7. Strings

7.1 A compound data type

So far we have seen five types: `int`, `float`, `bool`, `NoneType` and `str`. Strings are qualitatively different from the other four because they are made up of smaller pieces---characters.

Types that comprise smaller pieces are called **compound data types**. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This ambiguity is useful.

The bracket operator selects a single character from a string:

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print letter
```

The expression `fruit[1]` selects character number 1 from `fruit`. The variable `letter` refers to the result. When we display `letter`, we get a surprise:

```
a
```

The first letter of `"banana"` is not `a`, unless you are a computer scientist. For perverse reasons, computer scientists always start counting from zero. The 0th letter ("zero-eth") of `"banana"` is `b`. The 1th letter ("one-eth") is `a`, and the 2th ("two-eth") letter is `n`.

If you want the zero-eth letter of a string, you just put 0, or any expression with the value 0, in the brackets:

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

The expression in brackets is called an **index**. An index specifies a member of an ordered set, in this case the set of characters in the string. The index *indicates* which one you want, hence the name. It can be any integer expression.

7.2 Length

The `len` function 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]        # ERROR!
```

That won't work. It causes the runtime error `IndexError: string index out of range`. The reason is that there is no 6th letter in `"banana"`. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, we have to subtract 1 from `length`:

```
length = len(fruit)
last = fruit[length-1]
```

Alternatively, we 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.

7.3 Traversal and the `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 encode a traversal is with a `while` statement:

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

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.

Using an index to traverse a set of values is so common that Python provides an alternative, simpler syntax---the `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 and a `for` loop to generate an abecedarian series. "Abecedarian" refers to a series or list in which the elements appear in alphabetical order. For example, 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:

```
prefixes = "JKLMNOPQ"
suffix = "ack"

for letter in prefixes:
    print letter + suffix
```

The output of this program is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

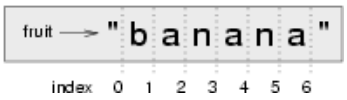
Of course, that's not quite right because "Ouack" and "Quack" are misspelled. You'll fix this as an exercise below.

7.4 String slices

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

```
>>> s = "Peter, Paul, and Mary"
>>> print s[0:5]
Peter
>>> print s[7:11]
Paul
>>> print s[17:21]
Mary
```

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; it makes more sense if you imagine the indices pointing *between* the characters, as in the following diagram:



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. Thus:

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

What do you think `s[:]` means?

7.5 String comparison

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

```
if word == "banana":
    print  "Yes, we have no bananas!"
```

Other comparison 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 "Yes, we have no bananas!"
```

You should be aware, though, that Python does not handle upper- and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters. As a result:

```
Your word, Zebra, 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. A more difficult problem is making the program realize that zebras are not fruit.

7.6 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'             # ERROR!
print greeting
```

Instead of producing the output `Jello, world!`, this code produces the runtime error `TypeError: 'str' object doesn't support item assignment`.

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!"
newGreeting = 'J' + greeting[1:]
print newGreeting
```

The solution here is to concatenate a new first letter onto a slice of `greeting`. This operation has no effect on the original string.

7.7 The `in` operator

The `in` operator tests if one string is a substring of another:

```
>>> 'p' in 'apple'
True
>>> 'i' in 'apple'
False
>>> 'ap' in 'apple'
True
>>> 'pa' in 'apple'
False
```

Note that a string is a substring of itself:

```
>>> 'a' in 'a'
True
>>> 'apple' in 'apple'
True
```

Combining the `in` operator with sting concatenation using +, we can write a function that removes all the vowels from a string:

```
def remove_vowels(s):
    vowels = "aeiouAEIOU"
    s_without_vowels = ""
    for letter in s:
        if letter not in vowels:
            s_without_vowels += letter
    return s_without_vowels
```

Test this function to confirm that it does what we wanted it to do.

7.8 A `find` function

What does the following function do?

```
def find(strng, ch):
    index = 0
    while index < len(strng):
        if strng[index] == ch:
            return 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 `strng[index] == ch`, the function returns immediately, breaking out of the loop prematurely.

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

This pattern of computation is sometimes called a "eureka" traversal because as soon as we find what we are looking for, we can cry "Eureka!" and stop looking.

7.9 Looping and counting

The following program counts the number of times the letter `a` appears in a string, and is another example of the counter pattern introduced in chapter 6:

```
fruit = "banana"
count = 0
for char in fruit:
    if char == 'a':
        count += 1
print count
```

7.10 Optional parameters

To find the locations of the second or third occurence of a character in a string, we can modify the `find` function, adding a third parameter for the starting postion in the search string:

```
def find2(strng, ch, start):
    index = start
    while index < len(strng):
        if strng[index] == ch:
            return index
        index += 1
    return -1
```

The call `find2('banana', 'a', 2)` now returns `3`, the index of the first occurance of 'a' in 'banana' after index 2. What does `find2('banana', 'n', 3)` return? If you said, 4, there is a good chance

you understand how `find2` works.

Better still, we can combine `find` and `find2` using an **optional parameter**:

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

The call `find('banana', 'a', 2)` to this version of `find` behaves just like `find2`, while in the call `find('banana', 'a')`, `start` will be set to the **default value** of `0`.

Adding another optional parameter to `find` makes it search both forward and backward:

```
def find(strng, ch, start=0, step=1):
    index = start
    while 0 <= index < len(strng):
        if strng[index] == ch:
            return index
        index += step
    return -1
```

Passing in a value of `-1` for `step` will make it search toward the beginning of the string instead of the end. Note that we needed to check for a lower bound for `index` in the while loop as well as an upper bound to accomodate this change.

7.11 The `string` module

The `string` module contains useful functions that manipulate strings. As usual, we have to import the module before we can use it:

```
>>> import string
```

To see what is inside it, use the `dir` function with the module name as an argument.

```
>>> dir(string)
```

which will return the list of items inside the string module:

```
['Template', '_TemplateMetaclass', '__builtins__', '__doc__', '__file__',
'__name__', '_float', '_idmap', '_idmapL', '_int', '_long', '_multimap',
'_re', 'ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'atof',
'atof_error', 'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize',
'capwords', 'center', 'count', 'digits', 'expandtabs', 'find',
'hexdigits', 'index', 'index_error', 'join', 'joinfields', 'letters',
'ljust', 'lower', 'lowercase', 'lstrip', 'maketrans', 'octdigits',
'printable', 'punctuation', 'replace', 'rfind', 'rindex', 'rjust',
```

```
'rsplit', 'rstrip', 'split', 'splitfields', 'strip', 'swapcase',
'translate', 'upper', 'uppercase', 'whitespace', 'zfill']
```

To find out more about an item in this list, we can use the `type` command. We need to specify the module name followed by the item using **dot notation**.

```
>>> type(string.digits)
<type 'str'>
>>> type(string.find)
<type 'function'>
```

Since `string.digits` is a string, we can print it to see what it contains:

```
>>> print string.digits
0123456789
```

Not surprisingly, it contains each of the decimal digits.

`string.find` is a function which does much the same thing as the function we wrote. To find out more about it, we can print out its **docstring**, `__doc__`, which contains documentation on the function:

```
>>> print string.find.__doc__
find(s, sub [,start [,end]]) -> in

    Return the lowest index in s where substring sub is found,
    such that sub is contained within s[start,end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

The parameters in square brackets are optional parameters. We can use `string.find` much as we did our own `find`:

```
>>> fruit = "banana"
>>> index = string.find(fruit, "a")
>>> print index
1
```

This example demonstrates one of the benefits of modules---they help avoid collisions between the names of built-in functions and user-defined functions. By using dot notation we can specify which version of `find` we want.

Actually, `string.find` is more general than our version. it can find substrings, not just characters:

```
>>> string.find("banana", "na")
2
```

Like ours, it takes an additional argument that specifies the index at which it should start:

```
>>> string.find("banana", "na", 3)
4
```

Unlike ours, its second optional parameter specifies the index at which the search should end:

```
>>> string.find("bob", "b", 1, 2)
-1
```

In this example, the search fails because the letter *b* does not appear in the index range from `1` to `2` (not including `2`).

7.12 Character classification

It is often helpful to examine a character and test whether it is upper- or lowercase, or whether it is a character or a digit. The `string` module provides several constants that are useful for these purposes. One of these, `string.digits`, we have already seen.

The string `string.lowercase` contains all of the letters that the system considers to be lowercase. Similarly, `string.uppercase` contains all of the uppercase letters. Try the following and see what you get:

```
print string.lowercase
print string.uppercase
print string.digits
```

We can use these constants and `find` to classify characters. For example, if `find(lowercase, ch)` returns a value other than `-1`, then `ch` must be lowercase:

```
def is_lower(ch):
    return string.find(string.lowercase, ch) != -1
```

Alternatively, we can take advantage of the `in` operator:

```
def is_lower(ch):
    return ch in string.lowercase
```

As yet another alternative, we can use the comparison operator:

```
def is_lower(ch):
    return 'a' <= ch <= 'z'
```

If `ch` is between *a* and *z*, it must be a lowercase letter.

Another constant defined in the `string` module may surprise you when you print it:

```
>>> print string.whitespace
```

Whitespace characters move the cursor without printing anything. They create the white space between visible characters (at least on white paper). The constant `string.whitespace` contains all the whitespace characters, including space, tab (\t), and newline (\n).

There are other useful functions in the `string` module, but this book isn't intended to be a reference manual. On the other hand, the *Python Library Reference* is. Along with a wealth of other documentation, it's available from the Python website, http://www.python.org.

7.13 String formatting

The most concise and powerful way to format a string in Python is to use the *string formatting operator*, `%`, together with Python's string formatting operations. To see how this works, let's start with a few examples:

```
>>> "His name is %s."  % "Arthur"
'His name is Arthur.'
>>> name = "Alice"
>>> age = 10
>>> "I am %s and I am %d years old." % (name, age)
'I am Alice and I am 10 years old.'
>>> n1 = 4
>>> n2 = 5
>>> "2**10 = %d and %d * %d = %f" % (2**10, n1, n2, n1 * n2)
'2**10 = 1024 and 4 * 5 = 20.000000'
>>>
```

The syntax for the string formatting operation looks like this:

```
"<FORMAT>" % (<VALUES>)
```

It begins with a *format* which contains a sequence of characters and *conversion specifications*. Conversion specifications start with a `%` operator. Following the format string is a single `%` and then a sequence of values, *one per conversion specification*, seperated by commas and enclosed in parenthesis. The parenthesis are optional if there is only a single value.

In the first example above, there is a single conversion specification, `%s`, which indicates a string. The single value, `"Arthur"`, maps to it, and is not enclosed in parenthesis.

In the second example, `name` has string value, `"Alice"`, and `age` has integer value, `10`. These map to the two converstion specifications, `%s` and `%d`. The `d` in the second converstion specification indicates that the value is a decimal integer.

In the third example variables `n1` and `n2` have integer values `4` and `5` respectively. There are four converstion specifications in the format string: three `%d`'s and a `%f`. The `f` indicates that the value should be represented as a floating point number. The four values that map to the four converstion specifications are: `2**10`, `n1`, `n2`, and `n1 * n2`.

`s`, `d`, and `f` are all the conversion types we will need for this book. To see a complete list, see the String

Formatting Operations section of the Python Library Reference.

The following example illustrates the real utility of string formatting:

```
i = 1
print "i\ti**2\ti**3\ti**5\ti**10\ti**20"
while i <= 10:
    print i, '\t', i**2, '\t', i**3, '\t', i**5, '\t', i**10, '\t', i**20
    i += 1
```

This program prints out a table of various powers of the numbers from 1 to 10. In its current form it relies on the tab character (`\t`) to align the columns of values, but this breaks down when the values in the table get larger than the 8 character tab width:

```
i       i**2    i**3    i**5    i**10   i**20
1       1       1       1       1       1
2       4       8       32      1024    1048576
3       9       27      243     59049   3486784401
4       16      64      1024    1048576         1099511627776
5       25      125     3125    9765625         95367431640625
6       36      216     7776    60466176        3656158440062976
7       49      343     16807   282475249       79792266297612001
8       64      512     32768   1073741824      11529215046068469 76
9       81      729     59049   3486784401      12157665459056928801
10      100     1000    100000  10000000000     100000000000000000000
```

One possible solution would be to change the tab width, but the first column already has more space than it needs. The best solution would be to set the width of each column independently. As you may have guessed by now, string formatting provides the solution:

```
i = 1
print "%-4s%-5s%-6s%-8s%-13s%-15s" %
      ('i', 'i**2', 'i**3', 'i**5', 'i**10', 'i**20')
while i <= 10:
    print "%-4d%-5d%-6d%-8d%-13d%-15d" % (i, i**2, i**3, i**5, i**10, i**20)
    i += 1
```

Running this version produces the following output:

```
i   i**2 i**3  i**5    i**10        i**20
1   1    1     1       1            1
2   4    8     32      1024         1048576
3   9    27    243     59049        3486784401
4   16   64    1024    1048576      1099511627776
5   25   125   3125    9765625      95367431640625
6   36   216   7776    60466176     3656158440062976
7   49   343   16807   282475249    79792266297612001
8   64   512   32768   1073741824   11529215046068469 76
9   81   729   59049   3486784401   12157665459056928801
```

```
10   100   1000   100000   10000000000   10000000000000000000
```

The - after each % in the converstion specifications indicates left justification. The numerical values specify the minimum length, so %-13d is a left justified number at least 13 characters wide.

7.14 Glossary

compound data type:
A data type in which the values are made up of components, or elements, that are themselves values.

index:
A variable or value used to select a member of an ordered set, such as a character from a string.

traverse:
To iterate through the elements of a set, performing a similar operation on each.

slice:
A part of a string (substring) specified by a range of indices. More generally, a subsequence of any sequence type in Python can be created using the slice operator (*sequence*[*start*:*stop*]).

immutable:
A compound data types whose elements can not be assigned new values.

optional parameter:
A parameter written in a function header with an assignment to a default value which it will receive if no corresponding argument is given for it in the function call.

default value:
The value given to an optional parameter if no argument for it is provided in the function call.

dot notation:
Use of the **dot operator**, **.**, to access functions inside a module.

docstring:
A string constant on the first line of a function or module definition (and as we will see later, in class and method definitions as well). Docstrings provide a convinient way to associate documentation with code. Docstrings are also used by the doctest module for automated testing.

whitespace:
Any of the characters that move the cursor without printing visible characters. The constant string.whitespace contains all the white-space characters.

7.15 Exercises

1. Modify:

```
prefixes = "JKLMNOPQ"
suffix = "ack"

for letter in prefixes:
    print letter + suffix
```

   so that Ouack and Quack are spelled correctly.

2. Encapsulate

```
fruit = "banana"
```

```
count = 0
for char in fruit:
    if char == 'a':
        count += 1
print count
```

in a function named `count_letters`, and generalize it so that it accepts the string and the letter as arguments.

3. Now rewrite the `count_letters` function so that instead of traversing the string, it repeatedly calls `find` (the version from section 8.10), with the optional third parameter to locate new occurences of the letter being counted.
4. Which version of `is_lower` do you think will be fastest? Can you think of other reasons besides speed to prefer one version or the other?
5. Create a file named `stringtools.py` and put the following in it:

```
def reverse(s):
    """
      >>> reverse('happy')
      'yppah'
      >>> reverse('Python')
      'nohtyP'
      >>> reverse("")
      ''
      >>> reverse("P")
      'P'
    """

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Add a function body to `reverse` to make the doctests pass.

6. Add `mirror` to `stringtools.py` .

```
def mirror(s):
    """
      >>> mirror("good")
      'gooddoog'
      >>> mirror("yes")
      'yessey'
      >>> mirror('Python')
      'PythonnohtyP'
      >>> mirror("")
      ''
      >>> mirror("a")
      'aa'
    """
```

Write a function body for it that will make it work as indicated by the doctests.

7. Include `remove_letter` in `stringtools.py` .

```
def remove_letter(letter, strng):
    """
      >>> remove_letter('a', 'apple')
      'pple'
      >>> remove_letter('a', 'banana')
      'bnn'
      >>> remove_letter('z', 'banana')
      'banana'
      >>> remove_letter('i', 'Mississippi')
      'Mssspp'
    """
```

Write a function body for it that will make it work as indicated by the doctests.

8. Finally, add bodies to each of the following functions, one at a time

```
def is_palindrome(s):
    """
      >>> is_palindrome('abba')
      True
      >>> is_palindrome('abab')
      False
      >>> is_palindrome('tenet')
      True
      >>> is_palindrome('banana')
      False
      >>> is_palindrome('straw warts')
      True
    """

def count(sub, s):
    """
      >>> count('is', 'Mississippi')
      2
      >>> count('an', 'banana')
      2
      >>> count('ana', 'banana')
      2
      >>> count('nana', 'banana')
      1
      >>> count('nanan', 'banana')
      0
    """

def remove(sub, s):
    """
      >>> remove('an', 'banana')
      'bana'
      >>> remove('cyc', 'bicycle')
      'bile'
      >>> remove('iss', 'Mississippi')
      'Missippi'
      >>> remove('egg', 'bicycle')
      'bicycle'
```

```
    """

def remove_all(sub, s):
    """
      >>> remove_all('an', 'banana')
      'ba'
      >>> remove_all('cyc', 'bicycle')
      'bile'
      >>> remove_all('iss', 'Mississippi')
      'Mippi'
      >>> remove_all('eggs', 'bicycle')
      'bicycle'
    """
```

until all the doctests pass.

9. Try each of the following formatted string operations in a Python shell and record the results:
   a. "%s %d %f" % (5, 5, 5)
   b. "%-.2f" % 3
   c. "%-10.2f%-10.2f" % (7, 1.0/2)
   d. print " $%5.2f\n $%5.2f\n $%5.2f" % (3, 4.5, 11.2)
10. The following formatted strings have errors. Fix them:
   a. "%s %s %s %s" % ('this', 'that', 'something')
   b. "%s %s %s" % ('yes', 'no', 'up', 'down')
   c. "%d %f %f" % (3, 3, 'three')

Table of Contents | Index