Lists as Records

Lists are data structures designed to keep an ordered sequence of data elements. A record is a list of attribute values about a single entity, typically with the attributes kept in a specific order. The similarities in those definitions suggest that records could be represented as a list.

Storing records as lists makes them easy to pass between functions in a program. The list structure will keep all the attributes together and in the correct order. A function that accepts a record as an argument only needs to keep track of the needed attributes and nothing more.

Note: as you study programming more you will encounter the concept of objects that also store records (called a data object). In some circumstances, objects will be the preferred way to go, but for simple applications, objects may be too heavy. Experience will help you decide.

Attribute Positions

For the code to use a list as a record, each attribute must be in a predefined position within the list. If the attribute is not at the correct index location, the code will not be able to interact with that attribute.

Consider an employee record. The record has an ID, last name, first name, department, and annual salary. An example might be: `[1, "Doe", "John", "Sales", 45000]`

For a program to interact with the record, each attribute must be in the correct place. The locations of the attributes must be the same for every record for this type of entity.

Missing Attributes

If a value is not available for an attribute, the location must be filled with a value that makes it clear that the value is missing. For example, all employee ID values must be greater than zero. If an employee does not have an ID yet, the record may reflect that with an invalid value such as zero or negative one:

```
[-1, "Smith", "John", "Sales", 45000]
```

If the data type is a string, consider using an empty string as a placeholder for the attribute. Assume that the employee does not have a department assigned. That may look like this:

```
[-1, "Smith", "John", "", 45000]
```

The important thing is that there must be some value in the location where the attribute belongs. If you leave an attribute out, the index values of the other attributes become wrong. For example, if we leave out the missing department, the salary gets the index value of department and there is no longer a value where the salary should be.

`[-1, "Smith", "John",  45000]` → This is wrong

Complex Attributes

In some cases, a record may have an attribute that contains multiple values. When this happens, we simply make the attribute another list.

Consider records where we have the quarterly sales for a salesperson. This salesperson may have zero or more sales, in addition to their ID, last name, first name, and department. To show the sales, we use a list for the attribute and put all the sales values inside.

```
[45, “Doe”, “Jane”, “Home Goods”, [125.00, 123.45, 10.00, 875.46] ]
[46, “Doe”, “John”, “Apparel”, [25.00, 37.65, 1.00] ]
```

Doing this lets the sales list be the element at index four (4). Then, the individual sales may be accessed using a second index. For example, if the current record is ID 45, then `record[4][2] = 10.00`.

Working with records like these is more complex, but with experience, you will develop the skills.

Constants for Indexes

Because many functions in the module will work with the records, it is important that they all use the same index values for the attributes. The easiest way to manage this is to define constants that hold the index values.

Continuing with the employee record, we may have constants:

- `ID_IDX = 0`
- `LNAME_IDX = 1`
- `FNAME_IDX = 2`
- `DEPT_IDX = 3`
- `SALARY_IDX = 4`

Note that the names are kept short but meaningful. In each case they contain “_IDX” to clarify that they are index values. We try to use shorter names because they will be placed inside of square brackets to access the attributes in the list. A long name will make the code more difficult to read.

Using these constants will clarify the code. There is a big difference between `record[3]` and `record[DEPT_IDX]`. As the code is maintained over its lifetime, these constants will greatly reduce confusion and the likelihood of an error.

Files as Lists of Lists

If one list can represent a record, then a list of lists can represent a list of records. A collection of related records is a file. Now we can read all the records contained in a file and store them in a two-dimensional list.

We often must work with collections of records at a time. File-based storage of the records is good, but we cannot interact with the data until we have read it into RAM. Using a list of records is how we can do this.

The basic algorithm for loading data is:

- Create an empty list to hold all the records – call this the records list (note the plural)
- Open the file for reading
- For each line in the file
    - Convert the line to a record (singular)
    - Append the record (singular) to the records list (plural)
- Close the file
- Return the records list to whoever called for it

Once the program has the list of records, it can loop through them to search for and modify records as needed.

Note that, as soon as any record is changed (updated, deleted, etc.), then the list of records no longer matches the file. The file should be the authoritative source for the data, so you need to store the records list back to the file as soon as possible.

The basic algorithm for saving the data is:

- Open the file for writing
- For each record (singular) in the records list (plural)
    - Convert the record into a string in the correct format for storage (e.g., CSV or similar)
    - Write the string to the file
- Close the file

By putting the read and save logic into functions of their own, the other functions do not need to contain that logic. The other functions just request the needed records and don't worry about the rest.

The benefits of this are:

1. Avoids code repetition by putting the commonly needed code into shared functions.
2. The reading and writing functions can be tested so that the other functions can rely on them.
3. Each function should have one responsibility and reading or writing files is a responsibility the other functions do not need to perform.
4. If you change the file format, you only need to change the functions that do the reading and writing of the files. The other functions still only work with records contained in lists. For example, we have used CSV, but we could change to XML and only need to rewrite the file-related functions.

Processing Records

All the usual list-processing operations work on records stored this way. The more important operations tend to involve looping.

A common operation is finding a specific record to perform some other work. It is best to do this by looping through the files and checking the attributes for a match.

The algorithm looks like this:

- Get the search value such as an ID
- Get the list of records
- For each record (singular) in records (plural)
    - If record attribute equals search attribute – found the record!
        - Perform the needed task with the record
        - Break the loop – no need to look at other records in most cases

Conclusion

Moving records as lists is a common task in programming. A well-designed program uses functions that work on parameter values. Looping through a list of records and passing those records as arguments to functions allows for fast and efficient operation.

File operations are slow and inefficient, but you often must work with files. Loading the data into RAM in a structured way improves the efficiency, at least a little. If a file is too big, this may not work!

In most applications, they don't care where you get the data. The date could come from a relational database management system (DBMS), across a network connection, or any number of different types of files. Only the functions that interact with the outside data source should know anything about that source. All other functions should only see the records as they are stored in RAM.

One of the big benefits of separating this read and write functionality from the rest of the program is that it allows you to change the data source later without needing to rewrite the rest of the program. If your CSV files have outgrown their usefulness and you need to switch to a database server, then you only need to recode the functions that talk to the data source. The rest of your program will remain oblivious.