Dictionaries

Joan Clarke

• Joan Clarke, played by Keira Knightly in The Imitation Game, studied mathematics at Cambridge, graduating with a double-first degree in 1939.
• Clarke was recruited to work at Bletchley, where one of her colleagues wrote, “It was a tribute to her ability that her equality with the men was never in question, even in those unenlightened days.”
• Although cryptography is still mostly male, recent initiatives are seeking to make a change.

Dictionaries and Maps

• One of the most important applications of Python objects uses them to associate pairs of data values. Although the more common term in computer science is map, Python calls this structure a dictionary.
• A dictionary associates a simple data value called a key (most often a string) with a value, which is often larger and more complex.
• Applications of the map idea exist everywhere in the real world. A classic example—which is where Python gets the name—is a dictionary. The keys are the words, and the values are the corresponding definitions.
• A more contemporary example is the World-Wide Web. In this example, the keys are the URLs, and the values are the contents of the corresponding pages.

Exercise: Maps in the Real World

• Talk with the people sitting near you and come up with at least two real-world applications in which you might use a map.

Dictionaries in Python

• Dictionaries in Python are similar in syntax to lists. In both data models, the fundamental operation is selection, which is indicated using square brackets. The difference is that index values for a dictionary need not be integers.
• When you look up a value in a dictionary, you supply the key as an expression using the square-bracket notation, as in:

```python
map[key]
```
If the key is defined in the dictionary, this selection returns the value. If no definition has been supplied, Python raises a KeyError exception.
• Dictionary selections are assignable. You can set the value associated with a key by executing an assignment statement:

```python
map[key] = value
```

Using Dictionaries in an Application

• Before going on to look at other applications, it seems worth going through the example from the text, which uses a dictionary to map three-letter airport codes to their locations.
• The association list is stored in a text file that looks like this:

```
ATL: Atlanta, GA, USA
PEK: Beijing, China
DUB: Dubai, United Arab Emirates
LAX: Los Angeles, CA, USA
HND: Tokyo, Japan
ORD: Chicago, IL, USA
LHR: London, England, United Kingdom
EWR: Hong Kong, Hong Kong
```
• The dictfile.py module shows how to read this type of data file into a Python object.
The `dictfile.py` Module

```python
def readDictionary(filename, separator=': '):
    """ Creates a dictionary by reading key-value pairs from the specified file in which
    the division between the key and value is marked by the specified separator,
    which defaults to a colon. The function discards any leading and trailing whitespace
    from both the key and the value.
    """
    dictionary = {}
    with open(filename) as f:
        for line in f:
            key = line[:index].strip()
            value = line[index + len(separator):].strip()
            dictionary[key] = value
    return dictionary
```

Finding an Airport from its Code

```python
# File: FindAirportCodes.py
from dictfile import readDictionary

def FindAirportCodes():
    airportDictionary = readDictionary('AirportCodes.txt')
    while True:
        code = input('Enter airport code: ')
        if code == '': break
        if code in airportDictionary:
            print(airportDictionary[code])
        else:
            print('There is no airport code " + code)

    # Startup code
    if __name__ == '__main__':
        FindAirportCodes()
```

Iterating Through Keys in an Object

- One of the common operations that clients need to perform when using a map is to iterate through the keys.
- Python supports this operation using the `for` statement, which has the following form:

  ```python
  for key in dictionary:
      ... code to work with the individual key and value ...
  ```

- You can also use the following idiom to iterate through the keys and values together:

  ```python
  for key, value in dictionary.items():
      ... code to work with the individual key and value ...
  ```

Finding Airports by Location

```python
# File: FindAirportsByLocation.py
from dictfile import readDictionary

def FindAirportsByLocation():
    airportDictionary = readDictionary('AirportCodes.txt')
    while True:
        str = input('Enter search string: ')[:-1]
        if str == '': break
        for code, location in airportDictionary.items():
            if str in location:
                print('code: {}, location:{}'.format(code, location))

    # Startup code
    if __name__ == '__main__':
        FindAirportsByLocation()
```

Symbol Tables

- Programming languages make use of dictionaries in several contexts, of which one of the easiest to recognize is a symbol table, which keeps track of the correspondence between variable names and their values.
- The `SymbolTable.py` application in the text implements a simple test of a symbol table that reads lines from the console, each of which is one of the following commands:
  - A simple assignment statement of the form `var = number`.
  - A variable alone on a line, which displays the variable’s value.
- Before running the program, we’re going to add two new features:
  - The command `list`, which lists all the variables.
  - The command `quit`, which exits from the program.

Sample Run of `SymbolTable.py`

```
>>> pi = 3.14159
>>> e = 2.71828
>>> x = 2
>>> pi
3.14159
>>> e
2.71828
>>> x
2
>>> a = 1.5
>>> list
a = 1.5
pi = 3.14159
e = 2.71828
x = 2
>>> quit
```
Implementation Strategies for Maps

There are several strategies you might choose to implement the map operations `get` and `put`. Those strategies include:

1. **Linear search.** This strategy keeps track of the name/value pairs in an array. In this model, both the `get` and `put` operations run in $O(N)$ time.

2. **Binary search.** If you keep the array sorted by the two-character code, you can use binary search to find the key. Using this strategy improves the performance of `get` to $O(\log N)$.

3. **Hashing.** This strategy implements a function on the space of possible keys, which tells the implementation exactly where to look for the matching item. This strategy has an average running time of $O(1)$.

Maps Can Be Fast

- If you think about the underlying implementation of maps, your first thought is likely to be that Python looks through a list of keys to find the key in question and returns the corresponding value. That approach takes time proportional to the number of keys.
- Maps, however, can be implemented much more efficiently than that. As I describe in the text, maps can be implemented so that the result is delivered almost instantly or, more accurately, so that the time required is constant no matter how many keys there are.

Hash Codes

- Python includes a built-in function called `hash` that takes an object and returns an integer, which is called the hash code for that object.
- Hash codes must adhere to the following guidelines:
  1. The hash code must be relatively easy to compute.
  2. The hash code for a particular object must always be the same for a given session of the Python interpreter.
  3. If two objects are equal, they must have the same hash code.
  4. The hash codes must be distributed as randomly as possible over the space of possible keys.
- The hashing algorithm uses the hash code for a key as a starting point in the process of searching for the associated value.

List-Based Implementation

```python
class Dictionary:
    def __init__(self):
        self._bindings = []
    def put(self, key, value):
        """Sets the binding for key to value."""
        for i in range(len(self._bindings)):
            if self._bindings[i][0] == key:
                self._bindings[i][1] = (key, value)
                return
        self._bindings.append((key, value))
    def get(self, key, value=None):
        """Retrieves the binding for key (or value)."""
        for pair in self._bindings:
            if pair[0] == key:
                return pair[1]
        return value
```

The Idea of Hashing

- Although the idea at first seems surprising, it is possible to implement the dictionary lookup operation so that it runs in constant time, independent of the number of keys in the dictionary. This $O(1)$ performance is possible only if the implementation can determine where to find a particular key.
- To get a sense of how you might achieve this goal in practice, it helps to think about how you find a word in a dictionary. Most dictionaries have thumb tabs that indicate where each letter appears. Words starting with `A` are in the `A` section, and so on.
- The most common implementations of maps use a strategy called hashing, which is conceptually similar to the thumb tabs in a dictionary. The critical idea is that you can improve performance enormously if you use the key to figure out where to look.

The Bucket Hashing Strategy

- One common strategy for implementing a map is to use the hash code for each key to select an index into an array that will contain all the keys with that hash code. Each element of that array is conventionally called a bucket.
- In practice, the array of buckets is smaller than the number of hash codes, making it necessary to convert the hash code into a bucket index, typically by executing a statement like
  ```python
  bucket = abs(hash(key)) % nBuckets
  ```
  The value in each element of the bucket array cannot be a single key/value pair given the chance that different keys fall into the same bucket. Such situations are called collisions.
- To take account of the possibility of collisions, each element of the bucket array is a list of the keys that fall into that bucket, as illustrated on the next slide.
Simulating Bucket Hashing

```python
stateMap.put('AK', 'Alaska')
hashCode('AK') → 5862129 (0 mod 7)
The key "AK" therefore goes in bucket 0.
```

```python
stateMap.put('AL', 'Alabama')
hashCode('AL') → 5862130 (0 mod 7)
The key "AL" therefore goes in bucket 1.
```

```python
stateMap.put('AR', 'Arkansas')
hashCode('AR') → 5862136 (0 mod 7)
The key "AR" also goes in bucket 0.
```

```python
Suppose you call stateMap.get('NV')
hashCode('NV') → 5862569 (6 mod 7)
The key "NV" must therefore be in bucket 6
and can be located by searching the list.
```

A Hash Table Implementation

```python
class Dictionary:
    INITIAL_NUMBER_OF_BUCKETS = 16
    def __init__(self):
        self._buckets = []
        for i in range(self.INITIAL_NUMBER_OF_BUCKETS):
            self._buckets.append([])
```

```python
def put(self, key, value):
    bucket = self._hash(key) % len(self._buckets)
    for i in range(len(self._buckets[bucket])):
        if self._buckets[bucket][i][0] == key:
            self._buckets[bucket][i] = (key, value)
            return
    self._buckets[bucket].append((key, value))
```

```python
def get(self, key, defaultValue=None):
    bucket = self._hash(key) % len(self._buckets)
    for i in range(len(self._buckets[bucket])):
        if self._buckets[bucket][i][0] == key:
            return self._buckets[bucket][i][1]
    return defaultValue
```

Achieving O(1) Performance

- The simulation on the previous side uses only seven buckets to emphasize what happens when collisions occur: the smaller the number of buckets, the more likely collisions become.
- In practice, the implementation of Dictionary would use a much larger value for nBuckets to minimize the opportunity for collisions. If the number of buckets is considerably larger than the number of keys, most of the bucket chains will either be empty or contain exactly one key/value pair.
- The ratio of the number of keys to the number of buckets is called the load factor of the map. Because a map achieves O(1) performance only if the load factor is small, the library implementation of Dictionary increases the number of buckets when the table becomes too full. This process is called rehashing.

Denial of Service Attacks

- Hashing algorithms must be designed to minimize the danger of a denial-of-service attack.
- If a hacker knows exactly how a hash algorithm works, it is easy to produce meaningless keys that collide.
- Making a series of requests with those keys slows down the server, increasing the effectiveness of the attack.
- To thwart this danger, the Python implementation of hash sets a random value as a seed each time the Python interpreter is started and then adds that seed to the hash code calculation.
- This strategy is effective because attackers can no longer tell which keys are going to collide.