Michele Sarchioto
Roma 15/05/2019

FATTURAZIONE

Analysis and Development

OVERVIEW

A billing software (Fatturazione) should import a csv file with the following fields:

- NrFattura
- DataFattura
- ModalitaDiPagamento

The delimiter should be a semicolon (“;”).

“Modalità di Pagamento” is used to determine the due date of the payment (DSP, Data Scadenza Pagamento).

- DF (Data Fattura) ==> DSP=DataFattura
- DFFM(Data Fattura Fine Mese) ==> DSP = last day of the month of the invoice date “Data Fattura”. E.g. If DataFattura=23/03/2013 and ModalitàPagamento DFFM, then DSP = 31/03/2013
- DF60(Data Fattura +60gg) ==> DSP = two solar months should be added to the date. E.g. If DataFattura=23/03/2013 and ModalitàPagamento=DF60, then DPS 23/05/2013. If the resulting date is not valid (30/12/2018 => 30/2/2019) then 60 days should be added.

Invoices (Fatture) should be sorted by due date (DSP) in a file with three fields:

- NrFattura
- DataFattura
- DataScadenzaPagamento

The software module should perform the following tasks:

- Read an input file
- Calculate due dates (DSP)
- Sort invoices by due date
- Write an output file

Constraints:

- Object-oriented programming
- Python Standard Libraries only

HIGH LEVEL DESIGN

This project will run under Python version 2.7 because it's the same Python version as what Odoo v10.0 uses, although I prefer Python version 3.X because of its many improvements.

The project will be broken down in two python files plus a config file.

Python files:

- main.py
- fatturazione.py

The script "main.py" is used mainly to:

- Capture the command line input
- Let the user select an input file and a config file
- Initialize logger
- Create an instance of the Fatturazione class

Syntax to start the process:

python main.py --inputfile inputfile.csv --conf conf_fatturazione.json

the "--inputfile" option lets the user select an input csv file, and the "--conf" option lets the user select a configuration file. This configuration file is present mainly for future-proofing and extensibility. It allows us to make our code more flexible and dynamic. When, for example, we change the name of a csv column, if we hard coded the column name we have to change the code. If instead we make it a parameter we only have to change that parameter itself and not the code. The software will provide a set of default values if they are not present in the config file. The config file, when properly written, will always overwrite the default values.

Help Syntax:

python main.py -h

Help output:

```
usage: main.py [-h] [--inputfile INPUTFILE] [--conf CONF]
Process input file and config file
optional arguments:
  -h, --help        show this help message and exit
  --inputfile INPUTFILE  input file name
  --conf CONF            config file name
```

The script “fatturazione.py” contains the “Fatturazione” class that will perform the tasks specified by the requirements. Namely:

- Read an input file
- Calculate due dates (DSP)
- Sort invoices by due date
- Write an output file

I will also add some checks to catch common mistakes and unintended exceptions, so it will perform the following checks:

- Input file name is empty
- config file is empty
- Columns of the input csv consistency
- “Data Fattura” is valid
- “Modalità di Pagamento” is valid

The “Fatturazione” class will stop any time there is an error or exception, and will return the error message. In the output file, It will mark the invoices (Fatture) that have an invalid “Data Fattura” or “Modalità di Pagamento” but will not halt execution.

INPUT FILE

The input file has the following example csv structure:

```
"NrFattura";"DataFattura";"ModalitaDiPagamento"
"FATT-0001";"2019-04-03";"DF"
"FATT-0002";"2019-01-03";"DFFM"
```

The **first** field is an alphanumeric ID.

The **second** field is a date with the standard sql format, for convenience (YYYY-MM-DD). This is a very useful format because the chronological order is the same as the alphabetical order. Other date formats such as “DD/MM/YYYY” could be used with minor adjustments to the code.

The **third** field has only three different values:

- DF
- DFFM
- DF60

Refer to the overview for details about this field.

CONFIG FILE

The config file has the following structure:

```
{
  "inputCols": ["NrFattura", "DataFattura", "ModalitaDiPagamento"],
  "outputCols": ["NrFattura", "DataFattura", "DataScadenzaPagamento"],
  "validModes": ["DF", "DFFM", "DF60"],
  "idField": "NrFattura",
  "dateField": "DataFattura",
  "modeField": "ModalitaDiPagamento",
  "csvDelimiter": ";"
}
```

Keys description:

- inputCols: List of column names in the input file
- outputCols: List of column names in the output file
- validModes: List of valid values for “Modalità di pagamento”
- idField: ID field name
- dateField: "Data Fattura" field name
- modeField: "Modalita Di Pagamento" field name
- csvDelimiter: CSV column delimiter

This keys are mainly used to make the code more dynamic and to make it more flexible to changes. This allows us to leave the code untouched if we, for example, change the name of the field “Modalità di Pagamento”.

OUTPUT FILE

The output file has the following csv structure:

```
NrFattura;DataFattura;DataScadenzaPagamento
FATT-0002;2019-01-03;2019-01-03
FATT-0006;2018-12-30;2019-02-28
```

If there input date or the input “Modalità di Pagamento” are not valid, an error will be written instead of the “Data Scadenza Pagamento” field value. Eg:

```
FATT-0011;2019-99-04;Invalid Date at ID FATT-0011: 2019-99-04
FATT-0010;2019-04-03;Invalid Mode at ID FATT-0010: DF_ERROR
```

The output file naming convention is the following:

DPS_<input_file_name>_YYYY-MM-DD_HH-MM-SS.csv

Where YYYY-MM-DD_HH-MM-SS is the current date and time.

LOGGING

The main.py script will init a log file with the same name of the script “main.log”. The log has a daily rotation and the debug level is set at “logging.INFO”. The structure is as follows:

<Timestamp> - <File name> - <Log Level> - <Message>

ALGORITHMS AND SPECIFIC SOLUTIONS

DF Handler “dfHandler()”

This is just a straight copy.

DFMM Handler “dfmmHandler()”

For this “End of the month” handler, we can use the following standard library and method:

from calendar import monthrange

This method returns the first and the last day of every month, so we can use the second value to to overwrite the day of the date. Eg:

```
>>> from calendar import monthrange
>>> monthrange(2019,4)
(0, 30)
>>> monthrange(2019,4)[1]
30
```

DF60 Handler “df60Handler()”

For this “Add two solar month” handler, we can simply extract the month from the date string with a “split(‘_’)” operation, add “2” and check if the month is over 12. If it is we add 1 to the year, then we divide the month by 12 and use the remainder as new month value. We then rebuild the date. If the rebuilt date is not valid (30/12/2018 => 30/2/2019) then 60 days should be added instead. For this operation we can use the standard library “datetime” and the method “timedelta(days=60)”.

Sorting Handler “sortOutput()”

To sort a two-dimensional array, like the one we produce internally, we can use the the string function “sort()” with they “key” parameter. As we can read in the documentation:

> *“The value of the key parameter should be a function that takes a single argument and returns a key to use for sorting purposes. This technique is fast because the key function is called exactly once for each input record.”*

Even if each element is a list itself we can use an anonymous function (lambda) to extract a key value that can be used as a sorting reference for the entire element.

The following example uses tuples but works just as well with list of lists:

```
>>> student_tuples = [
...         ('john', 'A', 15),
...         ('jane', 'B', 12),
...         ('dave', 'B', 10),
... ]
>>> student_tuples.sort(key=lambda student: student[2])   # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

Input/Output file handler

To handle the CSV input/output file we can use the standard “csv” library. For the input file I’ve chosen the “csv.DictReader()” function that imports the csv data as a list of dictionaries. This is more verbose but allows us to process the CSV file even if we add a column or the columns themselves are not in order. Example element:

```
{'NrFattura': 'Mock-Fattura-1', 'DataFattura': '2019-05-06', 'ModalitaDiPagamento': 'DM60'}
```

The output data is written on a list of lists because it’s easier to sort it with that structure.

Input flags from user input

To capture the input flags (--inputfile, --conf) we can use a simple standard library “argparse”, which only requires to define the inputs and it even creates a help dialogue for free.

Function Pointer “self.funcPointer”

When you have a list of values that drive different behaviours it's useful to define a dictionary of functions. This is particularly useful to reduce clutter when trying to implement a multi-choice "switch-case" scenario. Simple example:

```
funcPointer = {
        'DF' : self.dfHandler,
        'DFFM': self.dffmHandler,
        'DF60': self.df60Handler
}
myMode = 'DFFM'
functionHandler = funcPointer[myMode]
functionHandler() # this calls self.dffmHandler()
```

TESTS

A simple test file ("fatturazione_test.py) has been provided to test the following functionalities:

- Date check
- DF handler check
- DFFM handler check
- DF60 handler check
- Sorting Check

The test is based on the library "inittest".

The code coverage, after running "main.py" is:

- main.py ⇒ 83%
- fatturazione.py ⇒ 84%

This script has been tested on Linux CentOS 7 and Windows 7, always with Python 2.7..

POSSIBLE IMPROVEMENTS

Refactoring

For one, refactoring could have been pushed further, there are some hard-coded values like the "60" (days) or some error messages could be part of a static list, maybe in a file or a database, but for this project I think that readability is better than perfect refactoring.

Multi-Thread

Because the CSV lines are not related to each other another, a multi-threaded approach could be used to speed up the execution. For Python though this is **not** a viable choice, because of GLI (Python Global Interpreter Lock). If a Python process spawns 10 threads only one thread can run at any time, because it acquires the GIL for itself and uses the python interpreter exclusively. If “true” multi-threading application is to be implemented, then we need to use the “multiprocessing” library that creates separate processes with separate interpreters. For the scale of this project, anyway, multi-threading is overkill. For files bigger than 500MB and on a server, maybe this development is worth the effort.

Date Formats

Date formats other than “YYYY-MM-DD” could be used by simply using the library “datetime” and converting the strings to datetime objects, and vice versa with the functions:

- strptime()
- strftime()

More Data Validation and Checks

We could insert more data validation, because, if a field is missing from a random row in the input data, it will probably trigger an Exception.