# **Chapter 8**

# Strings

# 8.1 A string is a sequence

A string is a **sequence** of characters. You can access the characters one at a time with the bracket operator:

>>> fruit = 'banana'
>>> letter = fruit[1]

The second statement selects character number 1 from fruit and assigns it to letter.

The expression in brackets is called an **index**. The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

>>> print letter a

For most people, the first letter of 'banana' is b, not a. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> print letter
b
```

So b is the 0th letter ("zero-eth") of 'banana', a is the 1th letter ("one-eth"), and n is the 2th ("two-eth") letter.

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

# 8.2 len

len is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the IndexError is that there is no letter in 'banana' with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

```
>>> last = fruit[length-1]
>>> print last
a
```

Alternatively, you can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

# 8.3 Traversal with a for loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to write a traversal is with a while loop:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print letter
    index = index + 1</pre>
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is index < len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

**Exercise 8.1.** Write a function that takes a string as an argument and displays the letters backward, one per line.

Another way to write a traversal is with a for loop:

```
for char in fruit:
print char
```

Each time through the loop, the next character in the string is assigned to the variable char. The loop continues until no characters are left.

The following example shows how to use concatenation (string addition) and a for loop to generate an abecedarian series (that is, in alphabetical order). In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:



Figure 8.1: Slice indices.

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
    print letter + suffix
The output is:
    Jack
    Kack
    Lack
    Mack
    Nack
    Oack
    Pack
    Qack
    Of course, that's not quite right because "Ouack" and "Quack" are misspelled.
    Exercise 8.2. Modify the program to fix this error.
```

# 8.4 String slices

A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print s[6:12]
Python
```

The operator [n:m] returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last. This behavior is counterintuitive, but it might help to imagine the indices pointing *between* the characters, as in Figure 8.1.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

**Exercise 8.3.** Given that fruit is a string, what does fruit[:] mean?

# 8.5 Strings are immutable

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
```

TypeError: object does not support item assignment

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an **object** is the same thing as a value, but we will refine that definition later. An **item** is one of the values in a sequence.

The reason for the error is that strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
Jello, world!
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

### 8.6 Searching

What does the following function do?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

In a sense, find is the opposite of the [] operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns -1.

This is the first example we have seen of a return statement inside a loop. If word[index] == letter, the function breaks out of the loop and returns immediately.

If the character doesn't appear in the string, the program exits the loop normally and returns -1.

This pattern of computation—traversing a sequence and returning when we find what we are looking for—is called a **search**.

**Exercise 8.4.** Modify find so that it has a third parameter, the index in word where it should start looking.

# 8.7 Looping and counting

The following program counts the number of times the letter a appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print count
```

This program demonstrates another pattern of computation called a **counter**. The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result—the total number of a's.

**Exercise 8.5.** Encapsulate this code in a function named count, and generalize it so that it accepts the string and the letter as arguments.

**Exercise 8.6.** *Rewrite this function so that instead of traversing the string, it uses the threeparameter version of* find *from the previous section.* 

# 8.8 String methods

A **method** is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the method upper takes a string and returns a new string with all uppercase letters:

Instead of the function syntax upper(word), it uses the method syntax word.upper().

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no argument.

A method call is called an **invocation**; in this case, we would say that we are invoking upper on the word.

As it turns out, there is a string method named find that is remarkably similar to the function we wrote:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
```

In this example, we invoke find on word and pass the letter we are looking for as a parameter.

Actually, the find method is more general than our function; it can find substrings, not just characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

And as a third argument the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because b does not appear in the index range from 1 to 2 (not including 2). **Exercise 8.7.** There is a string method called count that is similar to the function in the previous exercise. Read the documentation of this method and write an invocation that counts the number of as in 'banana'.

**Exercise 8.8.** Read the documentation of the string methods at docs.python.org/lib/ string-methods.html. You might want to experiment with some of them to make sure you understand how they work. strip and replace are particularly useful.

The documentation uses a syntax that might be confusing. For example, in find(sub[, start[, end]]), the brackets indicate optional arguments. So sub is required, but start is optional, and if you include start, then end is optional.

### 8.9 The in operator

The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

For example, the following function prints all the letters from word1 that also appear in word2:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print letter
```

With well-chosen variable names, Python sometimes reads like English. You could read this loop, "for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter."

Here's what you get if you compare apples and oranges:

```
>>> in_both('apples', 'oranges')
a
e
s
```

### 8.10 String comparison

The relational operators work on strings. To see if two strings are equal:

```
if word == 'banana':
    print 'All right, bananas.'
```

Other relational operations are useful for putting words in alphabetical order:

```
if word < 'banana':
    print 'Your word,' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word,' + word + ', comes after banana.'
else:
    print 'All right, bananas.'
```

Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so:

Your word, Pineapple, comes before banana.

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

# 8.11 Debugging

When you use indices to traverse the values in a sequence, it is tricky to get the beginning and end of the traversal right. Here is a function that is supposed to compare two words and return True if one of the words is the reverse of the other, but it contains two errors:

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)

while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
```

return True

The first if statement checks whether the words are the same length. If not, we can return False immediately and then, for the rest of the function, we can assume that the words are the same length. This is an example of the guardian pattern in Section 6.8.

i and j are indices: i traverses word1 forward while j traverses word2 backward. If we find two letters that don't match, we can return False immediately. If we get through the whole loop and all the letters match, we return True.

If we test this function with the words "pots" and "stop", we expect the return value True, but we get an IndexError:

>>> is\_reverse('pots', 'stop')
...



Figure 8.2: State diagram.

```
File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

For debugging this kind of error, my first move is to print the values of the indices immediately before the line where the error appears.

```
while j > 0:
    print i, j  # print here
    if word1[i] != word2[j]:
        return False
    i = i+1
    j = j-1
```

Now when I run the program again, I get more information:

```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

The first time through the loop, the value of j is 4, which is out of range for the string 'pots'. The index of the last character is 3, so the initial value for j should be len(word2)-1.

If I fix that error and run the program again, I get:

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

This time we get the right answer, but it looks like the loop only ran three times, which is suspicious. To get a better idea of what is happening, it is useful to draw a state diagram. During the first iteration, the frame for is\_reverse is shows in Figure 8.2.

I took a little license by arranging the variables in the frame and adding dotted lines to show that the values of i and j indicate characters in word1 and word2.

**Exercise 8.9.** Starting with this diagram, execute the program on paper, changing the values of i and j during each iteration. Find and fix the second error in this function.

### 8.12 Glossary

**object:** Something a variable can refer to. For now, you can use "object" and "value" interchangeably.

**sequence:** An ordered set; that is, a set of values where each value is identified by an integer index.

item: One of the values in a sequence.

index: An integer value used to select an item in a sequence, such as a character in a string.

slice: A part of a string specified by a range of indices.

**empty string:** A string with no characters and length 0, represented by two quotation marks.

immutable: The property of a sequence whose items cannot be assigned.

**traverse:** To iterate through the items in a sequence, performing a similar operation on each.

**search:** A pattern of traversal that stops when it finds what it is looking for.

**counter:** A variable used to count something, usually initialized to zero and then incremented.

**method:** A function that is associated with an object and called using dot notation.

invocation: A statement that calls a method.

#### 8.13 Exercises

**Exercise 8.10.** A string slice can take a third index that specifies the "step size;" that is, the number of spaces between successive characters. A step size of 2 means every other character; 3 means every third, etc.

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

A step size of -1 goes through the word backwards, so the slice [::-1] generates a reversed string.

Use this idiom to write a one-line version of is\_palindrome from Exercise 6.6.

**Exercise 8.11.** The following functions are all intended to check whether a string contains any lowercase letters, but at least some of them are wrong. For each function, describe what the function actually does (assuming that the parameter is a string).

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
```

```
else:
            return 'False'
def any_lowercase3(s):
    for c in s:
        flag = c.islower()
   return flag
def any_lowercase4(s):
   flag = False
   for c in s:
        flag = flag or c.islower()
   return flag
def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
   return True
```

**Exercise 8.12.** ROT13 is a weak form of encryption that involves "rotating" each letter in a word by 13 places. To rotate a letter means to shift it through the alphabet, wrapping around to the beginning if necessary, so 'A' shifted by 3 is 'D' and 'Z' shifted by 1 is 'A'.

Write a function called rotate\_word that takes a string and an integer as parameters, and that returns a new string that contains the letters from the original string "rotated" by the given amount.

For example, "cheer" rotated by 7 is "jolly" and "melon" rotated by -10 is "cubed".

You might want to use the built-in functions ord, which converts a character to a numeric code, and chr, which converts numeric codes to characters.

Potentially offensive jokes on the Internet are sometimes encoded in ROT13. If you are not easily offended, find and decode some of them. Solution: http://thinkpython.com/code/rotate. py.