Chapter 1: Introduction to Programming and Problem Solving

CSC 111 – Computer Programming I

Dr. Sofien GANNOUNI

Week 1

Department of Computer Science

Outline

Lecture 1: What is Programming?

Lecture 2: Execution, Testing, and Documentation

Outline

What is Programming?

Definition

Programming is the process of designing and writing instructions that tell a computer exactly what to do to solve a problem.

- A computer is powerful, but has no common sense — it only does exactly what it is told.
- Programming bridges the gap between a **human idea** and a **machine action**.
- Good programmers are, first and foremost, good **problem solvers**.

Computer Programs and Algorithms

Algorithm

A finite, ordered sequence of well-defined steps that solves a problem.

Program

The concrete implementation of an algorithm, written in a programming language a computer can run.

- An **algorithm** is language-independent: the *same* algorithm can be written in Python, Java, or C.
- Analogy: a **recipe** (algorithm) vs. the **meal you actually cook** (program) following it.

Example: An Algorithm

Problem

Given three numbers a, b, c, determine which one is the largest.

What does the algorithm do?

It starts by assuming that a is the largest number, then compares b and c with the current largest value and updates it when necessary.

Example: An Algorithm

The algorithm:

1. Assume that *a* is the largest number.
2. Compare *b* with the current largest value.
3. If *b* is larger, update the largest value.
4. Compare *c* with the current largest value.
5. If *c* is larger, update the largest value.
6. Output the largest value.

```
ALGORITHM FindLargest
INPUT a, b, c

largest = a

IF b > largest THEN
    largest = b
ENDIF

IF c > largest THEN
    largest = c
ENDIF

OUTPUT largest
END
```

Example: An Algorithm

The algorithm:

1. **Assume that *a* is the largest number.**
2. Compare *b* with the current largest value.
3. If *b* is larger, update the largest value.
4. Compare *c* with the current largest value.
5. If *c* is larger, update the largest value.
6. Output the largest value.

```
ALGORITHM FindLargest
INPUT a, b, c

largest = a

IF b > largest THEN
    largest = b
ENDIF

IF c > largest THEN
    largest = c
ENDIF

OUTPUT largest
END
```

Example: An Algorithm

The algorithm:

1. Assume that *a* is the largest number.
2. **Compare *b* with the current largest value.**
3. If *b* is larger, update the largest value.
4. Compare *c* with the current largest value.
5. If *c* is larger, update the largest value.
6. Output the largest value.

```
ALGORITHM FindLargest
INPUT a, b, c

largest = a

IF b > largest THEN ⟸
    largest = b
ENDIF

IF c > largest THEN
    largest = c
ENDIF

OUTPUT largest
END
```

Example: An Algorithm

The algorithm:

1. Assume that *a* is the largest number.
2. Compare *b* with the current largest value.
3. **If *b* is larger, update the largest value.**
4. Compare *c* with the current largest value.
5. If *c* is larger, update the largest value.
6. Output the largest value.

```
ALGORITHM FindLargest
INPUT a, b, c

largest = a

IF b > largest THEN
    largest = b
ENDIF

IF c > largest THEN
    largest = c
ENDIF

OUTPUT largest
END
```

Example: An Algorithm

The algorithm:

1. Assume that *a* is the largest number.
2. Compare *b* with the current largest value.
3. If *b* is larger, update the largest value.
4. **Compare *c* with the current largest value.**
5. If *c* is larger, update the largest value.
6. Output the largest value.

```
ALGORITHM FindLargest
INPUT a, b, c

largest = a

IF b > largest THEN
    largest = b
ENDIF

IF c > largest THEN ⟸
    largest = c
ENDIF

OUTPUT largest
END
```

Example: An Algorithm

The algorithm:

1. Assume that *a* is the largest number.
2. Compare *b* with the current largest value.
3. If *b* is larger, update the largest value.
4. Compare *c* with the current largest value.
5. **If *c* is larger, update the largest value.**
6. Output the largest value.

```
ALGORITHM FindLargest
INPUT a, b, c

largest = a

IF b > largest THEN
    largest = b
ENDIF

IF c > largest THEN
    largest = c     ⟸
ENDIF

OUTPUT largest
END
```

Example: An Algorithm

The algorithm:

1. Assume that *a* is the largest number.
2. Compare *b* with the current largest value.
3. If *b* is larger, update the largest value.
4. Compare *c* with the current largest value.
5. If *c* is larger, update the largest value.
6. **Output the largest value.**

```
ALGORITHM FindLargest
INPUT a, b, c

largest = a

IF b > largest THEN
    largest = b
ENDIF

IF c > largest THEN
    largest = c
ENDIF

OUTPUT largest
END
```

One Algorithm, Different Languages

The **algorithm stays the same**; only its implementation changes.

Python

```python
a = float(input("a: "))
b = float(input("b: "))
c = float(input("c: "))

largest = a
if b > largest:
    largest = b
if c > largest:
    largest = c
print(largest)
```

C

```c
#include <stdio.h>
int main(void) {
    float a, b, c, largest;
    scanf("%f %f %f", &a, &b, &c);
    largest = a;
    if (b > largest)
        largest = b;
    if (c > largest)
        largest = c;
    printf("%f\n", largest);
    return 0;
}
```

Same steps, same result, different programming languages!

Steps of Problem Solving

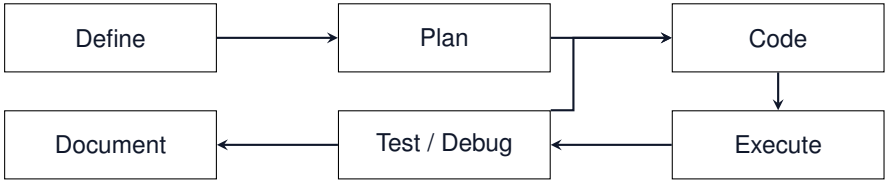
Solving a problem with a computer follows a systematic cycle:

1. Defining the problem
2. Planning the solution
3. Coding the program
4. Executing the program
5. Testing and debugging
6. Documentation

Key idea

Skipping steps 1–2 and jumping straight to code is the #1 cause of messy, buggy programs!

The Problem-Solving Cycle



Notice the loop:

testing often sends us back to fix the code.

Step 1: Defining the Problem

- Understand **exactly** what is being asked before writing anything.
- Identify:
  - **Inputs** — what data do we start with?
  - **Outputs** — what result do we need to produce?
  - **Constraints** — any rules or limits?

Example

"Compute the area of a circle."

Input: radius r. Output: area A. Rule: $A = \pi r^2$.

Step 2: Planning the Solution

- Break the problem into small, manageable steps **before** coding.
- Two common planning tools, both covered in this course:
  - **Pseudocode** — plain-language, structured steps.
  - **Flowcharts** — a visual diagram of the steps.
- Planning catches logical mistakes *early*, when they are cheap to fix.

Preview

We dedicate all of next lecture to flowcharts and pseudocode design.

Step 3: Coding the Program

- Translate the plan (pseudocode/flowchart) into a real programming language — **Python**, in this course.
- Follow the target language's syntax rules exactly.
- Good code is not just correct — it is also **readable**: clear names, consistent indentation, comments.

Remember

The computer will do precisely what your code says — not what you *meant*!

Interpreters vs. Compilers

Your **source code** must be translated into **something** the machine can execute.
Generally, source code is either compiled or interpreted by the computer.

Compiler

Translates the *entire* program into machine code **before** running it (e.g., C, C++).

Interpreter

Translates and executes the program **line by line** (e.g., Python).

- Python programs are run by the **Python interpreter**.
- **Trade-off:** interpreters are flexible and easy to test in; compiled code often runs faster.

Lecture 1 Summary

- Programming = turning ideas into precise instructions for a computer.
- An **algorithm** is language-independent; a **program** is its implementation.
- Problem solving follows six steps: *define, plan, code, execute, test/debug, document*.
- Interpreters (Python) translate and run code line by line; compilers translate the whole program first.

Next lecture: executing, testing/debugging, and documenting programs.

Outline

Recap of Lecture 1

- Programming turns a plan into instructions a computer can follow.
- We introduced the six-step problem-solving cycle.
- So far we covered: **define**, **plan**, **code**.
- Today: **execute**, **test/debug**, **document**.

Step 4: Executing the Program

- Running the code so the computer actually carries out the instructions.
- For Python, this typically means:
  - Running a script: `python program.py`
  - Running interactively in a shell / notebook / IDE
- Execution reveals whether the program actually produces the expected output.

Step 5: Testing and Debugging

Testing

Running the program with different inputs to check whether the output is correct.

Debugging

Finding and fixing the cause of an incorrect result (a **bug**).

Two broad categories of errors you will meet constantly:

- **Logical errors**
- **Runtime errors**

Logical Errors

Definition

The program runs without crashing, but produces the **wrong result** because the logic/algorithm itself is flawed.

Example

To compute the average of two numbers, a student writes:

$$\text{average} = a + b/2$$

This runs fine, but — due to operator precedence — it computes $a + \frac{b}{2}$, not $\frac{a+b}{2}$.

Logical errors are the hardest to catch: the program never complains!

Runtime Errors

Definition

An error that occurs **while the program is running**, causing it to crash or stop.

Common causes:

- Dividing by zero
- Using a variable that was never defined
- Asking for the 10th item of a list that only has 3 items
- Mismatched data types (e.g., adding a number to text)

Runtime errors are usually easier to catch than logical errors — the computer tells you (loudly!) when one happens.

Step 6: Documentation

- Writing clear explanations of **what** the program does and **how** it works.
- Includes:
  - Comments inside the code
  - External documentation (user guides, README files)
- Why it matters:
  - You will not remember every detail of your own code in 6 months.
  - Other people need to understand, use, and maintain your code.

Preview

Python comments (# and docstrings) are covered in the next chapter.

Worked Example: Circle Area

Let's apply the *full* cycle to one problem: calculating the area of a circle.

1. **Define:** input radius r; output area $A = \pi r^2$.
2. **Plan:**

   `INPUT r` $\rightarrow$ `COMPUTE A = pi * r * r` $\rightarrow$ `OUTPUT A`
3. **Code:** translate the plan into Python (next chapters!).
4. **Execute:** run it with $r = 2$; expect $A \approx 12.57$.
5. **Test:** try $r = 0$, a negative r, a very large r.
6. **Document:** explain what the program computes and how to use it.

Chapter Summary

- Problem solving is a **cycle**, not a straight line — testing often sends you back to coding.
- **Logical errors** give wrong answers silently; **runtime errors** crash the program.
- Documentation is not optional — it's part of professional programming.
- Interpreters (like Python's) execute your code line by line.

Coming up next: designing algorithms visually with *flowcharts*.