Dictionaries

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

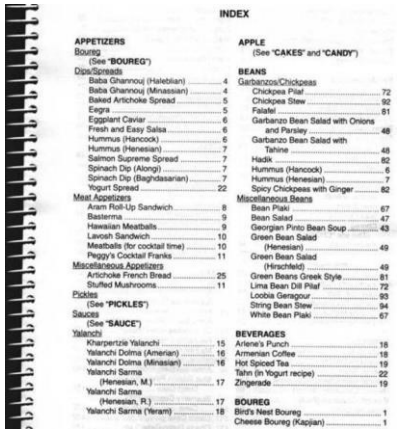
We can also improve the efficiency of an algorithm by changing the data structure we use to store incoming data. For example, a list is a good for storing values in sequential (and indexed) order. What other types of data might we work with?

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 a library directory (which maps ISBN to book title and author).



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 "DKU", 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, but values can be any type of data. Why? We'll explain next time, when we talk about dictionary search.

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. 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.

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

We can also access all the keys or all the values separately:

```
d.keys()
d.values()
```

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["bananas"] = 7 # adds a new key-value pair
d["apples"] = d["apples"] + 1 # updates the value
```

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?

Activity: Trace the code

After running the following code, what key-value pairs will the dictionary hold?

```
d = { "PA" : "Pittsburgh", "NY" : "New York City" }
d["WA"] = "Seattle"
d["NY"] = "Buffalo"
if "Pittsburgh" in d:
        d.pop("Pittsburgh")
```

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 index into the value to get an individual part and use a for loop over a range to visit each part in turn.

A dictionary doesn't have indexes; it has keys. That means we can't loop over dictionaries using a for loop with a range, as it isn't clear how we would set up the range.

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 is set to 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: countItems(foodCounts)

You do: write the function countItems(foodCounts) that takes a dictionary mapping foods (strings) to counts (integers), loops over the key-value pairs, and returns the total amount of food stored in the dictionary. The function should also print the number of each individual food type as it counts up the total.

For example, if d = { "apples" : 5, "beets" : 2, "lemons" : 1 }, the function might print

5 apples
2 beets
1 lemons

then return 8.

Coding with Dictionaries

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 list of students and their college (represented as "student,college"), how many students are in each college?

We will create a dictionary with college as the key and the student count as the value.

```
def countByCollege(studentLst):
    collegeDict = { }
    for student in studentLst:
        name = student.split(",")[0]
        college = student.split(",")[1]
        if college not in collegeDict:
            collegeDict[college] = 0
        collegeDict[college] += 1
    return collegeDict
```

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
```

Coding with Dictionaries – Nested Dictionaries

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
m = createMultDict(4)
print(m[2][3]) # 6
```