Hashing and Dictionaries

15-110 – Monday 03/02

Learning Goals

- Understand how and why **hashing** makes it possible to search for values in **O(1) time**
- Compute indexes in a **hashtable** using a specific **hash function**
- Define the concept of a **key-value pair**
- Use **dictionaries** when writing and reading code that uses key-value pairs

Improving Search

We've now discussed linear search (which runs in O(n)), and binary search (which runs in O(log n)).

We use search all the time, so we want to search as quickly as possible. **Can we search for an item in O(1) time?**

We can't *always* search for things in constant time, but there are certain circumstances where we can...

Search in Real Life – Post Boxes

Consider how you receive mail. Your mail is sent to the post boxes at the lower level of the UC. Do you have to check every box to find your mail?

No- just check the one assigned to you.

This is possible because your mail has an **address** on the front that includes your mailbox number. Your mail will only be put into a box that has the same number as that address, not other random boxes.

Picking up your mail is a O(1) operation!

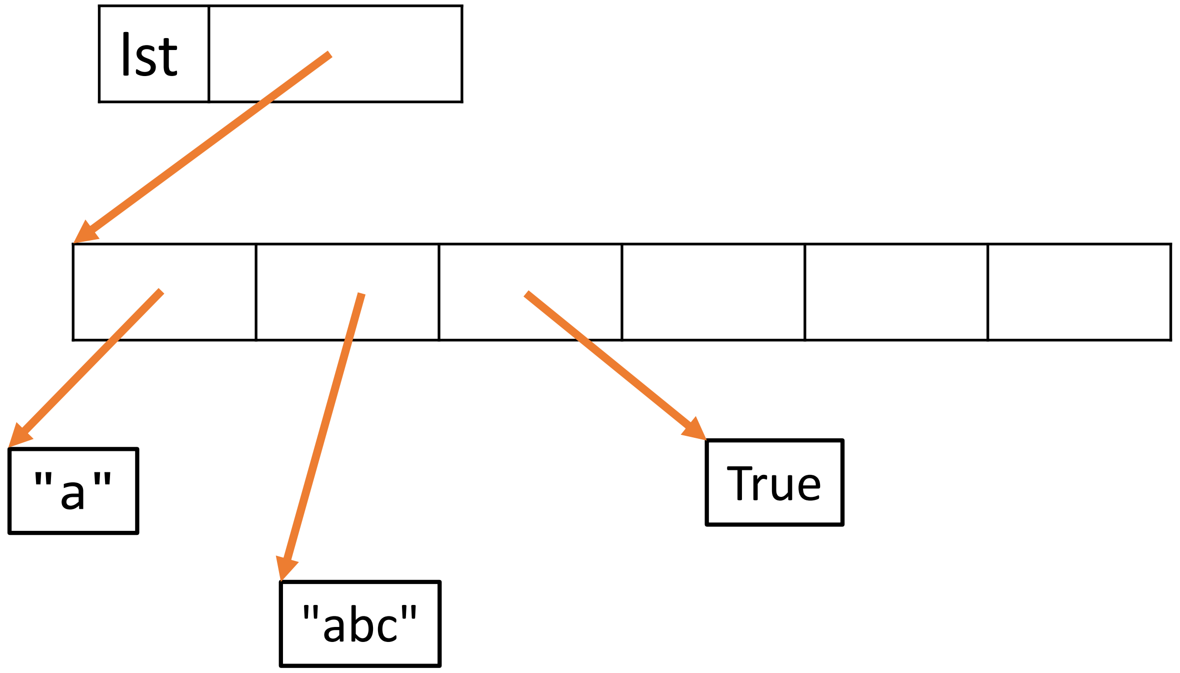
Search in Programming – List Indexes

We can't search a list for an item in constant time, but we **can** look up an item based on an index in constant time.

Reminder: Python stores lists in memory as a series of **adjacent parts**. Each part holds a reference to a single value in the list, and all the references use the same amount of space.

Example:

```
lst = ["a", "abc", True]
```



Search in Programming – List Indexes

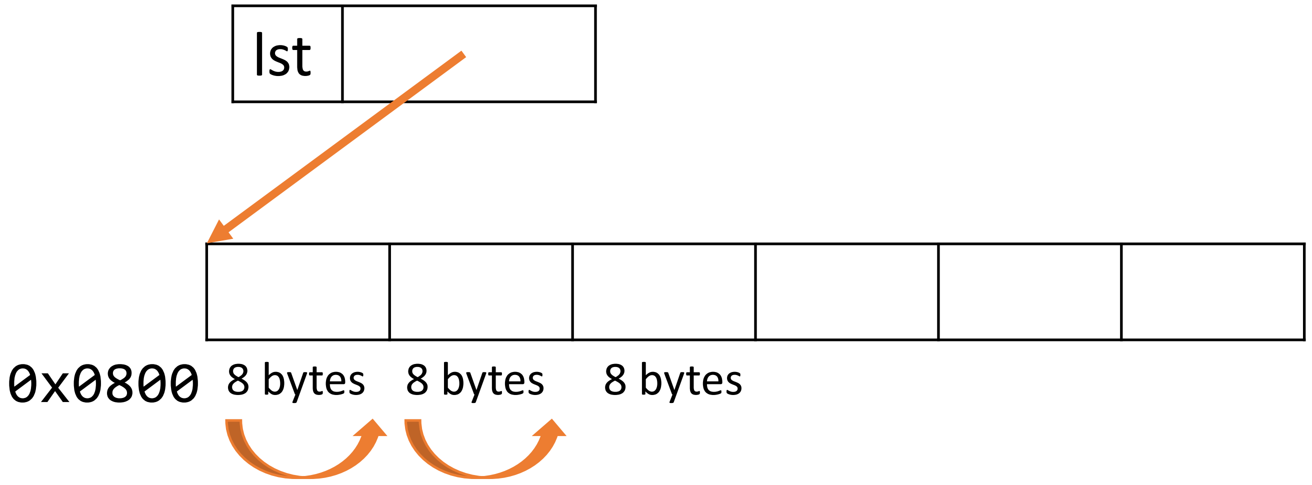
We can calculate the exact starting location of an index's memory based on the first address where `lst` is stored. If the size of a part is N, we can find an index's address with the formula:

```
start + N * index
```

Example: in the list to the right, each part is 8 bytes in size and the memory values start at `x0800`. To access `lst[2]`, compute:

```
x0800 + 8 * 2 = x0816
```

Given a memory address, we can get the value from that address in constant time. Looking up an index in a list is O(1)!



Combine the Concepts

To implement constant-time search, we want to combine the ideas of post boxes and list index lookup. Specifically, we want to be able to determine **which index a value is stored in based on the value itself**.

If we can calculate the index based on the value, we can retrieve the value in constant time.

Hashing

Hash Functions Map Values to Integers

In order to determine which list index should be used based on the value itself, we'll need to **map values to indexes**, i.e, non-negative integers.

We call a function that maps values to integers a **hash function**. This function must follow two rules:

- Given a specific value `x`, `hash(x)` must **always** return the same output `i`
- Given two different values `x` and `y`, `hash(x)` and `hash(y)` should **usually** return two different outputs, `i` and `j`

Activity: Design a Hash Function

How would you design a hash function for strings? Talk to a partner to come up with your algorithm.

Remember to follow the rules:

- Given a specific value `x`, `hash(x)` must **always** return the same output `i`
- Given two different values `x` and `y`, `hash(x)` and `hash(y)` should **usually** return two different outputs, `i` and `j`

Built-in Hash Function

We don't need to write our own hash functions most of the time- Python already has one!

```
x = "abc"
print(hash(x))
```

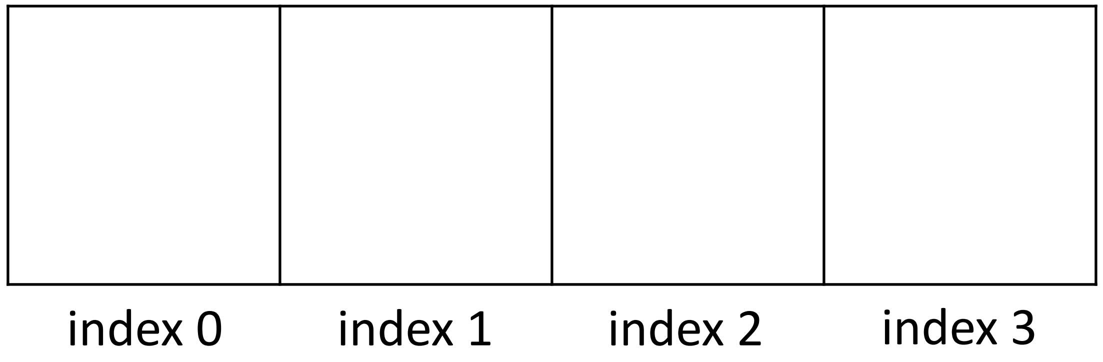
`hash()` works on integers, floats, Booleans, strings, and some other types as well.

Hashtables Organize Values

Now that we have a hash function, we can use it to organize values in a special data structure.

A **hashtable** is a list with a fixed number of indexes. When we place a value in the list, we put it into an index **based on its hash value**, instead of placing it at the end of the list.

We often call these indexes 'buckets'. For example, the hashtable to the right has four buckets. Note that actual hashtables have far more buckets than this.



Adding Values to a Hashtable

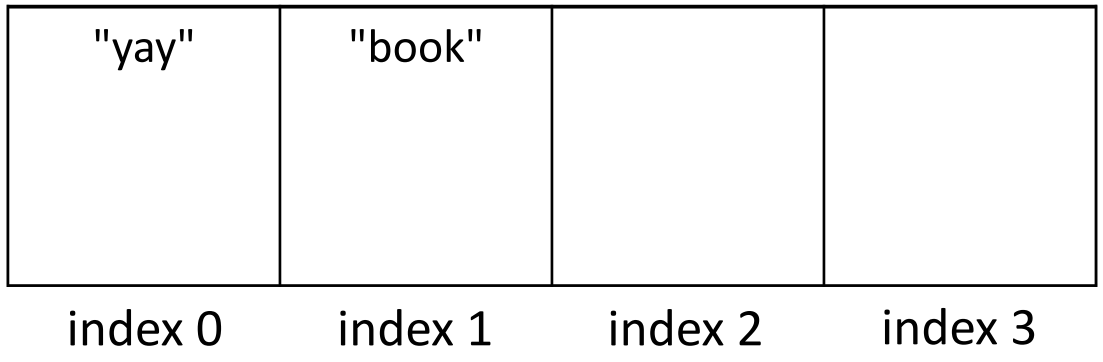
Let's say this hashtable uses a hash function that maps strings to indexes using the first letter of the string, as shown to the right.

```
def hash(s):
    return ord(s[0]) - ord('a')
```

First, add `"book"` to the table. `hash("book")` is `1`, so we'll put the value in bucket 1.

Next, add `"yay"`. The `hash("yay")` is `24`, which is outside the range of our table. How do we assign it?

Use `value % tableSize` to map integers larger than the size of the table to an index. `24 % 4 = 0`, so we put `"yay"` in bucket 0.



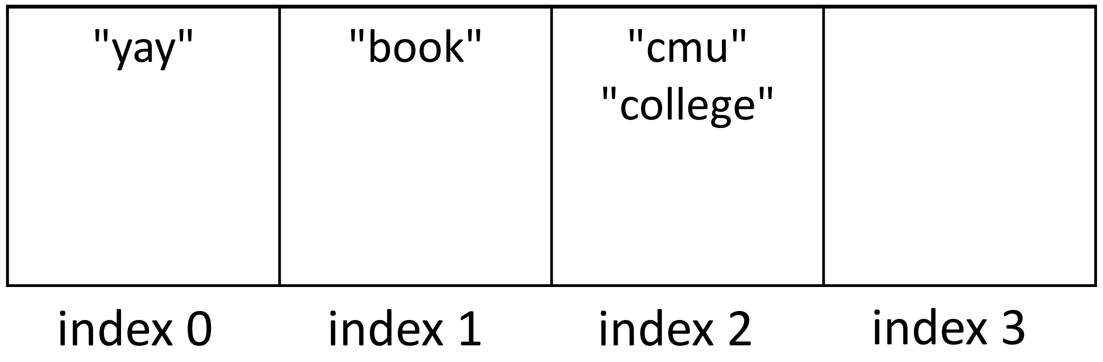
Dealing with Collisions

When you add lots of values to a hashtable, two elements **collide** if they are assigned to the same index. For example, if we try to add both `"cmu"` and `"college"` to our table, they will collide.

Hashtables are designed to handle collisions. One way is to put the collided values in a list and put that list in the bucket. If your table size is reasonably big and the indexes returned by the hash function are reasonably spread out, there will only be a **constant** number of values in each bucket.

Note: our example hash function is not good, because it only looks at the first letter. A function that uses all the letters would be better.

```
def hash(s):
    return ord(s[0]) - ord('a')
```



Searching a Hashtable is O(1)!

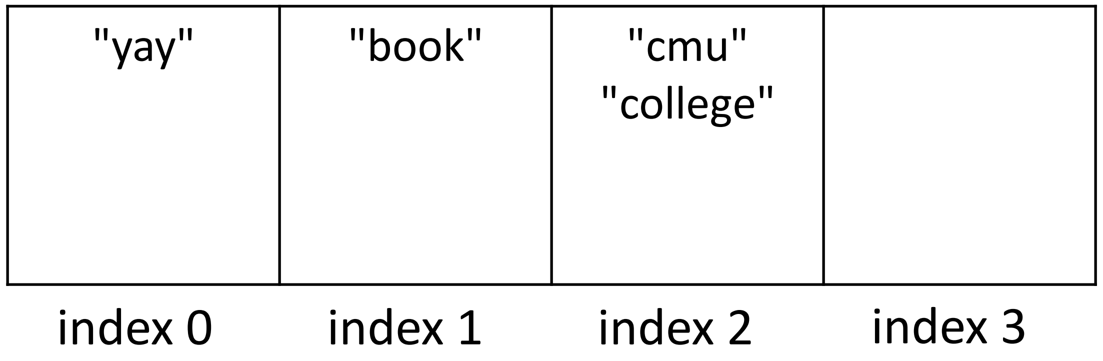
To search for a value, call the hash function on it, then mod the result by the table size. **The index produced is the only index you need to check!**

```
def hash(s):
    return ord(s[0]) - ord('a')
```

For example, we can check if `"book"` is in the table just by checking bucket 1.

If the value is in the table, it will be at that index. If it isn't, it won't be anywhere else either. To check for `"stella"` just look in in bucket 2.

Because we only need to check one index, and each index holds a constant number of items, finding a value is O(1).



Activity: Compute the Hashed Index

Assume you're using a really simple hash function that maps floats to indexes by hashing them to the value in the ones place. For example, 42.5 would hash to 2.

We want to place the number 17.46 in a four-bucket hash table.

Which bucket should it go into- 0, 1, 2, or 3?

Enter your answer on Piazza when you're ready.

Caveat: Don't Hash Mutable Values!

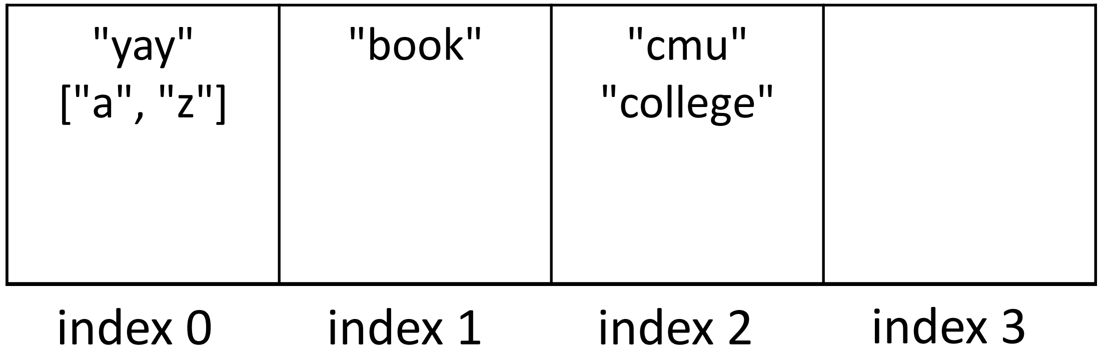
What happens if you try to put a list in a hashtable? Let's try adding the list `["a", "z"]` using the hash to the right.

This might seem fine at first, but it will become a problem if you change the list before searching. Let's say we set `lst[0]` equal to `"d"`.

Now when we hash the list, the hashed value is `3`, not `0`. But the list isn't stored in bucket 3! We can't find it reliably.

For this reason, **we don't put mutable values into hashtables**. If you try to run the built-in hash() on a list, it will crash.

```
def hash(s):
    return ord(s[0]) - ord('a')
```



Dictionaries

More Uses of Hash Functions

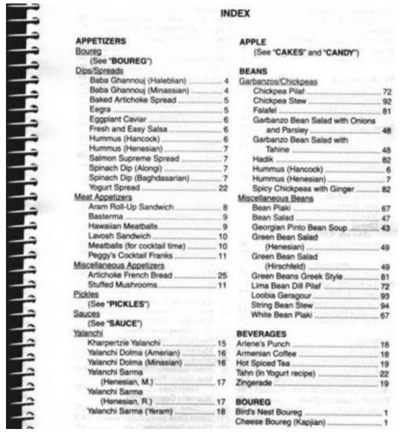
Now that we've demonstrated how hash functions work, we can use them to store data in new ways.

Our current hashtable is not a direct replication of the post box system we discussed earlier. Could we implement a post-box-like system, where we map addresses to letters?

Python Dictionaries Map Keys to Values

To implement a post box system, we'd want to use a **dictionary**, or **hashmap**. Dictionaries map **keys** (which are hashed items) to **values** (which can be anything).

We use dictionary-like data in the real world all the time! Post boxes mapping addresses to mail boxes are one example. Other 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

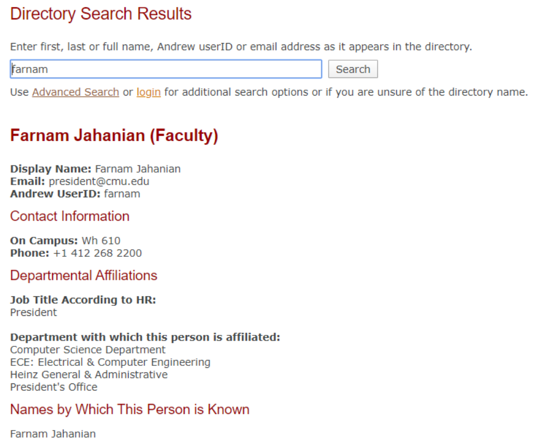
APPETIZERS
Boureg
(See "BOUREG")
Dips/Spreads
Baba Ghannouj (Haleblian) 4
Baba Ghannouj (Minassian) 4
Baked Artichoke Spread 5
Eegra 5
Eggplant Caviar 6
Fresh and Easy Salsa 6
Hummus (Hancock) 6
Hummus (Henesian) 7
Salmon Supreme Spread 7
Spinach Dip (Alongi) 7
Spinach Dip (Baghdasarian) 7
Yogurt Spread 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

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 except that, instead of being indexed by their position, dictionaries are indexed by their keys:

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

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

```
print(d.keys())
print(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 - Iteration

Dictionaries are iterable. We can use a For-Each loop to iterate over the keys of the dictionary.

```
for key in d:
    print(key, d[key]) # prints key and value
```

But we can't use a For-Range loop, because the indexes of the dictionary **are** the keys, not a range of integers.

Python Dictionaries – Search

Finally, we can **search** for a key in a dictionary in constant time, by using the built-in `in` operation.

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

Search is only constant-time because the keys of the dictionary are hashed. If we use `in` on a list or a string, it takes linear time.

We can't do constant-time lookups of the dictionary's values; we need to do a linear-time loop of the keys and check each key's value instead.

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 an iterable, 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 – Has Duplicates

We might also want to check whether a list contains duplicate values. One approach is to use nested loops. For each item in the list, search through the rest of the items to see if there is a duplicate. The nested loop has O(n^2) runtime.

We can use a dictionary to check if any student appears more than once in list of students in O(n) runtime.

```
def hasDuplicates(students):
    studentDict = { }
    for student in students:
        name = student.split(",")[0]
        college = student.split(",")[1]
        if name in studentDict:
            return True
        else:
            studentDict[name] = college
    return False
```

Learning Goals

- Understand how and why **hashing** makes it possible to search for values in **O(1) time**
- Compute indexes in a **hashtable** using a specific **hash function**
- Define the concept of a **key-value pair**
- Use **dictionaries** when writing and reading code that uses key-value pairs