Split each student's list into a list of (course, grade).
- 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.
- Compute GPA: $\text{GPA} = \frac{\text{total UxG}}{\text{total U}}$. 
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

- 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

- read the student grades file
  - split it into a list, one element per student
- read a file of courses + no. of units
  - split each student’s list into a list of (course, grade)
- 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 } U \times G}{\text{total } U}
- write out each student's GPA

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

```python
# main(): read student grades file, compute GPAs, write them out
def main():
    pass
main()
```
Example: GPA computation (programming)

**Conceptual decomposition**
- read a file containing student grades, compute GPAs, and write them out
- read the student grades file
- compute and write out each student's GPA

**Incremental Program Development**

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

main()
```
Example: GPA computation (programming)

Conceptual decomposition

compute and write out each student's GPA
compute GPA
write out 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}}

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**

- **compute GPA**
- **write out 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 = \( \frac{\text{total UxG}}{\text{total U}} \)

**Incremental Program Development**

```python
# 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

Compute GPA
Write out 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}}

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**

- Compute and write out each student's GPA
  - Compute GPA
  - Write out 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
- 
  \[
  \text{GPA} = \frac{\text{total UxG}}{\text{total U}}
  \]

**Incremental Program Development**

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

- read the student grades file

  - split it into a list, one element per student
  - split each student's list into a list of (course, grade)

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

• while not done:
  – pick a task \( A \) and break it down into simpler tasks \( A_1, ..., A_n \)
  – flesh out the stub for \( A \) to execute the code for \( A_1, ..., A_n \) (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":
  ```python
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
    o 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:
    ```python
    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.

```python
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.

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

```python
# 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

# 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

```python
# 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
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

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
# lookup_units(course, course_units) : looks up the # no. of units for a course

```python
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

```python
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
    o use stubs for as-yet-undeveloped parts of the program

• Program defensively
  – figure out invariants that must hold in the program
  – use asserts to express invariants in the code