CSci 157 - Introduction to Programming and Modeling

Iteration Statements

Dr. Stephen P. Carl

Iteration Statements

An *iteration statement* is used to execute a block of one or more statements repeatedly. Iteration statements are more commonly called *looping statements* or just *loops*

No more clicking the mouse repeatedly to get a shape to move across the screen. No more copying and pasting chunks of code which differ in only small ways.

Python provides the **`for`** statement for definite loops and the **`while`** statement for indefinite loops. We've worked with for loops a bit already, so let's dive into the **`while`** statement.

SEWANEE
The University of the South

The WHILE statement

The **`while`** statement (AKA while loop) appears quite simple at first. Consider the following template:

```
while <condition>:
    <body>
```

Here **<condition>** is a boolean expression and **<body>** is an indented code block that contains the statements to be iterated by the loop.

First, **<condition>** is evaluated. If false, control skips over the statements in **<body>.** Otherwise the statements are executed, after which control returns to the **<condition>** which is evaluated again. **<body>** is executed until **<condition>** becomes false.

SEWANEE
The University of the South

While Loop Design

In practice, while loops needs a bit more design. Here's a simple example that tries to sum up the first *N* integers starting with 1.

```
while i <= N:
    sum = sum + i
```

Obviously, by itself this won't work. What values do *i* and *sum* need to start with for this to give the right answer? Even with these initial values, what else is missing?

SEWANEE
The University of the South

Initialize/Test/Update

Here, we want **<condition>** to be true the first time through (though in general that's not required). Prior to the `while` we need to **Initialize** the *loop variables.* For example:

```
sum = 0
i = 1   # initialize loop variable
```

A *loop variable* is any variable that helps determine if the **<condition>** is `True` or `False`

SEWANEE
The University of the South

Initialize/Test/Update

Assume for now that N is a positive integer greater than or equal to 1.

On entering a loop the first time, we *Test* the condition. In this case, `i <= N` will be true, so the loop body is executed.

Once each statement in the loop body has been executed, we "loop back" to the top and once again **Test** the condition. This continues until the condition becomes `False`

SEWANEE
The University of the South

Initialize/Test/Update

What happens if `i <= N` is never false? The loop never terminates! This possibility is called an *infinite loop* - rarely a good thing.

Since the initial value of `i` made the condition true, we need to **Update** it so, eventually, its value becomes greater than `N`.

```
while i <= N:      # Test
   sum = sum + i
   i = i + 1       # Update
```

SEWANEE
The University of the South

Complete Example

Here is a complete function for summing from 1 to *N,* where *N* is passed as an argument:

```
## sum_to_N : int -> int
def sum_to_N(N):
   '''computes the sum of 1..N'''
   sum = 0
   i = 1          # Initialize
   while i <= N:  # Test
      sum = sum + i
      i = i + 1    # Update
   return sum
```

SEWANEE
The University of the South

Example: The Rising Sun

Sometimes, the **Initialize** and **Update** bits are done for us:

```
## onMouseClick : graphics.Circle -> None
def onMouseClick(sun):
'''moves the sun to the top of the window'''
    while sun.getY() > 0:
        sun.move(0, -5)  # move sun 5 pixels up
```

The `sun` object is initialized with the actual argument when the function is called, and `move` changes the y-coordinate of `sun` for us.

Note: this function is not quite complete...

SEWANEE
The University of the South

Example: Validating Input

In the `sum_to_N` example, we required that `N` be positive and greater than or equal to 1. Imagine a program that uses this function and gets the value of `N` as input from the user.

People are notorious for goofing up data entry. How do we ensure input value of `N` is positive? Or within a some specified range of values? Use a loop!

```
number = int(input("Enter a positive value: ")

while (number < 1):
   print("Not a positive value! Try again:")
   number = int(input("Enter a positive value: ")

print(sum_to_N(number))
```

SEWANEE
The University of the South

Multiple loop variables

If more than one variable is involved in **Test,** they all need to be initialized and updated properly.

For example, consider the problem of whether a `Point` we generate is inside the bounds of a window. We have to check both x- and y-coordinates against the width and height of the canvas, so there will be two loop variables.

Or maybe we want to move a graphic object until it hits the side of the window, and have it 'bounce' off.

SEWANEE
The University of the South

Example: Checking Window Bounds

Say we call a function `fractalPts` that generates `Point` objects for plotting a math function, continuing until one leaves the visible window:

```
Point pt = fractalPts(f1, f2)
x = pt.getX() // Init #1
y = pt.getY() // Init #2
# canvas refers to the GraphWin object
while (x >= 0 and x <= canvas.getWidth()) and
      (y >= 0 and y <= canvas.getHeight()):
   c = Circle(pt, 5, 5)
   c.draw(canvas)
   pt = fractalPts(f1, f2) // generate next pt
   x = pt.getX() // Update #1
   y = pt.getY() // Update #2
```

SEWANEE
The University of the South

The do/while loop

Unlike some programming languages, Python does not have a **do...while** loop - similar to while loops except the condition test occurs *after* the loop body. This would be useful when we know we want the loop body to execute at least once.

This and other common patterns can be emulated using the `while` statement. These patterns are discussed in detail in the text; have a look at the **Common Loop Patterns** section.

SEWANEE
The University of the South

The FOR statement

As we've seen, `for` is used when we want to iterate over a *sequence.* The general form is:

```
for <var> in <sequence>:
    <body>
```

The variable given by **<var>** is assigned each element of the **<seqence>** in turn; for each such value, the statements in **<body>** are executed. When the sequence has been exhausted, execution continues with the statement following **<body>**

SEWANEE
The University of the South

For Loop Example

Here's another version of the function for summing from 1 to N, using `for` instead:

```
def sum_to_N(N):
    sum = 0
    for i in range(1, N+1):
        sum = sum + i
    return sum
```

Because it takes a little thought to get the limits of the `range` right, I think `while` works better here.

SEWANEE
The University of the South

Contrasting for and while

As we've seen, `for` is used when we want to iterate a known number of times, controlled by the length of a sequence or the numbers in a range. We call this a *definite* loop.

By contrast, the *while loop* is an *indefinite* loop, used when we may not know how many iterations are needed.

SEWANEE
The University of the South