IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# *Codecamp*

**Back-to-school week – TAF ILSD**

## 1 Rules of the *codecamp*

The goal of the *codecamp* is to work in groups to develop a piece of software including a set of features.

Groups will be formed by the teachers and will each have between 4 and 5 students.

Each group develops in Python a similar piece of software. The first common step is described later in section 2. It is up to you to decide how you want to proceed. Information on essential Python tooling can be found in section 3.

Each step must be validated before moving on to the next. Validation includes verification by the group that the program is working properly, as well as validation of the requirements laid out in this document. This code review must be carried out by at least one member of the group. Then, final validation must be performed by an external review (by a teacher or another team). It is then possible to draw the next extension from the teachers.

To avoid wasting too much time, we recommend that you think carefully about the design of each extension before moving on to its implementation. We also advise against coming up with overly complex solutions. The goal is to implement as many extensions as possible, while keeping a relatively clean code.

In general, it is preferable not to rely on external libraries. Please try to make maximum use of the mechanisms provided by Python's standard libraries. In particular, the use of data processing libraries, such as Pandas, is prohibited.

## 2 Initial problem

The goal of this *codecamp* is to produce a simple piece of software that makes it easier for a caterer to manage orders placed by their customers. The starting point is the same for all groups and involves implementing the following features: adding, modifying, deleting and displaying customer orders. An order has an *identifier* provided by the system upon creation, a *customer name*, a *service date*, a *total amount*, and a *details* area (the contents of the order).

User interaction must remain simple; we opt for command line interaction in a terminal (CLI – Command Line Interface).

Each time the program is run, it receives a file name as a parameter, containing the previously created orders. Executing the program will generally modify the contents of this file.

You are free to choose the file format, but we recommend keeping it simple (for example, a text format with one line per order).

This first version should therefore allow you to run the following commands:

- `orders orders.txt add <···>` : adds to the `orders.txt` file the new order, then returns its identifier;
- `orders orders.txt modify id <···>` : replaces the information of the order matching the identifier `id` in `orders.txt`, or returns a message if the order is not found;
- `orders orders.txt rm id` : removes from the `orders.txt` file the order matching the identifier `id`, or returns a message if the order is not found;
- `orders orders.txt show` : shows the list of orders saved in the `orders.txt` file in the format shown below, sorted by identifier.

```
+-----+--------+------+--------+---------+
| id  | client | date | amount | details |
+-----+--------+------+--------+---------+
| ... | ...    | ...  | ...    | ...     |
+-----+--------+------+--------+---------+
```

The details area is free. You must provide options that the user can specify in order to use your program (for example, `-c "Martin family" -d 2025-12-24 -m 60.00 "2 × Prestige menu"`. Some values are constrained:

**date** a valid date
**amount** a valid amount

# 3 Python tooling

## 3.1 Reading and writing files

To make the project easy to implement, we recommend using a simple text format in which one order is stored per line. The following Python functions can then be used:

- `open(`*filename*`,` *mode*`)`, which opens the file in a given mode (`r`: read the file, `w`: create the file in needed and overwrite previous contents, `a`: create the file if needed and write new data at the end of the file).
- `read()` and `readlines()`, which return respectively the contents of the file and a list of content lines. ⚠ The file must be opened in `r` mode; if you read the contents of the file twice, the behavior will probably not be the one you expect.
- `write(`*string*`)`, which writes the given string into the file. ⚠ The file must be opened in either `w` or `a` mode.
- `close()`, which closes the file.

Here is an example showing the “pythonic” way of handling files. Note that Python automatically calls the `close` function when exiting the `with` block when you do this.

```
# Reading a file
with open('myfile.txt', 'r') as f:
    lines = f.readlines()

# Writing a file
with open('myfile.txt', 'w') as f:
    f.write('My contents\n') # \n is the new line character
```

### 3.2 Handling the command line

We suggest keeping things simple and using the standard `argparse` library. Its documentation is available online, and we provide a basic template in appendix A.

The way this library works is as follows:

- We create a parser with `ArgumentParser(<···>)`;
- We define a set of parameters to be given to the command line with the `add_argument(<···>)` method;
- We use the parser with `parse_args()`.

There are three types of arguments:

1. Options, which can be used at any time;
2. Positional options, which must be used at a specific position;
3. Subcommands, which themselves require a subparser to process what follows (which can be added with `add_subparsers(<···>)`).

## 4 Extensions

The various extensions are categorized below into four levels. Each extension may result in a change to the file format, how information is displayed, how command line arguments are handled, *etc.* Each extension is cumulative; once drawn, it must be implemented alongside the other extensions drawn.

The specification remains deliberately vague; your first task is, of course, to clarify requirements and specify the extension you are going to implement. Remember to *document* and *design*.

**Level 1**

a. Order status: each order can have a status (for example, *to be paid*, *paid*, *etc.*).

b. Adding labels: it is possible to associate a set of labels with an order (for example, *vegetarian*, *vegan*, *etc.*).

c. Differentiation between dates: each order can have a creation date, billing date, payment date, and delivery date, each of which can be different.

d. Adding a discount: good customers can be rewarded with a discount on the total price of their order.

**Level 2**

a. Orders breakdown: the details field now follows a standard format that allows further automated processing.

b. Configuration management: the program can now be configured using a configuration file tailored to its features.

c. Log management: every action performed is added to a log file, along with the results returned by the program.

d. Export formats: you can export information from the program into one or more formats of your choice.

**Level 3**

a. Automatic amount computation [*req.* extension 2.a]: the amount field is now automatically computed based on the order details.

b. Dishes breakdown [*req.* extension 2.a]: all dishes come with a list of ingredients and quantities.

c. Menu creation: it is possible to define menus made of several dishes, and use them when adding new orders (with the possibility of adjustments, for example, adding "no peppers" to the starter).

d. Search and filtering: it is possible to search and sort orders according to a set of criteria (for example, words contained in the details field, order status, date of service, *etc.*).

e. Files and contents compliance checks: the program includes mechanisms to check data compliance and integrity (are files in the expected format? are contents consistent? *etc.*).

**Level 4**

a. VAT handling [*req.* extension 2.a] : three VAT rates apply to food and beverages; the program allows you to specify them.

b. Workstation preparation [*req.* extension 2.a]: to simplify the cooking of items for sale, the program generates, for each dish, a document that helps tracking the steps and preparations remaining (for example, $n$ capons to cook, $m$ morel sauces, $k$ 4-person trays to portion, *etc.*).

c. Product traceability [*req.* extension 3.b]: to properly handle product recalls, each ingredient used in the preparation of dishes is traced with its supplier batch number.

d. Shopping list [*req.* extension 3.b]: the program automatically generates a shopping list summarizing all the ingredients to be purchased; ingredients already purchased are marked as such.

e. Import formats [*req.* extension 2.d]: you can import information from one or more formats of your choice; changes must be reflected in the original file.

Dependencies between extensions are summarized in figure 1. Note that extensions may have non-trivial interactions with one another, independent of these dependencies.

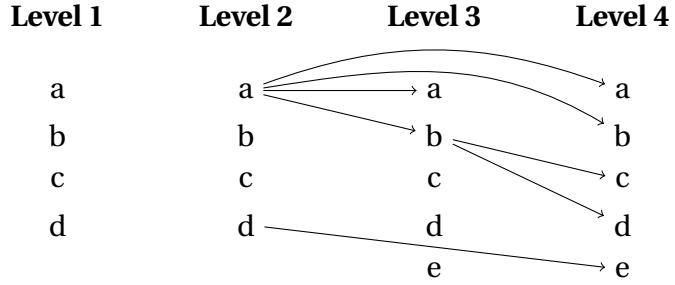


Figure 1: Extension dependencies

# 5 Validation criteria

1. Code quality:
   a. No global variable should be used, functions should take all necessary data as parameters and return all results.
   b. Functions are kept to a reasonable size.
   c. There is no unnecessary element (for example, unused function or variable).
   d. The names of variables, functions, and files are relevant (in line with their contents and usage). Code is written in a consistent manner.
   e. A functional style[1] is adopted (as an exercise in style), *i.e.*
      - Variables used are immutable; once created, they cannot be modified.
      - Loops are avoided, in favor of recursion. Iterators such as `map()` should be used.
      - Functions have no side effects (except I/O).
      - Consider introducing contracts/assertions.
      - Why not use lambdas to generalize certain functions...
   f. Functions are tested using unit tests that you provide.
2. Features:
   a. The code runs (easily) on a machine outside of your group.
   b. The code correctly supports the advertised features.
3. Documentation:
   a. Each function in your code must be described, along with its parameters, results, constraints, effects, possibly author(s)...
   b. Each usable command must be described in a help page (`-h` or `--help`).
   c. An up-to-date *readme* is provided, containing a quick description of the features and their usage.
4. Process and tools:
   a. You are using Git to work as a group, in a single repository per group.
   b. You are using a development environment suitable for programming.

[1] `https://docs.python.org/3/howto/functional.html`

# A Parser usage

## A.1 Example program

We assume that the parser is located in an `options.py` file.

```
#!/usr/bin/env python3
import commands
from options import create_parser

# Create command line parser
options = create_parser().parse_args()
try:
    # Read orders file, if it exists
    with open(options.file, 'r') as f:
        orders = f.readlines()
    # Run the command
    if options.command == 'add':
        commands.add(' '.join(options.details), options.file, orders)
    elif options.command == 'modify':
        commands.modify(options.id, ' '.join(options.details), options.file,
                        orders)
    elif options.command == 'rm':
        commands.rm(options.id, options.file, orders)
    elif options.command == 'show':
        commands.show(orders)

except FileNotFoundError:
    print(f"The file {options.file} was not found")
```

## A.2 Command file example

We assume that the file describing commands is `commands.py` and that core functions are implemented in `core.py`.

```
import core

# Implementation of commands, using functions in the core module
def add(details, filename, orders):
    with open(filename, 'a') as f:
        order = core.add(orders, details)
        f.write(order)
    print(f"Succesfully added order {order[0]} ({order[1]})")

...
```

## A.3 Parser example

The following code provides a starting point for implementing the parser (`options.py`).

```
import argparse

def create_parser():
    # Create command line parser
    parser = argparse.ArgumentParser(description='Simple order manager')
    # Add a positional argument (the file storing the orders)
    parser.add_argument('file', help='The orders file')
    # Add a subparser for subcommands
    subparsers = parser.add_subparsers(help='The commands to manage orders',
                                       dest='command', required=True)
    # Create parser for the add command
    parser_add = subparsers.add_parser('add', help='Add a new order.
      The rest of the command line is used for the order details, the default
      being "no details".')
    parser_add.add_argument('details', nargs='*', default="no details",
                            help="order details")
    # Create parser for the modify command
    parser_modify = subparsers.add_parser('modify',help='Modify an order given
      its id. The rest of the command line is used for the order details, the
      default being "no details"')
    parser_modify.add_argument('id', help="the order id")
    parser_modify.add_argument('details', nargs='*', default="no details",
                               help="the new details")
    # Create parser for the rm command
    parser_rm = subparsers.add_parser('rm', help='Remove an order given its id')
    parser_rm.add_argument('id', help="the order id")
    # Create parser for the show command
    parser_show = subparsers.add_parser('show', help='Show the orders')
    return parser
```