Joining two lists together: a brief description of programming models

Introduction

There are many programming languages. Most, however, represent a particular model or paradigm of programming. These models represent a particular way of doing and thinking about programming and programming languages. Generally speaking, there are four models: imperative, object-oriented, functional and logic. We give a brief example and explanation of each model below, using a programming language that represents that model. The example will be how to join together two lists.

Imperative

Programming language: Python

```python
list1 = ['apple','banana','apricot']
list2 = ['pear','satsuma','kiwi','fig']

list3 = list1 + list2
print(list3)
```

An imperative programming language uses statements, which control the logic and the state of the program. In the example above, there are 4 statements in a sequence. It is typical of an imperative programming language to use the ";" character to mark the end of a statement (although not all imperative languages do this). The third statement changes the state of the program by changing the value of a variable. By state, we typically mean the data currently held by the program in its variables.

Object-Oriented

Programming language: Java

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;

public class Fruit {
    public static void main(String[] args) {
        String[] fruits1 = {"apple","banana","apricot"};
        String[] fruits2 = {"pear","satsuma","kiwi","fig"};
        List<String> list1 = Arrays.asList(fruits1);
        List<String> list2 = Arrays.asList(fruits2);
        List<String> list3 = new ArrayList<String>();
        list3.addAll(list1);
        list3.addAll(list2);
        System.out.println(Arrays.toString(list3.toArray()));
    }
}
```

This example closely resembles the imperative style. There are statements, and they are separated by ";" characters. There are also variables whose values can be changed – hence the program also

has state. But the object-oriented model builds on top of the imperative model by adding the concepts of class and object.

An object is some entity or thing that has three characteristics: state, behaviour and identity. An object can be identified as being a particular thing, within which there is some state and some behaviour that is specific to it. By behaviour, we typically mean the functions that are associated with the object. We can ask an object to do something or change something. Object-oriented programs are sometimes described as a set of collaborating objects, and we also sometimes talk about one object sending a message to another object to perform some service.

The role of classes is to say how objects should be created by describing their structure and what behaviour they should contain.

For example, the line 'list3.addAll(list1);' is essentially a request to the object whose identity is stored in the variable 'list3' to perform the 'addAll' function, using the data in variable 'list1'. The object referred to by 'list3' is described by the class 'List<String>' – this class will describe what behaviour we can expect from such an object.

The object-oriented model is an advance on the imperative model because it makes it easier to structure and organize large complex programs. Because of these advantages, and because they build on the well-established and well-known imperative model of programming, object-oriented programming languages are now the main programming model in use today.

Functional

Programming language: Haskell

```
list1 :: [String]
list1 = ["apple","banana","apricot"]
list2 :: [String]
list2 = ["pear","satsuma","kiwi","fig"]

joinList :: [String] -> [String] -> [String]
joinList [] f2 = f2
joinList (f:fs) f2 = f : joinList fs f2

list3 = joinList list1 list2
```

Now for something completely different. This is an example of a functional programming language. Functional programming languages are a completely different model of programming from imperative and object-oriented. For example, they have no state. They do not change the value of variables. In fact, there are no variables. It might seem that there are variables in the example code, but these are really zero-argument functions that always return the same value. 'list1', for example, is just an identifier permanently bound to the value '["apple","banana","apricot"]'.

There is an easier way to join two lists in Haskell, but I have given a basic implementation because it shows a particular way of iterating that is common in functional programming but rare in imperative and object-oriented programming: recursion. By recursion, we mean a function that is defined by calling itself. The 'joinList' function uses recursion to iterate over the first list until it reaches the end, at which point it returns the second list.

Lists in Haskell are created using the ':' operator – pronounced 'cons' (for 'construct', I believe). If we called 'joinList [1,2,3] [4,5,6]', then we see, if we trace the execution of the function, that we effectively create the following list:

1 : 2 : 3 : [4,5,6]

which, in Haskell terms, is the list [1,2,3,4,5,6].

Haskell functions are designed to look like mathematical functions. They usually look like the 'joinList' function, in that they first have a type signature such as

```
[String] -> [String] -> [String]
```

Which says that the function takes two arguments that are string arrays and returns a result which is also a string array. They then have a function body, which you can see in the code above.

As well as not having state, functional programming languages have partial evaluation. For example, we can write

```
(joinlist list1)
```

This is actually a function that has type [String] -> [String], a function that takes a string array and returns a string array. We have given 'joinList' just one of the two arguments it was expecting, but instead of reporting an error (as an imperative or object-oriented function would do), it just waits for the remaining argument.

These features, as well as others, make functional programming very powerful. The absence of state makes programs easy to reason about. Partial evaluation is a powerful tool that makes programs shorter and more expressive. However, there is a steep learning curve for functional programming, and although lack of state makes programs easy to reason about, it can make programs slower to run – if you want to change a value in a list, you have to make a copy of the old list with the change in the new version, rather than just update the old one.

Logic

Programming language: Prolog

```
append([], List, List).
append([Head|Tail], List, [Head|Rest]) :-
    append(Tail, List, Rest).

append([1,2,4],[4,5,6],AList)
```

Logic programming is a specialised model of programming based on formal logic. A Prolog program specifies a number of facts or assertions, such as the first two lines in the example above.

The first line specifies the fact that if the first value is an empty list, then the second and third values are equal. The second fact says that if the first value has a value at its head and a tail, then the third value has the same value at its head. Moreover, the fact if true implies the fact to the right of the entailment relation ':-'.

The third line is a query – can we find a resolution or value for 'Alist' such that it is true given the facts in lines 1 and 2? Clearly, we can if the value of AList is '[1,2,3,4,5,6]'. So a Prolog execution is really a search for such a value.

Conclusion

All the examples of code in this document do the same thing: they join together two lists to create a third list. The purpose of showing you these examples is to give you some idea that there are many ways of doing the same thing. Here, the different ways are in the form of different models of computing or programming.

Finally, we might ask why are there different models? A simple answer is that these models reflect different theoretical approaches to the general topic of what it means to compute something. Turing machines are one such approach – they are the root of imperative, as well as object-oriented, programming. Another theoretical approach to defining computation is the lambda calculus – and this is the root of functional programming. Finally, logic itself is the root of logic programming. Logic and the lambda calculus are more closely related to each other than each is to Turing machines, hence functional and logic programming are sometimes considered to belong to the same family of programming models, known as the declarative programming languages.