Chapter 14
Tuples, Sets, and Dictionaries

Motivations

The **No Fly List** is a list, created and maintained by the United States government's Terrorist Screening Center, of people who are not permitted to board a commercial aircraft for travel in or out of the United States. Suppose we need to write a program that checks whether a person is in the No Fly List. You can use a Python list to store the persons in the No Fly List. However, a more efficient data structure for this application is a set.

Objectives

- To use tuples as immutable lists (§14.2).
- To use sets for storing and fast accessing non-duplicate elements (§14.3).
- To understand the performance differences between sets and lists (§14.4).
- To store key/value pairs in a dictionary and access value using the key (§14.5).
- To use dictionaries to develop applications (§14.6).

Tuples

Tuples are like lists except they are immutable. Once they are created, their contents cannot be changed.

If the contents of a list in your application do not change, you should use a tuple to prevent data from being modified accidentally. Furthermore, tuples are more efficient than lists.

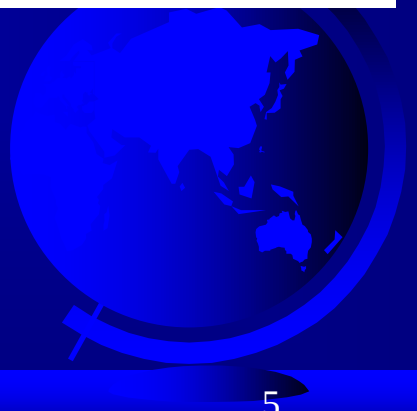
Creating Tuples

```
t1 = () # Create an empty tuple

t2 = (1, 3, 5) # Create a set with three elements

# Create a tuple from a list
t3 = tuple([2 * x for x in range(1, 5)])

# Create a tuple from a string
t4 = tuple("abac") # t4 is ['a', 'b', 'a', 'c']
```



Tuples

Tuples can be used like lists except they are immutable.

TupleDemo

Run

Sets

Sets are like lists to store a collection of items. Unlike lists, the elements in a set are unique and are not placed in any particular ordered. If your application does not care about the order of the elements, using a set to store elements is more efficient than using lists. The syntax for sets is braces {}.

Creating Sets

```
s1 = set() # Create an empty set

s2 = {1, 3, 5} # Create a set with three elements

s3 = set([1, 3, 5]) # Create a set from a tuple

# Create a set from a list
s4 = set([x * 2 for x in range(1, 10)])

# Create a set from a string
s5 = set("abac") # s5 is {'a', 'b', 'c'}
```

Manipulating and Accessing Sets

```
>>> s1 = {1, 2, 4}
>>> s1.add(6)
>>> s1
{1, 2, 4, 6}
>>> len(s1)
4
>>> max(s1)
6
>>> min(s1)
1
>>> sum(s1)
13
>>> 3 in s1
False
>>> s1.remove(4)
>>> s1
{1, 2, 6}
>>>
```

Subset and Superset

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 4, 5, 2, 6}
>>> s1.issubset(s2) # s1 is a subset of s2
True
>>>
```

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 4, 5, 2, 6}
>>> s2.issuperset(s1) # s2 is a superset of
s1
True
>>>
```

Equality Test

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 4, 2}
>>> s1 == s2
True
>>> s1 != s2
False
>>>
```

Comparison Operators

Note that it makes no sense to compare the sets using the conventional comparison operators (>, >=, <=, <), because the elements in a set are not ordered. However, these operators have special meaning when used for sets.

s1 > s2 returns true is s1 is a proper superset of s2.

s1 >= s2 returns true is s1 is a superset of s2.

s1 < s2 returns true is s1 is a proper subset of s2.

s1 <= s2 returns true is s1 is a subset of s2.

Set Operations (union, |)

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.union(s2)
{1, 2, 3, 4, 5}
>>>
>>> s1 | s2
{1, 2, 3, 4, 5}
>>>
```

Set Operations (intersection, &)

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.intersection(s2)
{1}
>>>
>>> s1 & s2
{1}
>>>
```

Set Operations (difference, -)

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.difference(s2)
{2, 4}
>>>
>>> s1 - s2
{2, 4}
>>>
```

Set Operations (symetric_difference, ^)

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.symmetric_difference(s2)
{2, 3, 4, 5}
>>>
>>> s1 ^ s2
{2, 3, 4, 5}
>>>
```

Sets

SetDemo

Run

Comparing Performance of Sets and Lists

SetListPerformanceTest

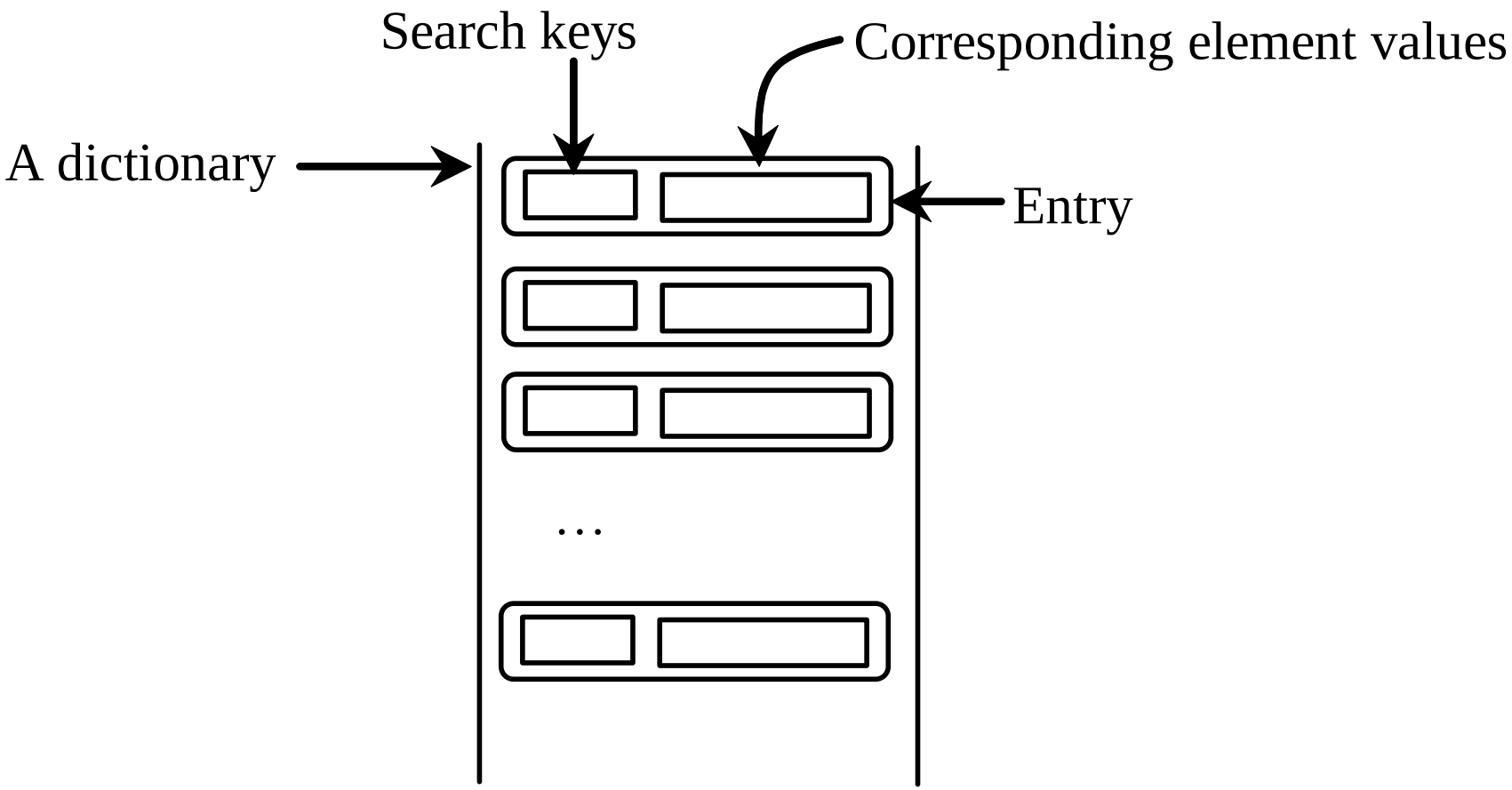
Run

Dictionary

Why dictionary?

Suppose your program stores a million students and frequently searches for a student using the social security number. An efficient data structure for this task is the *dictionary*. A dictionary is a collection that stores the elements along with the keys. The keys are like an indexer.

Key/value pairs



Creating a Dictionary

```
dictionary = {} # Create an empty dictionary
dictionary = {"john":40, "peter":45} # Create a dictionary
```

Adding/Modifying Entries

To add an entry to a dictionary, use

```
dictionary[key] = value
```

For example,

```
dictionary["susan"] = 50
```

Deleting Entries

To delete an entry from a dictionary, use

```
del dictionary[key]
```

For example,

```
del dictionary["susan"]
```

Looping Entries

```
for key in dictionary:
    print(key + ":" + str(dictionary[key]))
```

The len and in operators

len(dictionary) returns the number of the elements in the dictionary.

```
>>> dictionary = {"john":40, "peter":45}
>>> "john" in dictionary
True
>>> "johnson" in dictionary
False
```

The Dictionary Methods

dict	
keys(): tuple	Returns a sequence of keys.
values(): tuple	Returns a sequence of values.
items(): tuple	Returns a sequence of tuples (key, value).
clear(): void	Deletes all entries.
get(key): value	Returns the value for the key.
pop(key): value	Removes the entry for the key and returns its value.
popitem(): tuple	Returns a randomly-selected key/value pair as a tuple and removes the selected entry.

Case Studies: Occurrences of Words

This case study writes a program that counts the occurrences of words in a text file and displays the words and their occurrences in alphabetical order of words. The program uses a dictionary to store an entry consisting of a word and its count. For each word, check whether it is already a key in the dictionary. If not, add to the dictionary an entry with the word as the key and value 1. Otherwise, increase the value for the word (key) by 1 in the dictionary.

CountOccurrenceOfWords