3 - SupportBank (Python)

Learning Goals

The goals of this exercise are to learn about:

- A variety of data formats (CSV, JSON, XML)
- Exception handling
- Logging
- Object Oriented Design

Problem background

Softwire has a dedicated support team. They're a fun-loving bunch and do lots of social things. They mostly operate on an IOU basis and keep records of who owes money to whom. Over time though, these records have gotten a bit out of hand. Your job is to write a program which reads their records and works out how much money each member of the support team owes.

Part 1: Reading CSV files

The support team keep their records in CSV format. Here are their records for 2014: Transactions2014.csv

Download the file and open it in Excel. Note that there's a header row, telling you what each column means. Every record has a date, and represents a payment from one person to another person. There's also a narrative field which describes what the payment was for.

Write a program which creates an account for each person, and then creates transactions between the accounts. The person in the 'From' column is paying money, so the amount needs to be deducted from their account. The person in the 'To' column is being paid, so the amount needs to be added to their account.

Your program should support two commands, which can be typed in on the console:

- **List All** should output the names of each person, and the total amount the owe, or are owed.
- **List [Account]** should also print a list of every transaction, with the date and narrative, for that account with that name.

Before you start, consider (and discuss with your trainer) what data type you should use to represent currency amounts.

Part 2: Logging and exception handling

Modify your program so that it also loads all of the transactions for 2015: DodgyTransactions2015.csv

You'll probably notice that some dodgy data is present in the file and your program fails in some interesting way. In your work as a software developer, users will try to cram any old data into your program. You need to make sure that you explain, politely but firmly, what they've done wrong.

Firstly, let's add some logging. For this we are going to use a module from the Python standard library called `logging`. Add the following lines of code to the start of your program:

```
import logging

logging.basicConfig(filename='SupportBank.log', filemode='w',
level=logging.DEBUG)
```

This code tells Python that when we use the `logging` module to write logs, we would like them to be written to the file `SupportBank.log`, and we would like all logs at level `DEBUG` or higher to be saved. To actually write some logs, you can use a line of code like this:

```
logging.info('An informative log!')
```

Look at the documentation from the `logging` module. Notice that there are several levels of severity at which you can log errors. Try logging

something at the point when your program starts up, and check that a log file has been created.

Now add logging to the rest of your program. Get to a point where you could work out what went wrong by reading your log files. (Don't try to fix the problem yet!)

Awesome. You now have forensic evidence to work out why things went wrong. Now change your program so that it fails gracefully and tells the user which line of the CSV caused the problem. Think about what failing gracefully means, in this situation. Should we import the remaining transactions from the file? Should we just stop at the line that failed? Could we validate the rest of the file and tell the user up-front where all of the errors are? What would make sense if you were using the software? Discuss with your trainers and work out what to do in this situation.

An aside on logging

Your program tells a story. An heroic story of a plucky little program, trying its best to withstand the user's effort to break it. Sadly, many programs do not survive the process, leaving behind only a cryptic error message and a bemused user.

Logging should be like your program's diary, recording interesting happenings and its fondest thoughts and desires.

> *Dear diary, the user started running me at 10:30am. She seems nice, I think we'll get along!*
>
> *Dear diary, she's playing with the transaction import screen! That's my favourite screen!*
>
> *Dear diary, ooh, she's feeding me a file. It's called DodgyTransactions2015.csv - that sounds tasty, I can't wait!*
>
> *Dear diary, line 57! That's not a valid datetime! Oh how it burns! Why, user, why?!?*
>
> *Dear diary, I die. :-(*

Often your logs will be all that's left when things go wrong. Make sure that they're descriptive enough so that you know why your program failed.

Part 3: What's a JSON?

So your program works great. The support team accountants can rest easy, knowing that all of their debts can be reconciled... Except for one minor detail. Back in 2013, the support team didn't store their records in CSV format. They stored them in a different format, called JSON. Open the 2013 transaction file Transactions2013.json and take a look. Hopefully it's fairly obvious how this data format works, and how the transactions in JSON format correspond to the old CSV transactions. JSON is one of the most widely used data formats worldwide. It's short for JavaScript Object Notation, and it's used on the web for servers to communicate with clients, and also with each other.

Next step - you guessed it. Modify your programme to accept JSON files in addition to CSV files. Take a look at the documentation for the `json` module in the Python standard library to get started.

Extend the interface of your program by adding a new command: **Import File [filename]** which reads the file from disk. You'll need different behaviour for CSV and JSON files, so make sure that you do the right thing based on the type of the file supplied.

> As you work through this part of the exercise, start thinking about the functions, classes and methods in your program and the relationship between them. Have you split your code up into a few modules to keep related functionality together? Try to keep your functions small and focused, and make their operation obvious based on their name. Where several functions do similar things, can you make them use a common function rather than repeating your code?

Part 4: XML!?

This is just getting silly. The support team's transactions for 2012 were in XML format. This stands for eXtensible Markup Language, and is another commonly-used data format. It supports lots of different features and is much more powerful than CSV or JSON, but as a result is somewhat harder to work with. No wonder they moved away to using JSON instead.

Open the 2012 transactions file Transactions2012.xml and take a look at the structure. Again, it should be fairly obvious how this corresponds to the other format. It may not come as a surprise to you now, but there's a module in the Python standard library that will parse XML files for you. Take a look at the documentation for the `xml.etree` module.

Stretch goals

Add a new command: **Export File [filename]** which writes your transactions out in a format of your choosing.