*12.5 A DICTIONARY ADT

A dictionary abstract data type stores (key, value) pairs, and allows us to look up a value by specifying its key. In the Python implementation of a dictionary, we insert a (key, value) pair with the assignment statement `dictionary[key] = value` and retrieve a value with `dictionary[key]`. We have used dictionaries in a variety of applications: word frequencies in Section 7.3, Lindenmayer systems in Section 9.6, networks in Chapter 11, and the flocking simulation earlier in this chapter. Another simple application is shown below: storing the history of World Series champions by associating each year with the name of the winning team.

```
def main():
    worldSeries = { }
    worldSeries[1903] = 'Boston Americans'
    ⋮
    worldSeries[1979] = 'Pittsburgh Pirates'
    ⋮
    worldSeries[2014] = 'San Francisco Giants'

    print(worldSeries[1979])  # prints "Pittsburgh Pirates"
```

In this section, we will implement our own Dictionary abstract data type to illustrate how more complex collection types can be implemented. Formally, a dictionary ADT simply contains a collection of (key, value) pairs and a length.

Instance Variable	Description
pairs	a collection of (key, value) pairs
length	the number of (key, value) pairs

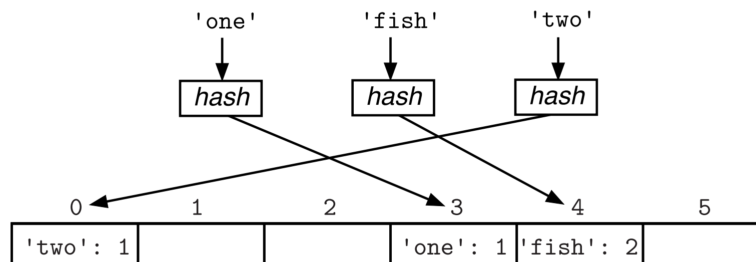
The most fundamental operations supported by a Dictionary are insert, delete, and lookup (all of which are implemented using indexing in the built-in Python dictionary). These, and a few other useful operations, are defined below.

Name	Arguments	Description
create	—	create a new empty Dictionary instance
insert	key, value	insert a (key, value) pair
delete	key	delete the pair with a given key
lookup	key	return the value associated with a key
contains	key	return true if the key exists, false otherwise
items	—	return a list of all (key, value) pairs
keys	—	return a list of all keys
values	—	return a list of all values
length	—	return the number of (key, value) pairs
is empty	—	return true if the dictionary is empty, false otherwise

As we discussed in Tangent 7.2, Python dictionaries are implemented using a data structure

Copyright Taylor and Francis, 2021

called a *hash table*. A hash table consists of a fixed number of *slots* in which the key:value pairs are stored. A *hash function* is used to map a key to the index of a slot. For example, in the illustration below from Tangent 7.2, the key:value pair `18.9:2` is mapped by the hash function to slot 3.



Hash tables

Although there are several other ways that a dictionary ADT could be implemented (such as with a binary search tree, as in Project 10.2), let's look at how to do so with our own hash table implementation. When implementing a hash table, there are two main issues that we need to consider. First, we need to decide what our hash function should be. Second, we need to decide how to handle *collisions*, which occur when more than one key is mapped to the same slot by the hash function. We will leave an answer to the second question as Project 12.2.

A hash function should ideally spread keys uniformly throughout the hash table to efficiently use the allocated space and to prevent collisions. Assuming that all key values are integers, the simplest hash functions have the form

$$\text{hash}(\text{key}) = \text{key} \bmod n,$$

where n is the number of slots in the hash table. Hash functions of this form are said to use the ***division method***.

Reflection 1 *What range of possible slot indices is given by this hash function?*

To prevent patterns from emerging in the slot assignments, we typically want n to be a prime number.

Reflection 2 *If n were even, what undesirable pattern might emerge in the slot indices assigned to keys? (For example, what if all keys were odd?)*

Reflection 3 *Suppose $n = 11$. What is the value of* hash(key) *for key values 7, 11, 14, 25, and 100?*

There are many better hash functions that have been developed, but the topic is outside the scope of our discussion here. If you are interested, we give some pointers to additional reading at the end of the chapter. In the exercises, we discuss how you might define a hash function for keys that are strings.

Implementing a hash table

To demonstrate how hash tables work, we will start by implementing the dictionary ADT with a small hash table containing only $n = 11$ slots. We will also assume that no collisions occur, leaving the resolution of this issue to Project 12.2.

```
class Dictionary:
    """A dictionary class."""

    _KEY = 0      # index of key in each pair
    _VALUE = 1    # index of value in each pair

    def __init__(self):
        """Construct a new Dictionary object."""

        self._length = 0      # number of (key, value) pairs
        self._size = 11       # number of slots in the hash table
        self._table = [None] * self._size  # hash table containing
                                           #  (key, value) pairs
    def _hash(self, key):
        return key % self._size

    def insert(self, key, value):
        """Insert a key:value pair into self."""

        index = self._hash(key)
        self._table[index] = (key, value)
        self._length = self._length + 1
```

The constructor creates an empty hash table with 11 slots. We represent an empty slot with `None`. When a (key, value) pair is inserted into a slot, it will be represented with a `(key, value)` tuple. The `_hash` method implements our simple hash function. The leading underscore (`_`) character signifies that this method is *private* and should never be called from outside the class. (The leading underscore also tells the `help` function not to list this method.) We will discuss this further below. The `insert` method begins by using the hash function to map the given `key` value to the `index` of a slot in the hash table. Then the inserted `(key, value)` tuple is assigned to this slot (`self._table[index]`) and the number of entries in the dictionary is incremented.

Reflection 4 *With this implementation, how would you create a new dictionary and insert* (1903, Boston Americans), (1979, Pittsburgh Pirates), *and* (2014, San Francisco Giants) *into it (as in the* `main` *function at the beginning of this section)?*

With this implementation, these three insertions would be implemented as follows:

```
worldSeries = Dictionary()
worldSeries.insert(1903, 'Boston Americans')
worldSeries.insert(1979, 'Pittsburgh Pirates')
worldSeries.insert(2014, 'San Francisco Giants')
```

Reflection 5 *Using the* `insert` *method as a template, write the* `delete` *method.*

The `delete` method is very similar to the `insert` method *if* the key to delete exists in the hash table. In this case, the only differences are that we do not need to pass in a value, we assign `None` to the slot instead of a tuple, and we decrement the number of items.

Reflection 6 *How do we determine whether the key that we want to delete is contained in the hash table?*

If the slot to which the key is mapped by the hash function contains the value `None`, then the key must not exist. But even if the slot does not contain `None`, it may be that a different

key resides there, so we still need to compare the value of the key in the slot with the value of the key we wish to delete. Since the key is the first value in the tuple assigned to `self._table[index]`, the key is in `self._table[index][0]`. Therefore, the required test looks like this:

```
if (self._table[index] != None) and (self._table[index][0] == key):
    # delete pair
else:
    # raise a KeyError exception
```

The `KeyError` exception is the exception raised when a key is not found in Python's built-in dictionary. The fleshed-out `delete` method is shown below.

```
    def delete(self, key):
        """Delete the pair with a given key."""

        index = self._hash(key)
        if (self._table[index] != None) and \
           (self._table[index][self._KEY] == key):
            self._table[index] = None
            self._length = self._length - 1
        else:
            raise KeyError('key was not found')
```

To prevent the use of "magic numbers," we defined two constant class variables, `_KEY` and `_VALUE`, that correspond to the indices of the key and value in a tuple.

Reflection 7 *How do we delete* `(2014, 'San Francisco Giants')` *from the* `worldSeries` `Dictionary` *object that we created previously?*

To delete this pair, we simply call

```
worldSeries.delete(2014)
```

Reflection 8 *Now write the* `lookup` *method for the* `Dictionary` *class.*

To retrieve a value corresponding to a key, we once again find the index corresponding to the key and check whether the key is present. If it is, we return the corresponding value. Otherwise, we raise an exception.

```
    def lookup(self, key):
        """Return the value associated with a key."""

        index = self._hash(key)
        if (self._table[index] != None) and \
           (self._table[index][self._KEY] == key):
            return self._table[index][self._VALUE]
        else:
            raise KeyError('key was not found')
```

Reflection 9 *How do we look up the 1979 World Series champion in the* `worldSeries` `Dictionary` *object that we created previously?*

To look up the winner of the 1979 World Series, we simply call the `lookup` method with the key `1979`:

```
champion = worldSeries.lookup(1979)
print(champion)                     # prints "Pittsburgh Pirates"
```

We round out the class with a method that returns the number of (key, value) pairs in the dictionary. In the built-in collection types, the length is returned by the built-in `len` function. We can define the same behavior for our `Dictionary` class by defining the special `__len__` method.

```
    def __len__(self):
        """Return the number of (key, value) pairs."""

        return self._length

    def isEmpty(self):
        """Return true if the dictionary is empty, false otherwise."""

        return len(self) == 0
```

With the `__len__` method defined, we can find out how many entries are in the `worldSeries` dictionary with `len(worldSeries)`. Notice that in the `isEmpty` method above, we also call `len`. Since `self` refers to a `Dictionary` object, `len(self)` implicitly invokes the `__len__` method of `Dictionary` as well.

The following `main` function combines the previous examples to illustrate the use of our class.

```
def main():
    worldSeries = Dictionary()
    worldSeries.insert(1903, 'Boston Americans')
    worldSeries.insert(1979, 'Pittsburgh Pirates')
    worldSeries.insert(2014, 'San Francisco Giants')
    print(worldSeries.lookup(1979))  # prints "Pittsburgh Pirates"
    worldSeries.delete(2014)
    print(len(worldSeries))          # prints 2
    print(worldSeries.lookup(2014))  # KeyError
```

Indexing

As we already discussed, Python's built-in dictionary class implements insertion and retrieval of (key, value) pairs with indexing rather than explicit methods as we used above. We can also mimic this behavior by defining the `__getitem__` and `__setitem__` methods, as we did for the `Pair` class earlier. The `__getitem__` method takes a single index parameter and the `__setitem__` method takes an index and a value as parameters, just as our `lookup` and `insert` methods do. Therefore, to use indexing with our `Dictionary` class, we only have to rename our existing methods, as follows.

```
    def __setitem__(self, key, value):
        """Insert a (key, value) pair into self."""

        index = self._hash(key)
        self._table[index] = (key, value)
        self._length = self._length + 1
```

```
    def __getitem__(self, key):
        """Return the value associated with a key."""

        index = self._hash(key)
        if (self._table[index] != None) and \
           (self._table[index][self._KEY] == key):
            return self._table[index][self._VALUE]
        else:
            raise KeyError('key was not found')
```

The most general method for deleting an item from a Python collection is to use the `del` operator. For example,

```
del frequency[18.9]
```

deletes the (key, value) pair with key equal to `18.9` from the dictionary named `frequency`. We can mimic this behavior by renaming our `delete` method to `__delitem__`, as follows.

```
    def __delitem__(self, key):
        """Delete the pair with a given key."""

        index = self._hash(key)
        if self._table[index] != None and \
           self._table[index][self._KEY] == key:
            self._table[index] = None
            self._length = self._length - 1
        else:
            raise KeyError('key was not found')
```

With these three changes, the following `main` function is equivalent to the one above.

```
def main():
    worldSeries = Dictionary()
    worldSeries[1903] = 'Boston Americans'
    worldSeries[1979] = 'Pittsburgh Pirates'
    worldSeries[2014] = 'San Francisco Giants'
    print(worldSeries[1979])  # prints "Pittsburgh Pirates"
    del worldSeries[2014]
    print(len(worldSeries))   # prints 2
    print(worldSeries[2014])  # KeyError
```

ADTs vs. data structures

The difference between an abstract data type and a data structure is perhaps one of the most misunderstood concepts in computer science. As we have discussed, an ADT is an abstract description of a data type that is independent of any particular implementation. A data structure is a particular implementation of an abstract data type. In this section, we implemented a Dictionary ADT using a hash table data structure. But we could also have used a simple list of (key, value) pairs or something more complicated like a binary search tree from Project 10.2. *Our choice of data structure does not change the definition of the abstract data type.*

When someone uses a class that implements an abstract data type, the data structure should remain completely hidden. In other words, we must not make any public methods (those without leading underscores) dependent upon the underlying data structure. For example, we did not make the number of slots in the hash table a parameter to the constructor because this parameter is not part of the ADT and would not make sense if the ADT were implemented with a different data structure. Likewise, we indicated with a leading underscore that the `_hash` method should be private because this method is only necessary if the Dictionary is implemented with a hash table. Similarly, we indicated that instance and class variables in the `Dictionary` class should also remain private (with leading underscores) because they only make sense in the context of the chosen data structure.

Collectively, the non-private methods of the class constitute the only information that a programmer using the class should need to know. This is precisely what is displayed by `help(Dictionary)`.

```
class Dictionary(builtins.object)
 |  A dictionary class.
 |
 |  Methods defined here:
 |
 |  __delitem__(self, key)
 |      Delete the pair with a given key.
 |
 |  __getitem__(self, key)
 |      Return the value associated with a key.
 |
 |  __init__(self)
 |      Construct a new Dictionary object.
 |
 |  __len__(self)
 |      Return the number of (key, value) pairs.
 |
 |  __setitem__(self, key, value)
 |      Insert a (key, value) pair into self.
 |
 |  isEmpty(self)
 |      Return true if the dictionary is empty, false otherwise.
```

Exercises

12.5.1* Add a private method named `_printTable` to the `Dictionary` class that prints the contents of the underlying hash table. For example, for the dictionary created in the `main` function above, the method should print something like this:

```
0:  (1903, 'Boston Americans')
1:  (2014, 'San Francisco Giants')
2:  None
3:  None
4:  None
5:  None
6:  None
7:  None
8:  None
9:  None
10: (1979, 'Pittsburgh Pirates')
```

12.5.2* Add a `__str__` method to the `Dictionary` class. The method should return a string similar to what is printed for a built-in Python dictionary. It should not divulge any information about the underlying hash table implementation. For example, for the dictionary displayed in the previous exercise, the method should return a string like this:

```
"{1903: 'Boston Americans', 2014: 'San Francisco Giants',
1979: 'Pittsburgh Pirates'}"
```

12.5.3. Implement use of the `in` operator for the `Dictionary` class by adding a method named `__contains__`. The method should return `True` if a key is contained in the `Dictionary` object, or `False` otherwise.

12.5.4. Implement `Dictionary` methods named `items`, `keys`, and `values` that return the list of (key, value) tuples, keys, and values, respectively. In any sequence of calls to these methods, with no modifications to the `Dictionary` object between the calls, they must return lists in which the orders of the items correspond. In other words, the first value returned by `values` will correspond to the first key returned by `keys`, the second value returned by `values` will correspond to the second key returned by `keys`, etc. The order of the tuples in the list returned by `items` must be the same as the order of the lists returned by `keys` and `values`.

12.5.5. Show how to use the `keys` method that you wrote in the previous exercise to print a list of keys and values in a `Dictionary` object in alphabetical order by key.

12.5.6* Write a function that uses the `Dictionary` class to find the frequencies of all the items in a list.

12.5.7. Use the `Dictionary` class to implement the `removeDuplicates5` function on page O7.5-9.

12.5.8. If keys are string values, then a new hash function is needed. One simple idea is to sum the Unicode values corresponding to the characters in the string and then return the sum modulo n. Implement this new hash function for the `Dictionary` class.

12.5.9. Design and implement a hash function for keys that are tuples.

12.5.10* Write a class named `Presidents` that maintains a list of all of the U.S. presidents. The constructor should take the number of presidents as a parameter and initialize a list of empty slots, each containing the value `None`. Then add `__setitem__` and `__getitem__` methods that insert and return, respectively, a `President` object (from Exercise 12.1.10) with the given chronological number (starting at 1). Be sure to check in each method whether the parameters are valid. Also add a `__str__` method that returns a complete list of the presidents in chronological order. If a president is missing, replace the name with question marks. For example, the following code

```python
presidents = Presidents(44)
washington = President('George Washington')
kennedy = President('John F. Kennedy')
presidents[1] = washington
presidents[35] = kennedy
print(presidents[35])   # prints Kennedy
print(presidents)
```

should print

```
John F. Kennedy
1. George Washington
2. ???
3. ???
⋮
34. ???
35. John F. Kennedy
36. ???
⋮
44. ???
```

Also include a method that does the same thing as the function from Exercise 12.1.10. In other words, your method should, given an age, print a table with all presidents who where at least that old when they took office, along with their ages when they took office.

12.5.11. Write a class that stores all the movies that have won the Academy Award for Best Picture. Inside the class, use a list of `Movie` objects from Exercise 12.1.11. Your class should include `__getitem__` and `__setitem__` methods that return and assign the movie winning the award in the given edition of the ceremony. (The 87th Academy Awards were held in 2015.) In addition, include a `__str__` method that returns a string containing the complete list of the winning titles in chronological order. If a movie is missing, replace the title with question marks. Finally, include a method that checks whether the winners in two given editions have actors in common (by calling the method from Exercise 12.1.11). You can find a complete list of winners at `http://awardsdatabase.oscars.org/`

12.5.12. Write a class named `Roster` that stores information about all of the students enrolled in a course. In the `Roster` class, store the information about the students using a list of `Student` objects from Exercise 12.1.13. Each student will also have an associated ID number, which can be used to access the student through the class' `__getitem__` and `__setitem__` methods. Also include a `__len__` method that returns the number of students enrolled, a method that returns the average of all of the exam grades for all of the students, a method that returns the average overall grade (using the weights in Exercise 12.1.13), and a `__str__` method that returns a string representing the complete roster with current grades. Use of this class is illustrated with the following short segment of code:

```
roster = Roster()
alice = Student('Alice Jones')
bob = Student('Bob Smith')
roster[101] = alice
roster[102] = bob
roster[101].addExam(100)
roster[102].addExam(85)
print(roster.examAverage())   # prints 92.5
print(roster.averageGrade())  # prints 46.25
print(roster)                 # prints:
                              # 101  Alice Jones      50.00
                              # 102  Bob Smith        42.50
```

Selected Exercise Solutions

12.5.1
```
def _printTable(self):
    for index in range(self._size):
        print(str(index) + ': ' + str(self._table[index]))
```

12.5.2
```
def __str__(self):
    allItems = []
    for slot in self._table:
        if slot != None:
            allItems.append(slot)

    dictString = '{'
    for pair in allItems[:-1]:
        dictString = dictString + repr(pair[self._KEY]) + ': '
                                + repr(pair[self._VALUE]) + ', '
    if len(allItems) > 0:
        dictString = dictString + repr(allItems[-1][self._KEY]) + ': '
                                + repr(allItems[-1][self._VALUE])
    return dictString + '}'
```

12.5.6
```
def frequencies(data):
    itemFreqs = Dictionary()
    for item in data:
        if item in itemFreqs:
            itemFreqs[item] = itemFreqs[item] + 1
        else:
            itemFreqs[item] = 1
    return itemFreqs
```

12.5.10
```
class Presidents:
    def __init__(self, numPresidents):
        self._list = [None] * numPresidents

    def __getitem__(self, number):
        if number >= 1 and number <= len(self._list):
            return self._list[number - 1]
        else:
            return None

    def __setitem__(self, number, president):
        if number >= 1 and number <= len(self._list):
            self._list[number - 1] = president

    def __str__(self):
        presList = ''
        for number in range(len(self._list)):
            if self._list[number] != None:
                presList = presList + str(number + 1) + '. '
                                    + str(self._list[number]) + '\n'
            else:
                presList = presList + str(number + 1) + '. ???\n'
        return presList

    def olderPresidents(self, age):
        print('Name                 Age')
```

```
print('-------------------- ---')
for president in self._list:
    if president.getAge() >= age:
       print('{0:<20} {1:>3}'.format(president.getName(),
                president.getAge()))
```

Copyright Taylor and Francis, 2021