Python: Looping Processing

10th Lecture

1. Introduction

All the programs you have studied so far in this class have consisted of short sequences of instructions that are executed one after the other. Even if we allowed the sequence of instructions to be quite long, this type of program would not be very useful. As human beings, computers must be able to repeat a set of actions. They also must be able to select an action to perform in a particular situation. This lecture focuses on control statements—statements that allow the computer to repeat an action.

2. Definite Iteration: The for Loop

Repetition statements (also known as loops), which repeat an action. Each repetition of the action is known as a pass or an iteration. There are two types of loops—those that repeat an action a predefined number of times (***definite iteration***) and those that perform the action until the program determines that it needs to stop (***indefinite iteration***). In this section, we examine Python's for loop, the control statement that most easily supports definite iteration.

2.1. Executing a Statement a Given Number of Times

The **for** loop is a generic sequence iterator in Python: it can step through the items in any ordered sequence object. The form of this type of loop is:

```
for <variable> in range(<an integer expression>):
    <statement_1>
    <statement_n>
```

The first line of code in a loop is sometimes called the loop header. For now, the only relevant information in the header is the integer expression, which denotes the number of iterations that the loop performs. The colon (**:**) ends the loop header. The loop body comprises the statements in the remaining lines of code, below the header. Note that the statements in the loop body *must be indented and aligned in the same column*. *These statements are executed in sequence on each pass through the loop*. Here is a for loop that does so four times:

```
for eachPass in range(4):
    print("It's alive!", end=" ")
```

This loop repeatedly calls one function—the print function. The constant 4 on the first line tells the loop how many times to call this function. If we want to print 10 or 100 exclamations, we just change the 4 to 10 or to 100.

Now let's explore how Python's exponentiation operator might be implemented in a loop. Recall that this operator raises a number to a given power. For instance, the expression 2 ** 3 computes the value of 2^3, or 2 * 2 * 2. The following session uses a loop to compute exponentiation for a non-negative exponent. We use three variables to designate the number, the exponent, and the product. The product is initially 1. On each pass through the loop, the product is multiplied by the number and reset to the result. To allow us to trace this process, the value of the product is also printed on each pass.

```
number = 2
exponent = 3
product = 1
for eachPass in range(exponent):
    product = product * number
    print(product, end = " ")
```

The output

```
2 4 8
```

2.2. Count-Controlled Loops

When Python executes the type of for loop just discussed, it actually counts from 0 to the value of the header's integer expression minus 1. On each pass through the loop, the header's variable is bound to the current value of this count. The next code segment demonstrates this fact:

```
for count in range(4):
    print(count, end = " ")
```

The output

```
0 1 2 3
```

Loops that count through a range of numbers are also called count- controlled loops. The value of the count on each pass is often used in computations.

For example, consider the factorial of 4, which is 1 * 2 * 3 * 4 = 24. A code segment to compute this value starts with a product of 1 and resets this variable to the result of multiplying it and the loop's count plus 1 on each pass, as follows:

```
product = 1
for count in range(4):
  product = product * (count + 1)
print("Product: ", product)
```

The output

```
Product  24
```

Note that the value of count + 1 is used on each pass, to ensure that the numbers used are 1 through 4 rather than 0 through 3.

To count from an explicit lower bound, the programmer can supply a second integer expression in the loop header. *When two arguments are supplied to range, the count ranges from the first argument to the second argument minus 1*. The next code segment uses this variation to simplify the code in the loop body:

```
product = 1
for count in range(1, 5):
  product = product * (count)
print("Product: ", product)
```

The output

```
Product  24
```

The only thing in this version to be careful about is the second argument of **range**, which should specify an integer greater by 1 than the desired upper bound of the count. Here is the form of this version of the for loop:

```
for <variable> in range(<lower bound>, <upper bound + 1>):
    <loop body>
```

Accumulating a single result value from a series of values is a common operation in computing. Here is an example of a summation, which accumulates the sum of a sequence of numbers from a lower bound through an upper bound:

```
lower = int(input("Enter the lower bound: "))
upper = int(input("Enter the upper bound: "))
sum = 0
for count in range(lower, upper + 1):
    sum = sum + count
print("Sum: ", sum)
```

The output

```
Enter the lower bound: 1
Enter the upper bound: 10
Sum:  55
```

3. Augmented Assignment

Expressions such as x = x + 1 or x = x + 2 occur so frequently in loops that Python includes abbreviated forms for them. The assignment symbol can be combined with the arithmetic and concatenation operators to provide augmented assignment operations. Following are several examples:

```
a = 17
s = "hi"

a += 3          # Equivalent to a = a + 3
a -= 3          # Equivalent to a = a - 3
a *= 3          # Equivalent to a = a * 3
a /= 3          # Equivalent to a = a / 3
a %= 3          # Equivalent to a = a % 3
s += " there"   # Equivalent to s = s + " there"
```

All these examples have the format

<variable> <operator>= <expression>

Note that there is no space between <operator> and =. The augmented assignment operations and the standard assignment operation have the same precedence.

4. Loop Errors: Off-by-One Error

The for loop is not only easy to write, but also fairly easy to write correctly. Once we get the syntax correct, we need to be concerned about only one other possible error: *the loop fails to perform the expected number of iterations*. Because this number is typically off by one, the error is called an **off-by-one error**. For the most **part, off-by-one errors** result when the *programmer incorrectly specifies the upper bound of the loop*. The programmer might intend the following loop to count from 1 through 4, but it actually counts from 1 through 3.

```
for count in range(1, 4): # Count from 1 through 4, we think
print(count)
```

Note that this is not a **syntax error**, but rather a **logic error**. Unlike syntax errors, logic errors are not detected by the Python interpreter, but only by the eyes of a programmer who carefully inspects a program's output.

5. Traversing the Contents of a Data Sequence

Although we have been using the **for** loop as a simple **count-controlled loop**, the loop itself actually visits each number in a sequence of numbers generated by the range function. The values contained in any sequence can be visited by running a **for** loop, as follows:

```
for <variable> in <sequence>:
    <do something with variable>
```

On each pass through the loop, the variable is bound to or assigned the next value in the sequence, starting with the first one and ending with the last one. The following code segment traverses or visits all the elements in two sequences and prints the values contained in them on single lines:

```
for number in [1, 2, 3]:
    print(number, end = " ")
```

The output
```
1 2 3
```

```
for character in "Hi there!":
    print(character, end = " ")
```

The output
```
H i t h e r e !
```

6. Specifying the Steps in the Range

The **count-controlled loops** we have seen thus far count through consecutive numbers in a series. However, in some programs we might want a loop to skip some numbers, perhaps visiting every other one or every third one. A variant of Python's range function expects a third argument that allows you to nicely skip some numbers. The third argument specifies a step value, or the interval between the numbers used in the range, as shown in the examples that follow:

for count **in range**(1, 6, 1): # Same as using two arguments → [1, 2, 3, 4, 5]

for count **in range**(1, 6, 1): # Use every other number → [1, 3, 5]

for count **in range**(1, 6, 1): # Use every third number → [1, 4]

Now, suppose you had to compute the sum of the even numbers between 1 and 10. Here is the code that solves this problem:

```
sum = 0
for count in range(2, 11, 2):
    sum += count
print("Sum: ", sum)
```

The output

```
Sum 30
```

7. Loops that Count Down

All of our loops until now have counted up from a lower bound to an upper bound. Once in a while, a problem calls for counting in the opposite direction, from the upper bound down to the lower bound. *For example*, when the top-10 singles tunes are released, they might be presented in order from lowest (10th) to highest (1st) rank. In the next session, a loop displays the count from 10 down to 1 to show how this would be done:

```
for count in range(10, 0, -1):
    print(count, end=" ")
```

The output

```
10 9 8 7 6 5 4 3 2 1
```

When the step argument is a negative number, the range function generates a sequence of numbers from the first argument down to the second argument plus 1. Thus, in this case, the first argument should express the upper bound, and the second argument should express the lower bound minus 1.

<Best Regards>

Dr. Raaid Alubady