Dictionaries

15-110 – Friday 10/06

Announcements

- **Hw3** due on Monday
- Sign up for code reviews!
  - Sign-ups close today at 5pm

Updates on AI Assistance

- It can be hard to know how to use ChatGPT and other tools appropriately, in a way that supports your learning
- Never ok:
  - Putting parts of an assignment writeup into an AI tool
  - Asking AI to write code to help you solve an assignment problem
  - Using code from an AI tool’s output in your assignment
    - Either by directly copy/pasting, or by typing out its response
- Not forbidden, but we do not recommend:
  - Asking an AI tool why some code isn’t working

Resources

- Office hours
  - Schedule: https://www.cs.cmu.edu/~110/syllabus.html
  - Ask questions about assignments or concepts
  - Hang out and work
- Piazza
  - Post questions (privately to "Instructors" if including details of your solution) and get answers
- Revision policy
- Submit partial work, or assignments with missing problems
  - This is ok!
- Check out the website for more: https://www.cs.cmu.edu/~110/index.html

Learning Goals

- Identify the **keys** and **values** in a dictionary
- Use **dictionaries** when writing and reading code that uses pairs of data
- Use **for loops** to iterate over the parts of an **iterable** value

Data Structures Organize Data

So far, we've talked about efficiency in terms of algorithm design. We can solve the same problem multiple ways, and some approaches are more efficient than others.

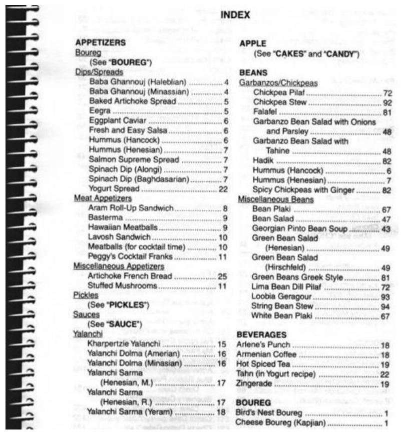
We can also improve the efficiency of an algorithm by changing the **data structure** we use to store incoming data. For example, a **list** is a good for storing values in sequential (and indexed) order. What other types of data might we work with?

Dictionaries

Python Dictionaries Map Keys to Values

The first data structure we'll discuss is the **dictionary**. Dictionaries store data in **pairs** by mapping **keys** to **values**.

We use dictionary-like data in the real world all the time! Examples include phonebooks (which map names to phone numbers), the index of a book (which maps terms to page numbers), or the CMU directory (which maps andrewIDs to information about people).



INDEX

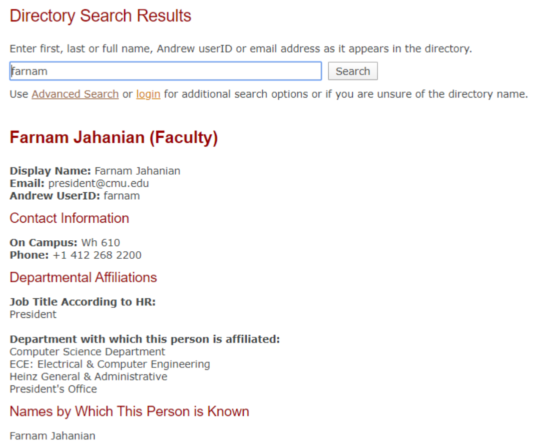
APPETIZERS
Boureg
(See "**BOUREG**")
Dips/Spreads
Baba Ghannouj (Haleblian) 4
Baba Ghannouj (Minassian) 4
Baked Artichoke Spread 5
Eegra 5
Eggplant Caviar 6
Fresh and Easy Salsa 6
Hummus (Hancock) 6
Hummus (Henesian) 7
Salmon Supreme Spread 7
Spinach Dip (Alongi) 7
Spinach Dip (Baghdasarian) 7
Yogurt Spread 22
Meat Appetizers
Aram Roll-Up Sandwich 8
Basterma 9
Hawaiian Meatballs 9
Lavosh Sandwich 10
Meatballs (for cocktail time) 10
Peggy's Cocktail Franks 11
Miscellaneous Appetizers
Artichoke French Bread 25
Stuffed Mushrooms 11
Pickles
(See "**PICKLES**")
Sauces
(See "**SAUCE**")
Yalanchi
Kharpertzie Yalanchi 15
Yalanchi Dolma (Amerian) 16
Yalanchi Dolma (Minasian) 16
Yalanchi Sarma
(Henesian, M.) 17
Yalanchi Sarma
(Henesian, R.) 17
Yalanchi Sarma (Yeram) 18

APPLE
(See "**CAKES**" and "**CANDY**")

BEANS
Garbanzos/Chickpeas
Chickpea Pilaf 72
Chickpea Stew 92
Falafel 81
Garbanzo Bean Salad with Onions
and Parsley 48
Garbanzo Bean Salad with
Tahine 48
Hadik 82
Hummus (Hancock) 6
Hummus (Henesian) 7
Spicy Chickpeas with Ginger 82
Miscellaneous Beans
Bean Plaki 67
Bean Salad 47
Georgian Pinto Bean Soup 43
Green Bean Salad
(Henesian) 49
Green Bean Salad
(Hirschfeld) 49
Green Beans Greek Style 81
Lima Bean Dill Pilaf 72
Loobia Geragour 93
String Bean Stew 94
White Bean Plaki 67

BEVERAGES
Arlene's Punch 18
Armenian Coffee 18
Hot Spiced Tea 19
Tahn (in Yogurt recipe) 22
Zingerade 19

BOUREG
Bird's Nest Boureg 1
Cheese Boureg (Kapjian) 1



Directory Search Results

Enter first, last or full name, Andrew userID or email address as it appears in the directory.

farnam

Use Advanced Search or login for additional search options or if you are unsure of the directory name.

Farnam Jahanian (Faculty)

Display Name: Farnam Jahanian
Email: president@cmu.edu
Andrew UserID: farnam

Contact Information

On Campus: Wh 610
Phone: +1 412 268 2200

Departmental Affiliations

Job Title According to HR:
President

Department with which this person is affiliated:
Computer Science Department
ECE: Electrical & Computer Engineering
Heinz General & Administrative
President's Office

Names by Which This Person is Known

Farnam Jahanian

Key-Value Pairs

In a dictionary, a **key-value pair** is two values that have been paired together for organizational purposes. We'll be able to access the value by looking up the key, like how we can access a list value using its index.

For example, if we stored a phonebook in a dictionary, a **key** might be the string `"CMU"`, and its **value** would be the string `"412-268-2000"`. It wouldn't make sense to switch the roles because our default action is to look up a phone number based on a name, not vice versa.

Note: keys must be **immutable** (numbers, strings, or Booleans), but values can be any type of data. Why? We'll explain later, when we discuss more search algorithms.

Python Dictionaries

Dictionaries have already been implemented for us in Python.

```
# make an empty dictionary
d = { }

# make a dictionary mapping strings to integers
d = { "apples" : 3, "pears" : 4 }
```

Python Dictionaries – Getting Values

Dictionaries are similar to lists, but they are **unordered** – key-value pairs don't have positions.

Instead of indexing by position, index by key:

```
d = { "apples" : 3, "pears" : 4 }
d["apples"] # the value paired with this key
len(d) # number of key-value pairs
```

If you try to access a key that doesn't exist, you'll get a runtime error. The same thing happens if you try to index by value instead of key.

```
d["ice cream"] # KeyError
d[4] # KeyError
```

Python Dictionaries – Adding and Removing

How do we add a new key-value pair? Use **index assignment** with the key.

This works whether or not that key has been assigned a value yet. If the key is already in the dictionary, the value for the key is updated; it does not add a new key-value pair.

```
d = { "apples" : 3, "pears" : 4 }
d["bananas"] = 7 # adds a new key-value pair
d["apples"] = d["apples"] + 1 # updates the key-value pair
```

To remove a key-value pair, use **pop** with just the key as a parameter.

```
d.pop("pears") # destructively removes
```

Python Dictionaries – Search

We can **search** for a key in a dictionary using the built-in `in` operation.

```
d = { "apples" : 3, "pears" : 4 }
"apples" in d # True
"kiwis" in d # False
```

We can't use `in` to look up the dictionary's values; we need to loop over the keys and check each key's value instead. How do we loop over a dictionary? We'll get there in just a moment!

Activity: Trace the code

In the following code, the keys represent student IDs and the values represent student names. After running the code, what key-value pairs will the dictionary hold?

```
d = { 26: "Chen", 23: "Patrick" }
d[88] = "Rosa"
d[23] = "Pat"
d[51] = d[23]
if "Chen" in d:
    d.pop("Chen")
```

For Loops over Iterables

Iterable Values and Loops

An **iterable value** is a value that can be looped over directly by a for loop. They are often composed of some number of individual pieces of data (though not always).

So far, strings and lists have been iterable: a string is a sequence of characters and a list is a sequence of values. Dictionaries are also iterable, as they're composed of some number of key-value pairs.

With both strings and lists, the pieces of data were stored in an ordered sequence. That meant we could identify the **position** of each value and use a **for loop over a range** to visit each position in turn.

A dictionary is **unordered**. That means we can't loop over dictionaries using a for loop with a range, as there are no positions to visit.

For Loops Can Repeat Over Iterable Values

We don't need a `range` to use a `for` loop. We can loop over the parts of an iterable value directly by providing the value instead of a `range`.

```
for <itemVariable> in <iterableValue>:
    <itemActionBody>
```

For example, if we run the following code, it will print out each string in a list with an exclamation point after it.

```
wordList = [ "Hello", "World" ]
for word in wordList:
    print(word + "!")
```

For Loops on Dictionaries

When we run a for loop directly over a dictionary, the loop visits all key-value pairs in some order. The loop control variable will hold the **key** of each key-value pair. To access the value, you must index into the dictionary with that key.

```
d = { "apples" : 5, "beets" : 2, "lemons" : 1 }
for k in d:
    print("Key:", k)
    print("Value:", d[k])
```

For-Range vs For-Iterable

When should you use a For-Iterable loop instead of a For-Range loop?

For dictionaries, **always use a For-Iterable loop**. There are no indexes, so you can't use For-Range.

For strings and lists, you can iterate directly over the values **if you don't need the indexes**. For example, to sum a list, you could use either:

```
result = 0
for item in lst:
    result = result + item
```

or:

```
result = 0
for i in range(len(lst)):
    result = result + lst[i]
```

Activity: Design a dictionary

You do: You’re creating a dictionary to help distribute t-shirts to students. Students will arrive at the office and tell you their AndrewID, and you’ll use the dictionary to see whether they ordered a red or a black shirt.

What type will you make the keys? The values?

Pitt heard about your great t-shirt program and wants to use it. They have numerical student IDs, though, and their shirts are blue or yellow, not red and black.

What need to change about the keys? The values?

Coding with Dictionaries

Example: Processing Dictionaries

Problem-solving with dictionaries often (but not always) involves looping over the dictionary and doing something with each key-value pair.

For example, the following program can sum all the values in a dictionary by capturing each value from its key.

```
def addValues(d):
    total = 0
    for key in d:
        total = total + d[key]
    return total
```

Example: Building Dictionaries

Another common task is to create a new dictionary based on existing values. In this case it is important to track **which keys have already been added** and need to be updated.

For example, we can create an alphabet dictionary for a list of strings. For each letter, create a new list if the letter has not been seen before, or add to the existing list if it has been seen.

```
def makeAlphabetDict(words):
    d = { }
    for word in words:
        letter = word[0]
        if letter not in d:
            d[letter] = [word]
        else:
            d[letter].append(word)
    return d
```

Example: Nested Dictionary

We can even use **nested** dictionaries in a similar way to how we use nested (2D) lists. Just map each key to another dictionary (which will map other keys to specific values).

For example, we can create a multiplication table in a nested dictionary (outer keys are x, inner keys are y, values are x*y).

```
def createMultDict(n):
    d = { }
    for x in range(1, n+1):
        innerD = { }
        for y in range(1, n+1):
            innerD[y] = x * y
        d[x] = innerD
    return d
```

[if time] Activity: `hasShortKeys(d, limit)`

You do: write a program that takes a dictionary mapping strings to numbers and a limit (a number) and returns `True` if all the keys are at most the limit in length, and `False` otherwise.

For example, `hasShortKeys({ "abc" : 2, "de" : 5}, 3)` would return `True`, but `hasShortKeys({ "abc" : 2, "defgh" : 2}, 4)` would return `False`.

Learning Goals

- Identify the **keys** and **values** in a dictionary
- Use **dictionaries** when writing and reading code that uses pairs of data
- Use **for loops** to iterate over the parts of an **iterable** value

Advanced Examples

Bonus slides

Coding with Dictionaries – Track Information

We often use dictionaries when problem-solving. One common use of dictionaries is to **track information** about a list of values.

For example, given a 2D list of students and their colleges (a list of two-element lists of `"student"` and `"college"`), how many students are in each college?

We will create a dictionary with colleges as the keys and the student counts as the values.

```
def countByCollege(studentLst):
    collegeDict = { }
    for pair in studentLst:
        name = pair[0]
        college = pair[1]
        if college not in collegeDict:
            collegeDict[college] = 0
        collegeDict[college] += 1
    return collegeDict

countByCollege([ ["erhurst" ,"CIT"],
["neerajsa","SCS"], ["cosorio","DC"],
["dtoussai", "CIT"]])
```

Coding with Dictionaries – Find Most Common

We also use dictionaries to find the most common element of a list, by mapping elements to counts.

For example, given the dictionary returned by the previous function, which college is the most popular?

```
def mostPopularCollege(collegeDict):
    best = None
    bestScore = -1
    for college in collegeDict:
        if collegeDict[college] > bestScore:
            bestScore = collegeDict[college]
            best = college
    return best
```