Sets and Maps

5

In the last chapter we studied sequences which are used to keep track of lists of things where duplicate values are allowed. For instance, there can be two sixes in a sequence or list of integers. In this chapter we look at *sets* where duplicate values are not allowed. After examining sets we'll move on to talk about *maps*. *Maps* may also be called *dictionaries* or *hash tables*.

The term *hash table* actually suggests an implementation of a set or map. The primary focus of this chapter is in understanding *hashing*. Hashing is a very important concept in Computer Science because it is a very efficient method of searching for a value. To begin the chapter we'll motivate our interest in hashing, then we'll develop a hashing algorithm for finding values in a set. We'll also apply hashing to the building of sets and maps. Then we'll look at an important technique that uses hashing called *memoization* and we'll apply that technique to a couple of problems.

5.1 Chapter Goals

In this chapter you learn how to implement a couple of abstract datatypes: sets and maps. You will read about the importance of hashing for both of these datatypes. You'll also learn about the importance of understanding the difference between mutable and immutable data. By the end of the chapter you should be able to answer these questions.

- What is the complexity of finding a value in a set?
- What is the *load factor* and how does it affect the overall efficiency of lookup in a hash table?
- When would you use an immutable set?
- When might you want a mutable set?
- When is there an advantage to using memoization in a problem?
- Under what circumstances can it be useful to use a map or dictionary?

K.D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-13072-9_5

Again, there will be some interesting programming problem challenges in this chapter including optimization of the tic tac toe game first presented in the last chapter and a Sudoku puzzle solver. Read on to discover what you need to know to solve these interesting problems.

5.2 Playing Sudoku

Many people enjoy solving Sudoku puzzles. To solve a Sudoku puzzle you must find the correct numbers to fill a 9×9 matrix. All numbers must be 1–9. Each row must have one each of 1–9 in it. The same is true for each column. Finally, there are nine 3×3 squares within the 9×9 matrix that must also have each of 1–9 in them. To begin, you are given a puzzle with some of the locations known as shown in Fig. 5.1. Your job is to find the rest of the numbers given the numbers that already appear in the puzzle.

A common way to solve these puzzles is by the process of elimination. It helps to write down the possible values for a puzzle and then eliminate the possible values one by one. For instance, the puzzle above can be annotated with possible values in for the unknowns as shown in Fig. 5.2.

To begin solving the puzzle, we can immediately eliminate 8, 3 and 9 from the second column of the puzzle for those cells that do not contain 8, 3 or 9. None of those numbers could appear in any other cell in the second column because they are already known. Likewise, the numbers 8, 6 and 4 can be eliminated from some cells

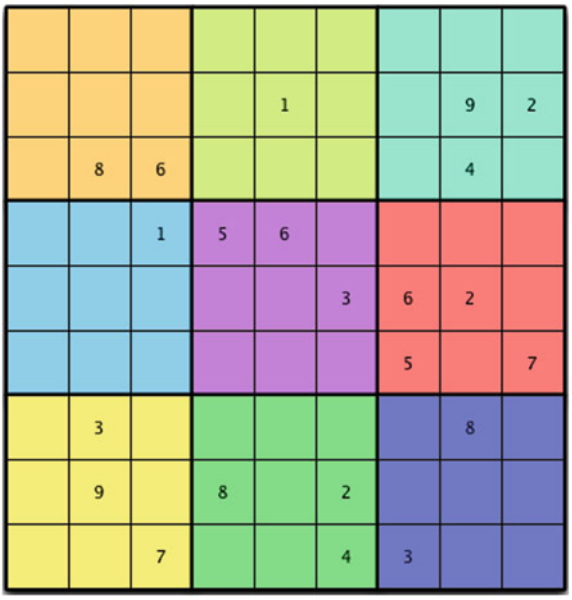

Fig. 5.1 A Sudoku Puzzle

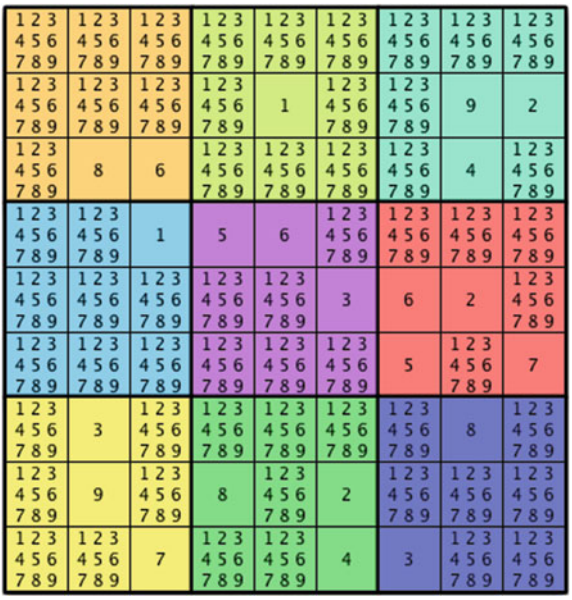

Fig. 5.2 Annotated Sudoku Puzzle

in the third row in the puzzle because those numbers already appear in other cells. Applying rules like these reduces the number of possible values for each cell in the puzzle. Figure 5.3 shows the puzzle after applying some of these rules.

If we spend some time thinking about Sudoku and how to solve it we can derive two rules that can be used to solve many Sudoku puzzles. These two rules can be applied to any group within the puzzle. A group is a collection of nine cells that appear in a row, column, or square within the puzzle. Within each cell is a set of numbers representing the possible values for the cell at some point in the process of reducing the puzzle. Here are the two rules.

- **RULE 1**. The first rule is a generalization of the process that we used above to remove some values from cells. Within a group look for cells that contain the same set of possible values. If the cardinality of the set (i.e. the number of items in the set) matches the number of duplicate sets found, then the items of the duplicate sets may safely be removed from all non-duplicate sets in the group. This rule applies even in the degenerative case where the number of duplicate sets is 1 and the size of that set is 1. The degenerative case is what we used above to remove single items from other sets. This rule can be applied to Fig. 5.3 to remove the 2 from all cells in the 7th row of the puzzle except the 7th column where 2 appears by itself.
- **RULE 2**. The second rule looks at each cell within a group and throws away all items that appear in other cells in the group. If we are left with only one value in the chosen cell, then it must appear in this cell and the cell may be updated by throwing

1 2 3 4 5 6 7 8 9	1 2 4 5 7	2 3 4 5 9	2 3 4 6 7 9	2 3 4 5 7 8 9	5 6 7 8 9	1 7 8	1 3 5 6 7	1 3 5 6 8
3 4 5 7	4 5 7	3 4 5	3 4 6 7	1	5 6 7 8	7 8	9	2
1 2 3 5 7 9	8	6	2 3 7 9	2 3 5 7 9	5 7 9	1 7	4	1 3 5
2 3 4 7 8 9	2 4 7	1	5	6	7 8 9	4 8 9	3	4 8 9
4 5 7 8 9	4 5 7	4 5 8 9	1 4 7 9	4 7 8 9	3	6	2	4 8 9
2 3 4 6 8 9	2 4 6	2 3 4 8 9	1 2 4 9	2 4 8 9	1 8 9	5	1	7
1 2 4 5 6	3	2 4 5	1 6 7 9	5 7 9	1 5 6 7 9	2	8	1 4 5 6 9
1 4 5 6	9	4 5	8	3	2	1 4 7	1 5 6 7	1 4 5 6
8	1 2 5 6	7	1 6 9	5 9	4	3	1 5 6	1 5 6 9

Fig. 5.3 Sudoku Puzzle After One Pass

away all other values that appear in the chosen cell. Applying this rule to the fifth row in Fig. 5.3 results in the fourth column being reduced to containing a 1 because 1 does not appear in any other cell in the 5th row. This rule also applies in the last row of the puzzle where 2 is only possible in the second column after removing 1, 5 and 6 from that cell because they appear within other cells in that row.

Note that the puzzle in Fig. 5.3 is not fully reduced. The reduction process can be applied iteratively until no more reductions are possible. The Sudoku solver algorithm keeps applying this reduction process until no more changes are made during a pass of reductions on all the groups within the puzzle. Applying these two rules in this manner will fully reduce many Sudoku puzzles.

5.3 Sets

The reduction algorithm for Sudoku puzzles manipulates sets of numbers and eliminates possible values from those sets as the reduction progresses. A set is a collection that does not allow duplicate values. Sets can be composed of any values. Integers, employee objects, characters, strings, literally any object in Python could be an element of some set. A set has a *cardinality*. The *cardinality* of a set is the number of items in it.

Operation	Complexity	Usage	Description
Set Creation	O(1)	s=set([iterable])	Calls the set constructor to create a set. Iterable is an optional initial contents in which case we have O(n) complexity.
Set Creation	O(1)	s=frozenset([iterable])	Calls the frozenset constructor for immutable set objects to create a frozenset object.
Cardinality	O(1)	len(s)	The number of elements in *s* is returned.
Membership	O(1)	e in s	Returns True if *e* is in *s* and False otherwise.
non-Membership	O(1)	e not in s	Returns True if *e* is not in *s* and False otherwise.
Disjoint	O(n)	s.isdisjoint(t)	Returns True if *s* and *t* share no elements, and False otherwise.
Subset	O(n)	s.issubset(t)	Returns True if *s* is a subset of *t*, and False otherwise.
Superset	O(n)	s.issuperset(t)	Returns True if *s* is a superset of *t* and False otherwise.
Union	O(n)	s.union(t)	Returns a new set which contains all elements in *s* and *t*.
Intersection	O(n)	s.intersection(t)	Returns a new set which contains only the elements in both *s* and *t*.
Set Difference	O(n)	s.difference(t)	Returns a new set which contains the elements of *s* that are not in *t*.
Symmetric Difference	O(n)	s.symmetric_difference(t)	Returns a new set which contains *s.difference(t).union(t.difference(s))*
Set Copy	O(n)	s.copy()	Returns a shallow copy of *s*.

Fig. 5.4 Set and Frozen Set Operations

Sets are objects that have several commonly defined operations on them. These operations are sometimes binary operations involving more than one set, and sometimes retrieve information about just one set. The table in Fig. 5.4 describes commonly defined operations on sets and their associated computational complexities. Python has built-in support for two types of sets, the *set* and *frozenset* classes. The *frozenset* class is immutable. Objects of the *set* class can be mutated. In Fig. 5.4 the variable *s* must be a set and the variable *t* must be an iterable sequence, which would include sets.

Infix operators are also defined as syntactic sugar for some of the operations defined in Fig. 5.4. For subset containment you can write $s <= t$. For proper subset you can write $s < t$. A proper subset is a subset that has at least one less element than its superset. For superset you can write $s >= t$ or $s > t$ for proper superset. For the union operation, writing $s \mid t$ is equivalent to writing *s.union(t)*. And for intersection, *s&t* is equivalent to writing *s.intersection(t)*. Writing $s - t$ is the same as writing *s.difference(t)* and s^t is equivalent to the symmetric difference operator.

The operations in Fig. 5.5 are not defined on the *frozenset* class since they mutate the set *s*. They are only defined on the *set* class.

Again, there are operators for some of the methods presented in Fig. 5.5. The mutator union method can be written $s|=t$. Intersection update can be written as $s\&=t$. Finally, the symmetric difference update operator is written $s\hat{}=t$. While these operators are convenient, they are not well-known and code written by calling the methods in the table above will be more descriptive.

Operation	Complexity	Usage	Description
Union	O(n)	s.update(t)	Adds the contents of *t* to *s*.
Intersection	O(n)	s.intersection_update(t)	Updates s to contain only the intersection of the elements from *s* and *t*.
Set Difference	O(n)	s.difference_update(t)	Subtracts from *s* the elements of *t*.
Symmetric Difference	O(n)	s.symmetric_difference_update(t)	Updates *s* with the symmetric difference of *s* and *t*.
Add	O(1)	s.add(e)	Add the element *e* to the set *s*.
Remove	O(1)	s.remove(e)	Remove the element *e* from the set *s*. This raises *KeyError* if *e* does not exist in *s*.
Discard	O(1)	s.discard(e)	Remove the element *e* if it exists in *s* and ignore it otherwise.
Pop	O(1)	s.pop()	Remove an arbitrary element of *s*.
Clear	O(1)	s.clear()	Remove all the elements of *s* leaving the set empty.

Fig. 5.5 Mutable Set Operations

The computational complexities presented above are surprising! How can set membership be tested in O(1) time? From what has been presented so far, it should take O(n) time to test set membership. After all, we would have to look at all the elements in the set, or at least half on average, to know if an item was in the set. How can the union of two sets be computed in O(n) time if we are to insure there are no duplicates in the set? It would seem that the union of two sets would take $O(n^2)$ time to compute unless the set could be sorted in some way. But sorting elements of a set is not always possible since not all elements of sets have an ordering.

5.4 Hashing

If it is possible to implement a set membership test in O(1) time, then we can implement the other operations above with the complexities we have indicated. Without a O(1) membership test, taking the union of two sets would take a lot longer as indicated above. Testing set membership in O(1) time is accomplished using *hashing*. Hashing is an extremely important concept in Computer Science and is related to random access in a computer. As we saw back in Chap. 2, accessing any location within a list can be accomplished in O(1) time. This is the principle of random access. A *randomly accessible* list means any location within the list can be accessed in O(1) time. To access a location in a list we need the index of the location we wish to access. The index serves as the address of an item in the list. Once we have stored an item in the list, we must remember its index if we wish to retrieve it in O(1) time. Without the index we would have to search for the item in the list which would take O(n) time, not O(1) time.

So, if we wanted to implement a set where we could test membership in O(1) time we might think of storing the items of the set in a list. We would somehow have to remember the index where each item was stored to find it again in O(1) time. This seems improbable at first. However, what if the item could be used to figure out its address? This is the insight that led to hashing. Each object in the computer must

be stored as a string of zeroes and ones since computers speak binary. These zeroes and ones can be interpreted however we like, including as the index into a list. This concept is so important that Python (and many other modern languages) has included a function called *hash* that can be called on any object to return an integer value for an object. We'll call this value the object's *hash code* or *hash value*. Consider these calls to the hash function.

```
Python 3.2 (r32:88452, Feb 20 2011, 10:19:59)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> hash("abc")
-1600925533
>>> hash("123")
1911471187
>>> hash(45)
45
>>> hash(45.0)
45
>>> hash(45.3)
1503225491
>>> hash(True)
1
>>> hash(False)
0
>>> hash([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

While most objects are hashable, not every object is. In particular, mutable objects like lists may not be hashable because when an object is mutated its hash value may also change. This has consequences when using hash values in data structures as we'll see later in this chapter. In addition to built-in types, Python let's the programmer have some control over hash codes by implementing a *__hash__* method on a class. If you write a *__hash__* method for a class you can return whatever hash value integer you like for instances of that class.

Notice that calling hash on the string "abc" returned a negative value while other calls to hash returned extremely large integers. Clearly some work has to be done to convert this hash integer into an acceptable index into a list. Read on to discover how hash values are converted into list indices.

5.5 The HashSet Class

We can use a hash value to compute an index into a list to obtain O(1) item lookup complexity. To hide the details of the list and the calling of the *hash* function to find the index of the item, a set class can be written. We'll call our set class *HashSet*, not to be confused with the built-in *set* class of Python. The built-in *set* class uses hashing, too. The *HashSet* class presented in this section shows you how the *set*

class is implemented. To begin, *HashSet* objects will contain a list and the number of items in the list. Initially the list will contain a bunch of *None* values. The list must be built with some kind of value in it. *None* serves as a null value for places in the list where no value has been stored. The list isn't nearly big enough to have a location for every possible hash value. Yet, the list can't possibly be big enough for all possible hash values anyway. In fact, as we saw in the last section, some hash values are negative and clearly indices into a list are not negative. The conversion of a hash value to a list index is explained in more detail in Sect. 5.5.2. The *HashSet* constructor is given in Sect. 5.5.1.

5.5.1 The HashSet Constructor

```
1  class HashSet:
2      def __init__(self,contents=[]):
3          self.items = [None] * 10
4          self.numItems = 0
5
6          for item in contents:
7              self.add(item)
```

5.5.2 Storing an Item

To store an item in a hash set we first compute its index using the hash function. There are two problems that must be dealt with. First, the list that items are stored in must be finite in length and definitely cannot be as long as the unique hash values we would generate by calling the hash function. Since the list must be shorter than the maximum hash value, we pick a size for our list and then divide hash values by the length of this list. The remainder (i.e. the result of the % operator, called the *mod* operator) is used as the index into the list. The remainder after dividing by the length of the list will always be between 0 and the length of the list minus one even if the hash value is a negative integer. Using the *mod* operator will give us valid indices into a list of whatever size we choose.

There is another problem we must deal with. Hash values are not necessarily unique. Hash values are integers and there are only finitely many integers possible in a computer. In addition, because we divide hash values by the length of the list, the remainders, or list indices, will be even less unique than the original hash values. If the list length is 10, then a hash value of 44 and -6 will both result in trying to store a value at index 4 in the list. This isn't possible of course.

5.5.3 Collision Resolution

Consider trying to store both "Cow" and "Fox" using hashing in a list whose length is 10. The hash value of "Cow" is -1432160826 and the hash value of "Fox" is 1462539404. When we *mod* both values by 10 the remainder is 4 for both hash values indicating they should both be stored at the fifth location in a list.

When two objects need to be stored at the same index within the hash set list, because their computed indices are identical, we call this a *collision*. It is necessary to define a *collision resolution* scheme to deal with this. There are many different schemes that are possible. We'll explore a scheme called *Linear Probing*. When a collision occurs while using linear probing, we advance to the next location in the list to see if that location might be available. We can tell if a location is available if we find a *None* value in that spot in the list. It turns out that there is one other value we might find in the list that means that location is available. A special type of object called a *__Placeholder* object might also be stored in the list. The reason for this class will become evident in the next section. For now, a *None* or a *__Placeholder* object indicates an open location within the hash set list. The code in Sect. 5.5.4 takes care of adding an item into the HashSet list and is a helper function for the actual *add* method.

5.5.4 HashSet Add Helper Function

```
def __add(item,items):
    idx = hash(item) % len(items)
    loc = -1

    while items[idx] != None:
        if items[idx] == item:
            # item already in set
            return False


        if loc < 0 and type(items[idx]) == HashSet.__Placeholder:
            loc = idx

        idx = (idx + 1) % len(items)

    if loc < 0:
        loc = idx

    items[loc] = item

    return True
```

The code in Sect. 5.5.4 does not add an item that is already in the list. The while loop is the *linear probing* part of the code. The index *idx* is incremented, mod the length of the list, until either the item is found or a *None* value is encountered. Finding a *None* value indicates the end of the linear chain and hence the end of any linear searching that must be done to determine if the item is already in the set. If the item is not in the list, then the item is added either at the location of the first *__Placeholder* object found in the search, or at the location of the *None* value at the end of the chain.

There is one more issue that must be dealt with when adding a value. Imagine that only one position was open in the hash set list. What would happen in the code above? The linear search would result in searching the entire list. If the list were full, the result would be an infinite loop. We don't want either to happen. In fact, we want to be able to add an item in amortized O(1) time. To insure that we get an amortized complexity of O(1), the list must never be full or almost full.

5.5.5 The Load Factor

The fullness of the hash set list is called its load factor. We can find the load factor of a hash set by dividing the number of items stored in the list by its length. A really small load factor means the list is much bigger than the number of items stored in it and the chance there is a collision is small. A high load factor means more efficient space utilization, but higher chance of a collision. Experimentation can help to determine optimal load factors, but a reasonable maximum load factor is 75 % full. When adding a value into the list, if the resulting load factor is greater than 75 % then all the values in the list must be transferred to a new list. To transfer the values to a new list the values must be hashed again because the new list is a different length. This process is called *rehashing*. In the hash set implementation we chose to double the size of the list when rehashing was necessary.

The code in Sect. 5.5.6 calls the *__add* function from Sect. 5.5.4. This code and the *__add* method are in the *HashSet* class. The *__add* and *__rehash* functions are hidden helper functions used by the publicly accessible *add* method.

5.5.6 HashSet Add

```
def __rehash(oldList, newList):
    for x in oldList:
        if x != None and type(x) != HashSet.__Placeholder:
            HashSet.__add(x,newList)

    return newList

def add(self, item):
    if HashSet.__add(item,self.items):
        self.numItems += 1
        load = self.numItems / len(self.items)
        if load >= 0.75:
            self.items = HashSet.__rehash(self.items,[None]*2*len(self.items))
```

Since the load factor is managed, the amortized complexity of adding a value to the list is O(1). This means the length of any chain within the list will be a finite length independent of the number of items in the hash set.

5.5.7 Deleting an Item

Deleting a value from a hash set means first finding the item. This may involve doing a linear search in the chain of values that reside at a location in the list. If the value to be deleted is the last in a chain then it can be replaced with a None. If it is in the middle of a chain then we cannot replace it with *None* because this would cut the chain of values. Instead, the item is replaced with a *__Placeholder* object. A place holder object does not break a chain and a linear probe continues to search skipping over placeholder objects when necessary. The remove helper function is given in Sect. 5.5.8.

5.5.8 HashSet Remove Helper Function

```
class __Placeholder:
    def __init__(self):
        pass

    def __eq__(self,other):
        return False

def __remove(item,items):
    idx = hash(item) % len(items)

    while items[idx] != None:
        if items[idx] == item:
            nextIdx = (idx + 1) % len(items)
            if items[nextIdx] == None:
                items[idx] = None
            else:
                items[idx] = HashSet.__Placeholder()
            return True

        idx = (idx + 1) % len(items)

    return False
```

When removing an item, the load factor may get too low to be efficiently using space in memory. When the load factor dips below 25 %, the list is again rehashed to decrease the list size by one half to increase the load factor. The remove method is provided in Sect. 5.5.9.

5.5.9 HashSet Remove

```
def remove(self, item):
    if HashSet.__remove(item,self.items):
        self.numItems -= 1
        load = max(self.numItems, 10) / len(self.items)
        if load <= 0.25:
            self.items = HashSet.__rehash(self.items,[None]*int(len(self.items)/2))
    else:
        raise KeyError("Item not in HashSet")
```

For the same reason that adding a value can be done in O(1) time, deleting a value can also be done with an amortized complexity of O(1). The *discard* method is nearly the same of the remove method presented in Sect. 5.5.9 except that no exception is raised if the item is not in the set when it is discarded.

5.5.10 Finding an Item

To find an item in a hash set involves hashing the item to find its address and then searching the possible chain of values. The chain terminates with a *None*. If the item is in the chain somewhere then the *__contains__* method will return True and False will be returned otherwise. The method in Sect. 5.5.11 is called when *item in set* is written in a program.

5.5.11 HashSet Membership

```
def __contains__(self, item):
    idx = hash(item) % len(self.items)
    while self.items[idx] != None:
        if self.items[idx] == item:
            return True

        idx = (idx + 1) % len(self.items)

    return False
```

Finding an item results in O(1) amortized complexity as well. The chains are kept short as long as most hash values are evenly distributed and the load factor is kept from approaching 1.

5.5.12 Iterating Over a Set

To iterate over the items of a set we need to define the *__iter__* method to yield the elements of the HashSet. The method traverses the list of items skipping over placeholder elements and None references. Here is the code for the iterator.

```
def __iter__(self):
    for i in range(len(self.items)):
        if self.items[i] != None and type(self.items[i]) != HashSet.__Placeholder:
            yield self.items[i]
```

5.5.13 Other Set Operations

Many of the other set operations on the HashSet are left as an exercise for the reader. However, most of them can be implemented in terms of the methods already presented in this chapter. Consider the *difference_update* method. It can be implemented using the iterator, the membership test, and the discard method. The code in Sect. 5.5.14 provides the implementation for the *difference_update* method.

5.5.14 HashSet Difference Update

```
def difference_update(self, other):
    for item in other:
        self.discard(item)
```

The *difference_update* method presented in Sect. 5.5.14 is a mutator method because it alters the sequence referenced by *self*. Compare that with the *difference* method in Sect. 5.5.15 which does not mutate the object referenced by *self*. Instead, the *difference* method returns a new set which consists of the difference of *self* and the *other* set.

5.5.15 HashSet Difference

```
def difference(self, other):
    result = HashSet(self)
    result.difference_update(other)
    return result
```

The *difference* method is implemented using the *difference_update* method on the *result* HashSet. Notice that a new set is returned. The hash set referenced by self is not updated. The code is simple and it has the added benefit that if *difference_update* is correctly written, so will this method. Programmers should always avoid writing duplicate code when possible and *difference* and *difference_udpate* are nearly identical except that the *difference* method performs the difference on a newly constructed set instead of the set that *self* references.

5.6 Solving Sudoku

Using a *set* or a *HashSet* datatype, we now have the tools to solve most Sudoku puzzles. A puzzle can be read from a file where known values are represented by their digit and unknown values are X's as in the puzzle below.

```
x x x x x x x x x
x x x x 1 x x 9 2
x 8 6 x x x x 4 x
x x 1 5 6 x x x x
x x x x x 3 6 2 x
x x x x x x 5 x 7
x 3 x x x x x 8 x
x 9 x 8 x 2 x x x
x x 7 x x 4 3 x x
```

Reading the dateable can be done a line at a time. Splitting the line will provide a string with each known and unknown value as a separate item in the list. When an X is encountered a *set* containing all values 1–9 can be constructed, just as you would if solving Sudoku by hand. When a known value is found a *set* with the known number in it can be constructed.

The sets are added to a two-dimensional matrix. The matrix is a list of lists. So reading a line corresponds to reading a row of the matrix. Each line becomes a list of sets. Each list of sets is added to a list we'll call the *matrix*. So, *matrix[row][col]* is one set within the Sudoku puzzle.

There are 81 sets within a Sudoku puzzle. However, each set is a member of three groups: the row, the column, and the square in which it resides. The rules presented in the Sudoku puzzle description above each deal with reducing a set within one of these groups. After reading the input from the file and forming the matrix with the 81 sets, 27 groups are formed by creating a list of groups.

A shallow copy of each row is first appended to the list of groups. A shallow copy means that each of the sets is the same set that was created when the puzzle was read.

A deep copy would create a copy of each of the 81 sets. A shallow copy of a list does not copy the sets within the list. Calling *list* on a list will make a shallow copy.

Another group is formed for each column and those groups are appended to the list of groups. Finally, a group is formed for each square and those groups are appended to the list of groups. When all done, there is one *groups* list with 27 groups in it. When forming these groups it is critical that the same set appears in each of three groups. This is because when a row is reduced, we want the changes in that row to be reflected in the columns and squares where the elements of the row also appear.

Solving a Sudoku puzzle means reducing the number of items in each set of a group according to the two rules presented in Sect. 5.2. Writing a function called *reduceGroup* that is given a list of 9 sets to reduce can help. The function *reduceGroup* should return **True** if it was able to reduce the group and **False** if it was not. Given this *reduceGroup* function, the *reduce* function is as defined in Sect. 5.6.1.

5.6.1 The Sudoku Reduce Function

```
1     def reduce(matrix):
2       changed = True
3       groups = getGroups(matrix)
4
5       while changed:
6         changed = reduceGroups(groups)
```

This algorithm is quite simple, and yet very powerful. It is different than any other algorithms that have been presented so far in this text. The concept is quite simple: keep reducing until no more reductions are possible. We are guaranteed that it will terminate since each iteration of the algorithm reduces the number of items in some of the sets of the puzzle. We never increase the size of any of these sets. Once we return from this function we may or may not have a solution. There are some puzzles that will not be solved by this Sudoku solver because the two rules that are presented above are not powerful enough for all puzzles. There are some situations where the number of items in a set cannot be reduced by looking at just one group. You would have to look at more than one group at the same time to figure out how to reduce the number of sets. *Hold on, you don't need to figure out any more rules. The next chapter will present an algorithm for solving all Sudoku puzzles, even the very hardest of them.*

The rules presented in this chapter will solve the Sudoku puzzle given in this section and many others. Sudoku puzzles one through six on the text's website can be solved by this Sudoku solver. The *reduceGroups* function above would presumably call *reduceGroup* on each of the groups in its list and it would return *True* if any of the groups are reduced and *False* otherwise.

5.7 Maps

A *map* in computer science is not like the map you used to read when going someplace in your car. The term *map* is a more mathematical term referring to a function that maps a *domain* to a *range*. You may have already used a map in Python. Maps are called by many names including dictionaries, hash tables, and hash maps. They are all the same data structure.

A map or dictionary maps a set of unique keys to their associated values much the way a function maps a value in the domain to the range. A key is what we provide to a map when we want to look for the key/value pair. The keys of a map are unique. There can only be one copy of a specific key value in the dictionary at a time. As we saw in chapter one, Python has built-in support for dictionaries or maps. Here is some sample interaction with a dictionary in the Python shell.

```
Python 3.2 (r32:88452, Feb 20 2011, 10:19:59)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> d = {}
>>> d["dog"] = "cat"
>>> d["batman"] = "joker"
>>> d["superman"] = "lex luther"
>>> for key in d:
...     print(key)
...
batman
dog
superman
>>> for key in d:
...     print(key,d[key])
...
batman joker
dog cat
superman lex luther
>>> len(d)
3
>>> d["dog"] = "skunk"
>>> d["dog"]
'skunk'
>>>
```

A map, or dictionary, is a lot like a set. A set and a dictionary both contain unique values. The set datatype contains a group of unique values. A map contains a set of unique keys that map to associated values. Like sets, we can look up a key in the map, and its associated value, in O(1) time. As you might expect, maps, like sets, are implemented using hashing. While the underlying implementation is the same, maps and sets are used differently. The table below provides the methods and operators of maps or dictionaries and their associated complexities.

The operations in the table above have the expected complexities given a hashing implementation as was presented in Sect. 5.5. The interesting difference is that key/value pairs are stored in the dictionary as opposed to just the items of a set. The key part of the key/value pair is used to determine if a key is in the dictionary as you might expect. The value is returned when appropriate.

5.7.1 The HashMap Class

A *HashMap* class, like the *dict* class in Python, uses hashing to achieve the complexities outlined in the table in Fig. 5.6. A private *__KVPair* class is defined. Instances of *__KVPair* hold the key/value pairs as they are added to the *HashMap* object. With the addition of a *__getitem__* method on the *HashSet* class, the *HashSet* class could be used for the *HashMap* class implementation. The additional *__getitem__* method for the *HashSet* is given in Sect. 5.7.3.

5.7.2 HashSet Get Item

```
# One extra HashSet method for use with the HashMap class.
def __getitem__(self, item):
    idx = hash(item) % len(self.items)
    while self.items[idx] != None:
        if self.items[idx] == item:
            return self.items[idx]

        idx = (idx + 1) % len(self.items)

    return None
```

Operation	Complexity	Usage	Description
Dictionary Creation	O(1)	d = {[iterable]}	Calls the constructor to create a dictionary. Iterable is an optional initial contents in which case it is O(n) complexity.
Size	O(1)	len(d)	The number of key/value pairs in the dictionary.
Membership	O(1)	k in d	Returns True if *k* is a key in *d* and False otherwise.
non-Membership	O(1)	k not in d	Returns True if *k* is not a key in *d* and False otherwise.
Add	O(1)	d[k] = v	Adds *(k,v)* as a key/value pair in *d*.
Lookup	O(1)	d[k]	Returns the value associated with the key, *k*. A *KeyError* exception is raised if *k* is not in *d*.
Lookup	O(1)	d.get(k[,default])	Returns *v* for the key/value pair *(k,v)*. If *k* is not in *d* returns *default* or *None* if not specified.
Remove Key/-Value Pair	O(1)	del d[k]	Removes the *(k,v)* key value pair from *d*. Raises *KeyError* if *k* is not in *d*.
Items	O(1)	d.items()	Returns a view of the key/value pairs in *d*. The view updates as *d* changes.
Keys	O(1)	d.keys()	Returns a view of the keys in *d*. The view updates as *d* changes.
Values	O(1)	d.values()	Returns a view of the values in *d*. The view updates as *d* changes.
Pop	O(1)	d.pop(k)	Returns the value associated with key *k* and deletes the item. Raises *KeyError* if *k* is not in *d*.
Pop Item	O(1)	d.popitem()	Return an abritrary key/value pair, *(k,v)*, from *d*.
Set Default	O(1)	d.setdefault(k[, default])	Sets *k* as a key in *d* and maps *k* to *default* or *None* if not specified.
Update	O(n)	d.update(e)	Updates the dictionary, *d*, with the contents of dictionary *e*.
Clear	O(1)	d.clear()	Removes all key/value pairs from *d*.
Dictionary Copy	O(n)	d.copy()	Returns a shallow copy of *d*.

Fig. 5.6 Dictionary Operations

Then, to implement the *HashMap* we can use a *HashSet* as shown in Sect. 5.7.3. In the *__KVPair* class definition it is necessary to define the *__eq__* method so that keys are compared when comparing two items in the hash map. The *__hash__* method of *__KVPair* hashes only the key value since keys are used to look up key/value pairs in the hash map. The implementation provided in Sect. 5.7.3 is partial. The other methods are left as an exercise for the reader.

5.7.3 The HashMap Class

```
class HashMap:
    class __KVPair:
        def __init__(self,key,value):
            self.key = key
            self.value = value

        def __eq__(self,other):
            if type(self) != type(other):
                return False

            return self.key == other.key

        def getKey(self):
            return self.key

        def getValue(self):
            return self.value

        def __hash__(self):
            return hash(self.key)

    def __init__(self):
        self.hSet = hashset.HashSet()

    def __len__(self):
        return len(self.hSet)

    def __contains__(self,item):
        return HashSet.__KVPair(item,None) in self.hSet

    def not__contains__(self,item):
        return item not in self.hSet

    def __setitem__(self,key,value):
        self.hSet.add(HashMap.__KVPair(key,value))

    def __getitem__(self,key):
        if HashMap.__KVPair(key,None) in self.hSet:
            val = self.hSet[HashMap.__KVPair(key,None)].getValue()
            return val

        raise KeyError("Key " + str(key) + " not in HashMap")

    def __iter__(self):
        for x in self.hSet:
            yield x.getKey()
```

The provided implementation in Sect. 5.7.3 helps to demonstrate the similarities between the implementation of the *HashSet* class and the *HashMap* class, or between the *set* and *dict* classes in Python. The two types of data structures are both implemented using hashing. Both rely heavily on a O(1) membership test. While understanding how the *HashMap* class is implemented is important, most programming languages include some sort of hash map in their library of built-in types, as does Python. It is important to understand the complexity of the methods on a hash map, but just as important is understanding when to use a hash map and how it can be used. Read on to see how you can use a hash map in code you write to make it more efficient.

5.8 Memoization

Memoization is an interesting programming technique that can be employed when you write functions that may get called more than once with the same arguments. The idea behind memoization is to do the work of computing a value in a function once. Then, we make a note to ourselves so when the function is called with the same arguments again, we return the value we just computed again. This avoids going to the work of computing the value all over again.

A powerful example of this is the recursive Fibonacci function. The Fibonacci sequence is defined as follows.

- Fib(0) = 0
- Fib(1) = 1
- Fib(n) = Fib(n−1) + Fib(n−2)

This sequence can be computed recursively by writing a Python function as follows.

```
def fib(n):
    if n == 0:
        return 0

    if n == 1:
        return 1

    return fib(n-1) + fib(n-2)
```

However, we would never want to use this function for anything but a simple demonstration of a small Fibonacci number. The function cannot be used to computing something as big as fib(100) even. Running the function with an argument of 100 will take a very long time on even the fastest computers. Consider what happens to compute fib(5). To do that fib(4) and fib(3) must first be computed. Then the two results can be added together to find fib(5). But, to compute fib(4) the values fib(3) and fib(2) must be computed. Now we are computing fib(3) twice to compute fib(5), But to compute fib(3) we must compute fib(2) and fib(1). But, fib(2) must be computed to find fib(4) as well. Figure 5.7 shows all the calls to fib to compute fib(5).

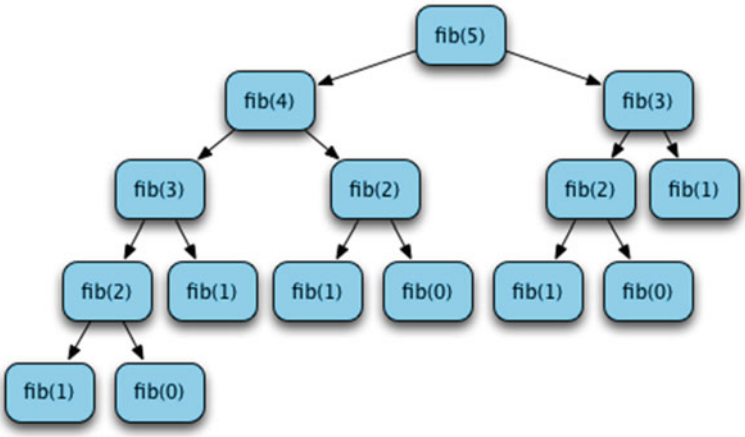

Fig. 5.7 Computing fib(5)

As you can see from Fig. 5.7, it takes a lot of calls to the *fib* function to compute fib(5). Now imagine how many calls it would take to compute fib(6). To compute fib(6) we first have to compute fib(5) and then compute fib(4). It took 15 calls to *fib* to compute fib(5) and from the figure we can see that it takes 9 calls to compute fib(4). Including the call to fib(6) it will take 25 calls to *fib* to compute fib(6). Computing fib(7) will take 15 + 25 + 1 calls or 41 calls. Computing fib(n) this way more than doubles the number of calls to compute fib(n−2). This is called exponential growth. The complexity of the *fib* function is $O(2^n)$. A function with exponential complexity is worthless except for very small values of *n*.

All is not lost. There are better ways of computing the Fibonacci sequence. The way to improve the efficiency is to avoid all that unnecessary work. Once fib(2) has been computed, we shouldn't compute it again. We already did that work. There are at least a couple of ways of improving the efficiency. One method involves removing the recursion and computing fib(n) with a loop, which is probably the best option. However, the recursive function is closer to the original definition. We can improve the recursive version of the function with memoization. In Sect. 5.8.1, the *memo* dictionary serves as our mapping from values of *n* to their *fib(n)* result.

5.8.1 A Memoized Fibonacci Function

```
memo = {}

def fib(n):
    if n in memo:
        return memo[n]

    if n == 0:
        memo[0] = 0
```

```
        return 0

    if n == 1:
        memo[1] = 1
        return 1

    val = fib(n-1) + fib(n-2)

    memo[n] = val

    return val

def main():
    print(fib(100))

if __name__ == "__main__":
    main()
```

The memoized *fib* function in Sect. 5.8.1 records any value returned by the function in its memo. The *memo* variable is accessed from the enclosing scope. The *memo* is not created locally because we want it to persist from one call of *fib* to the next. Each time *fib* is called with a new value of *n* the answer is recorded in the memo. When *fib(n)* is called a subsequent time for some *n*, the memoized result is looked up and returned. The result: the memoized *fib* function now has O(n) complexity and it can compute fib(100) almost instantly. Without memoization, it would take 1,146,295,688,027,634,168,201 calls to the *fib* function to compute fib(100). Assuming each function call completed in 10 microseconds, it would take roughly 363 million years to compute fib(100). With memoization it takes 100 calls to *fib* and assuming 10 microseconds per call, that's 1000 microseconds or 1/1000 of a second.

This is an extreme example of the benefit of memoization, but it can come in handy in many situations. For instance, in the tic tac toe problem of Chap. 4 the minimax function is called on many boards that are identical. The minimax function does not care if an X is placed in the upper-right corner first followed by the lower-left corner or vice-versa. Yet, the way minimax is written it will be called to compute the value of the same board multiple times. Memoizing minimax speeds up the playing of tic tac toe.

5.9 Correlating Two Sources of Information

Another use of a map or dictionary is in correlating data from different sources. Assume you are given a list of cities and the zip code or codes within those cities. You want to provide a service where people can look up the zip code for a city in the USA. So, you'll be given a city by the web page that provides you the information. You have to use that city to find a list of possible zip codes. You could search the list of cities to find the corresponding list of zip codes. Or, you could create a dictionary from city name to zip code list. Then when given a city name you check to see if it is in the dictionary and if so, you can look up the corresponding list of zip codes in O(1) time.

5.10 Chapter Summary

In this chapter we explored the implementation and some uses of sets and maps in Python. Hashing is an important concept. Hashing data structures must be able to handle collisions within the hash table by a collision resolution strategy. The resolution strategy explored in this chapter was linear probing. There are other collision resolution strategies possible. Any collision resolution strategy must have a way of handling new values being added to a chain and existing values being deleted from a chain.

The key feature of hashing is the amortized O(1) complexity for membership testing and lookup within the table. The ability to test membership or lookup a value in O(1) time makes many algorithms efficient that otherwise might not run efficiently on large data sets.

Memoization is one important use of a dictionary or map. By memoizing a function we avoid doing any redundant work. Another important use of maps or dictionaries is in correlating sources of information. When we are given information from two different sources and must match those two sources, a map or dictionary will make that correlation efficient.

5.11 Review Questions

Answer these short answer, multiple choice, and true/false questions to test your mastery of the chapter.

1. What type of value is a hash code?
2. Hash codes can be both positive and negative. How does a hash code get converted into a value that can be used in a hash table?
3. Once you find the proper location with a hash table, how do you know if the item you are looking for is in the table or not? Be careful to answer this completely.
4. Why is a collision resolution strategy needed when working with a hash table?
5. What is the difference between a map and a set?
6. In this chapter the HashSet was used to implement the HashMap class. What if we turned things around? How could a dictionary in Python be used to implement a set? Describe how this might be done by describing the add and membership methods of a set and how they would be implemented if internally the set used a dictionary.
7. How does the load factor affect the complexity of the membership test on the set datatype?
8. What is rehashing?
9. When is memoization an effective programming technique?
10. True or False: Memoization would help make the factorial function run faster? Justify your answer.

```
def fact(n):
    if n == 0:
        return 1

    return n * fact(n-1)

def main():
    x = fact(10)
    print("10! is",x)

if __name__ == "__main__":
    main()
```

5.12 Programming Problems

1. Complete the Sudoku puzzle as described in the chapter. The program should read a text file. Prompt the user for the name of the text file. The text file should be placed in the same directory or folder as the program so it can easily be found by your program. There are six sample Sudoku puzzles that you can solve available on the text's website. Write the program to read a text file like those you find on the text's website. Print both the unsolved and solved problem to the screen as shown below.

```
Please enter a Sudoku puzzle file name: sudoku2.txt
Solving this puzzle
-----------------
x x x x x x x x x
x x x x 1 x x 9 2
x 8 6 x x x x 4 x
x x 1 5 6 x x x x
x x x x x 3 6 2 x
x x x x x x 5 x 7
x 3 x x x x x 8 x
x 9 x 8 x 2 x x x
x x 7 x x 4 3 x x

Solution
-----------------
4 1 2 9 8 5 7 6 3
7 5 3 4 1 6 8 9 2
9 8 6 3 2 7 1 4 5
2 7 1 5 6 8 9 3 4
5 4 9 1 7 3 6 2 8
3 6 8 2 4 9 5 1 7
6 3 4 7 5 1 2 8 9
1 9 5 8 3 2 4 7 6
8 2 7 6 9 4 3 5 1

Valid Solution!
```

2. Complete the *HashSet* class found in the chapter by implementing the methods described in the two tables of set operations. Then, write a *main* function to test these operations. Save the class in a file called hashset.py so it can be imported into other programs. If you call your *main* function in hashset.py with the *if __name__ == "__main__"* statement, then when you import it into another program your hashset.py *main* function will not be executed, but when you run hashset.py on its own, its *main* function will run to test your *HashSet* class.
3. Memoize the tic tac toe program from Chap. 3 to improve its performance. To do this each board must have a hash value. You should implement a *__hash__* method for the *Board* class. The hash value should be unique to a board's configuration. In other words, the X's, O's, and Dummy objects should factor into the hash value for the board so that each board has its own unique hash value. Then memoize the minimax function to remember the value found for a particular board's configuration. The minimax function should start by checking whether or not the value for this board has already been computed and the function should return it if it has.
4. Write a version of the *HashSet* class that allows you to specify the maximum and minimum allowable load factor. Then run a number of tests where you plot the average time taken to add an item to a set given different maximum load factors. Also gather information about the average time it takes to test the membership of an item in a set for different maximum load factors. From this information you should be able to see some of the space/time trade-off in hash tables. Generate XML data in the plot format from these experimental results and plot the data to see what it tells you. From the gathered information, express your opinion about the optimal load factor for the *HashSet* class. Comment on the optimal maximum load factor at the top of the program that performs your tests.