1.10. Control Structures

As we noted earlier, algorithms require two important control structures: iteration and selection. Both of these are supported by Python in various forms. The programmer can choose the statement that is most useful for the given circumstance.

For iteration, Python provides a standard `while` statement and a very powerful `for` statement. The while statement repeats a body of code as long as a condition is true. For example,

```
>>> counter = 1
>>> while counter <= 5:
...     print("Hello, world")
...     counter = counter + 1


Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
```

prints out the phrase “Hello, world” five times. The condition on the `while` statement is evaluated at the start of each repetition. If the condition is `True`, the body of the statement will execute. It is easy to see the structure of a Python `while` statement due to the mandatory indentation pattern that the language enforces.

The `while` statement is a very general purpose iterative structure that we will use in a number of different algorithms. In many cases, a compound condition will control the iteration. A fragment such as

```
while counter <= 10 and not done:
...
```

would cause the body of the statement to be executed only in the case where both parts of the condition are satisfied. The value of the variable `counter` would need to be less than or equal to 10 and the value of the variable `done` would need to be `False` (`not False` is `True`) so that `True and True` results in `True`.

Even though this type of construct is very useful in a wide variety of situations, another iterative structure, the `for` statement, can be used in conjunction with many of the Python collections. The `for` statement can be used to iterate over the members of a collection, so long as the collection is a sequence. So, for example,

ndOutput.html)

```
>>> for item in [1,3,6,2,5]:
...     print(item)
...
1
3
6
2
5
```

assigns the variable `item` to be each successive value in the list [1,3,6,2,5]. The body of the iteration is then executed. This works for any collection that is a sequence (lists, tuples, and strings).

A common use of the `for` statement is to implement definite iteration over a range of values. The statement

```
>>> for item in range(5):
...     print(item**2)
...
0
1
4
9
16
>>>
```

will perform the `print` function five times. The `range` function will return a range object representing the sequence 0,1,2,3,4 and each value will be assigned to the variable `item` . This value is then squared and printed.

The other very useful version of this iteration structure is used to process each character of a string. The following code fragment iterates over a list of strings and for each string processes each character by appending it to a list. The result is a list of all the letters in all of the words.

Run Show in CodeLens

```
1 wordlist = ['cat','dog','rabbit']
2 letterlist = [ ]
3 for aword in wordlist:
4     for aletter in aword:
5         letterlist.append(aletter)
6 print(letterlist)
7
```

```
['c', 'a', 't', 'd', 'o', 'g', 'r', 'a', 'b', 'b', 'i', 't']
```

ndOutput.html)

ActiveCode: 1 Processing Each Character in a List of Strings (intro_8)

Selection statements allow programmers to ask questions and then, based on the result, perform different actions. Most programming languages provide two versions of this useful construct: the `ifelse` and the `if`. A simple example of a binary selection uses the `ifelse` statement.

```
if n<0:
   print("Sorry, value is negative")
else:
   print(math.sqrt(n))
```

In this example, the object referred to by `n` is checked to see if it is less than zero. If it is, a message is printed stating that it is negative. If it is not, the statement performs the `else` clause and computes the square root.

Selection constructs, as with any control construct, can be nested so that the result of one question helps decide whether to ask the next. For example, assume that `score` is a variable holding a reference to a score for a computer science test.

```
if score >= 90:
   print('A')
else:
   if score >=80:
      print('B')
   else:
      if score >= 70:
         print('C')
      else:
         if score >= 60:
            print('D')
         else:
            print('F')
```

This fragment will classify a value called `score` by printing the letter grade earned. If the score is greater than or equal to 90, the statement will print `A`. If it is not (`else`), the next question is asked. If the score is greater than or equal to 80 then it must be between 80 and 89 since the answer to the first question was false. In this case print `B` is printed. You can see that the Python indentation pattern helps to make sense of the association between `if` and `else` without requiring any additional syntactic elements.

An alternative syntax for this type of nested selection uses the `elif` keyword. The `else` and the next `if` are combined so as to eliminate the need for additional nesting levels. Note that the final `else` is still necessary to provide the default case if all other conditions fail.

ıdOutput.html)

```
if score >= 90:
   print('A')
elif score >=80:
   print('B')
elif score >= 70:
   print('C')
elif score >= 60:
   print('D')
else:
   print('F')
```

Python also has a single way selection construct, the `if` statement. With this statement, if the condition is true, an action is performed. In the case where the condition is false, processing simply continues on to the next statement after the `if` . For example, the following fragment will first check to see if the value of a variable `n` is negative. If it is, then it is modified by the absolute value function. Regardless, the next action is to compute the square root.

```
if n<0:
   n = abs(n)
print(math.sqrt(n))
```

Self Check

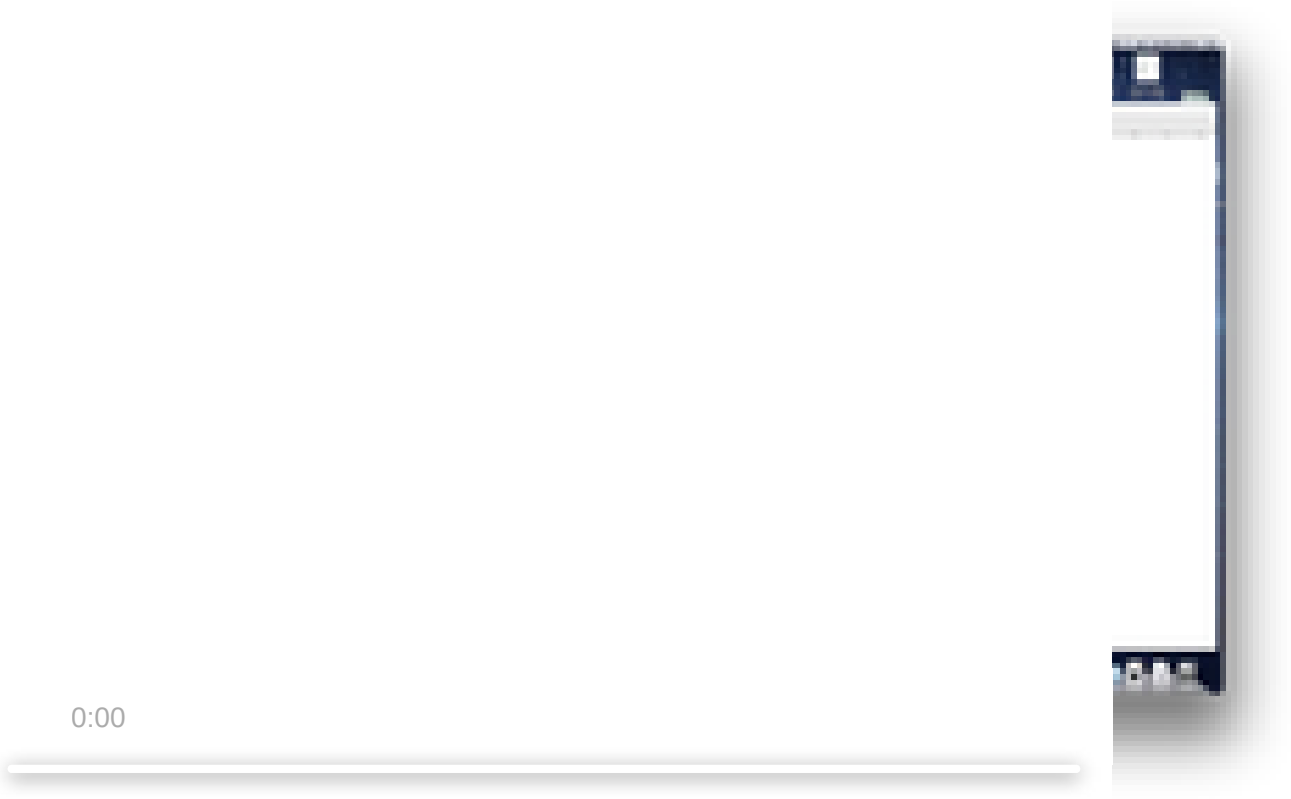
Test your understanding of what we have covered so far by trying the following exercise. Modify the code from Activecode 8 so that the final list only contains a single copy of each letter.

Run Show in CodeLens

```
1 # the answer is: ['c', 'a', 't', 'd', 'o', 'g', 'r', 'b', 'i
2
```

ActiveCode: 2 (self_check_1)

ndOutput.html)



Returning to lists, there is an alternative method for creating a list that uses iteration and selection constructs known as a **list comprehension**. A list comprehension allows you to easily create a list based on some processing or selection criteria. For example, if we would like to create a list of the first 10 perfect squares, we could use a `for` statement:

```
>>> sqlist=[]
>>> for x in range(1,11):
        sqlist.append(x*x)

>>> sqlist
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>>
```

Using a list comprehension, we can do this in one step as

```
>>> sqlist=[x*x for x in range(1,11)]
>>> sqlist
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>>
```

The variable `x` takes on the values 1 through 10 as specified by the `for` construct. The value of `x*x` is then computed and added to the list that is being constructed. The general syntax for a list comprehension also allows a selection criteria to be added so that only certain items get added. For example,

ndOutput.html)

```
>>> sqlist=[x*x for x in range(1,11) if x%2 != 0]
>>> sqlist
[1, 9, 25, 49, 81]
>>>
```

This list comprehension constructed a list that only contained the squares of the odd numbers in the range from 1 to 10. Any sequence that supports iteration can be used within a list comprehension to construct a new list.

```
>>>[ch.upper() for ch in 'comprehension' if ch not in 'aeiou']
['C', 'M', 'P', 'R', 'H', 'N', 'S', 'N']
>>>
```

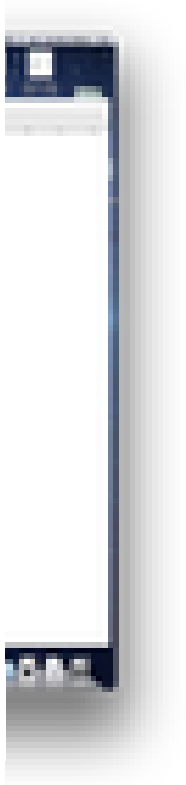
Self Check

Test your understanding of list comprehensions by redoing Activecode 8 using list comprehensions. For an extra challence, see if you can figure out how to remove the duplicates.

Run Load History Show CodeLens

```
1 # the answer is: ['c', 'a', 't', 'd', 'o', 'g', 'r', 'a', 'b
2
```

ActiveCode: 3 (self_check_2)



0:00

ndOutput.html)

(InputandOutput.html)

(ExceptionHandling.html)

 | Back to top

ıdOutput.html)