6.5 Sets

A *set* is a data structure that holds single copies of a number of elements. The most important property of a set is that it has no repeats: if an item is in the set and we try to add it again, the set is unchanged. Note that adding an element to the set that is already present in the set is not an error, it just doesn't change the change the set. You have encountered sets in mathematics classes where they seem fairly obvious and not very interesting, just an excuse for more terminology. They play a more interesting role in programming. Sets have some useful operations. For example, if word1 and word2 are two words, then **set**(word1)−**set**(word2) is the set of letters that are in word1 but not in word2, while **set**(word1)&**set**(word2) is the set of letters in both. Sets are also efficient ways to store data for lookups. If a set s has n elements, you can determine if an item is in s or not in time proportional to log(n). For a list this operation takes time proportional to n. The difference doesn't matter when n is small, but if n is large it might take 20 steps to find if something is in a set and one million steps to find if the item is in a list with the same elements as the set. That is enough of a difference to show up in the user experience of running your program.

We create an empty string with quotes, an empty list with square brackets, an empty tuple with parentheses, and an empty dictionary with curly brackets. We are out of delimiters. To make an empty set, we use the function **set**(). There are actually similar constructors for making the other structures: **list** (), **tuple**() and **dict**() make empty lists, tuples and dictionaries, respectively. If we have elements to put in the set at the time it is created we can write it with curly brackets, as in {1, 2, 3}. Note that the elements of a set must be *hashable*. It is hard to describe what constitutes being hashable without going into the details of how sets are implemented, but numbers and the immutable structures (strings and tuples) are all hashable. Sets themselves are mutable.

Here are some of the common operations with sets:

A. Making sets:
  S = set() (the empty set, which is the set with no elements)
  S = {3, 6, 9} makes S be a set consisting of the numbers 3, 6, and 9.
  S = set(L), where L is a list, string, or tuple: makes S be a set whose
    elements are the elements of L
    S = set("abc") is the same as S = {"a", "b", "c"}
  S.copy() returns a new set that is a copy of set S

B. Operations:
  A+B is the union of sets A and B. This can also be written A.union(B)
  A&B is the intersection of sets A and B.
    This can also be written A.intersect(B)
  A-B is the difference of sets A and B: the elements of A that are not in B.
    This can also be written A.difference(B)
  A<=B is True if A is a subset of B: all of the elements of A are also in B
  A<B is True if A is a *proper* subset of B:
    A is a subset of B and not equal to B

C. Changing the contents of a set:
  S.add(x) adds element x to set S if x is not already an element of S
  S.remove(x) deletes element x from set S;
    this causes an error if x is not an element of S
  S.discard(x) deletes element x from set S
    but does not cause an error if x is not an element of S
  S.clear() removes all elements of set S

D. Other methods
  len(S): the length, or number of entries, of set S
  for x in S: iterates a loop over all entries of set S
  x in S: returnsTrue ifx is an element of set S
  x not in S: returnsFalse ifx is an element of set S

We will now look at two examples of sets. The first example is a short program that builds up a set of the user's favorite colors. We make an empty set called favorites with the line

```
favorites=set()
```

and then go into our usual input loop that reads strings as variable color until we get to an empty string. We exit the loop on an empty string; when variable color is not empty we add it to the set with

```
favorites.add(color)
```

At the end we print the contents of the set with

```
for x in favorites:
    print(x)
```

Here is the complete program:

```
def main():
    # This builds up and then prints a set of
    # favorite colors
    done = False
    favorites = set()
    while not done:
        color = input( "Favorite colors? " )
        if color == "":
            done = True
        else:
            favorites.add(color)

    for x in favorites:
        print(x)

main()
```

Program 6.5.1: Creating and printing a set of favorite colors

Our second example of sets is more elaborate. Here is a statement of the problem:

> Write a program that inputs a single string from the user. The program should print every possible rearrangement of the letters of that string, with each rearrangement printed only once.

One way we will use sets in this program is for finding unique rearrangements. Each time we generate a rearrangement we'll add it to a set; this way duplicates will be ignored. At the end we'll just print the elements of this set. A more interesting use of sets occurs in creating the rearrangements. We will find all of the rearrangements by first finding all of the possible permutations of the numbers (0, 1, 2, ... n-1), where n is the length of our string, and using these permuations to make the rearrangements.

An example will help clarify this algorithm. Suppose the string we start with is "abc"; we'll call this string word. We start by finding the set of all permutations of the numbers (0, 1, 2): {(0,1,2),(0,2,1),(1,2,0),(1,0,2),(2,0,1),(2,1,0)} Each of these makes one rearrangement of word. For example, with permutation (1,2,0) our rearrangement is word[1]+word[2]+word[0], which is "bca". Here is code for a function that takes variable word and rearranges it according to permutation t:

```
def reArrange(word, t):
    newWord = ""
```

```
        for i in t:
            newWord = newWord + word[i]
        return newWord
```

So far we have reduced our problem to the problem of finding all permutations of the numbers (0,1,2,..., n−1). There are a number of ways to do this. An easy, though not particularly efficient way makes another use of sets. We will maintain a set of all of the permutations we have found so far; initially this contains only the single tuple (0,1,2,..., n−1). At each step we take one of the permutations from this set and run through all of the possible indexes i and j of this permutation. We interchange the elements at index i and index j, forming a new permutation, which we add to our set. This keeps going until we have found all of the permutations. How many are there? There are n numbers in (0,1,2,..., n−1), so there are n choices for the first entry. The one chosen can't be reused, so there are n−1 choices for the second entry, n−2 choices for the third entry, and so forth. Altogether there are n∗(n−1)∗(n−2)∗...∗1, or factorial (n) possible permutations.

This algorithm makes the following code:

```
    permutations = set()  #makes an empty set
    permutations.add(start) #start is the initial tuple (0,1,2,...N-1)
    while len(permutations) <  maxSize:  #maxSize is factorial(N)
        for t in permutations:
            for i in range(0, N-1):
                for j in range(i+1, N):
                    newT = switch(t, i, j) #switches i and j elements
                    permutations.add(newT)
```

There is one problem with this code. We have a **for**-loop: **for** t **in** permutations: and in the body of this loop we modify the permutations set. Python doesn't allow this. To fix this we change the loop to use a *copy* of the current permutations set, and we add newT to the *actual* permutations set:

```
    permutations = set()  #makes an empty set
    permutations.add(start) #start is the initial tuple (0,1,2,...N-1)
    while len(permutations) <  maxSize:  #maxSize is factorial(N)
        for t in permutations.copy():
            for i in range(0, N-1):
                for j in range(i+1, N):
                    newT = switch(t, i, j) #switches i and j elements
                    permutations.add(newT)
```

Now, suppose the size of our string is N=4. Factorial(4) is 24; suppose we have found 23 of the 24 permutations. If we loop through every one of these permutations and every possible pair of indices i and j, we might find the only remaining permutation quickly and then be stuck doing a lot of extra, unnecessary work. We can avoid this by placing a break statement inside the **for**-loop on t:

```
    if len(permutations) == maxSize:
        break
```

Our code so far calls a function switch(t, i, j) to interchange two elements of tuple t. Here is the algorithm for this. We have chosen indices i and j in such a way that we know i<j. So we first copy the elements of t with indices less than i, followed by the entry at index j, followed by the elements of t with indices from i+1 to j−1, followed by the entry at index i, and that followed by the portion of t with indices greater than j:

```
def switch(t, i, j):
    newT = t[0:i]+(t[j],)+t[i+1:j]+(t[i],)+t[j+1:]
    return newT
```

The following program puts all of this together:

```
# This program inputs a word and prints all of the possible
# anagrams of that word.

def factorial(n):
    # This returns the product of the numbers from 1 to n
    prod = 1
    for x in range(2, n+1):
        prod = prod*x
    return prod

def switch(t, i, j):
    # This assumes t is a tuple and
    # i and j are indexes into t, with i<j
    # This returns the result of interchanging
    # t[i] and t[j]
    newT = t[0:i]+(t[j],)+t[i+1:j]+(t[i],)+t[j+1:]
    return newT

def reArrange(word, t):
    # This arranges the letters of word according to the
    # order specified by permutation t.
    newWord = ""
    for i in t:
        newWord = newWord + word[i]
    return newWord

def main():
    word = input( "word? " )
    N = len(word)
    maxSize = factorial(N)
    start = tuple(range(N)) #(0,1,,2,...,N-1)
    # First we find all of the permutations of start
    permutations = set()
    permutations.add(start)
    while len(permutations) <  maxSize:
        for t in permutations.copy():
            if len(permutations) == maxSize:
                break
            for i in range(0, N-1):
                for j in range(i+1, N):
                    newT = switch(t, i, j)
                    permutations.add(newT)
    # Next we make a set of the anagrams of word
    anagrams = set()
    for t in permutations:
        anagrams.add( reArrange(word, t))
    # Finally we print that set.
    for w in anagrams:
        print(w)

main()
```

Program 6.5.2: Anagrams of a word

If we run this with word="read" we get the 24 permutations of those 4 letters:

```
dear , dare , aedr , ared , erad , eard , aerd , reda ,
eadr , ader , daer , drae , adre , raed , drea , erda ,
dera , arde , edra , rdea , read , rdae , edar , rade
```

However, if we run it with word="bob" we only get 3 since other permutations would result in duplicates:

```
obb , bob , bbo
```