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

Codecamp

Back-to-school week – TAF DCL

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 7 students. Each student draws his or her group number from a hat. The draw is made by group, to optimize group heterogeneity. The aim is to force students who don't know each other to meet up.

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 two** of the group. Then, final validation must be **performed by an external review (by 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 task manager software.

The starting point is the same for all groups and involves implementing the following features: adding, modifying, deleting tasks. A task has a *description* (text without carriage returns) and an *identifier* given when it is created. A means of viewing the task list must also be provided.

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 tasks. 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 task).

This first version should therefore allow you to run the following commands: (we assume that your program name is `task`):

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

```
+-----+----------------+
| id  | description    |
+-----+----------------+
| ... | ...            |
+-----+----------------+
```

3 Python tools

3.1 Reading and writing files

To make the project easy to implement, we recommend using a simple text format in which one task 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. Adding project: each task can be associated to a project
b. Task status: each task can have a status (for example, *started*, *suspended*, *completed*, *cancelled*, *etc.*)
c. Priority: each task can have a priority
d. Adding realization contexts: a task can be associated to a context (for example *home*, *work*, *etc.*)
e. Adding labels: it is possible to associate a set of labels with a task
f. Adding users: each task can be associated to users

Level 2

a. Adding duration: an estimated duration and a realized duration are associated to the tasks
b. Adding deadline: tasks have a deadline and a completion date
c. Attaching a file to a task

d. Configuration management: the program can be configured using a configuration file tailored to its features (for example, the possible states of a task or the list of users can be configured).

e. History management: when a task is modified or deleted, it is stored in a history file

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

Level 3

a. Recurring tasks: a task can be declared as recurring (for example, weekly or monthly)

b. Meta-task: task templates can be predefined and then instantiated; some task parameters can be modified on instantiation

c. Future task: it is possible to define a task that can only be carried out in the future, starting from a date or the end of a task

d. Search and filtering: it is possible to search and sort tasks according to a set of criteria (for example, words contained in the description or a characteristic value)

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.*).

f. Dependency between tasks: a task can depend on another task to be able to start, or wait for another task to finish.

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. Functions are tested.
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. un *readme* à jour est fourni, il contient une description rapide des fonctionnalités et de leur usage.

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 tasks file, if it exists
    with open(options.file, 'r') as f:
        tasks = f.readlines()
    # Run the command
    if options.command == 'add':
        commands.add(' '.join(options.details), options.file, tasks)
    elif options.command == 'modify':
        commands.modify(options.id, ' '.join(options.details), options.file,
                        tasks)
    elif options.command == 'rm':
        commands.rm(options.id, options.file, tasks)
    elif options.command == 'show':
        commands.show(tasks)

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, tasks):
    with open(filename, 'a') as f:
        task = core.add(tasks, details)
        f.write(task)
    print(f"Succesfully added task {task[0]} ({task[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 task manager')
    # Add a positional argument (the file storing the tasks)
    parser.add_argument('file', help='The tasks file')
    # Add a subparser for subcommands
    subparsers = parser.add_subparsers(help='The commands to manage tasks',
                                       dest='command', required=True)
    # Create parser for the add command
    parser_add = subparsers.add_parser('add', help='Add a new task.
      The rest of the command line is used for the task details, the default
      being "no details".')
    parser_add.add_argument('details', nargs='*', default="no details",
                            help="task details")
    # Create parser for the modify command
    parser_modify = subparsers.add_parser('modify',help='Modify a task given
      its id. The rest of the command line is used for the task details, the
      default being "no details"')
    parser_modify.add_argument('id', help="the task 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 a task given its id')
    parser_rm.add_argument('id', help="the task id")
    # Create parser for the show command
    parser_show = subparsers.add_parser('show', help='Show the tasks')
    return parser
```