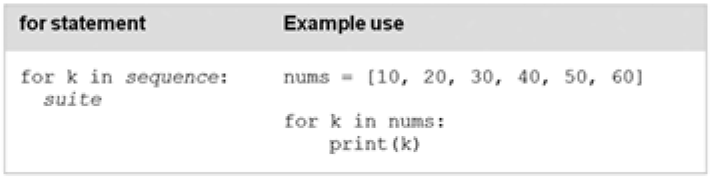
Iterating Over Lists (Sequences)

Python's for statement provides a convenient means of iterating over lists (and other sequences). In this section, you will explore both `for` loops and `while` loops for list iteration.

For Loops

A **for statement** is an iterative control statement that iterates once for each element in a specified sequence of elements. Thus, for loops are used to construct definite loops. For example, a for loop is given in Figure 4-9 that prints out the values of a specific list of integers.



for statement	Example use

`for k in` *`sequence`*`:`
  `suite`
	`nums = [10, 20, 30, 40, 50, 60]`
	
	`for k in nums:`
	    `print(k)`

Variable `k` is referred to as a **loop variable**. Since there are six elements in the provided list, the for loop iterates exactly six times. To contrast the use of for loops and while loops for list iteration, the same iteration is provided as a while loop below,

```
k = 0
while k < len(nums):
    print(nums[k])
    k = k + 1
```

In the while loop version, loop variable `k` must be initialized to `0` and incremented by 1 each time through the loop. In the for loop version, loop variable `k` *automatically* iterates over the provided sequence of values.

The for statement can be applied to all sequence types, including strings. Thus, iteration over a string can be done as follows (which prints each letter on a separate line).

```
for ch in 'Hello':

     print(ch)
```

Next you will look at the use of the built-in `range` function with for loops.

Your Turn

From the Python Shell, enter the following and observe the results.

```
>>> for k in [4, 2, 3, 1]      >>> for k in [‘Pie’,‘Cake’,‘Pear’]
>>> print k                    >>> print k
???                            ???

>>> for k in [4, 2, 3]         >>> for k in [‘Pear’]
>>> print k                    >>> print k
???                            ???
```

Part II - The Built-in `range` Function

Python provides a built-in **range function** that can be used for generating a sequence of integers that a for loop can iterate over, as shown below.

```
sum = 0
for k in range(1, 11):
sum =  sum 1 k
```

The values in the generated sequence include the starting value, up to *but not including* the ending value. For example, `range(1, 11)` generates the sequence `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].` Thus, this for loop adds up the integer values 1–10.

The `range` function is convenient when long sequences of integers are needed. Actually, `range` does not create a sequence of integers. It creates a *generator function* able to produce each next item of the sequence when needed. This saves memory, especially for long lists. Therefore, typing `range(0, 9)` in the Python shell does not produce a list as expected—it simply “echoes out” the call to `range.`

By default, the `range` function generates a sequence of consecutive integers. A “step” value can be provided, however. For example, `range(0, 11, 2)` produces the sequence `[0, 2, 4, 6, 8, 10],` with a step value of 2. A sequence can also be generated “backwards” when given a negative step value. For example, `range(10, 0, 21)` produces the sequence `[10, 9, 8, 7, 6, 5, 4, 3, 2, 1].` Note that since the generated sequence always begins with the provided starting value, "up to" but not including the final value, the final value here is 0, and not 1.

Your Turn

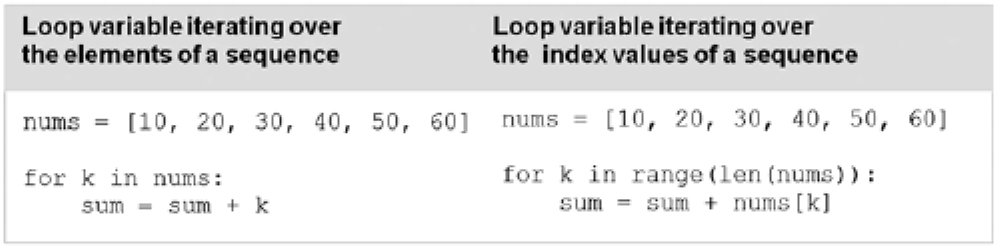
From the Python Shell, enter the following and observe the results.

```
>>> for k in range (0,11):          >>> for k in range (2, 102, 2):
        print k                               print k
???                                 ???

>>> for k in range [0,11]:          >>> for k in range (10, -1, -2):
>>> print k                         >>> print k
???                                ???
```

Part III - Iterating Over List Elements vs. List Index Values

When the elements of a list need to be accessed, but not altered, a loop variable that iterates over each list element is an appropriate approach. However, there are times when the loop variable must iterate over the index values of a list instead. A comparison of the two approaches is shown below:



Loop variable iterating over the elements of a sequence	Loop variable iterating over the index values of a sequence
`nums = [10, 20, 30, 40, 50, 60]`	`nums = [10, 20, 30, 40, 50, 60]`

`for k in nums:`
`    sum = sum + k`
	`for k in range(len(nums)):`
	`    sum = sum + nums[k]`

Suppose the average of a list of class grades named grades needs to be computed. In this case, a for loop can be constructed to iterate over the grades,

```
for k in grades: sum =  sum + k
print('Class average is', sum/len(grades))
```

However, suppose that the instructor made a mistake in grading, and a point needed to be added to each student's grade? In order to accomplish this, the index value (the location) of each element must be used to update each grade value. Thus, the loop variable of the for loop must iterate over the index values of the list,

```
    for k in range(len(grades)): grades[k] = grades[k] + 1
```

In such cases, the loop variable k is also functioning as an *index variable*. An **index variable** is a variable whose *changing value is used to access elements of an indexed data structure*. Note that the range function may be given only one argument. In that case, the starting value of the range defaults to 0. Thus, `range(len(grades))` is equivalent to `range(0, len(grades))`.

Your Turn

From the Python Shell, enter the following and observe the results.

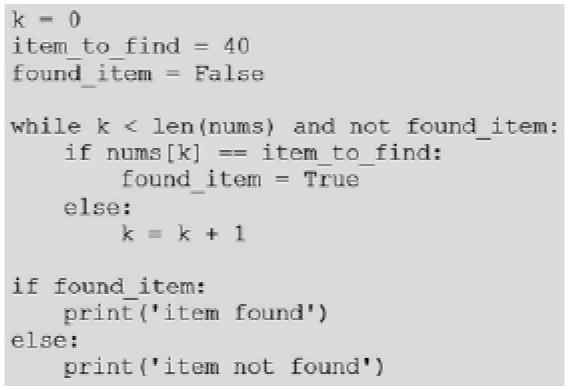
```
>>> nums = [10, 20, 30]
>>> for k in range (len(nums)):
    print (nums[k])
  ???

>>> for k in range (len(nums)-1, -1, -1):
>>> print (nums [k])
???
```

Part IV - While Loops and Lists (Sequences)

There are situations in which a sequence is to be traversed while a given condition is true. In such cases, a while loop is the appropriate control structure. (Another approach for the partial traversal of a sequence is by use of a for loop containing `break` statements. We avoid the use of `break` statements in this text, favoring the more structured while loop approach.)

Let's say that we need to determine whether the value `40` occurs in list `nums` (equal to [10, 20, 30]). In this case, once the value is found, the traversal of the list is terminated. An example of this is given below:



```
k = 0
item_to_find = 40
found_item = False

while k < len(nums) and not found_item:
    if nums[k] == item_to_find:
        found_item = True
    else:
        k = k + 1

if found_item:
    print('item found')
else:
    print('item not found')
```

Variable `k` is initialized to 0, and used as an index variable. Thus, the first time through the loop, `k` is `0`, and `nums[0]` (with the value 10) is compared to `item_to_find.` Since they are not equal, the second clause of the if statement is executed, incrementing `k` to `1`. The loop continues until either the item is found, or the complete list has been traversed. The final if statement determines which of the two possibilities for ending the loop occurred, displaying either `'item found'` or `'item not found'`. Finally, note that the correct loop condition is k , `len(nums)`, and not `k` <= `len(nums)`. Otherwise, an “index out of range” error would result.

Your Turn

Enter and execute the following Python code, and observe the results.

```
k = 0
sum = 0
nums = range (100)

while k < len(nums) and sum < 100:
sum = sum + nums[k]
k = k + 1

print (‘The first’, k ‘integers sum to 100 or greater’)
```

Concepts and Procedures

1. For `nums = [10,30,20,40]`, what does the following for loop output?

```
for k in
   nums:
   print(k)
```

2. For `nums = [10, 30, 20, 40]`, what does the following for loop output?

```
for k in range(1, 4):
   print(nums[k])
```

3. For `fruit = 'strawberry'`, what does the following for loop output?

```
for k in range(0, len(fruit), 2):
   print(fruit[k], end='')
```

4. For `nums = [12, 4, 11, 23, 18, 41, 27]`, what is the value of `k` when the while loop terminates?

```
k = 0
while k < len(nums) and nums[k] != 18:
     k 5 k 1 1
```

Problem Solving

1. For a list of integers named `nums`,
   a) Write a while loop that adds up all the values in `nums`.\
   b) Write a for loop that adds up all the values in `nums` in which the loop variable is assigned each value in the list.
   c) Write a for loop that adds up all the elements in `nums` in which the loop variable is assigned to the index value of each element in the list.
   d) Write a for loop that displays the elements in `nums` backwards.

e) Write a for loop that displays every other element in `nums`, starting with the first element.

2. Write a Python program that prompts the user for a list of integers, stores in another list only those values that are in tuple `valid_values`, and displays the resulting list.

3. Write a Python program that prompts the user for a list of integers and stores them in a list. For all values that are greater than `100`, the string `'over'` should be stored instead. The program should display the resulting list.