

Principles of Computer Science II

Python in practice: files, errors and recursion

Marco Zecchini

Sapienza University of Rome

Tutorial 2

Today's plan

- 1 **Warm-up:** Python exercises one notch harder than in Tutorial 1 (strings, lists, dictionaries, user input).
- 2 **Files:** reading and writing text and CSV files, where the data of bioinformatics lives.
- 3 **Errors:** how to read an error message, and how to handle errors with `try/except`.
- 4 **Recursion:** the exercises of Lecture 2, pseudocode first, then Python.

Same recipe as in Tutorial 1: a short reminder of the concept, then an exercise. The text of all the exercises is on the course website.

Before we start

- Open a new Colab notebook, File > Save a copy in Drive, run the cells in order with Shift+Enter.
- Today we also create **files**. In Colab, the folder icon on the left shows the files of the session: they are *deleted* when the session ends, so download what you want to keep.
- When something breaks, **read the error message** before asking for help: today we learn how.

Where we are

In Tutorial 1 we translated the pseudocode of the book into Python: assignment, arithmetic, if/else, for, while, arrays as lists, def/return.

Today we add the tools you need to work with real data:

- strings and **dictionaries** as data structures;
- **files** as the input and the output of a program;
- **exceptions**, because real data is messy;
- **recursion**, because many algorithms of the book are recursive (Sect. 2.5).

Strings and printing: reminder

```
bar = "*" * 3           # "***": repetition
print(bar)              # ***
print()                 # an empty line
print("A", end="")      # no newline after A
print("CGT")            # ACGT, on the same line
```

- A string is a sequence of characters: `len(s)`, `s[0]`, `s[-1]`,
for `c` in `s`:
- `+` concatenates, `*` repeats: `"ab" * 2` is `"abab"`.
- `print` always ends with a newline, unless you pass `end=""`.

Exercise 1: histogram

- Write a function `histogram(numbers)` that, given a list of numbers, prints a histogram made of asterisks: one bar per number, bars separated by an empty line.
- Example: `histogram([3, 7, 9, 5])` prints the output on the right.
- Hint: one for loop and the repetition `"*" * n`.

```
***  
  
*****  
  
*****  
  
*****
```

Bonus: `histogram(count_nucleotides(dna))`, with the function of Tutorial 1, draws the composition of a DNA string.

Building a string, asking the user: reminder

```
word = input("Word? ")    # what the user types
out = ""                  # result, built step by step
for c in word:
    if c in "aeiou":      # is c one of these?
        out = out + c.upper()  # build a new string
    else:
        out = out + c
print(out)                # "dna" -> "dnA"
```

- Also handy: `c.lower()`, `c.isalpha()`, `c == " "`, `len(word)`.
- `input()` returns a string: `int(input())` for a number.
- “Until the user says stop”: a while loop (Tutorial 1, Ex. 4 bonus).

Exercise 2: rövarspråket

Swedish children play with a secret language, *rövarspråket*: every consonant is doubled, with an “o” in the middle; vowels and spaces are left alone.

- Write a function `rovarsprak(text)` that translates a word or a phrase: `rovarsprak("mangiare")` returns `"momanongogiarore"`.
- Then write a program that asks the user for a phrase, prints the translation and asks whether to translate another one; if so, it waits for a new phrase, otherwise it stops.
- Hint: for each character decide whether it is a consonant (a letter that is not a vowel) and append either `c + "o" + c` or `c` to the result.

Modifying a list: reminder

```
a = [5, 3, 8]
for x in a:                # x is a COPY of each element
    x = x + 1              # a is still [5, 3, 8]
for i in range(len(a)):
    a[i] = a[i] + 1        # a is now [6, 4, 9]
b = []
for x in a:
    b.append(x + 1)        # new list, a is untouched
```

- To change the elements of a list you need their **index**: `a[i]`
=
- Two styles: modify the list *in place* (whoever called you sees the change) or build and return a *new* list.

Exercise 3: decrement a list

- Write a function `decrement(a)` that, given a list of integers, decreases every element by 1 and returns the list.
- Example: `decrement([5, 3, 8])` returns `[4, 2, 7]`.
- Write it twice: once modifying `a` in place with `for i in range(len(a))`, once building a new list with `append`.
- Question: after `b = decrement(a)`, what is `a` in the two versions? Check with `print`.

Dictionaries: reminder

```
pair = {"A": "T", "C": "G", "G": "C", "T": "A"}
print(pair["A"])           # T: look up a key
pair["N"] = "N"           # add (or overwrite) a key
print(len(pair))          # 5
for base in pair:         # loop over the keys
    print(base, pair[base])
print(pair.values())       # the values only
print("X" in pair)        # False: is X a key?
```

- A dictionary maps **keys** to **values**; a list is like a dictionary whose keys are 0, 1, 2, ...
- Looking up a missing key is an error: `pair["X"]` raises `KeyError` (more on this in Part 3).

Exercise 4: sum of the values of a dictionary

- Write a function `sum_values(d)` that returns the sum of all the values of a dictionary.
- Example: `sum_values({'a': 100, 'b': 200, 'c': 300})` returns 600.
- Do it with a for loop, without calling `sum()`.
- Bonus: rewrite `count_nucleotides(dna)` of Tutorial 1 so that it returns a dictionary, `{ 'A': 1, 'C': 2, 'G': 2, 'T': 1 }` for "ATGCGC"; then `sum_values` of the result must be `len(dna)`.

Why files?

- So far the input of our programs was typed in the code or by the user. Real data comes in **files**: a genome in FASTA format, a table of measurements, the dataset of a Rosalind problem.
- A program reads its input from a file, computes, and writes the result to another file, so that another program (or a colleague) can use it.
- Two formats today: plain **text** files (one sequence per line) and **CSV** files (tables).

Writing and reading a text file

```
with open("seqs.txt", "w") as f:    # "w": (over)write
    f.write("ATGCGC\n")
    f.write("GGATTA\n")
with open("seqs.txt", "r") as f:    # "r": read
    for line in f:                  # line by line
        dna = line.strip()          # remove "\n"
        print(dna, len(dna))
```

- `open(name, mode)`: mode "r" reads, "w" writes from scratch, "a" appends.
- `with ... as f:` closes the file at the end of the block.
- `f.read()` gives the whole content as one string.

Where is my file?

- A name like "seqs.txt" is looked up in the **current folder**: in Colab, the folder shown by the folder icon on the left (/content); on your computer, the folder from which you run the script.
- In Colab you can create a file by writing it from a cell:

```
%%writefile film.csv  
Title,Release Date,Director  
Monty Python's Life Of Brian,1979,Terry Jones
```

- or upload one with the upload button of the folder panel, or write it with Python as in the previous slide.

CSV: a table as text

film.csv

```
Title,Release Date,Director
Monty Python's Life Of Brian,1979,Terry Jones
Monty Python's The Meaning Of Life,1983,Terry Jones
...
```

- **Comma-Separated Values:** one row per line, columns separated by commas, usually a first line with the column names (the *header*).
- You could read it with `line.strip().split(",")`, but a title containing a comma would break everything.
- The `csv` module of the standard library handles those details for us.

Reading a CSV file: `csv.reader`

```
import csv

rows = []
with open("film.csv", "r") as csvfile:
    csvreader = csv.reader(csvfile)
    fields = next(csvreader)           # the header
    for row in csvreader:              # a list of strings
        rows.append(row)
print("Field names:", ", ".join(fields))
for row in rows:
    print(row[0], "-", row[1])         # title and year
```

- Every row is a **list**: `row[0]` is the title, `row[2]` the director.
- Everything is a string: convert with `int(row[1])` if you need a number.

Reading a CSV file: `csv.DictReader`

```
import csv

films = []
with open("film.csv", "r") as csvfile:
    for row in csv.DictReader(csvfile):
        films.append(row)           # row is a dictionary

for film in films:
    print(film["Title"], film["Director"])
```

- `DictReader` uses the header as keys: `row["Title"]` instead of `row[0]`, more readable and robust if the columns move.
- Exercise 4 again: dictionaries are everywhere.

Writing a CSV file: csv.writer

```
import csv

fields = ["Name", "Branch", "Year", "CGPA"]
rows = [
    ["Nikhil", "COE", "2", "9.0"],
    ["Sanchit", "COE", "2", "9.1"],
    ["Aditya", "IT", "2", "9.3"]
]

with open("university_records.csv", "w") as csvfile:
    csvwriter = csv.writer(csvfile)
    csvwriter.writerow(fields)           # one row
    csvwriter.writerows(rows)           # many rows
```

- `writerow` takes a list and writes one line, adding commas (and quotes when needed).
- Open the resulting file from the folder panel to check it.

Exercise 5: CSV files

Follow the steps on the course website (adapted from the GeeksforGeeks tutorial on CSV files):

- 1 Create `film.csv` with the five Monty Python films.
- 2 Run the three examples: `csv.reader`, `csv.DictReader`, `csv.writer`. Open the files you produced.
- 3 Write a script that (a) reads `film.csv`, (b) writes a second file `series.csv` with the six rows given on the website, (c) writes a third file with the content of both, header included once.

Bonus: with `DictReader`, print only the films whose director includes Terry Jones.

Reading an error message

```
Traceback (most recent call last):  
  File "tb.py", line 5, in <module>  
    print(third(F))  
  File "tb.py", line 2, in third  
    return F[3]  
IndexError: list index out of range
```

- Read it **from the bottom**: the last line says *what* went wrong, the type of error and a message.
- The lines above say *where*: the chain of calls, from the cell you ran down to the line that failed (`F[3]` inside `third`).
- The error is not always where the fix is: here `F` has 3 elements, so the bug may be in who called `third`.

The usual suspects

Error	Message	Typical cause
SyntaxError	expected ':'	missing colon or bracket
IndentationError	unexpected indent	wrong indentation
NameError	name 'lenght' is not defined	typo, not yet assigned
TypeError	can only concatenate str ...	"A" + 1
ValueError	invalid literal for int()	int("abc")
IndexError	list index out of range	a[len(a)]
KeyError	'X'	d["X"], no such key
ZeroDivisionError	division by zero	x / 0
FileNotFoundError	No such file or directory	wrong name or folder
RecursionError	maximum recursion depth ...	no base case (Part 4)

Syntax errors stop the program before it runs; all the others happen *while* it runs, on a specific input.

Handling an error: pseudocode

Format:

```
try
    B
except error E
    C
```

Effect: executes instructions *B*. If during *B* an error of type *E* happens, *B* is interrupted and instructions *C* are executed instead; then the program continues. If no error happens, *C* is skipped.

- An error is an **exception**: it is *raised* where it happens and travels up the chain of calls until someone *handles* it. If nobody does, the program stops with the traceback.
- This notation is not in the book: the pseudocode there assumes that the input is valid.

Handling an error: Python

```
try:
    n = int(input("How many? "))
    print(100 / n)
except ValueError:
    print("That is not an integer")
except ZeroDivisionError:
    print("n must be different from 0")
print("Done")           # reached in all three cases
```

- Several except blocks, one per type of error: the first that matches runs.
- `except Exception as e: print(e)` catches everything; use it sparingly, hiding an error is worse than seeing it.
- Raise your own: `raise ValueError("not a nucleotide")`.

Exceptions and files

```
def read_sequences(filename):  
    seqs = []  
    try:  
        with open(filename, "r") as f:  
            for line in f:  
                seqs.append(line.strip())  
    except FileNotFoundError:  
        print("Cannot find", filename)  
    return seqs # [] if the file is missing
```

- A missing file is the most common error with data: the function reports it and returns an empty list, instead of crashing the whole analysis.
- Handle only the errors you expect; let the others show the traceback.

Exercise 6: robust input and output

- 1 Write `ask_integer(prompt)` that asks the user for an integer and, until the user types a valid one, prints a message and asks again (catch `ValueError`). It returns the integer.
- 2 Write `load_rows(filename)` that returns the rows of a CSV file as a list of lists; if the file does not exist, print a message and return an empty list. Try it with `"film.csv"` and with a wrong name.
- 3 Bonus: complement(base) of Tutorial 1 with a dictionary `{"A": "T", ...}`: look up the base and return `"?"` when a `KeyError` is raised.

Recursion: reminder from Lecture 2

“Recursion is one of the most ubiquitous algorithmic concepts. Simply, an algorithm is recursive if it calls itself.” (p. 24)

- It has to terminate: a **base case**, an input whose answer is known without further calls.
- Every call works on a **smaller** instance, moving towards the base case: the algorithm “bottoms out” (p. 31).
- “Recursive algorithms can often be rewritten to use iterative loops instead, and vice versa; it is a matter of elegance and clarity” (p. 28).

Jones, Pevzner, Sections 2.5–2.6, pp.
24–33.



Recursion: pseudocode and Python

FACTORIAL(n)

```
1 if n = 1
2     return 1
3 return n · FACTORIAL(n - 1)
```

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

- The pseudocode of the book may call itself, like a Python function: the translation is the same as in Tutorial 1.
- Line 1 is the **base case**; line 3 calls the algorithm on a **smaller** input, $n - 1$.

What happens when it runs

```
def factorial(n):  
    print("called with n =", n)  
    if n == 1:  
        return 1  
    res = n * factorial(n - 1)  
    print("n =", n, "returns", res)  
    return res
```

```
factorial(4)
```

- Run it: four “called with” lines going *down* to the base case, then three “returns” lines coming back *up*.
- Each call waits for the inner one to finish: Python keeps all of them in memory, one on top of the other.

When the base case is missing

```
def factorial(n):  
    return n * factorial(n - 1)      # no base case!  
  
factorial(4)
```

```
RecursionError: maximum recursion depth exceeded
```

- The calls go on with $n = 4, 3, 2, 1, 0, -1, \dots$ forever; Python gives up after about 1000 nested calls and raises `RecursionError`.
- Same error if the base case is never reached: `factorial(0)` with the base case $n = 1$.
- Always check: does every call move towards the base case?

How to work on the exercises

For each exercise, in this order:

- 1 identify the **base case** and write it down;
- 2 write the **recursive case**: the answer for the input in terms of the answer for a smaller input;
- 3 write the **pseudocode**, in the style of the book;
- 4 write the **Python** function and test it on the examples;
- 5 (bonus) write the **iterative** version with a loop and compare the two.

Exercise 7: power

- Write a recursive function `power(x, n)` that returns x^n for an integer $n \geq 0$, without using `**`.
- Examples: `power(2, 10)` returns 1024; `power(5, 0)` returns 1.
- Hint: $x^n = x \cdot x^{n-1}$. What is x^0 ?

Exercise 8: sum of the first n natural numbers

- Write a recursive function `sum_to(n)` that returns $1 + 2 + \dots + n$.
- Example: `sum_to(10)` returns 55.
- Hint: $1 + 2 + \dots + n = n + (1 + 2 + \dots + (n - 1))$.
- Compare with `SUMINTEGERS` of Tutorial 1: same problem, iterative then and recursive now (Sect. 2.6).

Exercise 9: greatest common divisor

Euclid's observation: the greatest common divisor of a and b , with $b > 0$, is the same as that of b and $a \bmod b$ (the remainder, $a \% b$); and $\text{gcd}(a, 0) = a$.

- Write a recursive function `gcd(a, b)`: `gcd(48, 18)` returns 6; the calls are $(48, 18) \rightarrow (18, 12) \rightarrow (12, 6) \rightarrow (6, 0)$.
- Write a program that reads two integers from the keyboard with `ask_integer` of Exercise 6 and prints their gcd.
- Question: does every call make the problem smaller? Which argument shrinks?

Exercise 10: transcribing DNA into RNA (Rosalind)

Problem RNA on Rosalind: given a DNA string t , return its transcription, obtained by replacing every T with U.

- Example: GATGGAAGTTGACTACGTAAATT becomes GAUGGAACUUGACUACGUAAAUU.
- Write a recursive `transcribe(t)`: the base case is the empty string; otherwise transcribe the first character and append the transcription of the rest, `t[1:]`.
- Then download the dataset from Rosalind, read it from the file (Part 2), write the answer to a file and submit it.
- Question: what happens with a string of 10 000 nucleotides? Try it, read the error, and write the iterative version (a loop, or `t.replace("T", "U")`).

Cheat sheet

Task	Python
read a text file	<code>with open(name, "r") as f: then for line in f:</code>
write a text file	<code>with open(name, "w") as f: then f.write(s + "\n")</code>
read a CSV file	<code>csv.reader(f)</code> (lists), <code>csv.DictReader(f)</code> (dicts)
write a CSV file	<code>csv.writer(f).writerow(a_list)</code>
dictionary	<code>d[key], key in d, for key in d:, d.values()</code>
handle an error	<code>try: ... except SomeError: ...</code>
recursion	base case first, then the call on a smaller input

Before asking for help: read the last line of the traceback, then the line of code it points to.

What's next

- Rosalind: solve a couple more problems from the first page (*Counting DNA Nucleotides*, *Complementing a Strand of DNA*): you already have the functions from Tutorial 1, now read the dataset from the file.
- Recursion will come back with sorting, dynamic programming and graphs: the recipe stays the same.
- Keep the cheat sheets of both tutorials at hand: from now on, we translate the algorithms of the book.