

Lesson 18, While Statements

In the previous lesson we learned about more complex if statements. In this lesson we discuss while statements. In the next lesson we will incorporate our newly discovered conditional expressions into our menu project.

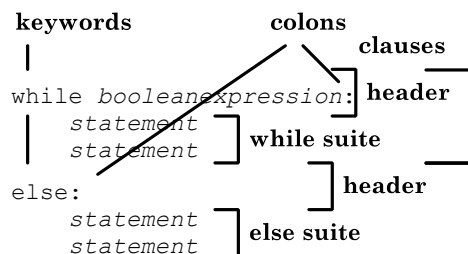
Discussion

We now know everything there is to know about if statements, including the use of optional elif and else clauses, as well as nesting if statements within each other. But still, our programs have merely run from top to bottom, like water flowing downhill. The if statement added the ability to choose which path the water took, but sometimes we need to be able to jump back higher up the hill to be able to do some process again and again.

Take as an example our sine calculator menu project we have been working on. Each time we want to test a new value we have to start the program all over again. Yet most programs you use don't work that way. Software is best when it helps make the user experience more comfortable; that is the point of having nice windowed programs and operating systems. A step in that direction would be for our project to continue to prompt for new angles after processing the previous value, until we tell it to stop. As another example, your washing machine doesn't have to be unplugged and plugged back in to get it to run another cycle. No, the firmware sits there waiting patiently until you command it to wash a load of clothes a certain way, and then afterward it goes back to waiting patiently for a new command.

So we need a way to get the software to go back to some earlier point in the program and start working again. To handle this requirement, there is another compound conditional statement which is as equally powerful as the if statement, and that is the while statement. You will find that both if statements and while statements are necessary for any program you write which is not trivially simple. Read the short article about the while statement in topic 7.2 *The while statement* under the *Language Reference* folder in the *Python Manuals* help file. As described in that topic, a **while statement** is a compound statement which allows looping back to the top of the suite of controlled statements as long as the boolean expression in the header of the while clause evaluates to `True`.

Consider the structure of the while statement shown to the right. Since the while statement is a compound statement, some of its structure will be familiar to you right away after reading that topic. All while statements must have the while clause, identified by the keyword `while`. The header of the while clause contains a boolean expression. The suite of statements controlled by the while clause is executed, over and over, as long as the boolean expression evaluates to `True`.



The first time the boolean expression evaluates to `False`, the optional else clause, if present, is executed, after which the while statement terminates. If the boolean expression evaluates to `False` the first time it is encountered, the suite of statements controlled by the while clause will not execute at all. And of course, the headers of both the while clause and the else clause, as part of compound statements, end with the now-familiar colon.

A while statement is a specific example of what is known as *looping code*. In software, a **loop** is code which jumps back to a particular location to execute over and over. This term is more informal, but helps to describe various looping situations where the precise definition of the statements involved are irrelevant to the discussion or easily determined from examination of the source code.

Two other statements, the `break` statement and the `continue` statement, help us fine-tune the use

of while statements, as we shall see in a, uhh, while. A while statement, that is. Read topic 6.10 *The break statement*, and 6.11 *The continue statement*, in the *Language Reference* folder in the *Python Manuals* help file. As always, don't worry if you don't understand the details in the help file yet, we will explain the essentials in this text.

A **break statement** is a simple statement (i.e., not compound) which, when placed within the while clause of a while statement, immediately stops processing of statements in the suite, and resumes processing of statements beyond the while statement without executing the suite of statements controlled by the else clause. That is a long worded precise way of saying that a break statement immediately jumps out of the while statement entirely. A break statement consists of the single word `break`. Similarly, a **continue statement** is a simple statement which, when placed within the while clause of a while statement, immediately stops processing of statements in the suite and returns control to the header of the while clause for the next evaluation of the boolean expression. A continue statement consists of the single word `continue`.

The distinction between the break statement and the continue statement is that a break terminates the entire while statement. But, the continue statement terminates only the current iteration of the while, and allows the boolean expression to decide whether to stop entirely. Programmers might say "break out of this loop" or "continue the loop", which helps to clarify the respective uses of these statements. Also note that should while statements be nested, these simple statements only affect the lowest-level while statement containing them.

Now let's use our knowledge of the while statement to improve our project. We'll also take advantage of some of the string operations we encountered on the way. Open our `MyMenu.py` project file from the console (be sure to be in the `C:\DevPython\Project1` folder first) and make the following changes.

```
import math

def Menu():
    strIn=raw_input('Enter angle in degrees>')
    while 0 != len(strIn):
        fDegrees=float(strIn)
        fSine=math.sin( math.radians( fDegrees ) )
        print 'sin(', fDegrees, ') =', fSine

        # The following code is for testing conditional logic
        if fDegrees < 0:
            print 'Negative degrees'
        elif fDegrees >= 360.0:
            print 'Circled'
        elif fDegrees >= 90.0:
            print 'Big angle'
        else:
            pass

    strIn=raw_input('Enter angle in degrees>')
```

Menu()

Make sure that you make all three of the following changes:

- Add the while statement as shown.
- Indent all the pre-existing italicized code one level (four spaces). This is the most error-prone change, so be sure to update the indentation exactly.
- Add the second `raw_input()` statement as shown.

Let's look at the boolean expression in the while statement for a moment. Note that it makes use

of the string length operation. As you will recall, this operation allows us to know whether our string has any data in it. This metadata about our string allows us to decide when to stop processing our loop. If the console user is done entering data, then all he has to do is hit `Enter` without typing anything and the `while` terminates. Note that we also had to go out and get a new console entry from the user when we were done processing the previous value entered by the user.

Save your work and execute this project from the console. Now it is relatively easy to whip through all seven test cases:

```
C:\DevPython\Project1>C:\Python25\python MyMenu.py
Enter angle in degrees>-21.7
sin( -21.7 ) = -0.369746757274
Negative degrees
Enter angle in degrees>0
sin( 0.0 ) = 0.0
Enter angle in degrees>48.5
sin( 48.5 ) = 0.748955720789
Enter angle in degrees>90
sin( 90.0 ) = 1.0
Big angle
Enter angle in degrees>128.6
sin( 128.6 ) = 0.781520472419
Big angle
Enter angle in degrees>360
sin( 360.0 ) = -2.44921270764e-016
Circled
Enter angle in degrees>528.4
sin( 528.4 ) = 0.201077921146
Circled
Enter angle in degrees>
C:\DevPython\Project1>
```

Wow! Congratulations, you just wrote your first looping console program! This is starting to look like a real console application which someone might use. And yet each portion of the code is simple to understand once you have the right foundation which has already been presented in this course. For the next several lessons we are going to add enhancements to the skeleton of this project as we learn more details about Python, but all the framework is now in place to do so.

Let's exercise the `else` clause by adding one to the `while`. As you recall, the `else` clause only executes when the boolean expression evaluates to `False`, and just before the `while` statement terminates. Add the following code.

```
def Menu():
    strIn=raw_input('Enter angle in degrees>')
    while 0 != len(strIn):
        fDegrees=float(strIn)
        ... other statements

        strIn=raw_input('Enter angle in degrees>')

    else:
        print 'Bye!'

Menu()
```

Note that the `else` clause is dedented to be at exactly the same indentation level as the `while` clause. Save your work and test the program:

```
C:\DevPython\Project1>C:\Python25\python MyMenu.py←
Enter angle in degrees>28.9←
sin( 28.9 ) = 0.483282383255
Enter angle in degrees>←
Bye!

C:\DevPython\Project1>
```

Confusing Dedentation

If you look at this version of the code and compare it to that from the previous changes, you will see that the else clause helps make a nice transition back to the indentation level containing the call to the Menu() function.

```
        strIn=raw_input('Enter angle in degrees>')
    else:
        print 'Bye!'

Menu()
```

Without the else clause, there is a disconcerting jump back two levels.

```
        strIn=raw_input('Enter angle in degrees>')

Menu()
```

It is for this reason that sometimes Python programmers may include a dummy else statement just to help document the indentation level, particularly if the while or if statement in question continues for many pages and many levels of indentation:

```
        strIn=raw_input('Enter angle in degrees>')
    else: pass

Menu()
```

As this example shows, you can put simple statements right after the colon of a clause if you want, which means we could have just as easily placed the print statement after the colon, too:

```
    else: print 'Program terminated'
```

In general, I highly recommend placing the suite of statements on indented lines which follow the header of a compound statement clause, even if it seems as if only one statement will ever be required. Often, months later a new requirement causes a single simple statement in a suite to balloon into many statements. Months later, the maintenance programmer, and many times this means you, will have to move that single statement back down to the next line and indent accordingly anyway. Anticipating this, avoid placing simple statements after the colon, except for possibly the pass statement.

Another way to help document the current indentation level is to place a comment at the end of the entire statement to show the termination.

For example, add the following comments:

```
    else:
        pass
    #end if
```

```

    strIn=raw_input('Enter angle in degrees>')

else:
    print 'Bye!'
#end while

```

Menu()

As always, save and test your work after making these changes. Now even if the final else clause weren't present it is easy to see that these comments greatly increase the ability of the maintenance programmer to see where the previous indentation level resides. I have found that taking the time to document the little details, and testing your code often as we just did, pays many dividends later.

One way to help automate this process is to type the terminating comment as you start the compound statement. For example (don't modify your project) we might begin a while statement as follows:

```

while :
#end while

```

Then, fill in the boolean expression. Finally, add blank lines between the statement header and the terminating content and add the controlled suite of statements and any additional clauses, as appropriate. We will practice this technique in future lessons.

Breaking the Loop

Now let's use the break statement to abnormally terminate execution of a while loop. As you will recall, a break statement immediately stops the loop without allowing the else clause to execute. Add a break statement before the pass statement in the else clause of the if statement:

```

else:
    break
    pass
#end if

```

Now when we enter a small angle the loop should terminate without printing the 'Bye!' contained in the else clause. Try a normal termination, then restart and try an abnormal termination:

```

C:\DevPython\Project1>C:\Python25\python MyMenu.py
Enter angle in degrees>-20
sin( -20.0 ) = -0.342020143326
Negative degrees
Enter angle in degrees>
Bye!

```

```

C:\DevPython\Project1>C:\Python25\python MyMenu.py
Enter angle in degrees>95.4
sin( 95.4 ) = 0.995561964603
Big angle
Enter angle in degrees>20
sin( 20.0 ) = 0.342020143326

```

```

C:\DevPython\Project1>

```

As expected, the small angle triggered the break statement, which terminated execution without passing through the else clause first. Before proceeding further, remove the break statement as shown below, and save your work.

```

else:
    break
    pass
#end if

```

Removing Repeated Code

For what it is, right now our project works fine. Yet there is one nagging detail which we need to resolve, and that is the topic of repeated code. When we added the while loop, we had to add another call to `raw_input()` at the bottom of the loop. Right now, both of these calls perform exactly the same function. But, what happens later during maintenance when the prompt needs to change? Then, two lines of code, by then perhaps pages apart, have to be changed identically.

Given this, there is a non-zero probability that one of those statements may get missed when it would be time to make this change. Or one may get changed incorrectly. Yes, we could test the code after the change, but realistically speaking such tests may get shortchanged by only being run once, or not being run at all. Worse, if one of these calls gets overlooked in one maintenance cycle, the next time the maintenance programmer might assume there is a reason why the prompts are different, and leave it that way forever. Source code has a tendency to decay over time unless proper attention is given to details that lead to rot.

Fortunately, many such rot-prone issues have simple solutions which are easy to do if applied soon enough. In the case of repeated code, the first thought which should come to mind is to add a function. So, let's add a function to handle the prompting and user input near the top of the module:

```

import math

def PromptUserInput():
# Prompts the console user, and returns the string of typed text
    strIn=raw_input('Enter angle in degrees>')
    return strIn

def Menu():

```

Note that we now place the prompt string inside this new function. Also note that we added a comment to explain to a maintenance programmer what this function is supposed to do. Now replace the calls to `raw_input()` with calls to this function, removing the old prompts:

```

def Menu():
    strIn=raw_input('Enter angle in degrees>')
    strIn = PromptUserInput()
    while 0 != len(strIn):
        ... other code lines
    strIn=raw_input('Enter angle in degrees>')
    strIn = PromptUserInput()

else:

```

Save your work and test from the console. The project should behave exactly as before:

```

C:\DevPython\Project1>C:\Python25\python MyMenu.py←
Enter angle in degrees>28.9←
sin( 28.9 ) = 0.483282383255
Enter angle in degrees>
Bye!

```

```

C:\DevPython\Project1>

```

Now if we ever have to change the prompt, we only have to do it in one place. By the way, there is a term for what we have done, and that is refactoring. When we **refactor** a project, we make structural changes to improve maintenance or to prepare for new features, without changing the capability of the program. Having good refactoring skills assists with writing good code. If you know how to change code with problems to make it better, then you are more likely to write better code from the beginning.

Vocabulary Terms

break statement
continue statement

loop
refactor

while statement

Practice Exercises

A. List the two ways for a while loop to terminate. Which of these allows the else clause to execute?

B. Consider the following while statement. Predict how many times 'did it' will print to the console. Check your prediction with the

interactive Python console.

```
x = 3
while x > 0:
    print 'did it'
    x = x - 1
```

Exercises

Complete the following exercises as homework on separate paper. Do not write in this book!

1. Bob thinks that his source code is too messy to leave to a maintenance programmer to add features. He should _____ it.

2. A _____ statement is a compound statement which allows looping back to the top of the suite of controlled statements as long as the boolean expression in its header evaluates to True.

3. In software terms, a _____ is a portion of code which jumps back to a specific location to execute over and over.

4. A _____ statement abnormally terminates a while statement, and skips over the else clause, if one exists.

5. A _____ statement causes an immediate re-evaluation of the boolean expression at the top of a while statement to see if processing the statement should resume.

6. Consider the following while statement. How many times will 'yep' print to the console? Confirm your answer with the Python console.

```
t = 4
while t < 6:
    print 'yep'
    t = t + 1
```

7. How many times would 'yep' print if the boolean expression in Exercise #6 was changed to `t <= 6`?

8. A _____ statement is a do-nothing statement which acts as a placeholder where a statement is syntactically required.

9. _____ testing is a source code test method which involves testing each branch of code execution, as well as the boundary conditions between each branch.

10. An _____ clause is used in the interior of an if statement to optionally execute a suite of statements based on a new boolean expression in the header of the clause.

11. An _____ clause identifies statements to be executed when the boolean expression of the associated if clause, and any preceding elif clauses, evaluate to False.

12. What will print on the console when the following code executes?

```
x = 3
if x <= 4:
    print 'Fee'
elif x < 8:
    print 'Fi'
elif x <= 12:
    print 'Foe'
else:
    print 'Fum'
```

13. What prints to the console when the following code executes?

```
a = 4
b = 6.3
c = False

if a < 3:
    print 'pig'
elif not c:
    print 'wig'
elif c and (b > 5.1):
    print 'big'
else:
    print 'fig'
```

14. A _____ number is a number in the base-16 number system, commonly used by computer hardware, where individual place values are represented by the numerals 0 through 9 and A through F.

15. The _____ character set is the 128 characters, numerals and console control codes which have been adopted as standard computer characters.

16. A _____ type is the class of

Python types, each of which contains an ordered set of subordinate data elements.

17. Predict the result of the following string operations given the following variables:

```
s = 'pepperoni'
t = 'pizza'

a. min(s)
b. len(t)
c. 3 * t
d. s[5]
e. s[1:5]
```

18. Given the string `s` from Exercise #16, write a Python expression statement using a slice operation to return the string 'peron':

19. Use Appendix A to find the characters which correspond to these hexadecimal numbers:

```
a. 0x3E
b. 0x32
c. 0x46
d. 0x26
e. 0x5D
```

20. Use Appendix A to find the hexadecimal equivalents of the following characters:

```
a. 'P'
b. Ctrl+J (linefeed)
c. '+'
d. 'u'
e. '"'
```