Computer Science I — CSCI-141
Moving Day — Lab 6

10/13/2020

1 Problem

It's moving day and you need to get all of your stuff packed in boxes. The moving company won't take any loose items - anything that doesn't make it into a box will be left behind.

You have a collection of boxes at your disposal. Each box has a corresponding capacity that represents the maximum weight of items that it can support without falling apart. You also have a collection of items that need to be put into the boxes. Each item has an associated weight. (Do not worry about the item or box volume for this problem.)

You don't want anything to be left behind! Can you come up with a strategy to place items in boxes so that everything will fit?

Of course you can exhaustively consider every possible combination of items that can be placed in each box. That will get you to the correct result, but you'll never finish in time.

Instead, you focus on possible *greedy* algorithms for the task of filling up the boxes. A greedy algorithm builds up a solution in steps, making fast, local decisions based on current conditions, sticking with those decisions and moving on. A greedy algorithm may lead to a globally optimal solution; . . . or it may not.

The movers will arrive soon. And you need to come up with a greedy strategy for filling boxes.

2 Problem-solving Session (15%)

Work in a team of students as determined by your instructor. Each team will work together to complete the following activities.

1. Here is one strategy for filling the boxes:

   > Iterate through the items one by one, from greatest weight to least. For each item, identify the box with the greatest remaining allowed weight that can support the item, and place the item in that box.

   Given these things:
   ```
   Cheese 9
   Barley 6
   Anise 2
   Doughnuts 5
   Eggplant 8
   ```
   Apply the strategy to pack these things into three boxes of capacity 15, 10, and 5. Identify what was packed in which box.

2. Using the same example as the above, describe a different strategy that fails to fit everything. Identify what is packed in each box and what thing(s) that are left out.

3. What user-defined data structure types will you define to represent the box and other things in this problem? For each, write a Python statement to define the data type. Recall that these definitions involve the following information:
   - A name for the type
   - A data type for each component of the data structure
   - A name for each component of the data structure

4. Write a Python function that reads an input file and constructs and returns appropriate data structures.
   Assume the input file has the following format: the first line is a space-separated listing of the capacities of the boxes. Each of the remaining lines contains the name of an item, followed by its weight. For example:
   ```
   12 14 9
   Computer 5
   Books 4
   Dumbbells 10
   Microwave 7
   Stereo 8
   ```
   The recommendation on the next page will be useful for this problem.

2.1 Recommendation: Maker Functions

Any time a user-defined data structure is initialized with default values for some of its components, it is worthwhile to write a *maker* function for that data structure. The maker function takes as input only the non-default values with which the data structure will be initialized. The function then creates the instance, populates it using the default and non-default values, and returns an instance of the data structure with all the slots filled. The typical definition pattern for a *maker* function is:

```
def make_<<structure-name>>( ... ):
```

where `<<structure-name>>` is the name of the structure that the function will construct.

As an example, consider a structure that maintains information about a bowling team. This structure has a team average slot and contains a list of bowler structures, each of which has an average score. When constructing the team with a list of bowlers, you want to compute the team average using the list of bowlers. A *maker* function could process the list to calculate the value for the team average and store it in the slot of the bowling team instance. The user would not have to calculate the average before passing the bowler list into the maker function.

This means we might have this:

```
def make_bowling_team( name, members ):
    team_average = # loop through members list to compute
    return Team( name, team_average, members)
```

3 Post-PSS Activity (10%)

Tasks:

1. Read the entire assignment to learn the strategies to implement and the program functionality.

2. Create a new project for this assignment and open a new file named `moving.py`.

3. Write the code that defines your data structures.

4. Write the function from problem solving to read the input file.

5. Write a `main` function that prompts for the file name, reads the file, and prints the data structures that it created from the file's content.

   The printing will show that you have constructed the data structures properly, but it will not be a part of the implementation. You can leave the printing in until you replace it with the functionality of the final implementation.

Upload `moving.py` to the post-PSS dropbox by the deadline.

4 Implementation (75%)

Each student will *individually implement* and submit their own solution to the problem as a Python program named `moving.py`.

4.1 Requirements

Your program must implement the following three strategies for filling the boxes. Use any sorting implementation you desire.

- **Greedy Strategy #1 (Roomiest):** Sort all items by decreasing weight. Iterate through the items one by one, from largest weight to smallest. For each item, identify the box with the greatest remaining allowed weight that can support the item, and place the item in that box. Ties can be broken arbitrarily. If no box can support the item, it is not placed in any box. Continue until all items have been considered, and either placed in a box or left out.

- **Greedy Strategy #2 (Tightest Fit):** Sort all items by decreasing weight. Iterate through the items one by one, from largest weight to smallest. For each item, identify the box with the least remaining allowed weight that can support the item, and place the item in that box. Ties can be broken arbitrarily. If no box can support the item, it is not placed in any box. Continue until all items have been considered, and either placed in a box or left out.

- **Greedy Strategy #3 (One Box at a Time):** Sort all items by decreasing weight. Fill the boxes one by one. For each box, iterate through all remaining items (not yet placed in a previously considered box) one by one. If there is room for an item to be placed in the current box, do so.
  (Note that this strategy can lead to different results depending on the order that boxes are considered, if the boxes have different weight capacities. Don't worry about this detail; process the boxes in any order. All of the boxes have equal weight capacity in the test cases we will use.)

The program must prompt the user to specify an input file. It must read the input file, evaluate each of the three strategies on the input data, and produce appropriate output messages.

Several input files are provided. The files illustrate cases for which each individual strategy succeeds while the others fail. They also include one case in which all strategies fail despite it being possible to pack all of the items in the given boxes. The input files can be accessed at: `http://www.cs.rit.edu/~csci141/Labs/06/TestData.zip`.

Below is one example of an input file, and output of appropriate information by the program. The output does not need to match exactly what is below, but it must contain the same information.

Example input file `items1.txt`:

```
12 12 12
Abacus 3
Blender 5
Chessboard 3
Dishes 6
Engraving 9
Flowerpot 3
Guitar 3
Helmet 4
```

Corresponding output information:

```
Enter data filename: items1.txt

Results from Greedy Strategy 1
All items successfully packed into boxes!
Box 1 of weight capacity 12 contains:
  Engraving of weight 9
  Chessboard of weight 3
Box 2 of weight capacity 12 contains:
  Dishes of weight 6
  Abacus of weight 3
  Flowerpot of weight 3
Box 3 of weight capacity 12 contains:
  Blender of weight 5
  Helmet of weight 4
  Guitar of weight 3

Results from Greedy Strategy 2
Unable to pack all items!
Box 1 of weight capacity 12 contains:
  Engraving of weight 9
  Abacus of weight 3
Box 2 of weight capacity 12 contains:
  Dishes of weight 6
  Blender of weight 5
Box 3 of weight capacity 12 contains:
  Helmet of weight 4
  Chessboard of weight 3
  Flowerpot of weight 3
Guitar of weight 3 got left behind.

Results from Greedy Strategy 3
```

```
Unable to pack all items!
Box 1 of weight capacity 12 contains:
  Engraving of weight 9
  Abacus of weight 3
Box 2 of weight capacity 12 contains:
  Dishes of weight 6
  Blender of weight 5
Box 3 of weight capacity 12 contains:
  Helmet of weight 4
  Chessboard of weight 3
  Flowerpot of weight 3
Guitar of weight 3 got left behind.
```

Your results may be slightly different for the input depending on how the sorting algorithm handles ties. This should not affect the final result (success packing everything).

4.2 Grading

The 75% implementation grade is based on these factors:

- 15%: The program uses `dataclasses` to define and use data structures.
- 10%: The program correctly reads input files.
- 45%: The program correctly computes the result for each strategy.
- 5%: The program generates appropriate output.
- Failure to follow course web site style and documentation guidelines, or to follow submission directions, will incur a 10% penalty.

4.3 Submission

Zip the `moving.py` file, call it `lab06.zip`, and submit it to the **MyCourses dropbox** by the due date for this assignment. **Use zip only. Do not use another compression mechanism.**