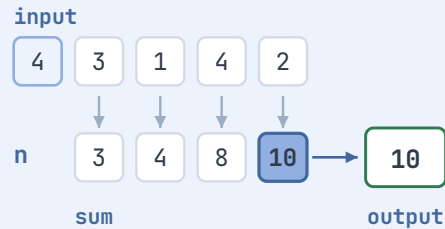


Input and output

Chapter 1.1 · Overall level: First programs



A program in a programming contest never sees a person typing. It receives a block of text, and its answer is checked by another program. *So how does it know which number is which, and what exactly should it print?*

This chapter answers that question and, along the way, gives you the small amount of Python, C++, Java or JavaScript you need to start: variables, whole-number arithmetic, loops, decisions and reading input. Every later chapter builds on these few pieces.

1.1.1 A program is a contract

Every problem statement contains an **input format** and an **output format**. Together they form a contract. The statement promises what the input will look like; your program promises to print exactly what the output format asks for, and nothing else.

The input is plain text made of **tokens**: groups of characters separated by spaces or line breaks. In the input `2 7` there are two tokens, `2` and `7`. Your output is also a list of tokens, and the judge compares your tokens with the expected ones, one by one.

Suppose a problem says “read two integers and print their sum”. A friendly program for a person might ask “Enter two numbers:” and reply “The answer is 9”. To a judge, that output starts with the token `Enter` where it expected `9`, so the answer is wrong even though the arithmetic is right.

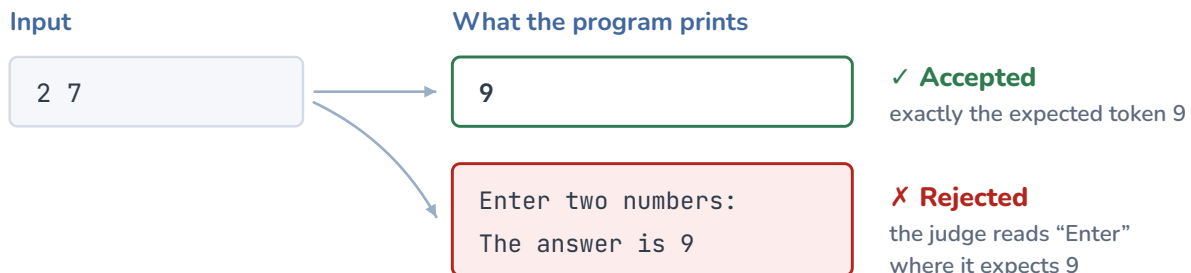


Figure 1.1.1 — A judge compares the specified output, so conversational prompts do not belong in the program output.

Two habits follow from the contract. First, read the input format carefully: it tells you how many tokens arrive and in what order. Second, print only the requested tokens, normally followed by a newline. Messages that help you debug belong on your screen while you test, never in the submitted output.

1.1.2 Variables, arithmetic, loops and decisions

A **variable** is a named box that holds a value, such as `count = 0`. Assigning a new value replaces the old one, so `count = count + 1` means “take the current value, add one, and store the result back in the same box”.

A **loop** repeats a block of statements. The loops in this library usually step a counter `i` through a range of whole numbers. An **if statement** runs a block only when a condition is true. The operator `%` gives the remainder after division, so `i % 2 == 0` tests whether `i` is even.

Here is one small program written in all four languages. It counts the even numbers from 1 to 10 and prints 5. Read the versions side by side: the ideas are identical, only the spelling changes.

Python

```
count = 0
for i in range(1, 11):
    if i % 2 == 0:
        count = count + 1
print(count)
```

C++ (inside main)

```
int count = 0;
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        count = count + 1;
    }
}
cout << count << '\n';
```

Java (inside main)

```
int count = 0;
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        count = count + 1;
    }
}
System.out.println(count);
```

JavaScript

```
let count = 0;
for (let i = 1; i <= 10; i++) {
    if (i % 2 === 0) {
        count = count + 1;
    }
}
console.log(count);
```

- **Variables.** C++ and Java state the type when a variable is created (`int count`); Python creates it on first assignment; JavaScript uses `let`.
- **Loops.** Python’s `range(1, 11)` stops *before* 11, so it gives 1 to 10. The other three test `i <= 10` before each round and add one with `i++` after it.
- **Blocks.** Python groups statements by indentation; the others use braces `{ }`. Indent in every language anyway: it shows the structure.
- **Comparison.** A single `=` stores a value; `==` (or `===` in JavaScript) asks whether two values are equal.

1.1.3 Whole-number arithmetic

Counting problems use whole numbers, called integers. Addition, subtraction and multiplication behave as you expect. Division needs care, because the four languages do not agree on what `/` means.

Expression	Python	C++ and Java (int)	JavaScript
17 divided by 5	<code>17 // 5</code> → 3	<code>17 / 5</code> → 3	<code>Math.floor(17 / 5)</code> → 3
remainder	<code>17 % 5</code> → 2	<code>17 % 5</code> → 2	<code>17 % 5</code> → 2
plain <code>/</code>	<code>17 / 5</code> → 3.4	whole part only	<code>17 / 5</code> → 3.4
-17 divided by 5	<code>-17 // 5</code> → -4	<code>-17 / 5</code> → -3	<code>Math.trunc(-17 / 5)</code> → -3

For non-negative numbers all four agree: the quotient is the whole part and `%` is what is left over, so $17 = 5 \times 3 + 2$. With negative numbers Python rounds down (to -4) while C++, Java and JavaScript's `Math.trunc` cut toward zero (to -3).

Rounding up

Many problems need the quotient rounded *up*: 23 students in cars that hold 4 need 6 cars, not 5, because the 3 students left over still need a car. Divide, then add one if anything is left over. For a total $t \geq 0$ and capacity $k \geq 1$, one integer formula does both:

$$\text{cars} = (t + k - 1) \text{ div } k \quad (23 + 3) \text{ div } 4 = 6, \quad (0 + 3) \text{ div } 4 = 0$$

Adding $k - 1$ pushes any total with a non-zero remainder past the next multiple of k , but never pushes an exact multiple that far. Here “div” is `//` in Python, `/` on integers in C++ and Java, and `Math.floor(... / k)` in JavaScript.

1.1.4 Reading tokens into variables

Reading input means turning characters into numbers. Each language has a tool that skips whitespace and hands you the next token. The safest pattern for beginners is to read *all* the tokens first and then use them in order.

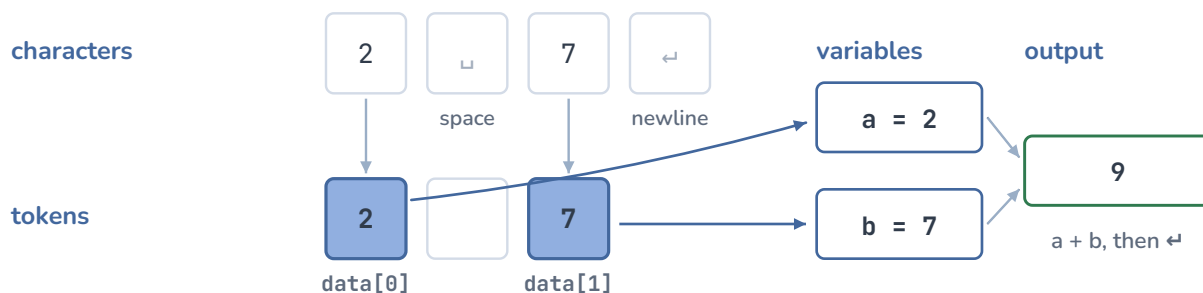


Figure 1.1.2 — Tracing the input “2 7”: whitespace separates the characters into tokens, and each token is converted into one named variable.

Language	Read every token	First token as an integer
Python	<code>data = sys.stdin.read().split()</code>	<code>a = int(data[0])</code>
C++	<code>cin >> a >> b;</code> reads one token per <code>>></code>	<code>int a;</code> declared first
Java	<code>next()</code> helper, shown in 1.1.8	<code>Integer.parseInt(next())</code>
JavaScript	<code>readFileSync(0, "utf8") .trim().split(/\s+/)</code>	<code>Number(data[0])</code>

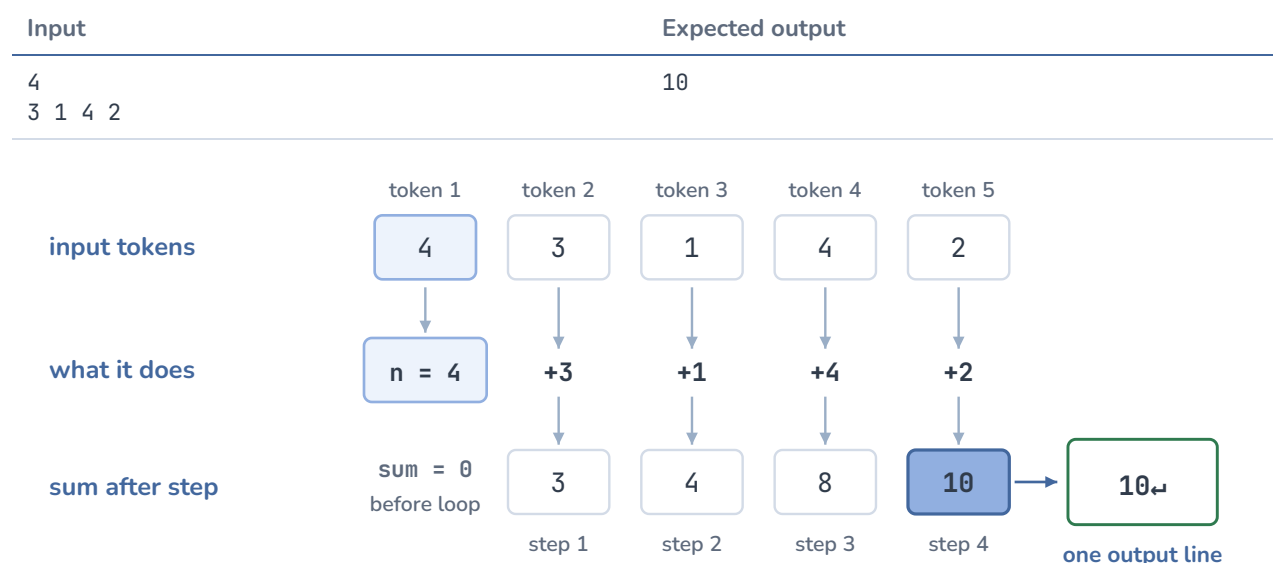
In Figure 1.1.2 the program reads two tokens, stores the first in `a` and the second in `b`, and prints `a + b`. The names are your choice; the *order* is fixed by the input format.

1.1.5 Reading a list whose length comes first

Most inputs contain a list. The statement then gives its length first. A typical format reads: “First line: *n*. Second line: *n* integers.” Three words are easy to mix up here:

- **A value** is one number the problem talks about, such as one daily count.
- **A token** is one group of characters in the input. Every value is one token, but so is *n*, which is not part of the list.
- **A line** is everything up to a newline. One line may hold one token or thousands of them.

The first token says how many value tokens follow, so the program knows when to stop. Take the input below. The first token is *n* = 4 and the four values are 3, 1, 4 and 2. To add them, keep an **accumulator**: a variable `sum` that starts at 0 and grows by one value per round of the loop.



The loop runs *n* = 4 times; each value token is read exactly once.

Figure 1.1.3 — Each input value is consumed once; the output contains only the requested answer.

Step	Token read	sum before	sum after
1	3	0	3
2	1	3	4
3	4	4	8
4	2	8	10

The output is exactly 10 followed by a newline: no label, no extra spaces, no list of partial sums.

Why the loop gives the right total

Look at what is true after each step. Before the loop, sum is 0, the total of no values. If after k steps sum equals the total of the first k values, then step $k + 1$ adds exactly the next value, so afterwards it equals the total of the first $k + 1$ values. A fact that stays true after every step is called an **invariant**. When the loop ends, $k = n$, so sum is the total of all n values: nothing skipped, nothing counted twice.

Pseudocode

Accumulating a list: $O(n)$ time, $O(1)$ extra memory

```

read n
sum ← 0
repeat n times:
    read the next token x
    sum ← sum + x
print sum

```

The program does a fixed amount of work per token, so the time grows in proportion to n , written $O(n)$. If each value is added as it is read, only n , x and sum are stored: $O(1)$ extra memory. Chapter 1.2 explains this notation properly.

1.1.6 Why line breaks usually do not matter

Token readers skip any run of spaces, tabs and newlines before a token. So these three inputs give the same five tokens and the same answer:

As specified	All on one line	Uneven spacing	One per line
4	4 3 1 4 2	4	4
3 1 4 2		3 1	3
		4 2	1
			4
			2

That is why the programs in this library read all tokens and ignore where the lines break. It also protects you from small surprises such as a trailing space or a missing final newline. A program that reads *lines* instead, for example Python's `input().split()` for a whole list, breaks if the list is spread over two lines. Lines matter only when a token may itself contain spaces, such as a full name; such problems say so in the input format.

1.1.7 Choosing an integer type that is wide enough

Python integers grow as large as needed. The other three languages store numbers in a fixed amount of memory, so each type has a largest value.

Type	Largest value it holds exactly	Use it for
<code>int</code> (C++, Java)	$2,147,483,647 \approx 2.1 \times 10^9$	single inputs and small totals
<code>long long</code> (C++), <code>long</code> (Java)	$\approx 9.2 \times 10^{18}$	sums, products and counts that may grow
JavaScript Number	$2^{53} - 1 \approx 9 \times 10^{15}$ (exact integers)	everything below that; <code>BigInt</code> beyond
Python <code>int</code>	no fixed limit	everything

Check the constraints before choosing. In the first practice problem both counts are at most 10^6 , so their sum is at most 2×10^6 and `int` is plenty. But a list of 200,000 values, each up to 10^6 , can total 2×10^{11} , roughly a hundred times the `int` limit. An `int` that goes past its limit does not stop with an error: it silently wraps around to a wrong, often negative, number. Declare the accumulator as `long long` or `long` whenever a total can exceed about two billion.

Also avoid fractional numbers (`double`, `float`) for counts. They are stored approximately, and a result such as 5.999999 printed as a count of boxes is simply wrong. Integer division and the rounding-up formula above keep everything exact.

1.1.8 Code in four languages

Each program solves the first practice problem, **Two counters**. The input is one line with two integers a and b (each between 0 and 10^6); the output is one integer, $a + b$, on its own line.

Sample input	Sample output
7 12	19

All four read every token, convert the first two to integers, and print one sum. Because they read tokens, they also work if a and b arrive on separate lines.

Python 3

```
import sys

def main():
    data = sys.stdin.read().split()
    a = int(data[0])
    b = int(data[1])
    print(a + b)

main()
```

Python: `sys.stdin.read()` returns the whole input as one string and `split()` cuts it at every run of whitespace.

C++17

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    cout << a + b << '\n';
    return 0;
}
```

C++: `cin >> a` skips whitespace, reads one token and converts it to the type of `a`. The line `#include <bits/stdc++.h>` brings in the whole standard library at once.

Checking a program yourself

Before submitting, run the program on the published sample and compare its output with the expected output character by character. Then invent a few inputs of your own whose answers you already know:

- the smallest values allowed, such as `0 0`, which must print `0`;
- the largest values allowed, such as `1000000 1000000`, to check that the type is wide enough;
- the same tokens with different spacing, for example `7` and `12` on separate lines.

Finally, look for anything printed besides the answer. The judge will not forgive a stray debug line, however helpful it was while you worked.

Java 17

```
import java.io.*;
import java.util.*;

public class Main {
    static BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
    static StringTokenizer tok = new StringTokenizer("");

    static String next() throws IOException {
        while (!tok.hasMoreTokens()) tok = new StringTokenizer(in.readLine());
        return tok.nextToken();
    }

    public static void main(String[] args) throws IOException {
        int a = Integer.parseInt(next());
        int b = Integer.parseInt(next());
        System.out.println(a + b);
    }
}
```

Java: `BufferedReader` reads the input a line at a time and `StringTokenizer` splits a line into tokens. The helper `next()` fetches a new line whenever the current one has no tokens left, so it ignores line breaks just like the other languages. Later chapters reuse this helper for long inputs, where it is much faster than `Scanner`.

JavaScript (Node)

```
const data = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const a = Number(data[0]);
const b = Number(data[1]);
console.log(a + b);
```

JavaScript (Node): `readFileSync(0, "utf8")` reads all of standard input; `trim()` removes the final newline and `split(/\s+/)` cuts at whitespace. `Number` converts a token to a number.

Each program does a constant amount of work: $O(1)$ time and memory. To solve a problem with a list, keep the same reading pattern and add the loop and accumulator from Section 1.1.5.

1.1.9 Common mistakes

- **Reading n as a list element.** In the input “4” then “3 1 4 2”, the first token is the length. Adding it to the sum gives 14 instead of 10. Read n first, then loop n times over the values.
- **Printing labels, prompts or debug lines.** “Enter n:”, “Answer: 10” and leftover test prints all add tokens the judge does not expect. Remove every print except the answer before submitting.
- **Using fractional division for whole items.** $9 / 4$ is 2.25 in Python and JavaScript. For counts, use integer division, and round up with $(t + k - 1) \text{ div } k$ when leftovers need their own box.
- **A total in a type that is too small.** Two million fits in an `int`; two hundred billion does not. Use `long` `long` or `Long` for running totals.
- **Reading lines when the format only promises tokens.** If the list is split over two lines, a line-based reader misses values. Read tokens unless a token may contain spaces.
- **Mixed-up order.** The input format fixes which token comes first. In “ n k ”, the first token is n , whatever name you give the variable.

1.1.10 When token reading is not enough

Reading all tokens at once suits almost every problem in this library, but it has limits. If a single input item contains spaces, such as a sentence, you must read whole lines and decide yourself where items begin. If the input is truly enormous, holding every token in memory at once may cost too much; the programs here read numbers as they go when that matters. A few contest problems are *interactive*: the judge answers each line you print before sending more input. Those need output flushed after every line and are outside this chapter.

Speed matters as well. Printing 200,000 answers one call at a time can be slow in Python and Java; collecting them and printing once, as the Chapter 2.1 programs do, avoids that. Chapter 1.2 shows how to estimate whether a whole program is fast enough before you write it.

Exercises

Difficulty: ● quick check, ●● needs an extra idea. Try each one before reading the answers.

1. ● For $n = 3$ and the values 2 0 5, what is the accumulator after the second value?
2. ● Why should the program not print “Enter n”?
3. ● Fill in the trace for $n = 5$ and the values 6 0 2 9 3: the value of `sum` after each of the five steps, and the exact output.
4. ● Give the value of `23 // 4` and `23 % 4` in Python, and of `-7 / 2` with C++ integers.
5. ●● A shop has 0 to 10^6 customers on each of 200,000 days. Which C++ type should hold the total, and why is `int` unsafe?
6. ●● Using integer arithmetic only, how many egg cartons of 6 are needed for 0, 6, 7 and 12 eggs?

Answers and explanations

Read these after attempting the exercises.

1. 2. The accumulator goes $0 \rightarrow 2$ after the first value and $2 + 0 = 2$ after the second; it becomes 7 only after the third.
2. That text is not in the output contract. The judge would read “Enter” where it expects the first answer token, so the output is wrong.
3. The first token, 5, is n . The sum goes 6, 6, 8, 17, 20, and the output is exactly **20** followed by a newline.
4. $23 // 4 = 5$ and $23 \% 4 = 3$ ($23 = 4 \times 5 + 3$). In C++, $-7 / 2 = -3$, because integer division cuts toward zero; Python’s $-7 // 2$ would give -4 .
5. long long. The total can reach $200,000 \times 10^6 = 2 \times 10^{11}$, far above the `int` limit of about 2.1×10^9 , and an `int` would wrap around silently.
6. With $(t + 5) \text{ div } 6$: 0 eggs $\rightarrow 0$, 6 $\rightarrow 1$, 7 $\rightarrow (12 \text{ div } 6) = 2$, 12 $\rightarrow (17 \text{ div } 6) = 2$. Only 7 has leftovers that need an extra carton.

Practice this topic

Four coding problems for this chapter are in the **Practice** tab. Open any of them in any order; the label shows difficulty within this topic.

Two counters

Apply

Two counters contain a and b items. Print their combined count.

Frame length

Connect

A rectangular display has integer width w and height h . Print its perimeter.

Collected tokens

Reason

A class records n daily token counts. Print the total number of tokens.

Packing batches

Stretch

There are n batches containing a_i badges each. Combine all badges and pack them into boxes holding at most k badges. Print the minimum number of boxes; zero badges need zero boxes.
