Data Structures

Overview

In this lab you'll process FIX (Financial Information eXchange) data. FIX is a standard data format for exchanging financial information electronically – we'll explain the details during the lab.

Your task will be to read a line of FIX data from a text file, split it into FIX tokens, and process each token. This will entail the use of various Python data structures (primarily dictionaries).

Source folders

Student folder : `C:\PythonDev\Student\05-DataStructures`

Solution folder: `C:\PythonDev\Solutions\05-DataStructures`

Roadmap

There are 5 exercises in this lab, of which the last two exercises are "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Getting started with FIX
2. Splitting a line into FIX tokens
3. Processing FIX tokens
4. Additional suggestions (if time permits)
5. Investigating other Python data structures (if time permits)

Exercise 1: Getting started with FIX

A common use of Python is to read data from a text file, process the data, and generate a report. This lab gives you an opportunity to perform these tasks on FIX data. FIX is a standard data format for representing financial information, so that the data can be exchanged electronically between IT systems. FIX has been around for decades and is widely used in financial systems.

If you want more information about FIX, there's an interesting article available on Wiki. See https://en.wikipedia.org/wiki/Financial_Information_eXchange.

Take a look in the *Student* folder. Notice the folder contains a file named `fixdata.txt`. Open this file in a text editor. The file contains a single line of text, which comprises a series of FIX tokens such as the following:

8=FIX.4.4

9=211

35=D

etc…

Each of these tokens is a *key=value* pair. The keys are special FIX numbers. Each of these numbers has a predefined meaning in FIX. For example:

The "8" token means "BeginString"

The "9" token means "BodyLength"

The "35" token means "MsgType"

etc… (see the Wiki site for more details)

Your task will be to write a Python script to extract all these tokens and process each one. This will entail two main steps:

- Split the line into tokens
- For each token, get the key (e.g. "8") and invoke the appropriate handler function. For example, if the key is "8", invoke a handler function named something like `proc8()`. The idea is that there'll be a dedicated handler function for each kind of token you're interested in, e.g. `proc8()`, `proc9()`, `proc35()`, etc.

To get started with your implementation, open `handlers.py` in a text editor. Follow the TODO comments in this file, to implement a series of handler functions for different FIX tokens. For example, here's a handler function to handle "8" tokens:

```
def proc8(value):
    print("In proc8 with value %s" % value)
```

Do likewise for some other FIX tokens, e.g. "9" and "35". Also define a catch-all function named `proc_other()`, which can mop up any other FIX tokens that you don't handle specifically.

Exercise 2: Splitting a line into FIX tokens

In this exercise, you'll write Python code to split a line of text into FIX tokens. You'll put all your code in `read_fixdata.py`, so open this file in a text editor now. Notice we've defined a couple of global variables near the top:

- `__tokenseparator` represents the "Start of header" token separator in FIX (it's actually ASCII hex code 01).
- `__kvseparator` represents the separator between the key and value within a token. It's just a simple "=" character. For example, in the token "8=FIX.4.4", the "=" character separates the key ("8") from the value (`"FIX.4.4").

Now see the "main" code near the bottom of the file. The code does the following:

- Opens `fixdata.txt` for reading.
- Loops through all the lines in the file (there's only one line in our simple example).
- Passes each line into the `tokenize_line()` function. You'll implement this function in this exercise, to tokenize a line of text and return a dictionary of key/value pairs.
- Prints the dictionary, for diagnostic purposes.
- Passes the dictionary into the `process_data()` function, to process all the key/value pairs. You'll implement this function in the Exercise 3 shortly.

You will now implement the `tokenize_line()` function, to tokenize an incoming line of FIX data and return a dictionary of key/value pairs. Follow the TODO comments in the code:

a. Split the line of text into tokens, using `__tokenseperator` as the separator.
b. Create an empty dictionary, ready to hold all the FIX keys/values.
c. Loop through all the tokens (from 2a above). Split each token at the = character, into a key/value pair, and then insert the key/value pair into the dictionary.
d. After the end of the loop, return the dictionary from the function.

Save your work and run the script as follows:

```
python read_fixdata.py
```

The script should display a dictionary containing all the FIX tokens as key/value pairs. Note that the display order is unpredictable – Python makes no promises about the order of items in a dictionary (nor should you care).

Exercise 3: Processing FIX tokens

In this exercise you'll implement the `process_data()` function, to process a dictionary of FIX key/value pairs. Follow the TODO comments in the code to achieve this task.

When you're done, save your work and run your script again. Verify your script detects the tokens you've chosen to handle (e.g. "8", "9", "35" etc.) and mops up all the other tokens in your "catch-all" handler function.

The output should look something like this (the order of tokens will be different for you):

```
{'40': '1', '11': 'TestOrder07', '38': '1000000', '55': '[N/A]', '8': 'FIX.4.4',
 '59': '0', '142': 'NEW YORK', '44': '95.5', '9': '211', '453': '1', '22': '4',
'35': 'D', '48': 'FR0010850701', '423': '1', '34': '34', '452': '11', '52': '201
60222-13:18:59', '54': '1', '60': '20160222-13:05:55', '448': 'autopmayo', '10':
 '069', '63': '0', '49': 'PMAYOAUTO', '56': 'TRADEWEBLDN'}
In proc_other with value 1
In proc_other with value TestOrder07
In proc_other with value 1000000
In proc_other with value [N/A]
In proc8 with value FIX.4.4
In proc_other with value 0
In proc_other with value NEW_YORK
In proc_other with value 95.5
In proc9 with value 211
In proc_other with value 1
In proc_other with value 4
In proc35 with value D
In proc_other with value FR0010850701
In proc_other with value 1
In proc_other with value 34
In proc_other with value 11
In proc_other with value 20160222-13:18:59
In proc_other with value 1
In proc_other with value 20160222-13:05:55
In proc_other with value autopmayo
In proc_other with value 069
In proc_other with value 0
In proc_other with value PMAYOAUTO
```

Exercise 4 (If time permits): Additional suggestions

There are a couple of Python language tricks and general programming techniques you can use to optimize your code:

- `tokenize_line()` currently creates an empty dictionary, then iterates through all the FIX tokens manually to add each token as a key/value pair to the dictionary. You can achieve the same effect in a single statement, by using a Python "comprehension". Think about how you might do this…
- `process_data()` currently has an `if/elif` statement that tests a key against a series of known values (e.g. `"8"`, `"9"`, `"35"` etc.) to determine which function to call. You can achieve the same effect much more efficiently by using a jump table, i.e. a dictionary that maps a key (such as `"8"`) directly to the handler function for that key (e.g. `proc8`). Note that a function name without parentheses designates the address of the function. Have a go at implementing this technique in your code…

We've provided a separate solution file that illustrates both these techniques, in case you get stuck or need some inspiration. See `read_fixdata_advanced.py` in the *Solution* folder.

Exercise 5 (If time permits): Investigating other Python data structures

If you have still some spare time, you might also like to investigate some additional techniques:

- The binary sequence types, i.e. `bytes`, `bytearray`, and `memoryview`.
- The `array` module, which is useful for storing numbers efficiently. For full details, see `https://docs.python.org/3.8/library/array.html`.
- The `collections` module, which provides specialized container types (i.e. alternatives to the general-purpose built-in containers `list`, `tuple`, `set`, and `dict`. For details, see `https://docs.python.org/3.8/library/collections.html`.