Lecture 2: Dictionaries and Sets

Finding the most frequent element

The file **freqlist.py** shows two different ways of finding the most frequent element in a list of numbers. `findFrequencies` only works with a list of numbers and further requires a data structure whose size needs to be the maximum value in the list of numbers plus 1. It will not work with negative numbers or strings!

If we wish to find the most frequent element in a list of strings, we could do this with a list data structure as shown in `findFrequenciesList`. We need to loop over the input list once for each element in the input list to compute frequencies. Note that `findFrequenciesList` will work for lists of numbers as well.

We can write simple and efficient code to solve this problem using dictionaries.

Dictionaries

Lists have to be indexed using non-negative integers, but dictionaries in Python are generalized lists that can be indexed using strings, integers, floating-point numbers (floats) or tuples. You will appreciate the power of dictionaries in this example and in the Bacon lab, as well as in numerous other applications you will encounter throughout the term.

Here's a simple dictionary that maps names to IDs. Note the curly braces, which tells you that we are declaring a dictionary.

```
NameToID = {'Alice': 23, 'Bob': 31, 'Dave': 5, 'John': 7}
```

`NameToID['Alice']` returns `23`, `NameToID['Dave']` returns `5`. `NameToID['David']` throws an error. The indices of our generalized list are called keys, and the above dictionary has four keys. Each key in a dictionary points to a value, and therefore a dictionary is composed of key: value pairs.

While `NameToID['David']` throws an error, we can check if a key exists or does not exist in a dictionary by writing:

```
'David' in NameToID
'David' not in NameToID
```

This will return **`False`** and **`True`**, respectively, for our dictionary. However, if we write:

```
NameToID['David'] = 24
```

```
print(NameToID)
```

We will see:

```
{'John': 7, 'Bob': 31, 'David': 24, 'Alice': 23, 'Dave': 5}
```

Notice that there are now five key: value pairs. If we write:

```
'David' in NameToID
```

This will now return **`True`** since we added `'David'` as a key to `NameToID`.

Note that the order of keys printed the second time we printed the dictionary was different from the first time. Dictionaries in Python do not guarantee any particular ordering of key: value pairs. Unlike a list where the indices are non-negative numbers and have a natural ordering, in a dictionary, keys can be integers, strings or tuples without a natural order. Here's a more interesting example of a dictionary:

```
crazyPairs = {-1: 'Bob', 'Bob': -1, 'Alice': (23, 11), (23, 11):
'Alice'}
```

We have a crazy collection of "objects", which are numbers, tuples or people, and we have paired them up in our dictionary. We can add another mapping, e.g., `crazyPairs['David'] = 24`, and change an existing mapping by writing `crazyPairs['Alice'] = (23, 12)`. If we do this, and execute `print(crazyPairs)`, we will get:

```
{(23, 11): 'Alice', 'Bob': -1, 'David': 24, 'Alice': (23, 12), -
1: 'Bob'}
```

Notice that that the key: value pair `(23, 11): 'Alice'` was unaffected since we only modified values associated with other keys.

We were careful to use immutable tuples in our example above. Lists are not allowed as keys in a dictionary in Python. The reason is that lists are mutable and all sorts of weird bugs would arise if you insert a key into a dictionary corresponding to a list, and then modify the list. This will "confuse" the dictionary and Python precludes this confusion by not allowing lists to be used as keys. They can, however, be used as values in a dictionary, and can be mutated after insertion into the dictionary.

Finally, to delete a key from a dictionary, you can write:

```
if 'Alice' in NameToID:
      del NameToID['Alice']
```

This will delete `'Alice'` from `NameToID` if it exists in `NameToID`.

Given a dictionary `d`, to obtain *views* of lists corresponding to dictionary keys, values and key: value pairs represented as tuples, we can call `d.keys()`, `d.values()` and `d.items()`. Here's an example:

```
d = {'A': 1, 'B': 2, 'C': 3, 'D': 4}
d.keys() # prints ['A', 'C', 'B', 'D']
d.values() # prints [1, 3, 2, 4]
d.items()  # prints [('A', 1), ('C', 3), ('B', 2), ('D', 4)]
```

Note that you only get a view of the list of keys if you write `k = d.keys()` and therefore you cannot run commands such as `k.sort()`, you will have to run `sorted(k)` to produce an actual list.

Iterators

Dictionaries, lists, and other Python data structures may be used uniformly with **`for`** loops. For instance, all three of the statements below generate the same output, where the last one is using *sets*, which are enforced to be duplicate-free.

```
for x in [1, 2, 3]:
      print (x)
for x in {1: 'A', 2: 'B', 3: 'C'}:
      print (x)
for x in {1, 2, 3}:
      print (x)
```

We can also apply an extra operation to the dictionary to let us loop over not just the *keys* (e.g., 1, 2, 3) but also the *values* (e.g., 'A', 'B', 'C').

```
for x, y in {1: 'A', 2: 'B', 3: 'C'}.items():
      print (x, y)
```

One might worry that the **`for`** loop seems to have, built into its behavior, all sorts of special cases for different collections that we might want to loop over. However, there is a more general mechanism at work here, called *iterators*. It is safe to indeed assume that **`for`** magically does the right thing with different kinds of collections.

Finding the most frequent element using dictionaries

Let's go back to our problem of finding the most frequent element in a list of strings or numbers. The most elegant and efficient way of solving this problem is to use dictionaries. The file **dictionary-set.py** has a function `findFrequenciesDict` that shows a simple implementation that creates a dictionary whose size will grow only with the number of unique elements in the input list. There is only one loop through the input list.

After the procedure is invoked, the keys in the dictionary will be the unique elements and the values will be the frequencies.

Sets

Sets in Python can be thought of as dictionaries without values. The elements of the set are the keys of the dictionary. A set only contains unique elements. Here is some simple code that uses sets to find the unique elements in a list that may contain repeated elements.

```
mylist = ['nowplaying', 'PBS', 'PBS', 'nowplaying', 'job',
          'debate', 'thenandnow']
myset = set(mylist)
print (myset)
mynewlist = list(myset)
print (mynewlist)
```

Alternately, one could do the following:

```
output = set()
for x in mylist:
    output.add(x)
print (output)
```

Sets in N Queens

The file **nqueens-sets.py** shows a compact implementation that solves the N Queens problem. It exploits the fact that a set does not contain repeated elements.

With the solution represented as a vector with one queen in each row, we don't have to check to see if two queens are on the same row. By using a permutation generator, namely, `permutations` (this requires us to import the `itertools` package) we know that no value in the vector is repeated, so we don't have to check to see if two queens are on the same column. Since rook (i.e., row and column) attacks don't need to be checked, we only need to check bishop (i.e., diagonal) moves.

The technique for checking the diagonals is to add or subtract the column number from each entry, so any two entries on the same diagonal will have the same value (in other words, the sum or difference is unique for each diagonal). Now all we have to do is make sure that the diagonals for each of the eight queens are **distinct**. So, we put them in a set (which eliminates duplicates) and check that the set length is `n` (which means no duplicates were removed).

Any permutation with non-overlapping diagonals is a solution. So, we pretty print it and continue checking other permutations.

One disadvantage with this solution is that we can't simply "skip" all the permutations that start with a certain prefix, after discovering that that prefix is incompatible. For example, it is easy to verify that no permutation of the form (1, 2, ...) could ever be a solution, but since we don't have control over the generation of the permutations, we can't just tell it to "skip" all the ones that start with (1, 2). This code is *much* slower than the recursive code for `n >= 10`.

Membership in Dictionaries/Sets Versus Lists

Not only can you look up strings and tuples in Python dictionaries, even checking that a number is one of the keys in a dictionary (i.e., checking membership) is much faster than checking that a number is in a list containing exactly the same keys as the dictionary. This is because dictionaries and sets are implemented using hash table – you will learn a lot about hash tables in 6.006. Briefly, a membership check in a dictionary or set takes, on average, a constant number of operations regardless of the size of the dictionary or set, whereas the membership check in an unsorted list takes, on average, a number of operations equaling half the size of the list.

Run `timingCheck(10000000)` in **dictionary-set.py** and see how long list membership and set membership take to run.

Let's look at a few puzzles that require the uses of dictionaries and sets.[1]

Pair Sum Problem

Given an integer k and a list A of n unordered integers, find out whether there is a pair $\{x, y\} \subset A$ such that $x + y = k$.

For a naive solution, just check all pairs (x, y) of elements of A. If x + y equals k, return yes and stop. Although no extra space is required, the number of additions and comparisons to be performed is n^2 in the worst case, where n is the number of elements in A.

We want to do better. First note that, if we select an element x of A, we can look for its complement $y = k - x$ in A, and the problem is now a search problem. If, for each x, we need to traverse the whole list to check whether it contains its complement, then our algorithm still demands n^2 operations.

We can use a Python set to store the elements of A. Then we have a fast algorithm that only does n membership checks in a set. The algorithm is "for each element x in

[1] Based on http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-74382015000200423

the list, if k – x is in the (initially empty) set S, then return yes, otherwise insert x in S." You can see code for this algorithm in **dictionary-set.py**.

Pair Sum of Pairs Problem

Given a list A find all non-trivial quadruples (x, y, z, w) of elements x, y, z, w $\in$ A such that x + y = z + w.

For a trivial solution, we may simply test each quadruple of distinct elements of the set A. We can do this using quadruply nested loops, taking n^4 operations where n is the length of A.

Now we look into a dictionary-based approach. We start with an empty dictionary H. We consider each unordered pair (x, y) of distinct elements of A, obtaining d = x + y, one pair at a time. If d is not a key stored in H, we insert the (key, value) pair (d, Sd) in H, where Sd is a collection (which can be implemented as a list, or a set) containing only the tuple (x, y) initially. On the other hand, if d is already stored in H, then we simply add (x, y) to the non-empty collection Sd which d already maps to in H. After all pairs of distinct elements have been considered, our dictionary will contain non-empty collections Sd whose elements are pairs (x, y) satisfying x + y = d. For each non-unitary collection Sd , we combine distinct pairs (x, y), (z, w) $\in$ Sd, two at a time, to produce our desired quadruples (x, y, z, w). You can see code for this algorithm in **dictionary-set.py**.

Permutation Bingo

In the standard game of Bingo, each of the contestants receives a card with some integers. The integers in each card constitute a subset of a rather small range A = [1, a]. Then, subsequent numbers are drawn at random from A, without replacement, until all numbers in the card of a contestant have been drawn. That contestant yells Bingo! and wins the prize.

In our variation of the game, the contestants receive an empty card that the contestants can write on. Subsequent numbers are drawn at random from some range B = [1, b], with replacement, forming a sequence S. The contestant who wins the game is the one who first spots a triple W = (x, y, z) of three distinct numbers such that all 3! = 6 permutations of its elements appear among the subsequences of three consecutive elements of S. As an example, the sequence

S = [1, 2, 3, 1, 2, 1, 3, 2, 1]

is a minimal winning sequence since the triple W = (1, 2, 3) appears in every permutation in the order (1, 2, 3), (2, 3, 1), (3, 1, 2), (2, 1, 3), (1, 3, 2) and (3, 2, 1). It

is minimal because if we remove 1 from the end of S, we do not have a winning sequence. Similarly, the more interesting sequence

$$S = [2, 19, 4, 1, 100, 1, 4, 19, 1, 4, 1, 19, 100, 192, 100, 4, 19, 2, 1, 19, 4]$$

is a minimal winning sequence. There is a triple, namely W = (1, 4, 19), such that every permutation of W appears as three consecutive elements of S. It is minimal because if we remove 4 from the end of S, we do not have a winning sequence. Of course, a permutation Bingo game could go on for a long time, but let's assume that S is carefully selected so the game ends fairly quickly.

We want to write a program to play permutation Bingo. Moreover, we want our program to win the game when it plays against other programs. In other words, we need an efficient algorithm to check, after each number is added to S, whether S happens to be a winning sequence. Our program should also determine the triple that appears in all permutations since that is a requirement for winning the prize.

One possible approach would be as follows. Store S as a list. After the n-th number is added, for all $n \geq 8$ (a lower bound for the size of a winning sequence) [2], traverse $S = [s_1, s_2, ..., s_n]$ entirely for each triple $R_i = (s_i, s_{i+1}, s_{i+2})$ of three consecutive distinct elements (with $1 \leq i \leq n - 2$), checking whether all permutations of R_i appear as consecutive elements of S. Checking a sequence of size n this way will require n^2 operations since each time a number is added we traverse the list.

We can do better using dictionaries and sets, only requiring a single traversal of S, during which we add each subsequence of three consecutive distinct elements of S into an appropriately structured dictionary, and checking if we have a Bingo situation. Our dictionary structure is updated efficiently each time a new number is announced. How best to structure this dictionary?

The dictionary's key-value pairs are such that each key is a standardized triple, i.e., a standardized subsequence of length 3, and each value is a set of subsequences that have appeared thus far. To make sure that two permutations go to the same dictionary key if and only if they have the same elements (in different orders), we create a standardized triple corresponding to the **ascending** sequence of its elements and use it as the dictionary key. Now, for each subsequence of three distinct elements in S, we compute its key, look up the key in the dictionary and place the subsequence into a **set** of subsequences that comprise the value for that key. Using a set for the value ensures that we do not add the same subsequence twice. Whenever a value set for a key reaches size six, we have a winner!

Each key in the dictionary is a Python tuple `(x, y, z)`, with `x ≤ y ≤ z`. The value for each key is a set of tuples. *Remember keys in dictionaries cannot be lists, and lists*

[2] 3 numbers corresponds to one permutation, and we need 1 more number for each of the 5 remaining permutations giving us 8.

cannot be elements of sets because lists are mutable. (We could have represented each value as a list of tuples or a list of lists as long as we check for duplicates each time we want to add to the value list.)

The code for playing permutation Bingo is in **dictionary-set.py**. There are two functions: `permutationBingocheck` checks if, given the current state of the dictionary and the latest triple of numbers, whether there is a Bingo situation or not. It does this by maintaining a dictionary structure as described above and incrementally updating it. `playBingo` is an interactive procedure that takes input from the user (announcer) one number at a time. As it encounters each new triple, it calls `permutationBingocheck`.