

UNIVERSITÉ
PARIS SCIENCES
& LETTRES

Input and Output

The Art of Computer Programming

Pierre Senellart



23 November 2026



Outline

- Motivation
- Functions and programs
- Streams, arguments, and exit status
- Bytes, buffers, and files
- Reading input
- Formatting output
- Files
- A complete example
- Practice
- Coda

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Motivation



Why input and output

- A program that only computes is of no use: it must **read** its data and **write** its results – to a user, a file, another program, the network
- Lecture 1 gave the bare minimum: `input()` and `print` in Python, `scanf` and `printf` in C. This lecture gives the rest, for **text**, the simplest form of input and output (next slide)
- Input and output (I/O) depend on what the program does not control:
  - the **operating system**, which owns files and devices
  - **performance**: a disk or a terminal is slow, so data goes through buffers
  - **users**, whose input is wrong as often as not, and files that are missing (lecture 8)
- The habits of this lecture are the same in the three languages; only the syntax changes

Many forms of input and output

Form	Examples
Text	terminal, text files, CSV, JSON
Graphical interfaces	windows, buttons, mouse, touch
Images, sound, video	a photo, a microphone, a camera
Network	Web pages, messages between programs
Databases	tables queried in SQL
Devices	sensors, motors, a game controller

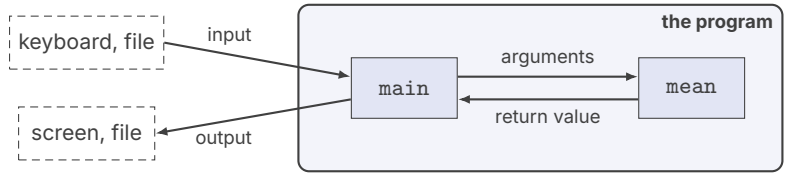
- As lecture 1 said, I/O is not only text: this lecture is restricted to text **to keep things simple**. Text is also what programs exchange most: configuration files, logs, CSV, JSON
- The other forms go through **libraries**, built on the notions of this lecture: bytes through the operating system, buffers, encodings, failures. A graphical program **waits for events** (a click, a key) instead of reading in order, and the library calls a function of ours for each (lecture 12)

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Functions



Arguments and return value, input and output



- **Arguments and return value** go from one function to another, **inside** the program: values of any type (a list, an object), nothing to convert, nothing that can go wrong on the way
- **Input and output** cross the boundary of the **whole program**: only text (or bytes), parsed on the way in, formatted on the way out, and anything can go wrong
- Most functions neither read nor write: they receive their data as **arguments**, and give their result by **returning** it – not with `input()` and `print`. A few functions, near `main`, read and write, and call the others

Computing is not printing

hard to reuse

```
def mean(t):
    print(sum(t) / len(t))
```

better

```
def mean(t):
    return sum(t) / len(t)

print(f"{mean(t):.2f}")
```

- A **returned value** goes back to the caller, who can compute with it, test it, or print it in the format **it** chooses
- A **printed value** leaves the program: the caller can do nothing more with it, and the function can only be tested by reading its output
- **The rule**: functions compute and return; reading and printing happen in **a few well-identified places** (often `main`)

What the caller receives

Python

```python
def square_print(x):
    print(x * x)          # no return

y = square_print(3)       # prints 9
print(y)                  # None
y + 1                     # TypeError
```

C

```c
int square_print(int x) {
  printf("%d\n", x * x);
}                         // no return: a warning

int y = square_print(3);  // prints 9
printf("%d\n", y + 1);    // 3 here!
```

- The 9 on the screen **is not** in `y`: it went to the terminal, and the program cannot get it back
- Python: without `return`, a function returns `None` (lecture 2); the error shows **later**, where `y` is used
- C: `control reaches end of non-void function` is only a warning (`-Wall`), and the value is **undefined behavior**: whatever was left in a register. With `void`, the assignment would not compile

What the interactive interpreter displays

Python

```
>>> square(3)
9
>>> square_print(3)
9
>>> x = square(3)
>>> y = square_print(3)
9
>>> print(x, y)
9 None
```

- In the **interactive** interpreter (and in a notebook such as Jupyter), the value of an expression typed alone is **displayed**: here, the value returned by `square`
- So `square(3)` and `square_print(3)` look the same: the first 9 is a **return value**, the second an **output**
- An assignment displays nothing but what the function prints
- In a **script**, `square(3)` alone on a line displays **nothing**: the value is thrown away

Reading is not taking an argument

hard to reuse

```
def is_prime():
    n = int(input("n? "))
    for d in range(2, n):
        if n % d == 0:
            return False
    return n >= 2
```

better

```
def is_prime(n):
    for d in range(2, n):
        if n % d == 0:
            return False
    return n >= 2

n = int(input("n? "))
print(is_prime(n))
print([k for k in range(30) if is_prime(k)])
```

- On the left, the **argument** has become an **input**: the function can only be used when someone types a number, and cannot list the primes below 30, nor be tested automatically (lecture 11)
- The data is a **parameter**; the caller decides where it comes from: keyboard, file, or computation

Functions and programs compared

	A function	A program
Called by	another function	the shell, a user, another program
Receives	its **arguments**	**command-line arguments**, and reads **input**
Gives back	its **return value**	an **exit status**, and writes **output**
Of type	any type	strings and bytes only; an integer for the status

- `main` is both a function and the entry point of the program: its parameters `argc` and `argv` are the program's command-line arguments, its return value is the program's exit status (next section)
- A **side effect** is what a function does beyond returning a value: printing, writing a file, modifying a global variable or an argument (lecture 4). Keep them few and visible

Which is which?

When asked to… use **I/O**, or **arguments and a return value**?

1. “Write a function that computes the mean of a list of numbers”
2. “Write a program that reads numbers, one per line, and prints their mean”
3. “Write a function that tells whether a number is prime”
4. “Write a recursive function that computes the *n*-th Fibonacci number”
5. “Write a function that asks the user for their age”

Which is which?

When asked to… use **I/O**, or **arguments and a return value**?

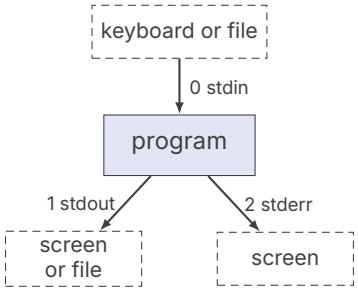
1. "Write a function that computes the mean of a list of numbers"
   → **a parameter for the list, a `return`; no `input`, no `print`**
2. "Write a program that reads numbers, one per line, and prints their mean"
   → **reading and printing, around a call to the function above**
3. "Write a function that tells whether a number is prime"
   → **return a `bool`; the caller decides what to print, if anything**
4. "Write a recursive function that computes the *n*-th Fibonacci number"
   → **return it; a `print` inside would run once per recursive call**
5. "Write a function that asks the user for their age"
   → **here reading *is* the task: `input` inside, and return the age as an `int`**

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Streams



The three standard streams



- Every program starts with three **streams** already open, numbered 0, 1, and 2:
  - **stdin** standard input, where data is read
  - **stdout** standard output, for the **results**
  - **stderr** standard error, for **diagnostics**
- By default, all three are the terminal; the program does not know, and need not know, what they really are

	C (`<stdio.h>`)	**C++** (`<iostream>`)	**Python** (`import sys`)
input	`stdin`	`std::cin`	`sys.stdin`
output	`stdout`	`std::cout`	`sys.stdout`
errors	`stderr`	`std::cerr`	`sys.stderr`

Redirections and pipes

Terminal

```
python3 stats.py < numbers.txt          # stdin read from a file
python3 stats.py < numbers.txt > out.txt 2> errors.txt
./generate | python3 stats.py           # stdout of one is stdin of the other
sort names.txt | uniq -c | sort -nr     # who appears most often?
```

- The **shell** connects the streams before the program starts: < for stdin, > for stdout (>> to append), 2> for stderr, | (a **pipe**) from one program to the next
- The same in `cmd` on Windows; PowerShell has all but < (use `Get-Content numbers.txt |` instead)
- Small programs that each do one thing, **combined** with pipes, e.g., on Linux and macOS, `sort` (sorts lines) and `uniq -c` (counts repeated lines)
- Hence two habits:
  - write **results** to stdout and **messages** to stderr: an error message must not end up in `out.txt`, mixed with the results
  - do not assume that stdin is a keyboard and stdout a screen

Command-line arguments

C

```c
int main(int argc, char *argv[]) {
  for (int i = 0; i < argc; ++i)
    printf("%d: %s\n", i, argv[i]);
}
```

Terminal

```
$ ./args data.txt -v "two words"
0: ./args
1: data.txt
2: -v
3: two words
```

Python

```python
import sys
for i, arg in enumerate(sys.argv):
    print(f"{i}: {arg}")
```

- The words after the program's name, split on spaces unless quoted, arrive as an **array of strings**; element 0 is the program's name. In C++, `main` takes the same `argc` and `argv`
- Typically used to tell a program **which file** to work on
- Always strings: convert those that stand for numbers, e.g., `int(sys.argv[1])`; and check that the program received as many as it expects

Exit status

C

```c
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "usage: %s file\n",
            argv[0]);
    return 2;            // or exit(2)
  }
  // ...
  return 0;              // success
}
```

Python

```python
import sys

if len(sys.argv) != 2:
    print(f"usage: {sys.argv[0]} file",
          file=sys.stderr)
    sys.exit(2)
# ...
# reaching the end: status 0
```

- A program returns an integer to whoever started it: its **exit status**. **0 means success**; anything else means failure, and the program may say which kind (keep codes small: Linux and macOS keep only 8 bits)
- An uncaught exception (lecture 8) gives a nonzero status
- In a Linux or macOS shell, `echo $?` shows it, and `make && ./prog` runs `./prog` only if `make` succeeded

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Bytes and buffers



Files and file descriptors

- A **file** is a named sequence of bytes, kept by the **operating system**; a **path** names it: `/home/alice/data.txt`, or `C:\Users\Alice\data.txt` on Windows (where `/` works too)
- A relative path, `data.txt`, is looked up in the **current directory** – the one the program was started from, not necessarily the one where the program is
- A program never touches the disk: it asks the OS to **open** the file, then to **read** or **write** bytes, then to **close** it
- Each open file is described by a small integer, a **file descriptor**: 0, 1, and 2 are the standard streams, the next file opened gets 3
- The number of open files is **limited** (by default 1024 per process on Linux, 256 on macOS): a program that forgets to close its files ends up unable to open any (lecture 8)
- On Linux and macOS, devices, pipes, and network connections are read and written through file descriptors too: “everything is a file”

The system calls underneath

C

```c
#include <fcntl.h>      // POSIX (Linux, macOS), not standard C
#include <unistd.h>

int fd = open("data.txt", O_RDONLY);           // a descriptor, or -1
if (fd == -1) { perror("data.txt"); return 1; }
char buffer[4096];
ssize_t n;
while ((n = read(fd, buffer, sizeof buffer)) > 0)   // 0: end of file
  write(1, buffer, n);                              // 1: stdout
close(fd);
```

- `open`, `read`, `write`, `close`: what the OS provides, on bytes only (here, to copy a file to stdout)
- Each call is a **system call**, which costs far more than a function call: reading byte by byte would be very slow
- Languages build on them a **higher-level** object, with formatting and a **buffer**: `FILE *` in C, streams in C++, file objects in Python

Buffering

- The library gathers what you write in a **buffer** in memory, and hands it to the OS in one system call when it is full, or when it is **flushed**
- Three modes, usually set up as follows:

unbuffered	written at once: stderr, so that error messages are never lost
line-buffered	written at each newline: stdout when it is a terminal
fully buffered	written when the buffer is full: files, and stdout when it is **redirected**

C

```c
printf("starting\n");
abort();            // crash
```

Terminal

```
$ ./crash
starting
$ ./crash > out.txt; cat out.txt
$                              # nothing!
```

Output still in the buffer when a program crashes is **lost**: the same program behaves differently on a terminal and in a file

Flushing

C

```c
printf("computing... ");
fflush(stdout);                         // shown now, without a newline
```

C++

```cpp
std::cout << "computing... " << std::flush;
std::cout << "done" << std::endl;      // std::endl: "\n" and a flush
```

Python

```python
print("computing... ", end="", flush=True)
```

- Flush when the output must be seen **now**: a progress message, a prompt before a long computation, a log before a possible crash
- Closing a file, or ending the program normally, flushes everything
- Do not flush **at every line**: writing many lines with `std::endl` instead of `"\n"` can make a program several times slower

Text, bytes, and encodings

- A file holds **bytes**; text is bytes **decoded** with an encoding (lecture 1): `é` is the two bytes `c3 a9` in utf-8
- Decoded with the wrong encoding, text is garbled:

Python

```
"é".encode("utf-8").decode("latin-1")      # 'Ã©'
```

- Python decodes for you in text mode, with an encoding that **depends on the system** if you do not give one: always write `open(name, encoding="utf-8")`. C and C++ just pass the bytes along
- **Newlines** differ: `\n` (byte `0a`) on Linux and macOS, `\r\n` (`0d 0a`) on Windows. In **text mode**, the library converts `\n` on the way in and out
- For data that is not text (an image, a compressed file), open in **binary mode**, which changes nothing: `fopen(name, "rb")`, `open(name, "rb")` (Python then gives `bytes`, not `str`)

Binary files

C

```c
FILE *f = fopen("t.bin", "wb");                  // (checks omitted)
size_t n = fwrite(t, sizeof t[0], 1000, f);      // the bytes of int t[1000]
fclose(f);
f = fopen("t.bin", "rb");
n = fread(t, sizeof t[0], 1000, f); fclose(f);  // n: how many were read
```

- `fwrite` and `fread` copy **memory as it is**: no formatting, no parsing
- The integers 0 to 999 999 take 4 bytes each in binary, 1 to 6 each as text (plus a separator)
- But the file depends on the machine: the size of an `int`, and the **byte order** (lecture 1: `01 00 00 00` is 1 on x86, but 2^{24} on a big-endian machine). A pointer written to a file means nothing when read back
- **Advice**: text, unless size or speed demand otherwise – it can be read, edited, and exchanged between machines

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Reading

Two ways to read

- **Line by line**: read a whole line as a string, then **parse** it – split it, convert its pieces
  - you see the whole line, and can report it, with its number, if it is wrong
  - a bad line does not disturb the next ones
- **Value by value**: ask for an integer, then a floating-point number... and let the library skip spaces and newlines: `scanf`, `std::cin >> x`
  - short to write, for well-formed input
  - what happens on bad input is subtle, and line numbers are lost
- Python only reads lines (or the whole file): parsing is always explicit
- **Advice**: read lines, then parse, as soon as the input comes from a human or may be malformed

Reading in Python

Python

```python
name = input("Your name? ")          # one line, without its newline

import sys
for line in sys.stdin:               # every line, until the end of input
    words = line.split()             # cut on spaces: '3 4\n' -> ['3', '4']
    try:
        x, y = int(words[0]), int(words[1])
    except (ValueError, IndexError):
        print(f"bad line: {line!r}", file=sys.stderr)
        continue
    print(x + y)
```

- A line read from a stream **keeps** its final `\n`: `line.strip()` or `line.rstrip("\n")`
- The **end of input** is not a line: the loop just stops; `input()` raises `EOFError`. On a terminal, type Ctrl+D (Linux, macOS) or Ctrl+Z then Enter (Windows)
- `int` and `float` raise `ValueError` on anything that is not a number (lecture 8)

Reading lines in C

C

```c
char line[256];
while (fgets(line, sizeof line, stdin) != nullptr) {   // nullptr: end of input
  char *newline = strchr(line, '\n');
  if (newline != nullptr) *newline = '\0';             // cut the newline, if any
  printf("read: %s\n", line);
}
if (ferror(stdin)) perror("stdin");                    // a real error
```

- `fgets(s, n, f)` reads one line, newline included, but **never more than** `n - 1` characters: a longer line comes in several pieces, never past the array (lecture 4)
- It returns `nullptr` at the end of the input, or on an error; `feof` and `ferror` tell which
- Never use `gets`, which has no size: it was **removed** from the language, after decades of buffer overflows
- From a file, the same with `fopen`'s `FILE *` instead of `stdin`

From a string to a number in C

C

```c
atoi("12x");   // 12, and atoi("x") is 0: no way to tell an error

struct long_result to_long(const char *s) {     // as in lecture 8
  char *end;
  errno = 0;
  long v = strtol(s, &end, 10);                 // base 10; end: first unused char
  bool ok = end != s                            // at least one digit read
            && *end == '\0'                     // nothing left over
            && errno == 0;                      // not too large for a long
  struct long_result r = {ok, v};
  return r;
}
```

- `atoi` and `atof` (lecture 1) **cannot report** a failure: avoid them on input
- `strtol`, `strtoll`, `strtod` (`<stdlib.h>`) say where they stopped, and set `errno` on overflow
- `"42"` → 42; `"12x"`, `""`, and `"9999999999999999999"` are rejected

`scanf` and its traps

C

```c
int age;
char name[100];
if (scanf("%d", &age) != 1) {   // 1 value?
  fprintf(stderr, "not a number\n");
  return 1;
}
fgets(name, sizeof name, stdin);
// name is "\n": the end of the
// line after the age!
```

- `scanf` returns the **number of values** read: always check it
- On `abc`, it reads nothing, and leaves `abc` in the input for the next read
- It stops **right after** the number: the newline stays, and the `fgets` that follows reads an empty line

- A string needs a **maximal width**, `scanf("%99s", name)`, or it overflows `name`
- Fine for well-formed input of numbers separated by spaces; for anything else, `fgets` then `strtol` or `sscanf` on the line – and do not mix `scanf` and `fgets` on one stream

Reading in C++

line by line

```
std::string line;
while (std::getline(std::cin, line)) {
  try {
    int x = std::stoi(line);
    std::cout << 2 * x << "\n";
  } catch (const std::exception &) {
    std::cerr << "bad: " << line << "\n";
  }
}
```

value by value

```
int x;
while (std::cin >> x)
  std::cout << 2 * x << "\n";
if (!std::cin.eof())
  std::cerr << "not a number\n";
```

- `std::getline` reads a line of any length, without its newline, into a `std::string`
- `std::stoi`, `std::stod` throw (lecture 8) on `"abc"`, but, like `strtol`, read `"7x"` as 7 unless you check where they stopped
- A stream is **false** once a read has failed (end of input, or text that is not a number, such as `abc`), and every later read fails until `std::cin.clear()`

Asking again until the input is valid

Python

```python
def ask_int(prompt, low, high):
    while True:
        text = input(prompt)
        try:
            n = int(text)
        except ValueError:
            print("not an integer")
            continue
        if low <= n <= high:
            return n
        print(f"from {low} to {high}")

age = ask_int("Age? ", 0, 150)
```

C++

```cpp
int ask_int(const std::string &prompt, int low, int high) {
  while (true) {
    std::cout << prompt;
    int n;
    if (std::cin >> n) {
      if (low <= n && n <= high) return n;
      std::cout << "from " << low << " to " << high << "\n";
    } else {
      if (std::cin.eof())
        throw std::runtime_error("end of input");
      std::cout << "not an integer\n";
      std::cin.clear();            // reset the state
    }
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
                    '\n');         // skip the line
  }
}
int age = ask_int("Age? ", 0, 150);
```

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Formatting



Format strings in C

Specifier	Type
`%d`	`int`
`%ld`, `%lld`	`long`, `long long`
`%zu`	`size_t`
`%x`	`unsigned`, in hexadecimal
`%f`, `%e`	`double`; `%lf` in `scanf`
`%c`, `%s`	`char`, string
`%p`	pointer
`%%`	the character `%`

C

```c
printf("[%8.3f]", 3.14159);
                              // [   3.142]
printf("[%-6s]", "ab");       // [ab    ]
printf("[%6s]", "ab");        // [    ab]
printf("[%05d]", 42);         // [00042]
printf("[%e]", 12345.678);
                              // [1.234568e+04]
```

- Between `%` and the letter: a **width** (padded on the left, on the right with `-`), then `.` **precision** (digits after the decimal point)
- The specifier must match the type of the argument: if not, **undefined behavior**, which `-Wall` usually detects

Formatting into a string in C

C

```c
char filename[32];
int n = snprintf(filename, sizeof filename, "result-%03d.txt", 7);
// filename is "result-007.txt"; n is 14, the length it needed
if (n < 0 || n >= (int) sizeof filename) { /* did not fit: truncated */ }
```

- `snprintf` formats like `printf`, into an array, and **never writes past** the size given: lecture 1's way from a number to a string
- It returns the length the result **would** have had: a value at least the size means the result was truncated
- `sprintf`, without the size, has the same flaw as `gets`: avoid it
- The other way, from a string: `strtol` (above), or `sscanf(line, "%d %d", &x, &y)`, which reads from a string as `scanf` reads from stdin

Formatting in Python

Python

```
(x, name, n) = (3.14159, "ab", 42)
f"[{x:8.3f}]"          # '[   3.142]'      same as %8.3f
f"[{name:<6}]"         # '[ab    ]'        < left, > right, ^ centered
f"[{n:05d}]"           # '[00042]'
f"{1234567:,}"         # '1,234,567'
f"{0.25:.0%}"          # '25%'
f"{x=}"                # 'x=3.14159'       to debug
f"{name!r}"            # "'ab'"            repr: shows the quotes
```

- After the `:` of an f-string (lecture 1), a **format specification**, close to C's: width, alignment, precision, type
- No type to match: an f-string cannot cause undefined behavior
- Older styles, still met in existing code: `"%8.3f" % x` and `"{:8.3f}".format(x)`
- `print(a, b, sep=", ", end="")` changes the separator and the end of line; `print(..., file=f)` writes to a file or to `sys.stderr`

Formatting in C++

C++

```
#include <iomanip>
std::cout << std::fixed << std::setprecision(3)                // stay
          << "[" << std::setw(8) << 3.14159 << "]";            // [   3.142]
std::cout << "[" << std::left << std::setw(6) << "ab" << "]"; // [ab    ]
```

C++

```
struct Point { double x, y; };
std::ostream &operator<<(std::ostream &os, const Point &p) {
  return os << "(" << p.x << ", " << p.y << ")";      // return os: to chain
}
Point p = {1, 2.5};
std::cout << "p = " << p << "\n";                     // p = (1, 2.5)
```

- **Manipulators**: `setw` for the next value only, the others until changed. Type-safe, but verbose
- `operator<<` for a class of our own (lecture 5) prints it on **any** stream, files included
- C++20 brings `std::format("{:8.3f}", x)`, and C++23 `std::print`, with Python's syntax

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Files



Files in C

C

```c
FILE *f = fopen("out.txt", "w");
if (f == nullptr) {
  perror("out.txt");                // lecture 8
  return 1;
}
for (int i = 1; i <= 10; ++i)
  fprintf(f, "%d\t%d\n", i, i * i);
if (fclose(f) != 0) {               // also fails!
  perror("out.txt");
  return 1;
}
```

Mode	Meaning
`"r"`	read; must exist
`"w"`	write; **emptied** first
`"a"`	write at the end
`"wx"`	write; must not exist
`"r+"`	read and write
`"…b"`	binary: `"rb"`...

`fprintf`, `fscanf`: like `printf`, `scanf`, on **any** open file. `stdin`, `stdout`, `stderr` are `FILE *` too: `printf(...)` is `fprintf(stdout, ...)`

- One `fclose` per successful `fopen`, **on every path** (lecture 8). It **flushes** the buffer, so a full disk may only be noticed there

Files in C++

C++

```
#include <fstream>

std::ifstream in("numbers.txt");          // opens for reading
if (!in) {
  std::cerr << "numbers.txt: cannot open\n";
  return 1;
}
std::ofstream out("doubled.txt");         // opens for writing; emptied
std::string line;
while (std::getline(in, line))
  out << 2 * std::stod(line) << "\n";     // the same << as for std::cout
                                          // both closed at the end of the block
```

- **Streams**: all that holds for `std::cin` and `std::cout` applies
- The destructor closes the file, even when an exception is thrown: **RAII** (lecture 8)
- Other modes as a second argument: `std::ofstream out(name, std::ios::app)` to append, `std::ios::binary`

Files in Python

Python

```python
with open("numbers.txt", encoding="utf-8") as f:   # "r" by default
    for line in f:                                 # one line at a time
        ...

with open("numbers.txt", encoding="utf-8") as f:
    text = f.read()                                # everything, as one string

with open("out.txt", "w", encoding="utf-8") as f:  # "w", "a", "x", "rb"...
    f.write("one line\n")                          # write adds no newline
    print(1, 2, 3, file=f)                         # print does
```

- Same modes as C's `fopen`; `open` raises `FileNotFoundError`, `PermissionError`... all subclasses of `OSError` (lecture 8)
- `with` closes the file on every path (lecture 8): never leave a file to the garbage collector
- `f.read()` is fine for a small file; a `for` loop handles a file larger than memory

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Example



The task

- `stats`: count, mean, minimum, and maximum of numbers given one per line, in a file named on the **command line**, or on **stdin** without a name (to work in a pipe)
- Blank lines skipped; **bad lines** reported on stderr with their number; **exit status** 0 on success, 1 if the file cannot be read or holds no number, 2 if called wrongly

numbers.txt

```
12
  3.5

abc
7x
-2
1e3
```

Terminal

```
$ python3 stats.py numbers.txt
numbers.txt:4: not a number: 'abc'
numbers.txt:5: not a number: '7x'
count      4
mean     253.375
min       -2.000
max     1000.000
$ python3 stats.py nosuch; echo $?
nosuch: No such file or directory
1
```

In Python

Python

```
import sys
def read_numbers(f, name):
    numbers = []
    for i, line in enumerate(f, start=1):
        text = line.strip()
        if text == "":
            continue
        try:
            numbers.append(float(text))
        except ValueError:
            print(f"{name}:{i}: not a number: "
                  f"{text!r}", file=sys.stderr)
    return numbers
```

`read_numbers` takes **any** file object and returns a list: testable without a terminal.

`sys.exit(main())` makes the result of `main` the exit status

Python

```
def main():
    if len(sys.argv) > 2:
        print(f"usage: {sys.argv[0]} [file]",
              file=sys.stderr)
        return 2
    if len(sys.argv) == 2:
        name = sys.argv[1]
        try:
            with open(name, encoding="utf-8") as f:
                numbers = read_numbers(f, name)
        except OSError as e:
            print(f"{name}: {e.strerror}",
                  file=sys.stderr)
            return 1
    else:
        numbers = read_numbers(sys.stdin, "<stdin>")
    # ... then "no numbers", or the four results
```

In C

C

```c
bool stats(FILE *f, const char *name) {
  char line[256];
  long count = 0;
  double sum = 0, min = 0, max = 0;
  for (long i = 1; fgets(line, sizeof line, f); ++i) {
    strip(line);                    // like Python's
    if (line[0] == '\0')
      continue;                     // blank
    double x;
    if (!to_double(line, &x)) {     // strtod
      fprintf(stderr, "%s:%ld: not a number: '%s'\n",
              name, i, line);
      continue;
    }
    // ... update count, sum, min, max
  }
  // ... ferror, "no numbers", or printf
```

C

```c
int main(int argc, char *argv[]) {
  if (argc > 2) {
    fprintf(stderr, "usage: %s [file]\n",
            argv[0]);
    return 2;
  }
  if (argc == 1)
    return stats(stdin, "<stdin>") ? 0 : 1;
  FILE *f = fopen(argv[1], "r");
  if (f == nullptr) {
    fprintf(stderr, "%s: %s\n", argv[1],
            strerror(errno));
    return 1;
  }
  bool ok = stats(f, argv[1]);
  fclose(f);
  return ok ? 0 : 1;
}
```

Paths and directories in Python

Python

```python
from pathlib import Path

p = Path("data") / "numbers.txt"     # data/numbers.txt, or data\numbers.txt
p.exists(), p.suffix, p.stem         # (True, '.txt', 'numbers')
text = p.read_text(encoding="utf-8") # open, read, close: for small files
for q in Path("data").glob("*.csv"): # every CSV file of the directory
    print(q.name)
Path("results").mkdir(exist_ok=True)
```

- `pathlib.Path` builds paths with `/`, with the right separator for the system: no string concatenation, no `\\`
- Also lists, creates, renames, and removes files and directories (older code: `os.path`)
- C++17 has the same in `<filesystem>` (`std::filesystem::path`); standard C has nothing but `remove` and `rename`

CSV files

grades.csv

```
name,city,grade
Alice,"Paris, France",15.5
Bob,Lyon,12
```

Python

```
import csv

with open("grades.csv", encoding="utf-8",
          newline="") as f:
    for row in csv.DictReader(f):
        print(row["city"], float(row["grade"]))
```

- **Comma-separated values**: one record per line, fields separated by commas, often a header line. What spreadsheets export
- `line.split(",")` is **wrong**: it cuts `"Paris, France"` in two. A field may contain commas, quotes, even newlines, within quotes
- Use a library that knows the rules: Python's `csv` (`csv.reader` gives lists, `csv.DictReader` dictionaries, `csv.writer` writes); in C and C++, an external library
- Everything read is a **string**: convert, and check

JSON

Python

```python
import json

data = {"name": "Alice", "grades": [15.5, 12], "tutor": None}
with open("alice.json", "w", encoding="utf-8") as f:
    json.dump(data, f)              # {"name": "Alice", "grades": [15.5, 12], ...
with open("alice.json", encoding="utf-8") as f:
    back = json.load(f)             # the same dictionary
```

- **JavaScript Object Notation**: numbers, strings, `true`/`false`/`null`, lists `[...]`, and objects `{...}` with string keys, nested at will
- Close to Python's own literals: `json` converts to and from `dict`, `list`, `str`…in one call (`json.loads`/`json.dumps` for strings)
- The usual format for configuration files, and for data exchanged over the Web
- **Advice**: before inventing a file format, check whether CSV or JSON will do – readers and writers already exist, in every language

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Practice



Input is not to be trusted

- Input comes from **outside**: a user who mistyped, a file from an older version, someone who wants to break the program. Check everything before use:
  - the **form**: is it a number, are all fields there?
  - the **range**: an age of -3, an index past the end, a size of 10^{12} to allocate
  - the **length**: a line longer than the array
- Never pass input as a **format string**:

C

```c
printf(name);          // a name containing %s or %n: a security hole
printf("%s", name);    // correct
```

- Report a bad input **precisely** – the file, the line, the text – and decide: skip it and carry on, or stop with a nonzero exit status (lecture 8)

Writing a file safely

Python

```
import os

def save(name, lines):
    tmp = name + ".tmp"
    with open(tmp, "w", encoding="utf-8") as f:
        for line in lines:
            print(line, file=f)
    os.replace(tmp, name)        # only now does name change, in one step
```

- Opening with `"w"` **empties** the file at once: if the program then fails (an exception, a full disk, a power cut), the old content is lost, and the new one incomplete
- Write a **temporary file** next to it, and replace the old file only once the new one is complete: `os.replace`, `rename` in C (on Linux and macOS)
- The file then holds either the old content or the new one, never half of each – how editors and databases save
- Mode `"x"` (`"wx"` in C) fails if the file already exists, instead of overwriting it

What output costs

C++ / C, 10^6 lines	time
`out << i << "\n"`	0.06 s
`out << i << std::endl`	0.45 s
`fprintf(f, "%d\n", i)`	0.05 s
then `fflush(f)`	0.46 s

Python, 10^6 lines	time
`print(i, file=f)`	0.31 s
with `flush=True`	1.19 s
`f.write(f"{i}\n")`	0.16 s

- Writing a million numbers to a file (one core of a laptop; the programs are in `data/`): flushing at every line, with a system call each time, costs **4 to 8 times** more
- In Python, `print` does more work than `write` (separators, conversion)
- **Writing to a terminal** is slower still: a program that prints a line per step of a long loop spends its time displaying. Print a summary, or every 1000 steps

Messages for the programmer: logging

Python

```
import logging
logging.basicConfig(level=logging.INFO)    # once, in the main program

logging.debug("x = %s", x)                 # hidden at level INFO
logging.info("reading %s", name)           # INFO:root:reading numbers.txt
logging.warning("line %d ignored", i)      # WARNING:root:line 5 ignored
```

- Results are for the **user** (stdout); messages on what the program did are for the **programmer** (stderr)
- `print` calls added to debug are forgotten, mixed with the results, and must be removed by hand
- A **logging** library writes messages on stderr, with a **level** (`DEBUG`, `INFO`, `WARNING`, `ERROR`): one line of configuration shows more or fewer of them, or sends them to a file, without touching the code
- In C and C++, `fprintf(stderr, ...)` and `std::cerr`, or an external library

The three languages side by side

	C	C++	Python
Read a line	`fgets`	`std::getline`	`for line in f`, `input()`
String to number	`strtol`, `strtod`	`std::stoi`, `std::stod`	`int`, `float`
Format	`printf`, `snprintf`	`<<`, `<iomanip>`	f-strings
Open a file	`fopen`	`std::ifstream`, `std::ofstream`	`open`
Close it	`fclose`, by hand	destructor	`with`
Failure	`nullptr`, `errno`	stream is false	`OSError`
Arguments	`argv`	`argv`	`sys.argv`
Exit status	`return` from `main`	`return` from `main`	`sys.exit`

- Keep computation and I/O **apart**: functions that return values, reading and printing in a few places
- Results to **stdout**, messages to **stderr**, an **exit status** that says whether the program succeeded
- Read lines, then parse; check every conversion; close every file, on every path

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Coda

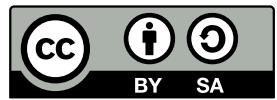


Wednesday's tutorial and references

- Wednesday's tutorial: command-line tools in Python and C that read files and standard input, report malformed lines, and return an exit status that says whether they succeeded; redirections and pipes; formatted tables; reading a CSV file – and the exceptions of lecture 8, put to use
- The complete programs of this lecture are on the course page: `stats` in the three languages (in C++, with `std::getline`, `std::stod`, and an `std::ifstream` closed by its destructor)
- **Next lecture**: building correct programs – testing, assertions, invariants
- References: Python Software Foundation [2026]; Stroustrup [2022], chapter 11 (input and output); King [2008], chapter 22 (input/output)

Licensing

This work is licensed under a Creative Commons “Attribution-ShareAlike 4.0 International” license.



Bibliography I

K. N. King. *C Programming: A Modern Approach*. W.W. Norton & Company, 2 edition, 2008. ISBN 0393979504. URL http://knking.com/books/c2/index.html.

Python Software Foundation. The python tutorial: Input and output. https://docs.python.org/3/tutorial/inputoutput.html, 2026.

Bjarne Stroustrup. *A Tour of C++*. Addison-Wesley, 3 edition, 2022. ISBN 978-0-13-681648-5. URL https://www.stroustrup.com/tour3.html.