# Loops

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.

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, andwhile loops, which we take up later, in [While Statements](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/whilestatements.html#while-statements).

Two preliminaries:

1. The value of already defined variables can be updated. This will be particularly important in loops. To prepare for that we first follow how variables can be updated in an even simpler, sequential situation.
2. Sequence types are used in for loops. We will look at a basic sequence type: list.

Then we put this all together. This is a long section. Go slowly and carefully.

|  |  |
| --- | --- |
|  | It is possible with function recursion, but we will avoid that advanced topic in this introduction. |

## 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:

|  |  |
| --- | --- |
| 1  2  3  4  5 | x = 3 *# simple sequential code*  y = x + 2 *# updating two variables*  y = 2\*y  x = y - x  print(x, y) |

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 is 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 | 10 | 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 the proper order of execution, carefully, keeping track of the current values of variables, will be calledplaying computer. A table like the one above is an organized way to keep track.

## 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. (In an edit window the indentation is automatic). Be sure to enter another empty line. (Just press Enter.) at the end to get the Shell to respond. :

|  |  |
| --- | --- |
| 1  2  3 | **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, 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; may use **item**

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. A more detailed sequence is given, playing computer, in the table:

| **Line** | **count** | **comment** |
| --- | --- | --- |
| 1 | 1 | start with the first element of the list |
| 2 | 1 | print 1 |
| 3 | 1 | ‘yes’ \* 1 is ‘yes’; print yes |
| 1 | 2 | change count to the next element in the list |
| 2 | 2 | print 2 |
| 3 | 2 | ‘yes’ \* 2 is ‘yesyes’; print yesyes; |
| 1 | 3 | change count to the next element in the list |
| 2 | 3 | print 3 |
| 3 | 3 | ‘yes’ \* 3 is ‘yesyesyes’; print yesyesyes; done with list |

When executing step by step, note that the for loop heading serves two purposes:

* Each time the heading line executes, it implicitly assigns a new value to the variable name you use in place of **item**.
* After each execution of the heading line, the statements in the indented block are executed, generally making use of the the new value for the variable assigned in the heading.

**Note**

When playing computer with a loop, the same line numbers can reappear over and over, because the for loop heading line and the indented body under it are each executed repeatedly, and each time it is executed must be listed separately, in time sequence!

When you used the Shell to enter a loop, there was a reason that 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]:

print(count)

print('Yes' \* count)

print('Done counting.')

**for** color **in** ['red', 'blue', 'green']:

print(color)

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 the fourth line is indented:

**for** count **in** [1, 2, 3]:

print(count)

print('Yes' \* count)

print('Done counting.') *# changed so indented*

**for** color **in** ['red', 'blue', 'green']:

print(color)

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

Loops are one of the most important features in programming. While the for loop 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, and give them names. One of the simplest patterns is illustrated in all of the for loop examples so far, a simplefor-each loop: **For each** element of the sequence, do the same sort of thing with it. Stated as mor Pythonic pseudocode:

for **item** in sequence:

do something with the current **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*

Seeing this feature requires the ability to abstract the general pattern from the group fo examples. This is essential for using loops effectively.

If you wish to see or run the whole program with this small modification, see the example madlibloop.py. A common naming convention is used in the program: Each element in the list is a cue, while the list with all the elements is named with the plural cues. In later situations I make a list name be the plural of the variable name used for an individual item of the list.

Note the logic of the transformation between the two program versions: The alternative pieces of data are collected in the list in the for loop heading. A single variable name (here I chose cue) is used in the heading as a placeholder to refer to the current choice being handled, and the body refers to this variable cue in place of the explicit data values included each time in the original no-loop version.

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 (not indented).

In this example the data values are just a few given literals, and there is only one line in the repeated pattern. Hence the use of a for loop is not a big deal, but it makes a simple example! This looping construction would be much 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 [The Revised Mad Lib Program](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/madlib2.html#the-revised-mad), after we introduce more about string manipulation.

### Pattern Loop Exercise

Write a two-line for-each loop in a file types2.py containing a call to the type function that will have the same effect as this code in example file types1.py:

print(2, type(2))

print(3.5, type(3.5))

print([], type([]))

print(**True**, type(**True**))

print(**None**, type(**None**))

### Triple Exercise

Complete the following function. This starting code is in tripleStub.py. Save it to the new name triple.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** tripleAll(nums):

*''' print triple each of the numbers in the list nums.*

*>>> tripleAll([2, 4, 1, 5])*

*6*

*12*

*3*

*15*

*>>> tripleAll([-6])*

*-18*

*'''*

|  |  |
| --- | --- |
|  | The elements of the list in the for loop heading are not all of the same type. |

|  |  |
| --- | --- |
|  | You need to use the loop variable twice in the loop body. |

## Simple Repeat Loop

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 exactsame 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 programrepeat1.py:

*''' A simple repeat loop'''*

**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:

*'''The number of repetitions is specified by the user.'''*

n = int(input('Enter the number of times to repeat: '))

**for** i **in** range(n):

print('This is repetitious!')

## 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:

*'''In this version number does not change.'''*

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:

*'''prints poorly numbered entries from the list'''*

items = ['red', 'orange', 'yellow', 'green']

number = 1

**for** item **in** items:

print(number, item)

number = 2 *# will change to 2 after printing 1*

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:

|  |  |
| --- | --- |
| 1  2  3  4  5 | items = ['red', 'orange', 'yellow', 'green']  number = 1  **for** item **in** items: *# print numbered entries*  print(number, item)  number = number + 1 *# crucial added line* |

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.

Again note that the program line numbers in the Line column of the playing computer table are not all sequential, because the for loop heading line and the indented body under it are each executed repeatedly.

For compactness, the variable items does not get its own column, since it always has the value shown in the comment in line 1:

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

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

Go through carefully and be sure you understand the meaning of each entry in the table, and the reason for the sequencing and the reason for the exact position of each entry in each step where it changes! In particular see how and why the line number for each successive row is not always one more than the previous row. In particular, see how the same sequence of numbered lines may be repeated in multiple places in the table. Without this understanding you will not be able to play computer yourself and really understand loops.

This short example illustrates a lot of ideas important to loops:

* 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 [Basic for Loops](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/loops.html#basic-for-loops), 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:

*''' use a function to number the entries in any list'''*

**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
4. items = ['red', 'orange', 'yellow', 'green']

and the function executes just like the flow followed in numberEntries3.py. This time, however, execution returns to main.

1. An empty line is printed in the second line of main.
2. The second call to numberList has a different actual parameter ['apples', 'pears', 'bananas'], so this effectively sets the formal parameter this time
3. 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.

1. Execution returns to main, but there is nothing more to do.

## 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **def** sumList(nums):  *'''Return the sum of the numbers in the list nums.'''*  sum = 0  **for** num **in** nums:  sum = sum + num  **return** sum |

The example program sumNums.py has the code above with the following line added at the end to test the function (not indented). Run sumNums.py.

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.

### Play Computer Odd Loop Exercise

\* Work in a word processor (not Idle!), starting from example playComputerStub.rtf, and save the file as playComputer.rtf. The file has tables set up for this and the following two exercise.

Play computer on the following code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | x = 0 *# Exercise Play Computer Loop*  y = 1  **for** n **in** [5, 4, 6]:  x = x + y\*n  y = y + 1  print(x) |

Reality check: 31 is printed when line 6 finally executes. The start of the table for this exercise is shown below.

| **Line** | **x** | **y** | **n** | **Comment** |
| --- | --- | --- | --- | --- |
| 1 | 0 |  |  |  |

# While Statements

## Simple while Loops

Other than the trick with using a return statement inside of a for loop, all of the loops so far have gone all the way through a specified list. In any case the for loop has required the use of a specific list. This is often too restrictive. A Python while loop behaves quite similarly to common English usage. If I say

While your tea is too hot, add a chip of ice.

Presumably you would test your tea. If it were too hot, you would add a little ice. If you test again and it is still too hot, you would add ice again. As long as you tested and found it was true that your tea was too hot, you would go back and add more ice. Python has a similar syntax:

while condition :

indentedBlock

Setting up the English example in a similar format would be:

while your tea is too hot :

add a chip of ice

To make things concrete and numerical, suppose the following: The tea starts at 115 degrees Fahrenheit. You want it at 112 degrees. A chip of ice turns out to lower the temperature one degree each time. You test the temperature each time, and also print out the temperature before reducing the temperature. In Python you could write and run the code below, saved in example program cool.py:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | temperature = 115  **while** temperature > 112: *# first while loop code*  print(temperature)  temperature = temperature - 1  print('The tea is cool enough.') |

I added a final line after the while loop to remind you that execution follows sequentially after a loop completes.

If you play computer and follow the path of execution, you could generate the following table. Remember, that each time you reach the end of the indented block after the while heading, execution returns to the while heading for another test:

| **Line** | **temperature** | **Comment** |
| --- | --- | --- |
| 1 | 115 |  |
| 2 |  | 115 > 112 is true, do loop |
| 3 |  | prints 115 |
| 4 | 114 | 115 - 1 is 114, loop back |
| 2 |  | 114 > 112 is true, do loop |
| 3 |  | prints 114 |
| 4 | 113 | 114 - 1 is 113, loop back |
| 2 |  | 113 > 112 is true, do loop |
| 3 |  | prints 113 |
| 4 | 112 | 113 - 1 is 112, loop back |
| 2 |  | 112 > 112 is false, skip loop |
| 6 |  | prints that the tea is cool |

Each time the end of the indented loop body is reached, execution returns to the while loop heading for another test. When the test is finally false, execution jumps past the indented body of the while loop to the next sequential statement.

A while loop generally follows the pattern of the successive modification loop introduced with for-each loops:

initialization

while continuationCondition :

do main action to be repeated

prepare variables for the next time through the loop

Test yourself: Following the code. Figure out what is printed. :

i = 4

**while** i < 9:

print(i)

i = i+2

Check yourself by running the example program testWhile.py.

**Note**

In Python, while is not used quite like in English. In English you could mean to stop as soon as the condition you want to test becomes false. In Python the test is only made when execution for the loop starts, not in the middle of the loop.

Predict what will happen with this slight variation on the previous example, switching the order in the loop body. Follow it carefully, one step at a time.

|  |  |
| --- | --- |
| 1  2  3  4 | i = 4 *# variation on testWhile.py*  **while** (i < 9):  i = i+2  print(i) |

Check yourself by running the example program testWhile2.py.

The sequence order is important. The variable i is increased before it is printed, so the first number printed is 6. Another common error is to assume that 10 will not be printed, since 10 is past 9, but the test that may stop the loop is not made in the middle of the loop. Once the body of the loop is started, it continues to the end, even when i becomes 10.

| **Line** | **i** | **Comment** |
| --- | --- | --- |
| 1 | 4 |  |
| 2 |  | 4 < 9 is true, do loop |
| 3 | 6 | 4+2=6 |
| 4 |  | print 6 |
| 2 |  | 6 < 9 is true, do loop |
| 3 | 8 | 6+2= 8 |
| 4 |  | print 8 |
| 2 |  | 8 < 9 is true, do loop |
| 3 | 10 | 8+2=10 No test here |
| 4 |  | print 10 |
| 2 |  | 10 < 9 is false, skip loop |

Predict what happens in this related little program:

nums = list()

i = 4

**while** (i < 9):

nums.append(i)

i = i+2

print(nums)