3.5 Designing code with loops

In the last two sections we have seen **while**-loops and **for**-loops. Beginning programmers are often mystified as to when to use which type of loop. The good news is that it often doesn't matter. In many situations we can set up natural, easy to read code with either type of loop. For example, the following blocks of code both sum the numbers from 1 to 10:

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

print( sum )
```

```
sum = 0
x = 1
while x <= 10:
    sum = sum + x
    x = x + 1

print( sum )
```

In either case the loop generates the list of numbers; inside the loop we have code to compute their sum. The first of these is a little shorter, but not enough to matter. To someone familiar with the syntax of **for**-loops and **while**-loops they seem equally clear. In a situation like this you can use either looping construct.

Suppose we want to generate a sequence of ten coin tosses: "Heads" or "Tails". At the end of section 2.4 we discussed the random library that has random number generators. In this case we will use randint(0, 1), which gives random numbers that are either 0 or 1. The loop we use is just a counting device; it will generate values, but we won't do anything with them other than to count how many there are. Again, we could do this with either looping construct:

```
from random import *

for x in range(10):
    number = randint(0, 1)
    if number == 0:
        print( "Heads" )
    else:
        print( "Tails" )
```

```
from random import *

x = 0
while x < 10:
    number = randint(0, 1)
    if number == 0:
        print( "Heads" )
    else:
        print( "Tails" )
    x = x + 1
```

Programs often use loops to generate sequences of values that are to be manipulated in some way. For example, suppose we want to count the number of leap years between two given years, such as 1878 and 1945. There are two separate tasks here: one is to generate the sequence of years and the other is to determine which of them are leaps years and keep track of their number. The first task requires a loop, and it doesn't make much difference whether we use a **for**-loop or **while**-loop for it. The second task needs an elaborate **if**-statement; we saw code for this in Program 3.1.3. Here again we show parallel constructions, with **for**- and **while**-loops.

```
startYear = 1878
finalYear = 1945
leapCount = 0
for year in range(startYear, finalYear+1):
    if year % 400 == 0:
        isLeap = True
    elif year % 100 == 0:
        isLeap = False
    elif year % 4 == 0:
        isLeap = True
    else:
        isLeap = False

    if isLeap:
        leapCount = leapCount + 1

print( "%d leap years." % leapCount )
```

Counting leap years, **for**-loop

```
startYear = 1878
finalYear = 1945
leapCount = 0
year = startYear
while year <= finalYear:
    if year % 400 == 0:
        isLeap = True
    elif year % 100 == 0:
        isLeap = False
    elif year % 4 == 0:
        isLeap = True
    else:
        isLeap = False

    if isLeap:
        leapCount = leapCount + 1

    year = year + 1
print( "%d leap years." % leapCount )
```

Counting leap years, **while**-loop

In section 3.4 we saw a similar program where we wanted to find the prime numbers in a range of integers. We used a **for**-loop to generate all of the integers in this range, but we could have just as easily used a **while**-loop. Later in that section we wanted to find the first N prime numbers for some value N. That would not have been easy to do with a **for**-loop because we didn't know the endpoint of the range of values to generate as candidates.

When there is a definite sequence of values that need to be processed, **for**-loops are often more natural; this was the reason **for**-loops were invented. Suppose, for example, that we want to count the number of instances of the letter 'a' in the word 'abracadabra'. We can walk through the letters of the string with a simple **for**-loop:

```
for letter in "abracadabra":
```

With a while-loop we need to use numeric indexes for the string:

```
index = 0
s = "abracadabra"
while index < len(s):
    letter = s[index]
```

Here are the resulting programs:

```
count = 0
for letter in "abracadabra":
    if letter == "a":
        count = count + 1

print( count )
```

```
index = 0
s = "abracadabra"
count = 0
while index < len(s):
    letter = s[index]
    if letter == "a":
        count = count + 1
    index = index + 1

print( count )
```

The version with **for**-loops is certainly shorter. More importantly, it is more natural and easier to read. This means the programmer is more likely to write it correctly.

The following rule of thumb can help you decide which looping construct to use:

If you need to generate a *definite* list of values where you can say in advance which values belong in this list, **for**-loops are usually easier to use. If you need to generate an *indefinite* list of values, for which you cannot say in advance where to stop, **while**-loops are usually easier.

We will illustrate these ideas by developing a program that prints a calendar for one month. In the next few chapters we will make several versions of this program to illustrate various programming constructs. This version asks the user to input some of the information about the calendar that we will later be able to compute directly.

Here is typical output from the program:

```
                July 2009
  Sun    Mon    Tu   Wed    Th   Fri   Sat
                       1     2     3     4
    5      6     7     8     9    10    11
   12     13    14    15    16    17    18
   19     20    21    22    23    24    25
   26     27    28    29    30    31
```

We will ask the user to enter the name of the month: ("July 2009" in this case), the number of days in the month (here 31) and the day of the week the month starts on (here 3 for Wednesday). We will call these three variables monthName, daysInMonth, and startsOn. Our job is to center the name, to print the header with the names of the days of the week, and then to print a table with the dates under the appropriate days.

The first decision is to settle on how we will print the table. It is easy to generate the numbers that need to be printed: 1 up to daysInMonth. We can do this with either

```
for day in range(1, daysInMonth + 1):
```

or

```
day = 1
while day <= daysInMonth:
    ⋮
    day = day + 1
```

The former seems a little clearer, so we will go with that. We can get the numbers to print in columns in the same way we did in Program 3.4.5: each number is printed with a formatted **print** statement using a fixed column width, followed by a comma. At the completion of each row we do a single **print** statement (with no comma) to terminate the line. Using a column width of 5, this gives us the following code:

```
dayOfWeek = 0
for day in range(1, daysInMonth + 1):
        print( "%5d" % day, end='''')
        dayOfWeek += 1
        if dayOfWeek == 7:
                print( )
                dayOfWeek = 0
```

This almost does what we need, but it makes every month start on Sunday. This is where our startsOn variable comes into play. If a month starts on a Wednesday, as does July 2009 in our example, startsOn will be 3. We want to print blanks corresponding to the first 3 days of the week (Sunday, Monday and

Tuesday) and print a 1 in the column for Wednesday. Note that since we print the numbers with

```
print( "%5d" % day, end='''' )
```

we can print blanks that occupy the same column width with

```
print( "%5s" % " ", end='''' )
```

Here is the loop that prints the blanks:

```
dayOfWeek = 0
for day in range(0, startsOn):
    print( "%5s" % " ", end='''' )
    dayOfWeek += 1
```

The rest of our program is easy. We need to print a header with the names of the days of the week. If we put them into a list we can use a **for**-loop to walk through it; we print the names with the same column widths as the body of the calendar:

```
for day in ["Sun","Mon","Tu","Wed","Th","Fri","Sat"]:
        print( "%5s" % day, end='''' )
print( )
```

This is so simple because Python gives us good tools for working with lists and strings.

Finally, we need to center the name of the month. Our calendar has 7 columns, each 5 spaces wide, so the calendar is a total of 35 spaces wide. The name will occupy **len**(monthName) of these spaces, leaving 35−**len**(monthName) left over. We want to print half of the leftover spaces prior to the name. There is no easy way to use the formatted **print**-statements with a variable width, so we use the * operator to make a string with the right number of blanks:

```
leftover = 35-len(monthName)
print( " "*(leftover/2), end='''')
print( monthName )
```

Here is the full program, with some comments inserted to make it more readable:

```
# This prints the calendar for one month

def main():
    monthName = input( "Name of month: " )
    daysInMonth = eval(input("Number of days in the month: "))
    print( "What day of the week does the month start on?" )
    startsOn=eval(input("Enter 0 for Sunday, 1 for Monday, etc: "))

    # First we center the month name.
    leftover = 42-len(monthName)
    print( " "*(leftover/2), end='''')
    print( monthName )

    # Then print the names of the days of the week,
    # in columns:
    for day in ["Sun","Mon","Tu","Wed","Th","Fri","Sat"]:
        print( "%5s" % day, end='''')
    print( )

    # This indents 5 blanks for each of the days
    # of the week prior to the day this month starts on:
    dayOfWeek = 0
    for day in range(0, startsOn):
        print( "%5s" % " ", end='''')
        dayOfWeek += 1

    # Finally, we print the days in the month,
    # with a line break after each Saturday:
    for day in range(1, daysInMonth + 1):
        print( "%5d" % day, end='''')
        dayOfWeek += 1
        if dayOfWeek == 7:
            print( )
            dayOfWeek = 0
    if dayOfWeek != 0:
        print( )

main()
```

Program 3.5.3: Calendar Program: FirstVersion