Comparison of shop simulation development using procedural Python, C, and Java

Fiachra O' Donoghue

G00398776

1 Introduction

1.1 The languages

This is a comparison of details of the development of a simple shop simulator in the Python, C, and Java programming languages, focussing on both general considerations and implementation details arising from the different paradigms within which one is implicitly forced to model a problem when using these languages. Python is here considered to be a solely procedural language, but a very high level one, with a huge variety of convenient constructions and abstractions. C is a low level procedural language, requiring manual memory management for non local variables, and with very few high level abstractions, such as functions and data structures. Java is a high level object oriented language with automatic garbage collection and a huge standard library of data structures and algorithms.

1.2 The problem

The problem specification is to create a shop simulation in which:

- Customer details, including shopping list should be loaded from a CSV file.
- The shop must maintain state; i.e. it must remember how much cash it has and its stock levels.
- The shop must react to requests which it both can and can't fulfill in an appropriate context sensitive manner.

1.3 Considerations

Arising from the problem specification above, some considerations regarding the implementation of a solution might be;

- Overall modelling of the problem. How might its parts be represented within the application? What sort of data structure might be used to represent, say, a shop, or a customer, or a business transaction?
- How might state be maintained. Must things be done differently from one paradigm to the other in order to achieve the same results?
- Is file I/O more or less the same process regardless of paradigm, or must the problem be reframed when switching languages.
- Similarly, text processing and parsing are fundamental to much computing activity. Which aspects of problem solving in this domain are resistant to paradigm switching, and which remain the same?

It seems probable that the answers to many of these questions rest more on levels of abstraction than on overarching programming paradigm. Much of computing involves solving numerous small problems and fitting those solutions together into larger and more complex solutions to smaller and more complex classes of problem. The problem that is solved by switching paradigms is the problem of how to fit those small solutions together — perhaps improving organisation for easy modularity, expansion, maintenance, or flexibility. The solutions to the small problems, of which all of the larger problems are constructed, such as how to read text from a file, sort strings, or sum a list of digits change more as a function of abstraction level than of philosophy.

2 Problem modelling

It seems obvious that to model a shop we need some kind of construction representing the shop and another representing the customer. The shop is a thing that has money; which is a number, a number of items for sale; which is a list of something, the ability to exchange some of those items for money; which is addition and subtraction, and, if it is to be sustainable, the ability to replenish its supply of items; which is addition. The customer is more or less the same as the shop except that it acquires items but does not sell them. An item is a thing that has a value.

It is significant that this problem was approached here by first proposing a solution in Python, then translating that solution to C, and then to Java. Moving from Python to C was simple in the sense that the overall solution was the same, even if the details were more complicated — that is, using the jigsaw analogy, the picture was the same but the pieces were much smaller. Moving from Python and C to Java, however, forced a change from one mode of representation to another. What was previously represented by structs and dataclasses was now represented by classes. However, in the case of the shop simulation, where there was little room — at least in the first iteration of a solution translated from C and Python — for the use of inheritance or polymorphism, two of the three so-called pillars of object orientation, it seems that these seemingly significant differences are superficial. [1]

More concretely, the problem was modelled in python as a series of blocks of functionality where, while some piece of functionality was being coded, it was discovered that a particular part of it was going to be needed in more than one place, that piece of functionality would be placed in a new function which, where possible, would be placed physically prior to the functions that

[1] Inicidentally, the third pillar of object orientation is encapsulation which was employed but it is hard to see that it made a difference in this case.

use it. This is not really necessary, even in C where a function must be declared at a point in the source before it is used, because a so-called function prototype can declare the function signature in advance of its full definition[2] For example we have a `stringify_product()`, which is used to build the string in `stringify_bill()`, which in turn is used to build the string in `stringify_customer()`.

In any case, in C and Python, the overall shape of the shop simulation is linear and generally cumulative. First some simple structures are defined: `Product`, `ProductStock` (a product with a quantity), `shop` and `customer`, along with a handful of frequently used small chunks of functionality for doing things like clearing the console or waiting for the user to press enter. This is followed by some functions for initalising the application — generating or loading the shop and customers. This is followed by a series of functions which output data of some kind and which often consume the data output by the previous functions to create their own output. Then there are the core pieces of functionality; `transact`, `restock`, `check_stock`, and culminating in `shop_visit` which incorporates all previous functionality to perform the core purpose of the application. Finally we have the simple functions which are exposed to the user and which provide an interface to doing more or less the same thing in a few different ways; `auto_mode`, `preset_mode`, `live_mode`, followed by a few ancillary functions for creating the user interface or generating data for future runs if the application.

The Java application, on the other, hand, has its functionality distributed across several classes; `Customer`, `Product`, `ProductStock`, `Shop`. These are directly analogous to the simple data structures defined in the procedural programmes. However these classes also contain functionality. So a `Product` can produce a string representation of itself and the `Shop` can restock itself. In practice, though, much of the functionality ends up in the `Shop` class because that's where it is most used and the code of that class looks remarkably similar that of the middle of the Python and C files.

3 I/O

File I/O is an important part of this application. The shop is generated from a CSV file on disk and customers are loaded from and saved to files. It is quite similar in all implementations. This, presumably, is either because loading bytes from disk is a fundamentally low level operation or because it is a fundamentally important operation in computing. Either way, even in C, it is simple to define a filhandle, open a file, read some lines, and close the file. In C, but not in Python or Java, memory must then be allocated to hold the data acquired if anything interesting is to be done with it, and that memory must later be freed — a process which can lead to some complex and time consuming debugging situations.

4 Text processing

All of the data loaded from disk in this application is text data; i.e. it is a series of characters. In Python and Java this is known as a datatype called a string which is a series of characters that can be manipulated more or less as if it was not an array like structure of individual characters.

[2]It was necessary to declare a `main_menu()` prototype in the C shop code.

C, however, does not supply this abstraction. A series of characters is an array, each element of which points to the address in memory where that character is stored.