Part II

Problem Solving

Unit 5

Data Structures

Learning Outcomes

- Manipulate string data.
- Use lists for storing and manipulating data.
- Describe the difference between mutable and immutable data structures.
- Identify mutable and immutable data structures, and describe the difference between them.
- Describe the use of references in assignment and argument passing.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Chapters 7 and 8 in *How to Think Like a Computer Scientist*.

Topic 5.1 Lists

So far, all of the variables that we have used have held a single item: one integer, floating point value, or string. These types are good at storing the information they are designed for, but you will often find that you want to store a collection of values in your programs.

For example, you may want to store a list of values that have been entered by the user or a collection of values that are needed to draw a graph.

In Python, *lists* can be used to store a collection of other values. Lists in Python can hold values of any type; they are written as a comma-separated list enclosed in square brackets:

```
numlist = [23, 10, -100, 2]
words = ['zero', 'one', 'two']
junk = [0, 1, 'two', [1,1,1], 4.0]
```

Here, `numlist` is a list holding four integers; `words` holds three strings; and `junk` has five values of different types (one of them is itself a list).

Lists are like strings

To get a particular value out of a list, it can be *subscripted*. This is done just like subscripting a string to extract a single character:

```
>>> testlist = [0, 10, 20, 30, 40, 50]
>>> print testlist[2]
20
>>> print testlist[0]
0
>>> print testlist[10]
IndexError: list index out of range
```

Like strings, the first element in a list is element 0.

You can determine the length of a list with the `len` function:

```
>>> print len(testlist)
6
```

So, we can walk through each element of a list the same way we iterated over the characters in a string:

```
>>> for i in range(len(testlist)):
...     print testlist[i],
...
0 10 20 30 40 50
```

Lists can be joined (*concatenated*) with the `+` operator:

```
>>> testlist + [60, 70, 80]
[0, 10, 20, 30, 40, 50, 60, 70, 80]
>>> ['one', 'two', 'three'] + [1, 2, 3]
['one', 'two', 'three', 1, 2, 3]
```

Lists can also be returned by functions:

```
>>> s = 'abc-def-ghi'
>>> s.split('-')
['abc', 'def', 'ghi']
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In all of the above examples, lists are similar to strings. As far as we've seen so far, you could just think of strings as just lists of characters. Lists (so far) work just like strings, except you can put anything in each element, not just a character.

If you ever program in C, you'll find that there really isn't any "string" type in C. Strings are just implemented as lists (called arrays in C) of characters. In Python, strings get a separate data type.

Lists are different from strings

Of course, the biggest difference between lists and strings is what they can hold. A string holds only characters, but a list can hold any type of Python data.

More than that, there are many operations that you can do on lists that aren't possible on string. It's not possible to change individual parts of a string without totally rebuilding it. When you want to change a string, you need to write an expression that created a new string, and over-wrote the old variable.

With a list, you can assign a new value to a *part* of the list, without having to rebuild the entire list. This is called *element assignment*.

```
>>> colours = ['red', 'yellow', 'blue']
>>> print colours
['red', 'yellow', 'blue']
>>> colours[1] = 'green'
>>> print colours
['red', 'green', 'blue']
```

The third statement here changes a single element of the lists, without having the change the entire list (by doing a `colours=...` assignment).

It is also possible to *delete* an element from a list, using the `del` statement.

```
>>> colours = ['red', 'yellow', 'blue']
>>> del colours[1]
>>> print colours
['red', 'blue']
>>> del colours[1]
>>> print colours
['red']
```

You can also add a new element to the end of a list with the `append` method.

```
>>> colours = ['red', 'yellow', 'blue']
>>> colours.append('orange')
>>> colours.append('green')
>>> print colours
['red', 'yellow', 'blue', 'orange', 'green']
```

In order to do something similar with a string, a new string must be built with the `+` operator:

```
>>> letters = 'abc'
>>> letters = letters + 'd'
>>> letters = letters + 'e'
>>> print letters
abcde
```

You can do the same thing with lists (rebuild them with `+` or other operators), but it's slower. As another example, see Figure 5.1

All of these operations can change *part* of a list. Changing one element (or a few elements) is more efficient than creating an entirely new list. These operations can make working with lists quite efficient.

If you try change part of a string, you will get an error. We will discuss this difference further in Topic 5.5.

There are many other list operations as well: lists are a very flexible data structure. See the online Python reference for more details.

```
print "Enter some numbers, 0 to stop:"

numbers = []
x=1
while x!=0:
    x = int(raw_input())
    if x!=0:
        numbers.append(x)

print "The numbers you entered are:"
print numbers
```

Figure 5.1: Building a list with the `append` method

Topic 5.2 Lists and `for` loops

In an earlier example, you might have noticed how the `range` function was used to create a list:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

But all along, we have been using the `range` function with the `for` loop:

```
for i in range(10):
    # do something with i
    ...
```

So, what's the relationship?

It turns out that the `for` loop in Python can iterate over *any* list, not just those produced by the `range` function. The `range` function is a convenient way to produce a list of integers, but not the only way. We have seen other ways to build a list, and those can be used with the `for` loop as well.

Have a look at the code in Figure 5.2. There, the `for` loop iterates over each element in the list `words`. The `for` loop does the same thing it does with a `range`: it runs the loop body once for each element. The output of Figure 5.2 is:

```
words = ["up", "down", "green", "cabbage"]
for word in words:
    print "Here's a word: " + word
```

Figure 5.2: Iterating over a list

```
Here's a word: up
Here's a word: down
Here's a word: green
Here's a word: cabbage
```

Iterating through the elements of a list can be quite convenient. It's common to have to do the same operation on each element of a list, and this gives an easy way to do it.

It also makes code very readable. You can interpret the loop in Figure 5.2 as "for every word in the list `words`, do this. . . ." So, the meaning of the code is very close to the way you'd read it. This is always a benefit when trying to read and maintain code.

Topic 5.3 Slicing and Dicing

Hopefully you are comfortable with indexing by now. You can access a single element from a string or list with indexing:

```
>>> colours = ['red', 'yellow', 'blue']
>>> colours[1] = 'green' # set an element with indexing
>>> print colours[2]     # index to retrieve an element
'blue'
```

It's also possible to access several elements of a list by *slicing*. Slicing looks like indexing, but you can specify an entire range of elements:

```
>>> colours = ['red', 'yellow', 'green', 'blue']
>>> print colours[1:3]
['yellow', 'green']
```

As you can see, the slice `[1:3]` refers to elements 1–2 from the list. In general, the slice `[a:b]` extracts elements `a` to `b-1`.

You might be thinking that the slice operator should extract all of the elements from `a` to `b` (including `b`), but it stops one before that. Maybe it's not intuitive, but it does match the behaviour of the `range` function: `range(a,b)` gives you all of the integers from `a` to `b-1`, and the slice `[a:b]` gives you the elements from `a` to `b-1`.

Special Slice Positions

In addition to selecting "elements `a` to `b-1`," there are special values that can be used in a slice.

Negative values count from the *end* of a list. So, `-1` refers to the *last* item in the list, `-2` to the second-last, and so on. You can use this to (for example) extract everything *except the last element*:

```
>>> colours = ['red', 'yellow', 'green', 'blue']
>>> print colours[0:-1]
['red', 'yellow', 'green']
```

If you leave out one of the values in the slice, it will default to the start or end of the list. For example, the slice `[:num]` refers to elements `0` to `num-1`. The slice `[2:]` gives elements from `2` to the end of the list. Here are some more examples:

```
>>> colours = ['red', 'yellow', 'green', 'blue', 'orange']
>>> print colours[2:]
['green', 'blue', 'orange']
>>> print colours[:3]
['red', 'yellow', 'green']
>>> print colours[:-1]
['red', 'yellow', 'green', 'blue']
```

The slice `[:-1]` will always give you everything except the last element; `[1:]` will give everything but the first element. These cases in particular come up fairly often when programming. It's common to use the first element of a list (work with `thelist[0]`) and then continue with the tail (`thelist[1:]`). Similarly, you can work with the last element((`thelist[-1]`), and then use the head of the list (`thelist[:-1]`).

Manipulating Slices

You can actually do almost anything with list slices that you can do with simple indexing. For example, you can assign to a slice:

```
>>> colours = ['red', 'yellow', 'green', 'blue']
>>> colours[1:3] = ['yellowish', 'greenish']
>>> print colours
['red', 'yellowish', 'greenish', 'blue']
>>> colours[1:3] = ['pink', 'purple', 'ecru']
>>> print colours
['red', 'pink', 'purple', 'ecru', 'blue']
```

Notice that in the second assignment above, we assigned a list of three elements to a slice of length two. The list expands to make room or the new elements: the slice `colours[1:3]` (`['yellowish', 'greenish']`) is replaced with the list `['pink', 'purple', 'ecru']`. If the list assigned had been shorter than the slice, the list would have shrunk.

You can also remove any slice from a list:

```
>>> colours = ['red', 'yellow', 'green', 'blue']
>>> del colours[1:3]
>>> print colours
['red', 'blue']
```

Slices give you another way to manipulate lists. With both slices and the operators and methods mentioned in Topic 5.1, you can do many things with lists.

Topic 5.4 Strings

Much of what we have seen in the previous sections about lists also applies to strings. Both are considered *sequence types* in Python. Strings are a sequence of characters; lists are a sequence of any combination of types.

In Topic 5.2, we saw that the `for` loop can iterate through any list. Lists aren't the only type that can be used in the `for` loop. Any type that represents a collection of values can be used as the "list" in a `for` loop.

Since a string represents a sequence of characters, it can be used. For example, this program iterates through the characters in a string:

```
for char in "abc":
    print "A character:", char
```

When this is executed, it produces this output:

```
A character: a
A character: b
A character: c
```

List looping over a list, this can make your code very readable, and is often a very useful way to process a string.

Slicing Strings

In Topic 5.3, we used slices to manipulate parts of lists. You can slice strings using the same syntax as lists:

```
>>> sentence = "Look, I'm a string!"
>>> print sentence[:5]
Look
>>> print sentence[6:11]
I'm a
>>> print sentence[-7:]
string!
```

But, you can't modify a string slice, just like you can't assign to a single character of a string.

```
>>> sentence = "Look, I'm a string!"
>>> sentence[:5] = "Wow"
TypeError: object doesn't support slice assignment
>>> del sentence[6:10]
TypeError: object doesn't support slice assignment
```

Just like lists, you can use the slice `[:-1]` to indicate "everything but the last character" and `[1:]` for "everything but the first character".

Topic 5.5 Mutability

You may be wondering why assigning to a slice (or single element) works for a list, but not a string. For example:

```
dots = dots + "."      # statement #1
values = values + [n]  # statement #2
values.append(n)       # statement #3
```

Figure 5.3: Manipulating strings and lists

```
>>> colours = ['red', 'yellow', 'green', 'blue']
>>> colours[1:3] = ['yellowish', 'greenish']
>>> print colours
['red', 'yellowish', 'greenish', 'blue']
>>> sentence = "Look, I'm a string!"
>>> sentence[:5] = "Wow"
TypeError: object doesn't support slice assignment
```

Why is it possible to do more with lists than strings?

In fact, lists are the only data structure we have seen that can be changed *in-place*. There are several ways to modify an existing list without totally replacing it: assigning to a slice, using `del` to remove an element or slice, extending with the `append` method, and so on. Notice that none of these require creating a new list.

On the other hand, any string manipulation requires you to build a new string which can then be stored in a variable. Consider the statements in Figure 5.3. Statement #1 first builds a new string object (by evaluating the right side of the assignment, `dots + "."`), and then stores that new string in `dots`. The old value in `dots` is discarded because it's no longer in use.

In statement #2, the same thing happens with a list. A new list is built (by evaluating `values + [n]`), and the variable `values` is set to refer to it instead of its old value. The old value is discarded since it's no longer used. Each of these statements requires a lot of work if the initial string/list is large: it must be copied, along with the new item, and the old data is dropped from memory.

Statement #3 has the same effect as #2, but it happens in a very different way. In this case, the `append` method uses the *existing* list, and just adds another element to the end. This requires much less work, since the list doesn't have to be copied as part of evaluating an expression.

At the end of statement #2 or #3, the `values` variable holds the same list. In the case of #3, the list has been modified but not entirely rebuilt. This should be clear since it is not an assignment statement (i.e. a `var=`

statement). Any statement that assigns to a variable must be building a new value in the expression on the right of the `=`. If there is no assignment, the variable is still holding the same object, but the object may have changed.

Data structures that can be changed in-place like this are called *mutable*. Lists are the only mutable data structure we have seen in detail. The strings and numbers are not mutable: they are *immutable*.

Objects in Python are mutable if they contain methods that can change them without a new assignment. The `date` objects that were used in Topic 4.4 are immutable since there are no methods that can modify an existing `date` object. There are other object types in Python modules that have methods that modify them in-place: these are mutable objects.

In Python 2.4, two set types were added that can be created with the `set()` and `frozenset()` functions. These hold values like lists, but they aren't in any order. They are just a collection of values. The only difference between a `set` and `frozenset` is that sets are mutable (contain methods like `add` to insert a new element), and frozen sets are immutable (those methods aren't included). There are instances where either is useful, so they are both available.

Topic 5.6 References

There are several cases where the contents of one variable are copied to another. In particular, here are two operations that require duplicating the variable `x`:

```
# x copied to a parameter variable in some_function:
print some_function(x)
# x copied into y:
y = x
```

You probably don't think of copying the contents of a variable as a difficult operation, but consider the case where `x` is a list with thousands of elements. Then, making an full copy of `x` would be a lot of work for the computer, and probably unnecessary since all of its contents are already in memory.

In fact, Python avoids making copies where possible. To understand how this happens, it's important to understand *references*.

Statements:

```
my_string = "one" + "two" + "three"
my_list = [0, 10, 20]
```

Result:

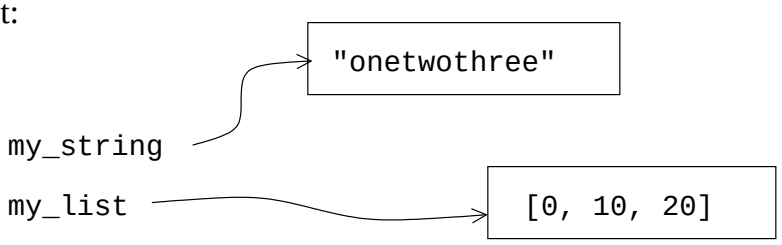

Figure 5.4: Variables referencing their contents

Every variable in Python is actually a reference to the place in memory where its contents are stored. Conceptually, you should think of a variable referencing its contents like an arrow pointing to the contents in memory. In Figure 5.4, you can see a representation of two variables referencing their contents.

When you use a variable in an expression, Python follows the reference to find its contents. When you assign to a variable, you are changing it so the variable now references different contents. (The old contents are thrown away since they are no longer being referenced.)

Usually, the expression on the right side of an assignment creates a new object in memory. The calculation is performed, and the result is stored in memory. The variable is then set to refer to this result. For example, `total = a+b` calculates `a+b`, stores this in memory, and sets `total` to reference that value.

The exception to this is when the right side of an assignment is simply a variable reference (like `total=a`). In this case, the result is already in memory and the variable can just reference the existing contents. For example, in Figure 5.5, `my_list` is created and refers to a list. When `my_list` is assigned to `list_copy`, the reference is copied, so the list is only stored once. This saves memory, and is faster since the contents don't have to be copied to another location in memory.

Statements:

```
my_list = [0, 10, 20]
list_copy = my_list
```

Result:

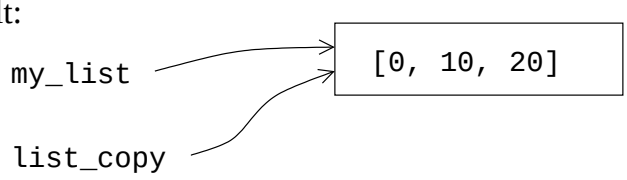

Figure 5.5: Reference copied during assignment: aliasing

Statements:

```
my_list = [0, 10, 20]
list_copy = my_list
list_copy.append(30)
my_list.append(40)
```

Result:

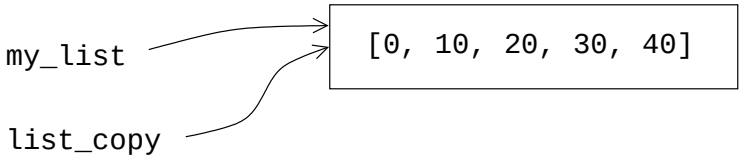

Figure 5.6: Changing either alias changes both

Aliases

When two variables refer to the same contents, they are *aliases* of each other. This has always happened when we assigned one variable to another, and it's generally good since it doesn't require copying the contents to another location in memory.

But, now that we have mutable data structures (lists and some objects), aliases complicate things. Since mutable data structures can be changed without totally rebuilding them, we can change the contents without moving the reference to a new object in memory.

That means that it's possible to change a variable, and the changes will

Statements:

```
my_string = "one" + "two" + "three"
string_copy = my_string
string_copy = string_copy + "four"
```

Result:

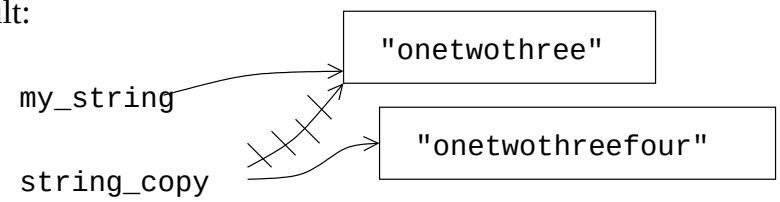

Figure 5.7: An expression creates a new reference and breaks the alias

affect any other variables *that reference the same contents.*

For example, in Figure 5.6, `my_list` and `list_copy` are aliases of the same contents. When either one is changed, both are affected. If you were to print out `my_list` and `list_copy` after these statement, you would find that they are both `[0, 10, 20, 30, 40]`.

The same things would happen if we used any methods that change the list (or any object that has such methods). So for any mutable data structure, aliasing is an issue.

Immutable data structures can also be aliased, but since they can't be changed, it never causes problems. For example, in Figure 5.7, a string is aliased when it is copied. But the only way to change it is to construct a new string during an assignment and the alias is removed.

Remember that any expression (that's more complicated than a variable reference) will result in a new reference being created. If this is assigned to a variable, then there is no aliasing. This is what happened in Figure 5.7. It also occurs with lists, as you can see in Figure 5.8.

Really Copying

If you want to make a copy of a variable that isn't a reference, it's necessary to force Python to actually copy its contents to a new place in memory. This is called *cloning.*

Cloning is more expensive than aliasing, but it's necessary when you do want to make a copy that can be separately modified.

Statements:

```
my_list = [0, 10, 20]
bigger_list = my_list + [30]
```

Result:

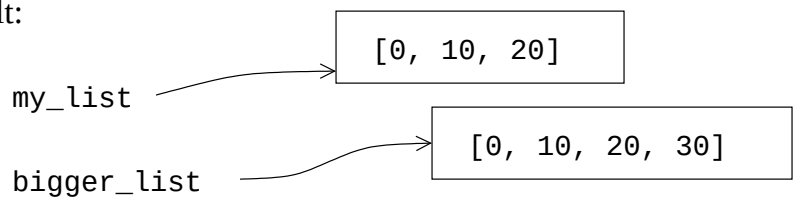

Figure 5.8: A calculation creates a new instance containing the results

Statements:

```
my_list = [0, 10, 20]
list_copy = my_list[:]
```

Result:

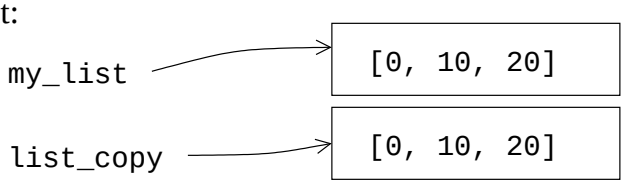

Figure 5.9: Slicing a list forces copying of its elements

For lists, the slice operator can be used to create a clone. Since cloning requires creating a new list, Python will copy whatever contents are needed by the slice. In Topic 5.3, we saw that in a slice like `[a:b]`, leaving out the `a` starts the slice at the start of the original list, and leaving out the `b` goes to the end of the list. If we combine these, we have a slice that refers to the entire list: `my_list[:]`.

For example, in Figure 5.9, you can see the slice operator being used to copy the contents of a list. This creates a new list and reference. Now, the lists can be changed independently.

You could also make a copy of a list with the `list` function that creates a new list (out of the old one). So, `list(my_list)` would give the same result as `my_list[:]`.

For other data types, the `copy` module contains a `copy` function. This function will take any Python object and clone its contents. If `obj` is a Python object, this code will produce a clone in `new_obj`:

```
import copy
new_obj = copy.copy(obj)
```

This should work with any mutable Python object, including lists. It will also clone immutable objects, but it's not clear why you would want to do that.

Summary

In this unit, you have learned the basics of lists in Python. You should also have picked up more tools that can be used to manipulate strings.

The concept of references might seem odd at first, but it's fundamental to many programming languages. It's one of those ideas that comes up often, and you'll have to be able to deal with it when it does.

Key Terms

- list
- append
- slice
- sequence
- mutable
- reference
- alias
- cloning