Note the variable names inside braces in `formatStr`, and the dictionary reference used as the format parameter is `**locals()`.

A string like `formatStr` is probably the most readable way to code the creation of a string from a collection of literal strings and program values. The ending part of the syntax, `.format(**locals())`, may appear a bit strange, but it is very useful! We will use this notation extensively to clearly indicate how values are embedded into strings.

The example program `hello_you4.py` does the same thing as the earlier hello_you versions, but with a dictionary reference:

```
person = input('Enter your name: ')
greeting = 'Hello {person}!'.format(**locals())
print(greeting)
```

## 1.13. Loops and Sequences

Modern computers can do millions or even billions of instructions a second. With the techniques discussed so far, it would be hard to get a program that would run by itself for more than a fraction of a second.[6] Practically, we cannot write millions of instructions to keep the computer busy. To keep a computer doing useful work we need *repetition*, looping back over the same block of code again and again. There are two Python statement types to do that: the simpler `for` loops, which we take up shortly, and `while` loops, which we take up later, in Section 3.3. Two preliminaries: First, the value of already defined variables can be updated. This will be particularly important in loops. We start by following how variables can be updated in an even simpler situation. Second, *for* loops involve sequence types, so we will first look at a basic sequence type: `list`. This is a long section. Go carefully.

**1.13.1. Updating Variables.** The programs so far have defined and used variables, but other than in early shell examples we have not changed the value of existing variables. For now consider a particularly simple example, just chosen as an illustration, in the example file `updateVar.py`:

```
x = 3           #1
y = x + 2       #2
y = 2*y         #3
x = y - x       #4
print(x, y)     #5
```

Can you *predict* the result? Run the program and check. Particularly if you did not guess right, it is important to understand what happens, one step at a time. That means keeping track of what changes to variables are made by each statement. In the table below, statements are referred to by the numbers labeling the lines in the code above. We can track the state of each variable after each line in executed. A dash is shown where a variable is not defined. For instance after line 1 is executed, a value is given to x, but y is still undefined. Then y gets a value in line 2. The comment on the right summarizes what is happening. Since x has the value 3 when line 2 starts, x+2 is the same as 3+2. In line three we use the fact that the right side of an assignment statement uses the values of variables when the line starts executing (what is left after the previous line of the table executed), but the assignment to the variable y on the left causes a change to y, and hence the updated value of y, 10, is shown in the table. Line 4 then changes x, using the *latest* value of y (10, not the initial value 5!). The result from line 5 confirms the values of x and y.

| Line | x | y | comment |
|---|---|---|---|
| 1 | 3 | - | |
| 2 | 3 | 5 | 5=3+2, using the value of x from the previous line |
| 3 | 3 | 10 | 10=2*5 on the right, use the value of y from the previous line |
| 4 | 7 | 19 | 7=10-3 on the right, use the value of x and y from the previous line |
| 5 | 7 | 10 | print: 7 10 |

The order of execution will always be the order of the lines in the table. In this simple sequential code, that also follows the textual order of the program. Following each line of execution of a program in order, carefully, keeping track of the current values of variables, will be called *playing computer*. A table like the one above is an organized way to keep track.

[6]It is possible with function recursion, but we will avoid that topic in this introduction.

**1.13.2. The `list` Type.** Lists are ordered sequences of arbitrary data. Lists are the first kind of data discussed so far that are *mutable*: the length of the sequence can be changed and elements substituted. We will delay the discussion of changes to lists until a further introduction to objects. Lists can be written explicitly. *Read* the following examples

```
['red', 'green', 'blue']
[1, 3, 5, 7, 9, 11]
['silly', 57, 'mixed', -23, 'example']
[] # the empty list
```

The basic format is square-bracket-enclosed, comma-separated lists of arbitrary data.

**1.13.3. The `range` Function, Part 1.** There is a built-in function `range`, that can be used to automatically generate regular arithmetic sequences. Try the following in the *Shell*:

```
list(range(4))
list(range(10))
```

The general pattern for use is

```
range(sizeOfSequence)
```

This syntax will generate the items, one at a time, as needed. If you want to see all the results at once as a list, you can convert to a list as in the examples above. The resulting sequence starts at 0 and ends *before* the parameter. We will see there are good reasons to start from 0 in Python. One important property of sequences generated by `range(n)` is that the total number of elements is `n`. The sequence omits the number `n` itself, but includes 0 instead.

With more parameters, the `range` function can be used to generate a much wider variety of sequences. The elaborations are discussed in Section 2.4.12 and Section 3.3.2.

**1.13.4. Basic `for` Loops.** Try the following in the *Shell*. You get a sequence of continuation lines before the Shell responds. Be sure to indent the second and third lines. (This is only needed inthe Shell, not in an edit window, where the indentation is automatic). *Be sure to enter another empty line (just* ENTER*) at the end to get the Shell to respond.*

```
for count in [1, 2, 3]:
    print(count)
    print('Yes' * count)
```

This is a `for` loop. It has the heading starting with `for`, followed by a variable name (`count` in this case), the word `in`, some sequence, and a final colon. As with function definitions and other heading lines ending with a colon, the colon at the end of the line indicates that a consistently indented block of statements follows to complete the `for` loop.

```
for item in sequence:
    indented statements to repeat
```

The block of lines is repeated once for each element of the sequence, so in this example the two lines in the indented block are repeated three times. Furthermore the variable in the heading (`count` here) may be used in the block, and each time through it takes on the *next* value in the sequence, so the first time through the loop `count` is 1, then 2, and finally 3. Look again at the output and see that it matches this sequence.

There is a reason the interpreter waited to respond until after you entered an empty line: The interpreter did not know how long the loop block was going to be! The empty line is a signal to the interpreter that you are done with the loop block.

Look at the following example program `for123.py`, and run it.

```
for count in [1, 2, 3]:                 #1
    print(count)                        #2
    print('Yes'*count)                  #3
print('Done counting.')                 #4
for color in ['red', 'blue', 'green']:  #5
    print(color)                        #6
```

In a file, where the interpreter does not need to respond immediately, the blank line is not necessary. Instead, as with a function definition or any other format with an indented block, you indicate being past the indented block by *de*denting to line up with the `for`-loop heading. Hence in the code above, "Done Counting." is printed once after the first loop completes all its repetitions. Execution ends with another simple loop.

As with the indented block in a function, it is important to get the indentation right. Alter the code above, so line 4 is indented:

```
for count in [1, 2, 3]:               #1
    print(count)                      #2
    print('Yes'*count)                #3
    print('Done counting.')               #4
for color in ['red', 'blue', 'green']:  #5
    print(color)                      #6
```

Predict the change, and run the code again to test.

Loops are one of the most important features in programming. While the syntax is pretty simple, using them creatively to solve problems (rather than just look at a demonstration) is among the biggest challenges for many learners at an introductory level. One way to simplify the learning curve is to classify common situations and patterns. One of the simplest patterns is illustrated above, simple *for-each* loops.

```
for item in sequence
    do some thing with item
```

(It would be even more like English if `for` were replace by `for each`, but the shorter version is the one used by Python.)

In the `for`-loop examples above, something is printed that is related to each item in the list. Printing is certainly one form of "do something", but the possibilities for "do something" are completely general!

We can use a for-each loop to revise our first example. *Recall* the code from madlib.py:

```
addPick('animal', userPicks)
addPick('food', userPicks)
addPick('city', userPicks)
```

Each line is doing exactly the same thing, except varying the string used as the cue, while repeating the rest of the line. This is the for-each pattern, but we need to list the sequence that the cues come from. *Read* the alternative:

```
for cue in ['animal', 'food', 'city']:  # heading
    addPick(cue, userPicks)             # body
```

If you wish to see or run the whole program with this small modification, see the example `madlibloop.py`.

It is important to understand the sequence of operations, how execution goes back and forth between the heading and the body. Here are the details:

(1) heading first time: variable `cue` is set to the first element of the sequence, `'animal'`
(2) body first time: since `cue` is now `'animal'`, effectively execute `addPick('animal', userPicks)` (Skip the details of the function call in this outline.)
(3) heading second time: variable `cue` is set to the next element of the sequence, `'food'`
(4) body second time: since `cue` is now `'food'`, effectively execute `addPick('food', userPicks)`
(5) heading third time: variable `cue` is set to the next (last) element of the sequence, `'city'`
(6) body third time: since `cue` is now `'city'`, effectively execute `addPick('city', userPicks)`
(7) heading done: Since there are no more elements in the sequence, the entire `for` loop is done and execution would continue with the statement after it.

This looping construction would be even handier if you were to modify the original mad lib example, and had a story with many more cues. Also this revision will allow for further improvements in Section 2.3.3, after we introduce more about string manipulation.

**1.13.5. Simple Repeat Loops.** The examples above all used the value of the variable in the `for`-loop heading. An even simpler `for`-loop usage is when you just want to repeat the exact same thing a specific number of times. In that case only the *length* of the sequence, not the individual elements are important. We have already seen that the `range` function provides an ease way to produce a sequence with a specified number of elements. Read and run the example program `repeat1.py`:

```
for i in range(10):
    print('Hello')
```

In this situation, the variable `i` is not used inside the body of the for-loop.

The user could choose the number of times to repeat. Read and run the example program `repeat2.py`:

```
n = int(input('Enter the number of times to repeat: '))
for i in range(n):
    print('This is repetitious!')
```

**1.13.6. Successive Modification Loops.** Suppose I have a list of items called `items`, and I want to print out each item and number them successively. For instance if `items` is ['red', 'orange', 'yellow', 'green'], I would *like* to see the *output*:

```
1 red
2 orange
3 yellow
4 green
```

*Read about the following thought process for developing this:*

If I allow myself to omit the numbers, it is easy: For any `item` in the list, I can process it with

```
print(item)
```

and I just go through the list and do it *for each* one. (Copy and run if you like.)

```
items = ['red', 'orange', 'yellow', 'green']
for item in items:
    print(item)
```

Clearly the more elaborate version with numbers has a pattern with some consistency, each line is at least in the form:

```
number item
```

but the number changes each time, and the numbers do *not* come straight from the list of items.

A variable can change, so it makes sense to have a variable `number`, so we have the potential to make it change correctly. We could easily get it right the first time, and then repeat the *same* number. Read and run the example program `numberEntries1.py`:

```
items = ['red', 'orange', 'yellow', 'green']
number = 1
for item in items:
    print(number, item)
```

Of course this is still not completely correct, since the idea was to *count.* After the first time number is printed, it needs to be changed to 2, to be right the next time through the loop, as in the following code: Read and run the example program `numberEntries2.py`:

```
items = ['red', 'orange', 'yellow', 'green']
number = 1
for item in items:
    print(number, item)
    number = 2
```

This is closer, but still not completely correct, since we never get to 3! We need a way to change the value of number *that will work each time through the loop.* The pattern of counting is simple, so simple in fact that you probably do not think consciously about how you go from one number to the next: You can describe the pattern by saying each successive number is *one more than the previous number.* We need to be able to change `number` so it is one more than it was before. That is the additional idea we need! Change the last line of the loop body to get the example program numberEntries3.py. See the addition and run it:

```
items = ['red', 'orange', 'yellow', 'green']  #1
number = 1                                     #2
for item in items:                             #3
    print(number, item)                        #4
    number = number + 1                        #5
```

It is important to understand the step-by-step changes during execution. Below is another table showing the results of playing computer. The line numbers are much more important here to keep track of the flow of control, because of the jumping around at the end of the loop.

| Line | items | item | number | comment |
|---|---|---|---|---|
| 1 | ['red', 'orange', 'yellow', 'green'] | - | - | |
| 2 | ['red', 'orange', 'yellow', 'green'] | - | 1 | |
| 3 | ['red', 'orange', 'yellow', 'green'] | 'red' | 1 | start with item as first in sequence |
| 4 | ['red', 'orange', 'yellow', 'green'] | 'red' | 1 | print: 1 red |
| 5 | ['red', 'orange', 'yellow', 'green'] | 'red' | 2 | 2 = 1+1 |
| 3 | ['red', 'orange', 'yellow', 'green'] | 'orange' | 2 | on to the next element in sequence |
| 4 | ['red', 'orange', 'yellow', 'green'] | 'orange' | 2 | print 2 orange |
| 5 | ['red', 'orange', 'yellow', 'green'] | 'orange' | 3 | 3=2+1 |
| 3 | ['red', 'orange', 'yellow', 'green'] | 'yellow' | 3 | on to the next element in sequence |
| 4 | ['red', 'orange', 'yellow', 'green'] | 'yellow' | 3 | print 3 yellow |
| 5 | ['red', 'orange', 'yellow', 'green'] | 'yellow' | 4 | 4=3+1 |
| 3 | ['red', 'orange', 'yellow', 'green'] | 'green' | 4 | on to the last element in sequence |
| 4 | ['red', 'orange', 'yellow', 'green'] | 'green' | 4 | print 4 green |
| 5 | ['red', 'orange', 'yellow', 'green'] | 'green' | 5 | 5=4+1 |
| 3 | ['red', 'orange', 'yellow', 'green'] | 'green' | 5 | sequence done, end loop and code |

The final value of number is never used, but that is OK. What we want is printed.

This short example illustrates a lot of ideas:

- Loops may contain several variables.
- One way a variable can change is by being the variable in a for-loop heading, that automatically goes through the values in the for-loop list.
- Another way to have variables change in a loop is to have an explicit statement that changes the variable inside the loop, causing *successive modifications*.

There is a general pattern to loops with successive modification of a variable like `number` above:

(1) The variables to be modified need *initial* values *before* the loop (line 1 in the example above).
(2) The loop heading causes the repetition. In a for-loop, the number of repetitions is the same as the size of the list.
(3) The body of the loop generally "does something" (like print above in line 4) that you want done repeatedly.
(4) There is code inside the body of the loop to set up for the *next* time through the loop, where the variable which needs to change gets transformed to its next value (line 5 in the example above).

This information can be put in a code outline:

```
Initialize variables to be modified
Loop heading controlling the repetition
    Do the desired action with the current variables
    Modify variables to be ready for the action the next time
```

If you compare this pattern to the for-each and simple repeat loops in Section 1.13.4, you see that the examples there were simpler. There was no explicit variable modification needed to prepare for the next time though the loop. We will refer to the latest, more general pattern as a *successive modification* loop.

Functions are handy for encapsulating an idea for use and reuse in a program, and also for testing. We can write a function to number a list, and easily test it with different data. Read and run the example program `numberEntries4.py`:

```
def numberList(items):
    '''Print each item in a list items, numbered in order.'''
    number = 1
    for item in items:
        print(number, item)
        number = number + 1
```

```
def main():
    numberList(['red', 'orange', 'yellow', 'green'])
    print()
    numberList(['apples', 'pears', 'bananas'])


main()
```

Make sure you can follow the whole sequence, step by step! This program has the most complicated flow of control so far, changing both for function calls and loops.

(1) Execution start with the very last line, since the previous lines are definitions
(2) Then `main` starts executing.
(3) The first call to `numberList` effectively sets the formal parameter
`items = ['red', 'orange', 'yellow', 'green']`
and the function executes just like the flow followed in `numberEntries3.py`. This time, however, execution returns to main.
(4) An empty line is printed in the second line of main.
(5) The second call to `numberList` has a different actual parameter `['apples', 'pears', 'bananas']`, so this effectively sets the formal parameter this time
`items = ['apples', 'pears', 'bananas']`
and the function executes in a similar pattern as in `numberEntries3.py`, but with different data and one less time through the loop.
(6) Execution returns to main, but there is nothing more to do.

**1.13.7. Accumulation Loops.** Suppose you want to add up all the numbers in a list, `nums`. Let us plan this as a function from the beginning, so *read* the code below. We can start with:

```
def sumList(nums):
    '''Return the sum of the numbers in nums.'''
```

If you do not see what to do right away, a useful thing to do is write down a *concrete* case, and think how you would solve it, in complete detail. If `nums` is `[2, 6, 3, 8]`, you would likely calculate

2+6 is 8
8 + 3 is 11
11 + 8 is 19
19 is the answer to be returned.

Since the list may be arbitrarily long, you need a loop. Hence you must find a pattern so that you can keep reusing the *same statements* in the loop. Obviously you are using each number in the sequence in order. You also generate a sum in each step, which you reuse in the next step. The pattern is different, however, in the first line, 2+6 is 8: there is no previous sum, and you use two elements from the list. The 2 is not added to a previous sum.

Although it is not the shortest way to do the calculation *by hand*, 2 is a sum of 0 + 2: We can make the pattern consistent and calculate:

start with a sum of 0
0 + 2 is 2
2 + 6 is 8
8 + 3 is 11
11 + 8 is 19
19 is the answer.

Then the second part of each sum is a number from the list, `nums`. If we call the number from the list `num`, the main calculation line in the loop could be

```
nextSum = sum + num
```

The trick is to use the same line of code the next time through the loop. That means what was `nextSum` in one pass becomes the `sum` in the next pass. One way to handle that is:

```
sum = 0
for num in nums:
    nextSum = sum + num
    sum = nextSum
```

Do you see the pattern? Again it is

```
initialization
loop heading
    main work to be repeated
    preparation for the next time through the loop
```

Sometimes the two general loop steps can be combined. This is such a case. Since `nextSum` is only used once, we can just substitute its value (`sum`) where it is used and simplify to:

```
sum = 0
for num in nums:
    sum = sum + num
```

so the whole function, with the `return` statement is:

```
def sumList(nums):                                  #1
    '''Return the sum of the numbers in nums.'''
    sum = 0                                         #2
    for num in nums:                                #3
        sum = sum + num                             #4
    return sum                                      #5
```

With the following (not indented) line below used to test the function, you have the example program sumNums.py. Run it.

```
print(sumList([5, 2, 4, 7]))
```

The pattern used here is certainly successive modification (of the `sum` variable). It is useful to give a more specialized name for this version of the pattern here. It follows an *accumulation* pattern:

```
initialize the accumulation to include none of the sequence (sum = 0 here)
for item in sequence :
    new value of accumulation = result of combining item with last value of accumulation
```

This pattern will work in many other situations besides adding numbers.

Exercise 1.13.7.1. * Suppose the function `sumList`, is called with the parameter [5, 2, 4, 7]. Play computer on this call. Make sure there is a row in the table for each line executed in the program, each time it is executed. In each row enter which program line is being executed and show all changes caused to variables by the execution of the line. A table is started for you below. The final line of your table should be for line 5, with the comment, "return 18". If you do something like this longhand, and the same long value repeats a number of times, it is more convenient to put a ditto (") for each repeated variable value or even leave it blank. If you want to do it on a computer you can start from the first table in example file `playComputerSumStub.rtf`. First save the file as playComputerSum.rtf.

| Line | nums | sum | num | comment |
|---|---|---|---|---|
| 1 | [5, 2, 4, 7] | - | - | |
| 2 | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

EXERCISE 1.13.7.2. * Write a program `testSumList.py` which includes a `main` function to test the sumList function several times. Include a test for the extreme case, with an empty list.

EXERCISE 1.13.7.3. ** Complete the following function. This starting code is in `joinAllStub.py`. Save it to the *new* name `joinAll.py`. Note the way an example is given in the documentation string. It simulates the use of the function in the Shell. This is a common convention:

```
def joinStrings(stringList):
    '''Join all the strings in stringList into one string,
    and return the result. For example:

    >>> print(joinStrings(['very', 'hot', 'day']))
    'veryhotday'
    '''
```

Hint1: [7] Hint2: [8]

**1.13.8. More Playing Computer.** Testing code by running it is fine, but looking at the results does not mean you really understand what is going on, particularly if there is an error! People who do not understand what is happening are likely to make random changes to their code in an attempt to fix errors. This is a *very bad*, increasingly self-defeating practice, since you are likely to never learn where the real problem lies, and the same problem is likely to come back to bite you.

It is important to be able to predict accurately what code will do. We have illustrated playing computer on a variety of small chunks of code.

Playing computer can help you find bugs (errors in your code). Some errors are syntax errors caught by the interpreter in translation. Some errors are only caught by the interpreter during execution, like failing to have a value for a variable you use. Other errors are not caught by the interpreter at all – you just get the wrong answer. These are called *logical* errors. Earlier logical errors can also trigger an execution error later. This is when playing computer is particularly useful.

A common error in trying to write the `numberList` function would be to have:

```
def numberList(items):    # WRONG code for illustration!!!!    #1
    '''Print each item in a list items, numbered in order.'''  #2
    for item in items:                                         #3
        number = 1                                             #4
        print(number, item)                                    #5
        number = number + 1                                    #6
```

You can run this code in `numberEntriesWRONG.py` and see that it produces the wrong answer. If you play computer on the call to `numberList(['apples', 'pears', 'bananas'])`, you can see the problem:

[7]This is a form of accumulation, but not quite the same as adding numbers.

[8]"Start with nothing accumulated" does not mean 0, here. Think what is appropriate.

| Line | items | item | number | comment |
|---|---|---|---|---|
| 1 | ['apples', 'pears', 'bananas'] | - | - | pass actual parameter value to items |
| 3 | ['apples', 'pears', 'bananas'] | 'apples' | - | start with item as first in sequence |
| 4 | ['apples', 'pears', 'bananas'] | 'apples' | 1 | |
| 5 | ['apples', 'pears', 'bananas'] | 'apples' | 1 | print: 1 apples |
| 6 | ['apples', 'pears', 'bananas'] | 'apples' | 2 | 2 = 1+1 |
| 3 | ['apples', 'pears', 'bananas'] | 'pears' | 2 | on to the next element in sequence |
| 4 | ['apples', 'pears', 'bananas'] | 'apples' | 1 | |
| 5 | ['apples', 'pears', 'bananas'] | 'pears' | 1 | print: 1 pears *OOPS!* |

If you go step by step you should see where the incorrect 1 came from: the initialization is repeated each time in the loop at line 4, undoing the incrementing of `number` in line 6, messing up your count. Always be careful that your *one-time* initialization for a loop goes *before* the loop, not in it!

Functions can also return values. Consider the Python for this mathematical sequence: define the function m(x) = 5x, let y = 3; find m(y) + m(2y-1).

```
def m(x):                    #1
    return 5*x               #2

y = 3                        #3
print(m(y) + m(2*y-1))       #4
```

A similar example was considered in Section 1.11.6, but now add the idea of playing computer and recording the sequence in a table. Like when you simplify a mathematical expression, Python must complete the innermost parts first. Tracking the changes means following the function calls carefully and using the values returned. Again a dash '-' is used in the table to indicate an undefined variable. Not only are local variables like formal parameters undefined before they are first used, they are also undefined after the termination of the function,

| Line | x | y | comment |
|---|---|---|---|
| 3 | - | 3 | (definitions only before this line) |
| 4 | - | 3 | start on: print m(y) + m(2*y-1); find m(y), which is m(3) |
| 1 | 3 | 3 | pass 3 to function m, so x =3 |
| 2 | 3 | 3 | return 5*3 = 15 |
| 4 | - | 3 | substitute result: print 15 + m(2*y-1), find m(2*y-1), which is m(2*3-1) = m(5) |
| 1 | 5 | 3 | pass 5 to function m, so x=5 |
| 2 | 5 | 3 | return 5*5 = 25 |
| 4 | - | 3 | substitute result: print 15 + 25, so calculate and print 40 |

Thus far most of the code given has been motivated first, so you are likely to have an idea what to expect. You may need to read code written by someone else (or even yourself a while back!) where you are not sure what is intended. Also you might make a mistake and accidental write code that does something unintended! If you really understand how Python works, one line at a time, you should be able to play computer and follow at least short code sequences that have not been explained before. It is useful to read another person's code and try to follow it. The next exercises also provides code that has not been explained first:or has a mistake.

EXERCISE 1.13.8.1. ** Play computer on the following code. Reality check: 31 is printed when line 6 finally executes. Table headings are shown below to get you started with a pencil. Alternately you can work in a word processor starting from playComputerStub.rtf, which has tables set up for this and the following exercise. Save the file with an alternate name playComputer.rtf.

```
x = 0                  #1
y = 1                  #2
for n in [5, 4, 6]:    #3
    x = x + y*n        #4
    y = y + 1          #5

print(x)               #6
```

| Line | x | y | n | Comment |
|---|---|---|---|---|

EXERCISE 1.13.8.2. ** The following code is supposed to compute the product of the numbers in a list. For instance `product([5, 4, 6])` should calculate and return 5*4*6=120 in steps, calculating 5, 5*4=20 and 20*6=120 . Play computer on a call to `product([5, 4, 6])` until you see that it makes a mistake. This code appears in the example file `numProductWrong.py`. Save it as `numProduct.py` and fix the error (and save again!). Table headings and the first row are shown below to get you started with a pencil. Alternately you can work in a word processor continuing to add to playComputer.rtf, started in the previous exercise.

```
def product(nums):     #1
    for n in nums:     #2
        prod = 1       #3
        prod = prod*n  #4
    return prod        #5
```

| Line | nums | n | prod | Comment |
|---|---|---|---|---|
| 1 | [5, 4, 6] | - | - | |

EXERCISE 1.13.8.3. ** Play computer on the following code. Table headings are shown for you. Reality check: 70 is printed. See the previous exercises if you enter your answer in a file.

```
def f(x):                 #1
  return x+4              #2

print(f(3)*f(6))          #3
```

| Line | x | Comment |
|---|---|---|

**1.13.9. The `print` function `end` keyword.** By default the print function adds a newline to the end of the string being printed. this can be overridden by including the keyword parameter `end`. The keyword end can be set equal to any string. The most common replacements are the empty string or a single blank. If you also use the keyword parameter `sep`, these keyword paramters may be in either order, but they msut come at the end of the parmater list. Read the illustrations:

```
print('all', 'on', 'same', 'line')
print('different line')
```

is equivalent to

```
print('all', 'on' , end = ' ')
print('same', end = ' ')
print('line')
print('different line')
```

This does not work directly in the shell (where you are always forced to a new line at the end). It does work in a program, but it is not very useful except in a loop! Suppose I want to print a line with all the elements of a list, separated by spaces, but not on separate lines. I can use the `end` keyword set to a space in the loop. Can you figure out in your head what this example file `endSpace1.py` does? Then try it:

```
def listOnOneLine(items):
    for item in items:
         print(item, end=' ')

listOnOneLine(['apple', 'banana', 'pear'])
print('This may not be what you expected!')
```

If you still want to go on to a new line at the *end* of the loop, you must include a print function that does advance to the next line, once, *after* the loop. Try this variation, `endSpace2.py`