Computer Science I
Girl Scout Cookies

CSCI-141
Lecture

10/24/2019

1 Problem

Your niece is a Girl Scout. She holds the troop record for most boxes of cookies sold, and she says that sometimes she has difficulties with all of her order forms. Customer handwriting can be hard to read, and she wants to make sure she gets all of the correct information back to the troop leader.

You offer to write a Python program to help her manage this information. The goal is identify data structures to store the details of each order, as well as to make it easy to look up information for an individual customer, and to report back the total sales for each type of cookie.

2 Analysis and Solution Design

The first task is to identify how to represent the data in this problem using an appropriate *data structure*.

An individual order contains the following information:

- name
- address
- total cost of order
- paid status
- alphabetical list of cookie variety purchases and box count for each

This can be broken down further. Each component of the list of cookies purchased contains the following information:

- cookie variety
- number of boxes

This representation includes two kinds of 'thing': an *Order*, and a *BoxCount*. The *Order* represents the information for an entire order, while the *BoxCount* refers to a number of boxes of a particular type of cookie.

These 'things' are known as *types*. The *concept of type* is not new; we have already used several of Python's built-in types, including `bool, int, float, str, list, and NoneType`. Now we will design our own data structure types.

2.1 Using a List Representation

We have seen a Python data structure that can represent a structure defined by parts: the **list**. We could represent the two values (cookie variety and count) for an individual `BoxCount` as a list, and alphabetically store the collection of all `BoxCount`s for a particular order in an outer list.

Finally, we could include all customer information for an order, including the `BoxCount`s, in an outermost list.

This yields the following example representation:

```
>>> cookie1 = ["Lemonades", 1]
>>> cookie2 = ["ThinMints", 2]
>>> purchase = [cookie1, cookie2]
>>> order = ["Ima Hungry", "3 Maple St.", 10.50, False, purchase]
>>> order
['Ima Hungry', '3 Maple St.', 10.5, False, [['Lemonades', 1],
    ['ThinMints', 2]]]
```

There are issues when using lists to represent the `Order` and `BoxCount` structures.

- It feels unnatural to use numeric indices when accessing the parts of an *instance* of the structure. It is easy to forget which index corresponds to which part since index numbers have *no mnemonic value*. Forgetting which part of a `BoxCount` is `[0]` and which part is `[1]` would introduce all sorts of errors.

- We also cannot distinguish between instances of different structures that have the same number of parts. For example, an automobile object instance such as `['BMW', 1972]` could be mistaken for a `['ThinMints', 2]` in a program that concerns buying and selling cars. An instance of a different structure that also has two parts and is composed of a string, and an integer, should be easy to distinguish from an instance of a `BoxCount` structure.

Ideally, we would like a data structure to represent `Order`s and `BoxCount`s that would give us the following features:

1. *Named Slots:* We desire variable names instead of indices associated with each of the components of the data structure. The structure should prevent addition of any additional information beyond the specified components.

2. *Easy Printability:* We would like to have the data structure print itself in a meaningful way. This should include the names of each component part for readability.
3. *Type Enforcement:* We would like to specify type restrictions on the kind of values that may be stored for each component of the data structure. Python is designed as a *dynamically typed language*. That means that Python allows a variable to hold a value of any type *and change the value* to a different type at any time. A dynamic type change can cause trouble for programmers, and it would be nice to restrict things so that a variable can refer to values of a more constrained type.[1]

2.2 User Defined Structures Using `class` Keyword and `dataclass` Annotation

Python 3.7 added a special *annotation*[2] to classes that makes them a choice for creating user defined structures. To use `dataclass`, we must import `dataclasses.dataclass`, which Python 3.7 introduced.

Here is how to create a new structure type, and add fields and their respective types. We define the `BoxCount` data structure like this:

```
from dataclasses import dataclass

@dataclass
class BoxCount:
    """
    BoxCount represents some number of a named cookie box.
    cookie is the name of the box.
    count is the number of those boxes.
    """
    cookie: str
    count: int
```

The `@` sign is the syntax of a *decorator* annotation in Python, and this decorator adds type annotations to a `class` definition.[3] The structure will then have expected types associated with the named `cookie` and `count` *slots* of `BoxCount` type instances.

We define the `Order` data structure as follows:

```
@dataclass
class Order:
    """
    Order represents an order of cookie boxes from a person.
```

---

1. While we would like this capability, type enforcement is not available from the tools we will use to build our own data structures in Python.
2. PEP3107 describes a Python syntax for adding arbitrary metadata annotations to Python functions.
3. For the curious, see `https://www.quora.com/What-is-a-decorator-in-programming` for a brief description and example.

```
    name is the name of the person.
    address is the address of the person.
    cost is the total cost of all the boxes in the order.
    paid is True if the order has been paid.
    cookies is the list of BoxCount items.
    """
    name: str
    address: str
    cost: float
    paid: bool
    cookies: list
```

After defining the `BoxCount` and `Order` data structures, we can create individual instances of each data structure with initial values as in the following examples:

```
>>> box1 = BoxCount("Lemonades", 1)
>>> box2 = BoxCount("ThinMints", 2)
>>> order1 = Order("Ima Hungry", "3 Maple. St.", 10.50, False, [box1, box2])
>>> print(order1)
Order(name='Ima Hungry', address='3 Maple. St.', cost=10.5,
paid=False, cookies=[BoxCount(cookie='Lemonades', count=1),
BoxCount(cookie='ThinMints', count=2)] )
```

We create an instance of the data structure by using the data structure name as the name of a function call, passing arguments inside the parentheses, and assigning the result to a reference variable.

It is important to pass the arguments in the same order that they are defined for the data structure.

The example also shows output of printing an instance of a data structure. While the output would normally appear on a single line, the line has been folded for document readability.

We must provide initial values for all components of the data structure when creating an instance:

```
>>> order2 = Order("", "", 0.0, False, [])
>>> print(order2)
Order(name='', address='', cost=0.0, paid=False, cookies=[])
>>> order3 = Order()
TypeError: __init__() missing 5 required positional arguments: 'name',
'address', 'cost', 'paid' and 'cookies'
```

[4]

One of the desired features for our data structure design is the ability to access components of the data structure by name instead of using indexing notation. With dataclasses, we access components of a data structure using **dot notation** similar to what we use to call

4. The error line has been folded.

functions located in an imported module. The left-hand side of the 'dot' is the structure instance, and the right-hand side of the 'dot' is the *slot name*.

Using the data structure instances designed above, we have the following examples:

```
>>> box1.cookie
'Lemonades'
>>> order1.name
'Ima Hungry'
>>> order1.cookies[1].cookie
'ThinMints'
```

Here is how to interpret the composition of dot notation and list indexing in the above example:

- `order1.cookies` is a reference to a list;
- `order1.cookies[1]` is a reference to the second element of that list, which is an instance of `BoxCount`; and
- `order1.cookies[1].cookie` accesses the `cookie` slot of that `BoxCount` instance.

2.3 Storing all Orders

With a Python dictionary, we can use a customer name as a unique key, and the `Order` object as the associated value.[5]

We can store all of the orders in a Python dictionary, using the string name as the key and the `Order` as the associated value. That way we can easily access an order by providing the person's name as the key.

2.4 Implementation

A solution for this problem is in `girl_scouts.py`.

These are the major tasks:

- Reading order information from a file;
- Computing the total number of boxes for each variety of cookie purchased across all orders; and
- Merging repeat orders from the same person.

2.4.1 Reading Order Information

The provided code illustrates how to read an input file with a specified format, and converts that input into corresponding `Order`s.

An example input file is shown below. The input file structure has an arbitrary number of orders, and each order spans five consecutive input lines.

---

5. For simplicity, we assume that any given name appears only once across all orders.

```
Ima Hungry
3 Maple St.
10.50
False
Lemonades 1 ThinMints 2
Cookie Monster
123 Sesame Street
35.00
True
CaramelDeLites 3 PeanutButterPatties 4 Shortbread 2 Thanks-A-Lot 1
Chuck Letty
101 Chocolate World Way
7.00
True
MangoCremes 2
Juliette Gordon Low
420 5th Avenue
17.50
True
Lemonades 2 Shortbread 1 ThinMints 2
```

The first four of these five lines contain the customer name, address, total order cost, and paid status, respectively. The final line contains a listing of the various cookie varieties and counts purchased. For simplicity, the cookie varieties are restricted not to contain any whitespace within the name. Furthermore, the cookie varieties are assumed to be listed in alphabetical order.

2.4.2 Computing the Total Boxes

For each variety of cookie purchased, we have to count all the boxes in all the orders.

The `compute_box_totals` function loops over all the orders. For each order, it also loops through the boxes to add the counts.

2.4.3 Gathering Repeat Orders

It is possible that the input file may have a name appear more than once, representing repeat orders by a single customer. For simplicity, we assume that repeat orders have the same paid status as the initial order, and aggregate the boxes into a single order for that customer.

The code for this operation is found in the functions `repeat_order` and `merge_lists`.

3 Testing

The provided file `orders.txt` can be used to test the program functionality.

Initially, we want to test that `BoxCount`s and `Order`s are being constructed correctly, and that non-repeat orders are correctly added to the dictionary.

Next, we can test that the function to calculate totals is correct. We want to have input that includes cookie varieties that are repeated across different orders as well as cookie varieties that are unique to a single order.

Finally, we can test that repeat orders are handled correctly. We want to check that the total cost for the customer is correctly updated, and that the list of cookie varieties and counts is correct. The repeat order should consider several cases to make sure that the merging algorithm is correct, including: the repeat order having a unique cookie variety that is alphabetically first, or alphabetically last, or alphabetically in the middle, as well as having duplicate cookie varieties.