CSci 1133
Lab Exercise 9

Associative Data Structures

This week, we continue our exploration of container classes in Python. Python *dictionaries* are mutable structures that provide a powerful association mechanism using *key* : *value* pairs. Associative data structures are ubiquitous across many computing applications and involve some sort of mapping from a key value to an associated data value(s). The Python *dictionary* structure provides a simple, efficient mechanism that can be used to implement a number of complex computational tasks such as multi-way value selection (tail-nested if-else), frequency counting, etc.

Warm-up

1) Imagine that you have been provided with the following lists of students and their respective test scores:

```
names = ['joe', 'tom', 'barb', 'sue', 'sally']
scores = [10,23,13,18,12]
```

Write a function named `makeDictionary` that takes the two lists and returns a dictionary with the names as the key and the scrores as the values. Assign the result of `makeDictionary` to a variable: `scoreDict`, which will be used in the exercises that follow.

2) Using `scoreDict`, display barb's score. (use the Python command-line interface)

3) Add a score of 19 to `scoreDict` for a student named john. (use the Python command-line interface)

4) Create and display a sorted list of all the scores in `scoreDict`. (use the Python command-line interface)

5) Calculate the average of all the scores in `scoreDict`. (use the Python command-line interface)

6) Tom has just dropped the class. Remove him (and his associated score) from `scoreDict`. (use the Python command-line interface)

7) Change sally's score to 13. (use the Python command-line interface)

8) Write another function named `getScore` that takes a name and a dictionary as arguments and returns the score for that name if it is in the dictionary. If the name is not in the dictionary, print an error message and return -1. Demonstrate your function to one of the TAs.

Stretch

1) **Morse Code**[1]

Write a Python program that will allow a user to type in a message and convert it into Morse code. Your program must use a *dictionary* to represent and implement the code. The letters in Morse code are encoded as follows:

A	. _	N	_ .
B	_ . . .	O	_ _ _
C	_ . _ .	P	. _ _ .
D	_ . .	Q	_ _ . _
E	.	R	. _ .
F	. . _ .	S	. . .
G	_ _ .	T	_
H	. . . .	U	. . _
I	. .	V	. . . _
J	. _ _ _	W	. _ _
K	_ . _	X	_ . . _
L	. _ . .	Y	_ . _ _
M	_ _	Z	_ _ . .

Format output (use periods and underscores separated by one space to represent dots and dashes) so that there is one letter per line, with a blank line following the last letter of each word, and two blank lines following the end of each sentence (except the last).

Example:

```
Enter a phrase: sos
. . .

_ _ _
. . .
```

[1] adapted from Charles Dierbach: *Introduction to Computer Science Using Python*, pp. 165-166

Workout

1) **Playing Cards**

Many games involve the use of "playing cards" drawn from a deck of 52 individual cards. Each card in the deck has a unique combination of value (or *rank*), and *suit*. The *rank* of each card is taken from the set: { 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace } and the *suit* is one of: { Clubs, Spades, Diamonds, Hearts }.

Construct a pure function named `rankandSuitCount(cards)` that will take a list of strings representing a collection of playing cards (a "hand" perhaps), and return two Python dictionaries with the frequency count of each unique card *rank* and *suit* (respectively) in the list of cards. Individual cards are represented by a 2-character string. The first character represents the card *rank*:

'2' - '9' ==> 2 - 9
'T' ==> 10
'J' ==> Jack
'Q' ==> Queen
'K' ==> King
'A' ==> Ace

and the second character represents the *suit:*

'C' ==> Clubs
'S' ==> Spades
'D' ==> Diamonds
'H' ==> Hearts

For example, a hand with the ace of spades, the ace of diamonds, the 2 of clubs, the 10 of hearts and the 10 of spades would be represented as:

['AS', 'AD', '2C', 'TH', TS]

And the result of `rankandSuitCount()` on this list of cards would look like this:

```
>>> rank, suit = rankandSuitCount(['AS','AD','2C','TH',TS])
>>> suit
{'S':2,'D':1,'C':1,'H':1}
>>> rank
{'A':2,'2':1,'T':2}
```

2) **Poker Hands**

Most variants of the card game *Poker* are played by ranking the various types of hands that can occur with 5 cards:

straight flush: all cards in rank sequence, same suit	(e.g., 8D, 9D, TD, JD, QD)
four-of-a-kind: 4 cards of the same rank	(e.g., 5C, 5S, 5D, 5H, 9C)
full-house: 3 cards of one rank and 2 of another	(e.g., 6H, 6S, 6C, QD, QH)
flush : all cards of same suit, not in sequence	(e.g., 3H, 6H, TH, JH, AH)
straight: all cards in rank sequence, not same suit	(e.g., 2S, 3D, 4H, 5H, 6S)
three-of-a-kind: 3 cards of the same rank	(e.g., 2C, 2H, 2D, 7S, 3D)
two pair: 2 pairs of matching rank cards	(e.g., AH, AD, QS, QH, 2C)
one pair: 1 pair of matching rank cards	(e.g., 8S, 8C, 9D, 3S, TH)
high-card: no matching rank cards	(e.g., 4H, 7D, KS, AS, 9C)

The value of any particular poker hand is inversely proportional to its probability of occurrence when 5 cards are dealt randomly from a 52-card deck. The highest ranked poker hand is an 'Ace-high straight flush", commonly referred to as a "royal" flush.

It turns out that the frequency analysis returned by the `rankandSuitCount` is sufficient to determine any particular poker hand. For example, if the cardinality of the card rank dictionary is 4, then the hand must be "one pair" because *exactly* two of the cards must be of the same rank (can you explain why?)

Similarly, if the cardinality of the card-rank dictionary is 3, then the hand must be either "two pair" or "three-of-a-kind" because these are the only ways you can create a hand of five cards and exactly 3 separate values.

In summary, given the cardinality of the car-rank dictionary, you can deduce the following:

5 : "flush", or "straight" or "straight flush" or "high-card"
4 : "one pair"
3 : "two pair" or "three-of-a-kind"
2 : "four-of-a-kind" or "full-house"
1 : (this is impossible! can you explain why?)

Write a pure function named `pokerHand(cards)` that will take a list of 5 "cards" (in the manner described in Workout problem 1) and return a string describing what kind of poker hand it is (i.e., "flush", "two-pair", etc.)

Your function should call the `rankandSuitCount` function from Workout problem 1 and use the information returned in the two dictionaries to determine the best hand represented by the 5 cards.

Challenge

Here is an interesting challenge problem. Try it if you have extra time or would like additional practice outside of lab.

1) **Fully Associative Memory Program**

A *fully-associative memory* is a storage mechanism in which *any* value can be used as a key to identify all the other values that have been previously associated with it. For example, using a fully-associative memory, you could associate names with telephone numbers. Later, you could input a name and obtain all the associated telephone numbers for that name or, conversely, you could input a telephone number to obtain all the names associated with the number.

A fully-associative memory can be constructed using a Python dictionary. The mechanism of adding an associated pair of elements to the "memory" is accomplished by doing *two* dictionary insertions: using the first element as a key for the second, and then using the second element as the key for the first. Each of the "values" in the dictionary will be maintained as a *list of values* that have been associated with the key. For example, consider the following associative memory actions:

```
add bob 755.3632
add carol 612.4435
add sue 755.3632
add bob 787.3345
```

The resulting dictionary would appear as follows:

```
{'bob':['755.3632', 787.3345'], 'carol':['612.4435'], 'sue':[755.3632'],
 '755.3632':['bob', 'sue'], '612.4435':['carol'], '787.3345':['bob']}
```

Write a program to implement a fully-associative memory. You should do the following:

- Write a function named `addDB` that will accept three arguments: a dictionary and two strings, and add the two associated strings to the fully-associative memory (dictionary). The association between the two string arguments must be added twice: the first string argument will be the key for the second, and the second string argument will be the key for the first. For each dictionary insertion, you should first determine if the value already appears in the dictionary. If so, simply add the associated value to the list of values. If not, then create a new dictionary entry with the value as a single *list* element. Thoroughly test your function and print the results to ensure it is working properly. Use the example data provided in the problem description.

- Write another function named `findDB` that will accept two arguments: a dictionary and a key (string), and return the list of values that are associated with the key. If the key is not in the dictionary, then return a null list.

- Write a third (void) function named `removeDB` that will accept three arguments: a dictionary and two strings, and remove the associated strings from the fully-associative memory (dictionary). Each value must be removed from its respective value list. For each dictionary deletion, you should first determine if the association already appears in the dictionary. [hint: the key must be in the dictionary *and* the value must be in the value list] If so, simply remove the associated value to the list of values. If the resulting list is empty, then remove the key from the dictionary.

- Using the functions you've created, construct a program that will interact with the user and allow him/her to add, find, and delete associations in the memory. Your program should provide a loop that will solicit command lines (a single string) containing an operation followed by zero, one or two "arguments". The program should implement the following commands:

`add`	Add an association to the memory between the two arguments
`find`	Print out the associated list of values for the single argument
`del`	Delete the association between the two arguments
`clear`	Clear all associations from the associative memory structure
`end`	End the program

- Input and process commands as described above [hint: use the .split() method to obtain the command and argument strings from the input]
- If the command is `end`, then terminate the program
- If the command is not valid, print a suitable message, otherwise process the command using the functions you've already created and tested
- Continue to process commands until the user enters the `end` command

Here's an example:

```
>>> main()
--> add Cindy 334-6892
--> add Steve 445-5691
--> add Bob 334-6710
--> find Cindy
['334-6892']
--> add Susan 334-6892
--> add Fred 556-7823
--> find Susan
['334-6892']
--> find 334-6892
['Cindy', 'Susan']
--> find Bob
['334-6710']
--> del Bob 334-6710
--> find Bob
[]
--> end
>>>
```