ICS-33: In-Lab Programming Exam #1

Summary:

This exam is worth 100 points. It requires you to write five functions according to the specifications below. Each problem is graded on correctness only (how many tests it passes); each problem will be worth 20 points.

The module you will download

1. Defines each method with reasonable annotated parameter names (which you can change) and a body that is **pass** (thus, it returns **None** and will pass no tests).
2. Includes a script that tests these functions individually on different legal inputs, printing the results, and printing whether or not they are correct; if they are not correct, further information is printed.

The next five sections explain the details of these functions. You should probably try writing/testing each function as you read these details. Spend about 15-20 minutes on each function; at that point, if you are not making good progress toward a solution, move on to work on later functions (which you might find easier) and return if you have time. Each problem is graded based on the percentage of tests it passes.

A Data Structure for Warehouse Management

Suppose that a company maintains multiple **warehouses**. Each warehouse stocks various **products** (not all warehouses stock the same products). The **inventory** of a product is the amount of any product available at a warehouse. For example, a warehouse in **Irvine** may stock a **brush** product, and it may have an inventory of **3** (3 brushes in the Irvine warehouse).

We will repersent all this information in a **dictionary** whose keys are the warehouses: associated with each warehouse is an inner **dictionary** whose keys are the stocked products (and whose associated values are the inventory of that product in the warehouse). The inventory must always be a **non-negative** value; an inventory of **0** is legal.

For example a simple/small database might be.

```
db = {'Irvine':  {'brush': 3,    'comb': 2,    'wallet': 2},
      'Newport': {'comb': 1,     'stapler': 0},
      'Tustin' : {'keychain': 3, 'pencil': 4,  'wallet': 3}}
```

This data structure means that

1. The **Irvine** warehouse stocks **3 brush**es, **2 comb**s, and **2 wallet**s
2. The **Newport** warehouse stocks. **1 comb**, and **0 stapler**s.
3. The **Tustin** warehouse stocks **3 keychain**s, **4 pencil**s, and **3 wallet**s.

I have printed this dictionary in an easy to read form. Python prints dictionaries on one line, and can print their key/values in any order, because dictionaries are not ordered.

1: Details of read_db:

The **read_db** function takes an open-file as an argument; it returns a dictionary (**dict** or **defaultdict**) in the form illustrated above).

The input file will consist of some number of lines; each line specifies a product, followed by any number of warehouses and the inventory at each warehouse (always a non-negative integer; don't check it). All these different items are separated by colons (**:**) and contain no internal spaces.

For example, the **db1.txt** file contains the lines

```
brush:Irvine:3
comb:Irvine:2:Newport:1
wallet:Irvine:2:Tustin:3
stapler:Newport:0
keychain:Tustin:3
pencil:Tustin:4
```

For this file, **read_db** returns a dictionary whose contents are:

```
{'Irvine':  {'brush': 3,    'comb': 2,    'wallet': 2},
 'Newport': {'comb': 1,     'stapler': 0},
 'Tustin' : {'keychain': 3, 'pencil': 4,  'wallet': 3}}
```

So, each line from the file will put some information into the dictionaries. Note that each line in the file will have a unique product: you will never see the same product on multiple lines; also, each line will contain no duplicate warehouses: you will never see multiple appearances of a warehouse on the same line.

Some warehouse may inventory the same products, some warehouses may inventory unique products: e.g., **wallet**s are warehoused in both **Irvine** and **Tustin**; **keychain**s are warehoused only in **Tustin**.

I have printed this dictionary in an easy to read form. Python prints dictionaries on one line, and can print their key/values in any order, because dictionaries are not ordered.

2: Details of inventory:

The **inventory** function takes a **dict** argument in the form illustrated on the top of page 2,

> each key in the **dict** is a **str** (the **warehouse**) whose associated value is another **dict**: this inner **dict** associates a **str** (the **product**) with an **int** (the **inventory**)

and returns a sorted **list** of **2-tuple**s: each **2-tuple** is a **product** followed by the combined inventory of that product: the sum of its inventory available in **all the warehouses**. The **list** is sorted in descending order of **inventory**: for equal inventory, it is sorted in ascending alphabetical order of **products**.

Calling this function with the **db** dictionary shown on the top of page 2

```
inventory( db )
```

returns a **list** of the following **2-tuple**s.

```
[('wallet', 5), ('pencil', 4), ('brush', 3), ('comb', 3), ('keychain', 3), ('stapler', 0)]
```

In this dictionary, there are **3 comb**s in the inventory: **2 comb**s are in the **Irvine** warehouse and **1 comb** is in the **Newport** warehouse. There are two other products with a combined inventory of **3** and they appear in the **list** in alphabetical order: **brush**, then **comb**, then **keychain**

3: Details of resupply:

The **resupply** function takes a **dict** argument in the form illustrated on the top of page 2,

> each key in the **dict** is a **str** (the **warehouse**) whose associated value is another **dict**: this inner **dict** associates a **str** (the **product**) with an **int** (the **inventory**)

and a second argument that is an **int** specifying the desired minimum inventory of every **product**: a **warehouse** must

have at least that inventory for every product in the **warehouse**.

It returns a dictionary (**dict** or **defaultdict**): each key in the dictionary is a **str** (the **product**) whose associated value is a **set** of **2-tuple**s: a **warehouse**, followed by the number of that **product** to order at that **warehouse**, to reach the required minimum inventory.

Calling this function with the **db** dictionary shown on the top of page 2

```
resupply( db , 3 )
```

returns a dictionary whose contents are:

```
{'comb': {('Irvine', 1), ('Newport', 2)}, 'wallet': {('Irvine', 1)}, 'stapler': {('Newport', 3)}}
```

I have printed this dictionary in an easy to read form. Python prints dictionaries/sets on one line, and can print their key/values in any order, because dictionaries/sets are not ordered.

Note that for **comb**s, **Irvine** needs to resupply **1** and **Newport** needs to resupply **2** to bring their inventories up to **3**. For **wallet**s, **Irvine** needs to resupply **1**. For **stapler**s, **Newport** needs to resupply **3**. For **pencil**s, **brush**es, and **keychain**s, all stores stocking these products have an inventory **>=3** so those products don't appear as keys in the resulting dictionary.

4: Details of update_purchases:

The **update_purchases** function takes a **dict** argument in the form illustrated on the top of page 2,

> each key in the **dict** is a **str** (the **warehouse**) whose associated value is another **dict**: this inner **dict** associates a **str** (the **product**) with an **int** (the **inventory**)

and a second argument that is a **list** of **2-tuple**s: two **str**s (**warehouse**, **product**). Each represents the sale of the specified product at the specified warehouse.

It **mutates** the **dict** argument, decreasing by **1** the inventory of the specified product at the specified warehouse, each time one product is sold: these sales occur in the order shown in the **list**.

It returns a **set** of **2-tuple**s for purchases that cannot be made because either

1. The **warehouse** is not legal (not in the **dict**)
2. The **warehouse** does not have that **product** available for sale: either it does not appear in the **dict**; or it appears, but has no inventory: none of that **product** is in stock.

Such illegal purchases **do no**t update the dictionary argument.

Calling this function with the **db** dictionary shown on the top of page 2

```
update_purchases( db, [('Irvine', 'brush'), ('Newport', 'comb'),
                       ('Tustin', 'car'),   ('Newport', 'comb')] )
```

mutates **db**, decreasing the **inventory** of **products** successfully purchased (a **brush** from **Irvine** and a **comb** from **Newport**), which becomes

```
{'Irvine':  {'brush': 2,    'comb': 2,    'wallet': 2},
 'Newport': {'comb': 0,     'stapler': 0},
 'Tustin' : {'keychain': 3, 'pencil': 4,  'wallet': 3}}
```

I have printed this dictionary in an easy to read form. Python prints dictionaries on one line, and can print their key/values in any order, because dictionaries are not ordered.

It returns **{('Tustin', 'car'), ('Newport', 'comb')}** indicating that there is no **car** product in **Tustin** and after **1 comb** is sold in **Newport** there are **0** remaining, so the second sale of **('Newport','comb')** cannot be fulfilled (see the **mutated**

db above).

5: Details of product_locations:

The **product_locations** function takes a **dict** argument in the form illustrated on the top of page 2,

> each key in the **dict** is a **str** (the **warehouse**) whose associated value is another **dict**: this inner **dict** associates a **str** (the **product**) with an **int** (the **inventory**)

and returns a **list** of **2-tuples**: the first index in the **2-tuple** is a **product**; the second index is another **list** of **2-tuples**: **(warehouse, inventory)**.

In the outermost **list**, the **product**s appear in increasing alphabetical order. In each inner **list** (for each **product**) the **warehouse**s appear in increasing alphabetical order.

Calling this function with the **db** dictionary shown on the top of page 2

```
product_locations( db )
```

returns the following **list**s/**tuple** data structure.

```
[('brush', [('Irvine', 3)]),
 ('comb', [('Irvine', 2), ('Newport', 1)]),
 ('keychain', [('Tustin', 3)]),
 ('pencil', [('Tustin', 4)]),
 ('stapler', [('Newport', 0)]),
 ('wallet', [('Irvine', 2), ('Tustin', 3)])
]
```

6: Details of unqiue_supplier (Extra Credit: 1 point):

The **unique_supplier** function takes a **dict** argument in the form illustrated above,

> each key in the **dict** is a **str** (the **warehouse**) whose associated value is another **dict**: this inner **dict** associates a **str** (the **product**) with an **int** (the **inventory**)

and returns a dictionary (**dict** or **defaultdict**) whose keys are **str** (the **warehouse**) and whose associated value is a **set** of all the **products** that are **stocked only from that warehouse** (even if the inventory is **0**).

Calling this function with the **db** dictionary shown on the top of page 2

```
unique_supplier( db )
```

returns a dictionary whose contents are:

```
{'Irvine': {'brush'}, 'Newport': {'stapler'}, 'Tustin': {'pencil', 'keychain'}}
```

This result shows that a **brush** is stocked only in **Irvine**, a **stapler** is stocked only in **Newport** (even though its inventory is 0),and a **pencil** and **keychain** is stocked only in **Tustin**.

Python can print the key/values in any order, because dictionaries are not ordered.