Computational Data Analysis

Coursework (2017)

Jonathan Schler

Introduction

The aim of the coursework is to design and develop a system for efficient management of the historic currency exchange rates.

The system consists of a module, containing most of the functionality, and several programs, allowing users to use all that functionality.

The problem

The module

Write a module `exrates` that implements fetching, saving, and analysis of the historical exchange rates. The module has to provide the following functions:

- `_fetch_currencies()` that fetches the currencies list from here (http://openexchangerates.org/api/currencies.json) and returns it as a dictionary (see the description of the format below).
- `_fetch_exrates(date)` that fetches the exchange rates for the date `date` from the Open Exchange Rates (https://openexchangerates.org/) website and returns it as a dictionary (see the description of the format below).
- `_save_currencies(currencies)` that saves the dictionary `currencies` in the currencies file, as described below.
- `_save_exrates(date, rates)` that saves the exchange rates data for date `date` in the appropriate exchange rates file, as described below.
- `_load_currencies()` that returns the currencies loaded from the currencies file.
- `_load_exrates(date)` that returns the exchange rates data for date `date` loaded from the appropriate exchange rates file.
- `get_currencies()` that returns the currencies loaded from the currencies file, as a dictionary. If the currencies file doesn't exists, the function fetches the data from the internet, saves it to the currencies file and then returns it.
- `get_exrates(date)` that returns the exchange rates data for date `date` loaded from the appropriate exchange rates file. If that file doesn't exists, the function fetches the data from the internet, saves it to the file, and then returns it.
- `convert(amount, from_curr, to_curr, date)` that returns the value obtained by converting the amount `amount` of the currency `from_curr` to the currency `to_curr` on date `date`. If `date` is not given, it defaults the current date (you can represent "today" as an empty string).
  The formula is `amount * to_value / from_value`, where `to_value` and `from_value` represent the values of the currencies `to_curr` and `from_curr`, respectively, on the date `date`.
  If the exchange rate for either of the currency codes `from_curr` and `to_curr` does not exist on the date `date`, the function must raise a custom exception `CurrencyDoesntExistError` with an appropriate message.

The functions have to raise proper exceptions on invalid input or if they cannot perform their tasks for some other reason (for example, a failed network connection or badly formated data).

When imported, the module has to read the App ID (read the notes below to see how to obtain one) from the file named `"app.id"` in the current directory. In case it fails, it has to print a descriptive message about the problem (preferably, using `sys.stderr`, but this is optional) and stop the program with the return value -17 (i.e., `sys.exit(-17)`). To make sure that the App ID is properly read, strip (https://docs.python.org/3/library/stdtypes.html#str.strip) it of the whitespaces after reading it.

The module must also create the data directory if it doesn't already exist. For this purpose, consider using the `os.makedirs` (https://docs.python.org/3/library/os.html#os.makedirs) and `os.path.exists` (https://docs.python.org/3/library/os.path.html#os.path.exists) functions.

The programs

Using the module `exrates`, write the following programs:

1. `cod` (which stands for Currencies On a Date), that inputs a date and prints the list of currencies for which there is a data on that date (i.e., the keys for the exchange rates dictionary on that date). The currencies should be printed in the format `"Name (code)"`, one per line, sorted by their code.
   Of course, the names are obtained from the currencies list. However, some may be missing there (for the currencies that don't exist anymore, like SIT that existed in the database between 2003-06-02 and 2006-12-22). Those should be printed as `"<unknonwn> (code)"`.
2. `ert` (which stands for Exchange Rates Table), that inputs a date and prints the exchange rates for that date in a tabular form, sorted by the currencies names, with the first column containing the string in the form `"Name (code)"` and the second one containing the exchange rate relative to the USD, aligned to the right and written to the 5 digits precision. The data has to be retrieved using the `get_exrates` function.
3. `convert`, that inputs two dates, an amount, and codes of two currencies (allow them to be given as lower or upper case strings). It then prints a table with the amount converted between those currencies, in both directions, i.e., conversion of `amount` from the first currency to the second one and from the second one to the first one (so, two lines should be printed), using the exchange rates on the given date period.
4. `hist` (which stands for History), that inputs two dates, a comma-separated list of currency codes (for example, `"ILS, GBP,eur,cAd"`), and a file name. It then saves the exchange rates relative to the USD between those two dates for the given currencies in CSV file of the given name (in the current directory if no path is given).
   Each date between the two given dates (including both of them) has to go in its own row, the first column must contain the date of that row, and each of the currencies must be in its own column. Only the currencies contained in the currencies list are taken into account (the rest are ignored; if there are no valid currencies, the input is considered invalid). If the value for a certain currency doesn't exist on some of the dates, save `"-"` as its value.
   The two input dates may be given in any order. If the first one is older, than the dates in the file must go in the chronological order. If the first one is newer, you are free to choose the order of the dates in the output file.
   Be careful when testing this program: each date's data has to be fetched, so you should avoid too big date spans as your access might be revoked if you clog the server!
5. `hmg` (which stands for History of one Month Graph changes), that inputs two integers, `year` and `month`, a comma-separated list of currency codes (for example, `"ILS,GBP,eur,cAd"`), and a file name. It then saves the graph of the `exchange rates changes` (in percentage (`%`)) relative to the USD for the given month and year, in a `png` file. Exchange rate change is defined as the ratio of the exchange rate on a given date compared to the previos exchange rate (**note** pay attention to proper handling of dates with missing exchange rates for certain currencies). For example: if today's ILS exchange rate is 3.8242 and previous rate was 3.8091 the `exchange rate change` is 0.396%. In addition, note that you need to start calculating the change only from the 2nd day (compared to the first day), first day is considered 0 change.
6. `analyze`, that inputs two dates, and list of currencies (allow them to be given as lower or upper case strings). It analyzes the change (in %) of the exchange rate during that period, and prints a table that lists for each currency the maximal exchange rate, the minimal exchange rate and the diff between the maximal exchange rate to the minimal exchange rate. The table is then sorted in descending order of the diff between max to min.

All the programs have to catch the functions' exceptions and handle them accordingly (at minimum, display an error message). You must also ensure that all the input is correct and, if it's not, ask for it again.

The input itself may be done with the `input` function or through the command-line arguments (or both), however you prefer.

Instructions

Due date

Last date of assignment of this work is 17-03-2017 at 12:00 (noon)

Data formats

The instructions below describe how the data should be processed. While the dates can be handled any way you want, as long as the functions accept a string input for them, keeping the currencies and the exchanges rates data in the dictionaries, as described below, is mandatory, as are the files' formats described under "Files".

Dates

Input and output dates are defined as strings of exactly ten characters in the format `"YYYY-MM-DD"`, where `YYYY` is a four-digit year, `MM` is a two digit month, and `DD` is a four digit day, each two neighbouring ones separated by a dash. For example, 31. January 1956. is represented as the string `"1956-01-31"`. Empty strings represent the current date (i.e., "today").

However, between input and output (i.e., when working with the data), it is advisable to keep the dates as `date` or `datetime` objects, but you are free to do it in any other way as well (as strings, as `(year,month,day)` tuples, etc).

The currencies list

The *currencies list* should be kept as a dictionary (so, not a Python list, despite its name), with the currency codes used as the keys and their respective names used as the values. They can be retrieved from here (http://openexchangerates.org/api/currencies.json) and, once retrieved, would look like this:

```
{
    "AED": "United Arab Emirates Dirham",
    "AFN": "Afghan Afghani",
    "ALL": "Albanian Lek",
    ...
    "ZMK": "Zambian Kwacha (pre-2013)",
    "ZMW": "Zambian Kwacha",
    "ZWL": "Zimbabwean Dollar"
}
```

While this list shows all the supported currencies, not all of them existed at all times, so various exchange rates tables will be missing some of those. How to handle those is explained in the notes below.

Exchange rates

The *exchange rates* for a certain date should be kept as dictionaries, with the currency codes used as the keys and their respective exchange rates (`float` numbers), relative to the United States Dollar (USD), used as the values. For example, that structure for the date 1999-03-26 would look like this:

```
{
    "ANG": 1.801151,
    "AUD": 1.572653,
    "AWG": 1.801151,
    ...
    "TWD": 33.163145,
    "USD": 1,
    "VEF": 0.582628,
    "ZAR": 6.241016
}
```

This was retrieved here (http://openexchangerates.org/api/historical/1999-03-26.json?app_id=) (the link will not work until you put in your own App ID, as explained below).

Files

All the data should be saved in a subdirectory with the name `"data"` in the current directory. The files with the exchange rates for date `YYYY-MM-DD` should be named `rates-YYYY-MM-DD.csv` (4 digits for year, 2 for month, and 2 for day, separated by dashes), and the file with the list of the currencies should be named `currencies.csv` (all of them in the directory `"data"`).

The currencies file should have columns "Code" and "Name" in which the currencies' codes and names are saved. The exchange rates files should have columns "Code" and "Rate", in which the currencies' codes and exchange rates (relative to USD) are saved. Both files should have column names in the first row.

The App ID file is named `"app.id"` and resides in the same directory as the module and the programs. It contains only App ID (which may or may not be surrounded by whitespaces, including newlines).

When working with files in subdirectories, be sure to use os.path.join (https://docs.python.org/3/library/os.path.html#os.path.join) and other `os.path` (https://docs.python.org/3/library/os.path.html) functions that you may need for constructing their full names.

Important: Do not use backslash `"\"` as a separator for path components and work only with relative paths as your programs will not be run on a Windows machine and your absolute paths will most likely not work.

Notes

Design of the module and the programs

To enable the fetching of data from the Internet, register at Open Exchange Rates (https://openexchangerates.org/signup/free) for free and write the obtained App ID in the file named `"app.id"` in the same directory where you create your module and your programs. The basic usage of this service is described in the site's API documentation (https://docs.openexchangerates.org/docs/).

The fetching of the data can be done using the module `urllib3.request` (https://docs.python.org/3/library/urllib3.request.html). The data returned by the "Open Exchange Rates" web site is returned in the JSON (http://json.org/) format which is easily converted to Python 3 data structures using the module `json` (https://docs.python.org/3/library/json.html).

All the used URLs should be global variables defined at the beginning of the module. It is advisable that the URLs for the exchange rates are formattable (i.e., they contain `"{}"` or something similar, with which the `format` function can easily insert a date, and App ID, or anything else it might need).

Date and time manipulations can be done directly (working with strings and integers), but it is much easier and more efficient to use the module `datetime` (https://docs.python.org/3/library/datetime.html) (trying to do it yourself will consume much more time). For example:

In [1]:

```
from datetime import timedelta, datetime

# This should be set soon after the module's doctring to allow easier modificati
on
date_format = "%Y-%m-%d"

# Some date to test the code on
some_date = "1956-01-31"
print("Some date as string:", some_date)

# Convert from string to an actual date
real_date = datetime.strptime(some_date, date_format)
print("\nThe same date as a datetime object:", real_date)

# Get the previous and the next day
prev_day = real_date - timedelta(days=1)
next_day = real_date + timedelta(days=1)
print("\nThe previous day as a datetime object:", prev_day)
print("The next day as a datetime object:", next_day)

# Convert them to strings
prev_day_str = prev_day.strftime(date_format)
next_day_str = next_day.strftime(date_format)
print("\nThe previous day as a string:", prev_day_str)
print("The next day as a string:", next_day_str)

# Which date is/was earlier?
if prev_day < next_day:
    print("{} is/was before {}.".format(prev_day, next_day))
else:
    print("{} is/was after {}.".format(prev_day, next_day))

print("There are {} days between previous and next day.".format((next_day - prev
_day).days))
```

```
Some date as string: 1956-01-31

The same date as a datetime object: 1956-01-31 00:00:00

The previous day as a datetime object: 1956-01-30 00:00:00
The next day as a datetime object: 1956-02-01 00:00:00

The previous day as a string: 1956-01-30
The next day as a string: 1956-02-01
1956-01-30 00:00:00 is/was before 1956-02-01 00:00:00.
There are 2 days between previous and next day.
```

Your module may have more functions than described above, if you need to use them in several programs or you simply feel that they fit in there.

For example, you might want to add a function for converting a string with a date to `datetime` object, and another one to convert a `datetime` object back to a string, both of them returning their input unprocessed if it's already of the proper type (use the function `isinstance` (https://docs.python.org/3/library/functions.html#isinstance) to verify this). Doing this allows you to keep the dates and work with them in a manner far easier and more natural than directly with strings (which are convenient only for input and output), while still allowing a user-convenient input in the string format.

Non-existing currencies

Through times, the currencies appeared and disappeard. For some of them, not all the data exists. For example, Serbian Dinar (RSD) was introduced to the Open Exchange Rates database on 2008-04-11. So, when you run `hist` for the dates 2008-04-05 through 2008-04-12, and for the currencies RSD and GBP, the output file should look like this:

```
Date,RSD,GBP
2008-04-05,-,0.500409
2008-04-06,-,0.500409
2008-04-07,-,0.50331
2008-04-08,-,0.507939
2008-04-09,-,0.506825
2008-04-10,-,0.506
2008-04-11,51.586749,0.506908
2008-04-12,51.586749,0.506674
```

So, RSD should be included (as it is one of the currencies listed in the currencies list (http://openexchangerates.org/api/currencies.json)), but for those dates for which its data is unavailable, a minus sign `"-"` should be put as its value.

The same should also be applied for the `hm` program. So, for the same list of currencies and the month 4 of the year 2008, the resulting file (with many lines ommited here) should look like this:

```
Date,RSD,GBP
2008-04-01,-,0.505054
2008-04-02,-,0.50392
...
2008-04-09,-,0.506825
2008-04-10,-,0.506
2008-04-11,51.586749,0.506908
2008-04-12,51.586749,0.506674
...
2008-04-29,50.513346,0.505873
2008-04-30,51.290013,0.505235
```

Deployment

Students need to submit the module and the programs through the Moodle system, without sending any extra files (i.e., submit only the seven `.py` files described above; no data files, nor an App ID file, nor any other files). Name the module and the program files exactly as they are named in this coursework, using only the lower case characters for the file names and extensions (for example, the sixth program should be `"hmg.py"`, and not `"HMG.PY"`, `"HMG.py"`, `"hMg.Py"`, `"hmg.PY"`, etc).

Marking

In order to be marked, all the code must run properly under **Python 3**. To verify that your Python installation is really Python 3 (and not ~~Python 2~~), you can run PyVer (https://github.com/vsego/misc/blob/master/pyver.py) and see what it prints. In case you have a wrong version of Python, uninstall Anaconda and install the proper version using the instructions from the course web page.

The module and the programs **must work**, i.e., the module must not crash the progam when being imported and the programs must run, all under **a standard Python 3 installation.** All of the modules in the Python 3 standard Library (https://docs.python.org/3/library/) can be used, along with the modules and packages from the SciPy ecosystem. If you are planning to use any other module, please ask via e-mail if it is OK.

All programs will be run via command line (and therefore have to run in this environment). For each program make sure that when you type from the cmd `python myscript.py` it works (where myscripts is any of the 7 files you submitted)

The solutions are viewed as the sets of elements: loading and reading data from the internet, saving the data to CSV files and reading from them, exchange rates conversion, programs, any out-of-the-functions work that module has to do, etc.

Each element of the coursework is marked separately, for a total of 70 points if all of them are done correctly. The students are allowed to skip any parts that they don't wish to do, for which they will get zero points, but the rest will still be marked. For example, you can skip loading the data from the internet and only load the data locally, from CSV files.

As a reminder, each module, file and function MUST be well documented. Minimal documentation consist of begining of file, beginning of module, documentation for each function, and every non-trivial code element

Each student is expected to do the assignment on his own. You can ask friends to help, but the final code need to be yours - You'll be proud of yourself once accomplished this! Students that will submit identical code - both students will get a 0 mark.

Have fun with it!

Concepts introduced in this coursework

These are remarks on some of the subjects covered above. There are no further problems here, only clarifications.

What the functions `get_currencies` and `get_exrates` are doing is called *caching* (not unlike that when the files are saved to a disk): when the data is requested, they read it from a local repository (cache); if the data cannot be found there, they compute it or fetch it, save it in the cache (for future requests), and then return it. This is a mechanism almost identical to what your browser does with most of the web pages and their elements.

The functions `_fetch_currencies`, `_fetch_exrates`, `_save_currencies`, `_save_exrates`, `_load_currencies`, and `_load_exrates` are considered *private*, in a way that they are not meant to be invoked outside of the module (and `from exrates import *` will not import them). Python does not provide mechanisms for restricting the access to any of the modules' functions, but the standard is that an underscore at the beginning of the function's name means an advice not to call that function outside of the module. If the name begins with two underscores, this advice is considered even more emphasised.