Tutorial 9: Input, Output, and Exceptional Situations

The Art of Computer Programming

25 November 2026

Objective. This tutorial puts into practice the tenth lecture (input and output) together with the eighth (exceptional situations), which had no tutorial of its own. You start from a short program that mixes computing and printing and ignores every error, and turn it into a command-line tool: functions that return values, a file named on the command line or standard input, malformed lines reported on standard error, and an exit status. You then read a CSV file into a formatted table, with an exception class of your own, and repair a C program that ignores its errors. Do Parts A and B in the first half of the session and Parts C and D in the second half; Part E is for those who finish early.

Reminders

- Create a folder `tutorial09` and run every program from a terminal opened in it. Commands are written for Linux and macOS; on Windows, type `python` for `python3` and `.\stats.exe` for `./stats`.
- `$ python3 stats.py < numbers.txt` feeds a file to standard input; PowerShell has no `<`: type `$ Get-Content numbers.txt | python stats.py`. On a terminal, end the input with Ctrl+D (Linux, macOS) or Ctrl+Z then Enter (Windows).
- The exit status of the last command is shown by `$ echo $?` (Linux, macOS) or `$ echo $LASTEXITCODE` (PowerShell).
- Save every file in utf-8 (the default of VS Code, shown at the bottom right of its window).
- Compile C programs with `$ gcc -std=c23 -Wall -Wextra stats.c -o stats`.

Part A: Computing is not printing

Q1. A program to repair. Type this program as `naive.py`, and create a file `numbers.txt` with the four lines `12`, `3.5`, `-2`, and `20.5`.

Python

```
def mean():
    f = open("numbers.txt")
    total = 0
    count = 0
    for line in f:
        total += float(line)
        count += 1
    print(total / count)

mean()
```

(a) Run it: it prints 8.5. Now replace the last line by `m = mean()` followed by `print(m + 1)`. What happens, and why?

(b) Could this function compute the mean of another file? Of numbers typed at the keyboard? Of a list that the program already has?

(c) Rewrite it as `stats.py`, with three functions: `mean(numbers)`, which takes a list and *returns* its mean, with no `print`; `read_numbers(f)`, which takes a file that is already open and returns the list of the numbers that it holds; and `main()`, which opens `numbers.txt` in a `with` block (with `encoding="utf-8"`), calls the other two, and prints the two lines below with f-strings: the count on 6 columns, the mean on 10 columns with 3 decimals. The file ends with a call to `main()`.

Text

```
count      4
mean       8.500
```

(d) The original program never closes its file. When does the `with` block close it, and what if `float` raises an exception in the middle of the loop?

Part B: When things go wrong

Q2. Test the error paths. Run `stats.py` in each of the three situations below. Each time, read the last line of the traceback: which *class* of exception is raised, and by which line of your program?

(a) The file is missing (rename it for a moment).
(b) The file is empty.
(c) The file holds the seven lines on the right: a blank line and two lines that are not numbers have been added.

None of these is a bug of the program: they are mistakes of its user, who cannot do much with a traceback. Keep the seven-line file for the rest of the tutorial.

numbers.txt

```
12
3.5

abc
7x
-2
20.5
```

Q3. Report, and carry on.

(a) In `read_numbers`, which now also takes the `name` of the file, skip blank lines (`line.strip()` is then empty), and catch the exception raised by a line that is not a number: print the message below on *standard error*, and go on with the next line. `enumerate(f, start=1)` gives the line numbers.

Text

```
numbers.txt:4: not a number: 'abc'
numbers.txt:5: not a number: '7x'
count      4
mean       8.500
```

(b) The `try` block should hold the conversion and nothing else. What would the program print if the whole `for` loop were inside it? And why `except ValueError` and not a bare `except`?
(c) Make `main` *return* the exit status of the program, and end the file with `sys.exit(main())`. A file that cannot be opened: catch `OSError as e`, print the name of the file and `e.strerror` on standard error, return 1. No number at all: print `no numbers` on standard error, return 1, without calling `mean`. Otherwise return 0. Run the three situations of Q2 again, and look at the exit status after each.

Q4. Command line and streams.

(a) Take the name of the file from the command line, `$ python3 stats.py numbers.txt`. Without a name, read `sys.stdin` instead (`read_numbers` needs no change: why?), under the name `<stdin>`. With more than one argument, print `usage: stats.py [file]` on standard error and return 2. Try it on a file, with numbers typed at the keyboard, and with a redirection of standard input.
(b) Run `$ python3 stats.py numbers.txt > out.txt`. What appears on the screen, and what is in `out.txt`? What would be in `out.txt` if the messages had been printed with a plain `print`? Then add `2> errors.txt` to the command.
(c) Write `gen.py`, which prints n random numbers, one per line, where n is its command-line argument; `random.uniform(0, 20)` draws one, to be printed with 2 decimals. Called without an argument, or with `abc`, it prints a usage message on standard error and returns 2: which two classes of exceptions does this take? Then connect the two programs with a pipe: `$ python3 gen.py 1000 | python3 stats.py`. Which mean do you expect?

Part C: A CSV file and a formatted table

grades.csv

```
name,city,grade
Alice,"Paris, France",15.5
Bob,Lyon,12
Chloé,"Lille, France",17.25
David,Lyon,abc
Emma,Paris,9
Farid,Paris,150
```

Create the file `grades.csv` on the left, and a program `grades.py` that takes its name on the command line. Opening the file and reporting that it cannot be opened are as in `stats.py`; for a CSV file, add `newline=""` to the arguments of `open`.

Q5. Reading and formatting.

(a) First cut each line with `line.strip().split(",")` and print the second field. What is wrong with Alice's city?

(b) Read the file with `csv.DictReader` instead, and print the table below, ignoring David and Farid for now: name and city aligned on the left, on 10 and 16 columns; grade on the right, on 6 columns with 2 decimals; and a last line with the mean. Print the header with the same widths (`f"{'name':<10}"`).

Text
```
name      city             grade
Alice     Paris, France    15.50
Bob       Lyon             12.00
Chloé     Lille, France    17.25
Emma      Paris             9.00
mean                       13.44
```

(c) *An exception class of your own.* Define `class GradeError(Exception)`, and a function `parse_grade`, which takes a string `text` and returns the grade as a `float`, or raises a `GradeError` with a message that contains `text`, in two cases: `text` is not a number (catch the `ValueError` as `e` and `raise GradeError(...) from e`), or the number is not between 0 and 20. Call it on David's grade without catching anything: which two exceptions does the traceback show?

(d) In the loop over the rows, catch `GradeError as e`, report it on standard error with the name of the file and the number of the line (the header is line 1), and skip the row:

Text
```
grades.csv:5: not a number: 'abc'
grades.csv:7: not between 0 and 20: '150'
```

(e) Replace `encoding="utf-8"` by `encoding="latin-1"`. What becomes of Chloé, and why? (Put utf-8 back afterwards.)

Part D: In C

Q6. The same program, without exceptions. Type this program as `stats.c`.

C
```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  FILE *f = fopen(argv[1], "r");
  char line[256];
  double sum = 0;
  int count = 0;
  while (fgets(line, sizeof line, f) != nullptr) {
    sum += atof(line);
    ++count;
  }
  printf("%f\n", sum / count);
}
```

(a) Run it on `numbers.txt`; without any argument; on a file that does not exist; on an empty file. What happens each time, and what is the exit status? Where does 5.857143 come from? Compare with what Python did in Q2, before you handled anything: which of the two behaviors is the more dangerous?

(b) Repair it, one check at a time, until it behaves as `stats.py` on a file named on the command line: the same messages on `stderr`, the same two lines of results (`"count %6d\n"` and `"mean  %10.3f\n"`), the same exit statuses.

- `argc` is not 2: a usage message, `return 2`.
- `fopen` returns `nullptr`: print the name of the file and `strerror(errno)`, `return 1` (headers `<string.h>` and `<errno.h>`).
- Each line: cut its newline with `line[strcspn(line, "\r\n")] = '\0';` and skip it if it is then empty. Convert it with a function `bool to_double(const char *s, double *x)`, written with `strtod` on the model of the lecture's `to_long`; when it returns `false`, report the line and go on.
- No number at all: `no numbers`, `return 1`.
- One `fclose` for the `fopen`: on which paths is it needed?

(c) Python stopped you at each unhandled error in Q2. What told you here that a check was missing?

Part E: If you have time: in C++

Q7. Exceptions and RAII. Write `stats.cpp`, the same program in C++, with three functions that report their failures by throwing:

- `mean`, which takes a `std::vector<double>` and throws `std::invalid_argument("no numbers")` if it is empty;
- `to_double`, which calls `std::stod(s, &used)` and throws `std::invalid_argument` too if `used`, the number of characters read, is not `s.size()`;
- `read_numbers`, which opens an `std::ifstream`, throws an `std::runtime_error` if it cannot be opened, reads it with `std::getline`, and catches the exceptions of `to_double` to report the bad lines.

`main` catches `const std::exception &`, prints `e.what()` on `std::cerr`, and returns 1. Where is the file closed, and how many lines of your program deal with closing it, compared with Part D? Compile with `$ g++ -std=c++17 -Wall -Wextra stats.cpp -o statscpp`.

Check

By the end of the session you should be comfortable with: writing functions that take arguments and return values, with reading and printing kept in `main`; reading a file or standard input line by line and checking every conversion; printing results on standard output and messages on standard error, and returning an exit status; redirections and pipes; formatting a table; catching the exception that you can handle, where you can handle it, and defining a class of your own; and checking every return value in C. If one of them is still unclear, ask us, or finish the corresponding exercises at home.