Assignment Guidelines:

- This assignment covers material up to Module 9.
- Submission details:
  - Solutions to these questions must be placed in files `a08q1.py`, `a08q2.py`, `a08q3.py`, and `a08q4.py`, respectively, and must be completed using Python 3.
  - Download the interface file from the course Web page to ensure that all function names are spelled correctly and each function has the correct number and order of parameters.
  - All solutions must be submitted to MarkUs. No solutions will be accepted through email, even if you are having issues with MarkUs.
  - Verify using MarkUs and your basic test results that your files were properly submitted and are readable on MarkUs.
  - For full style marks, your program must follow the Python section of the CS116 Style Guide.
  - Be sure to review the Academic Integrity policy on the Assignments page
  - Natural numbers in this course begin at 0.
  - Required functions need all design recipe elements. Functions you define (eg. helper functions) need all design recipe elements except for examples and tests.
- Download the testing module from the course web page. Include `import check` in each solution file.
  - When a function produces a floating point value, you *must* use `check.within` for your testing. Unless told otherwise, you may use a tolerance of 0.00001 in your tests.
  - Test data for all questions will always meet the stated assumptions for consumed values.
- Restrictions:
  - Do not import any modules other than `math` and `check`.
  - You are always allowed to define your own helper/wrapper functions, as long as they meet the assignment restrictions. Do **not** use Python constructs from later modules (e.g. `zip`, anything with `sets` or `enumerators`, list comprehension, commands `continue` or `break`). For this assignment, abstract list functions **and** recursion will **not** be allowed. Use only the functions and methods as follows:
    * `abs`, `len`, `max`, `min`, `sum`, `range` and `sorted`
    * Any method or constant in the `math` module
    * Type casting including `int()`, `str()`, `float()`, `bool()`, `list()`
    * The command `type()`
    * Any basic arithmetic operation (including `+, -, *, /, //, %, **`)
    * String or list slicing and indexing as well as string or list operations using the operators above
    * Any string or list methods.
    * `input` and `print` as well as the formatting parameter `end` and method `format`. Note that all prompts must match exactly in order to obtain marks so ensure that you do not alter these prompts.
    * Loops, specifically `for` and `while` loops.
    * Dictionaries, classes and methods.
  - Do **not** mutate any passed parameters unless instructions dictate otherwise. You may mutate lists you have created however.
  - While you may use global *constants* in your solutions, **do not** use global *variables* for anything other than testing.
  - Read each question carefully for additional restrictions.
  - **The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.**

1 Distinct

Write a function

`is_distinct(L)`

that consumes a list of integers `L` and returns `True` if and only if every entry in `L` is distinct and `False` otherwise. Your function should run in at worst $O(n \log n)$ time. If desired, you may mutate `L`.

Sample:

```
L = [-10, 9, 5, 4, -10]
is_distinct(L) => False
```

A reminder for this entire assignment, **recursion** and **abstract list functions** are not allowed.

2 Voting

In a democracy, people vote for their favourite candidates in specific districts to elect their representative for their respective region. In this setting, think of a region as being broken down into a bunch of smaller districts and candidates vote in their districts.

Consider the following data definitions:

```
A District is a (dictof Str Nat) which is a dictionary mapping a
   candidate's name to the number of votes the candidate obtained.
A Region is a (listof District).
```

Write a function

`election_winner(R)`

that consumes a `Region` `R` and returns the winner of an election when the results are tallied across all of the districts. Return an empty string if there is no such winner.

Sample:

```
district1 = {'Kirby':100, 'Luigi':75, 'Mario':125}
district2 = {'Luigi':200, 'Kirby':125, 'Mario':125}
election_winner([district1, district2]) => 'Luigi'
```

You may assume that each district has the same candidates to vote for and even if no one votes for them, their name will be in the dictionary with 0 votes tallied. You may assume there will not be any ties.

3 I Have A Secret

In Cryptography, people work to encode secret messages using encryption schemes. One of the easiest such schemes is a substitution cipher which replaces single characters with other characters. This is done using an encryption cipher (explained below). You are working with a top notch team of spies and they have managed to get a hold of both an encrypted word and its proper decryption (or unencrypted form) and your job is to construct the encryption cipher.

Consider the following data definition:

```
An Encryption is a (dictof Str Str) which is a dictionary mapping of
   length one strings to length one strings.
```

As an example, our cryptographers have gotten a hold of the word `banana` and its encryption `ymrmrm`. We can see that the character `b` was encrypted as the character `y`, the character `a` was encrypted as the character `m` and the character `n` was encrypted as the character `r`. This can be represented by the `Encryption` given by `{'b': 'y', 'a':'m', 'n':'r'}`. Write a function

`encryption(original, encrypted)`

which consumes two strings, `original` which corresponds to the unencrypted word and `encrypted` which corresponds to the encrypted word and returns the `Encryption` which was used to convert `original` to `encrypted`.

Sample:

```
encryption('banana', 'ymrmrm') => {'b': 'y', 'a':'m', 'n':'r'}
encryption('banana', 'ymrmrt') => {}
```

Notice in the second example above, the encoding is inconsistent; `a` maps to both to `m` and `t`. In this case, the cryptographer has made an error and in this case, you should return an empty dictionary. As a reminder, the order of terms in the dictionary might be displayed differently than above for your program but should still be equivalent.

4 Vending Machine

A vending machine `Item` consists of the following fields:

```
class Item:
  def __init__(self, n, c, quant, val):
    '''
    Fields: name (Str), code (Str), quantity (Nat), cost (Float)
    requires: cost >= 0.0
              code is of the form L## where L is any upper case letter
              and the two # symbols represent digits.
              codes do not repeat.
    '''
    self.name = n
    self.code = c
    self.quantity = quant
    self.cost = val
```

For example, one might have `Item('Coke Zero', 'A01', 10, 1.25)`. In this problem we will complete the definition of a Vending Machine class

```
class Vending_Machine:
  def __init__(self, list_of_items, starting_money):
    '''
    Fields: items (listof Item) money (Float)
    '''
    self.items = list_of_items
    self.money = starting_money
```

In this problem, you will create a vending machine class that replicates the basic functionality of a vending machine. You may use previous parts of this problem to help you with later parts if desired. You will create class methods for `Vending_Machine` as follows. **Note that in what follows, you will often need to test mutation of the class objects which you should do using `check.expect` and by creating new class objects as needed**.

(1) Create the `Vending_Machine` method

`out_of_item(self, name)`

which returns `True` if and only if `self.items` does not contain any items with the string `name` or if it does but the item has no quantity and `False` otherwise. Note that it is possible for the user to ask for an item your machine does not have and is possible for machines to have duplicates of items.

(2) Create the `Vending_Machine` method

`purchase(self, code)`

which consumes the string `code` corresponding to an `Item`'s code in `self.items` and returns one of two strings:

- `"ERROR"` if the item at `code` does not exist in `self` or if `self` is out of the corresponding item at the code identifier.
- `"Thanks!"` if the item at `code` can actually be purchased.

If the item can be purchased, you should update the `Vending_Machine` and corresponding `Item` to reflect the fact that one item has been purchased (think about which variables you would need to modify).

(3) Create the `Vending_Machine` method

`update_cost(self, name, new_cost)`

which consumes a string `name` and changes the cost of all items in `self.items` with the string `name` to `new_cost`. The function should return `True` if successful and `False` otherwise (for example, if the item is not in the machine).

(4) Create the `Vending_Machine` method

`restock(self, pairs)`

which consumes a `(dictof Str Nat)`, a dictionary mapping strings corresponding to the code of an item in the vending machine to natural numbers which correspond to the quantity of the item at code being added to the machine, and returns the entire total number of items in the machine of all the items in the machine after restocking it. You may assume each code in `pairs` exists is in the machine already.

A sample interaction of the above is as follows. Please note this does not test all behaviours and does not test everything that needs to be updated above (but does give you a sample of what things might need to change as you go through the problem).

```
items = [Item('Coke Zero', 'A01', 10, 1.25),
         Item('Sprite', 'A02', 1, 1.00)]
v = Vending_Machine(items, 1000.0)
print(v.purchase('A02'))
print(v.out_of_item('Sprite'))
print(v.money)
print(v.update_cost('Coke Zero', 1.00))
print(v.items[0].cost)
print(v.restock({'A01':1, 'A02':10}))
print(v.items[0].quantity)
```

which will display:

```
Thanks!
True
1001.0
True
1.0
21
11
```