3.4 For-loops

In the previous section we looked at **while**-loops, Python's basic looping structure. There is a second loop construct in Python called a **for**-loop. This is more specialized. We use **for**-loops when there is a particular set of values that we want to work through, doing some task once with each of these values.

A **for**-loop has the structure:

```
for <variable> in < sequence>:
        statement_block:
```

When this statement is executed, the variable takes on the first value in the sequence and the statement_block is executed. The variable then takes on the next value and the statement_block is executed. This process continues with each of the values in the sequence.

What is a *sequence*? It is any collection of "sequential" data values: for something to be a sequence we must be able to say what the first value is, and for each value but the last we need to be able to say what the next value is. The non-negative integers form a sequence: 0 is the first value, 1 comes next, 2 is after that, and so forth. The non-negative real numbers do not. For example, there is no "next" value after 1.3. There is 1.31, but 1.301 comes before that, and 1.3001 precedes that , and so forth. Python has many ways to construct a sequence; among the most imporant of these is the *list*. A list is a sequence specified by listing values inside square brackets, as in ["John", "George", "Paul", "Ringo"]. Python makes it easy to work with lists. For example, Program 3.4.1 will print

The Beatles were:
John
George
Paul
Ringo

```
# This prints a list of the Beatles
def main():
    print( "The Beatles were:" )
    for name in ["John","George","Paul","Ringo"]:
        print( "   %s" % name )

main()
```

Program 3.4.1: The Beatles

Any values can be put into lists: strings, numbers, even other lists. For example, [1, 2, [3, 4]] is a list where the first two elements are numbers and

the third is a list with two elements.

It is often useful to have a loop walk through a range of numbers. Python's **range()** function generates sequences of numbers for this purpose. There are three ways to use this function:

- **range**(a, b) generates a list-like structure containing the numbers from a to b: including a but not including b. For example, **range**(3, 7) is the sequence 3, 4, 5, 6s. This assumes a is smaller than b; if the reverse is true the list generated is empty.
- **range**(b) is the same as **range**(0, b). For example, **range**(5) is [0, 1, 2, 3, 4].
- **range**(a, b, c) uses an increment of c to go from one number to the next. So **range**(2, 12, 3) is [2, 5, 8, 11] and **range**(5, 0, −1) is [5, 4, 3, 2, 1].

In Python 2 **range**(a, b) is an actual list. This is simple but runs into trouble if you work with very large ranges. For example, if you try to roll a pair of dice a billion times with the code

```
for x in range(1000000000):
    value = RollDice()
    ....
```

Python 2 would actually generate in memory a list with a billion items. This would probably cause your program to crash. In Python 3 the **range**-function returns a **range−object** that can be iterated through with a for-loop and works like a list, but it has the same small size regardless of the size of the range. The only time you are likely to see a difference between lists and range-objects is if you try to print a range-object.

```
print( range(0, 10) )
```

prints the words **range**(0, 10) rather than the numbers from 0 through 9. If you want to print the elements of the range you can convert it into an actual list with the **list**( ) function:

```
print( list(range(0, 10)) )
```

prints [0,1,2,3,4,5,6,7,8,9]

The following code fragment uses the **range**() function to print the factors of the number 36. Note that we need to extend the range up to 37 so that the last number in the list is 36.

```
for x in range(1, 37):
    if 36%x == 0:
        print( x )
```

Factoring 36

We can also treat strings as sequences — the letters are taken in the order in which they appear in the string. The next program reads a string from the user and reports how many instances of the letter "a" it contains. As is common with programs that count things, this has a variable count that starts at 0 and is incremented each time the appropriate letter is found.

```
# This counts the a's in a string
def main():
    string = input( "Enter a string: " )
    count = 0
    for letter in string:
        if letter == "a":
            count = count + 1
    print( "The letter 'a' occurs %d times."%count )
main()
```

Program 3.4.2: Counting instances of a letter

We will illustrate **for**-loops with several programs that find prime numbers. Remember that a number is *prime* if its only factors are 1 and itself. 2, 3, 5, 7, 11, and 13 are all prime numbers, 15 is not. First, we will write a program that lets the user input a number; the program will say if this number is prime. The basic structure inputs a single number and says if it is prime:

```
def main():
    number = eval(input("Enter a number: "))

    # test if it is prime; for now we
    # always say no.
    isPrime = False

    if isPrime:
        print( "%d is prime." % number )
    else:
        print( "%d is not prime." % number )

main()
```

Program 3.4.3: Initial version

Next, we need to fill in code for testing whether the number we have is prime. One way to do this is to think about the definition. A number is prime

if it is divisible only by 1 and itself, so it is not prime if we can find any divisor greater than 1 and less than the number itself. This is a natural application for a **for**-loop we know exactly the range of values we want to check, so we use the **range()** function to generate them and a **for**-loop to check them. We will start the isPrime variable out at True and change it to False if we ever find a divisor.

```
isPrime = True
for factor in range(2, number):
        if number % factor == 0:
                isPrime = False
```

Putting this altogether, we have the first complete version of our prime finder:

```
# This lets the user enter a number and reports
# whether that number is prime.
def main():
    number = eval(input( "Enter a number: " ))

    isPrime = True
    for factor in range(2, number):
        if number % factor == 0:
            isPrime = False

    if isPrime:
        print( "%d is prime." % number )
    else:
        print( "%d is not prime." % number )

main()
```

Program 3.4.3: Final version

We can turn this into a program that generates prime numbers if we surround it with a loop in place of the single input statement. For example, if we want to find all prime numbers between 2 and 100 we would use the loop:

```
for number in range(2, 101):
        isPrime = True
        for factor in range(2, number):
            if number % factor == 0:
                isPrime = False
```

Note that for each new value of number we need to reset isPrime to True; otherwise, once we found a non-prime number there would never be a way to find a prime one: isPrime would always be False.

One more change to Program 3.4.3 will help make it a better prime number generator. The prime numbers are fairly sparse; most integers are not prime. If we print "is prime" or "is not prime" with every number, there are so many non-primes that it becomes hard to see the actual primes. We accordingly change the **if**-statement to report only the prime numbers and be silent on the non-primes. Here is the resulting program:

```
# This prints all of the prime numbers up to
# a limit supplied by the user
def main():
    max = eval(input("Enter the largest number to check: "))
    for number in range(2, max+1):
        isPrime = True
        for factor in range(2, number):
            if number % factor == 0:
                isPrime = False

        if isPrime:
            print( "%d is prime." % number )

main()
```

Program 3.4.4: Final version

As a final example for this section, here is a third prime number program. This time we make two changes that illustrate some of the differences between **for**-loops and **while**-loops. The first change is that instead of asking the user how high in the list of integers we should check for prime numbers, this time we ask the user how many prime numbers we should find. This might sound like a minor change, but it forces us to rethink our loops. In Program 3.4.4 our basic structure is this:

```
for number in range(2, max+1):
        <if number is prime, print it>
```

If N is the number of primes we need to find, we could change our main loop to:

```
for i in [1,2, 3,     . N]:
     <Find the ith prime and print it>
```

This isn't so simple. How do we find the *ith* prime number? This seems unnecessarily complicated. An easier solution is to have a loop that generates numbers to test, with the exit condition based on whether we have found enough primes. This needs to be a **while**-loop rather than a for-loop because we don't know how many times we will execute its body; we just keep going around until we have enough primes. This looks something like the following:

```
number = 2
primeCount = 0
while primeCount < N:
     <if number is prime:>
          print number
          primeCount = primeCount + 1
     number = number + 1
```

It should be an easy matter to convert the body of this **while**-loop to Python code. First, we are going to make another change in the program. What we have so far will make a long list of prime numbers. The output will be shorter if we allow for multiple columns, such as

```
  2    3    5    7   11   13
 17   19   23   29   31   37
 41   43   47   53   59
```

The first thought of many beginners at this point would be "We need another loop." But what would this loop do? We already have a loop to generate numbers. We don't need a loop to count the entries in a line; a simple variable can do that. We will print the numbers just as before, only with a formatting statement to control the number of spaces used for each number regardless of how many digits it has (so we get nice columns of output), and we will end each print statement with a comma, so that successive prints appear on the same line. Each time we print a number we will increment a lineCount variable that tells us how many numbers are on the current line. When this is large enough we execute a **print** statement (with no comma) to terminate the line and start over. Of course, we will also start our lineCount variable over at 0.

Now our code looks like this:

```
number = 2
primeCount = 0
linecount = 0
while primeCount < N:
     <if number is prime:>
          print("%7d " % number, end = "")
          primeCount = primeCount + 1
          lineCount = lineCount + 1
          if lineCount == C:
               print( )
               lineCount = 0
     number = number + 1
```

Here is the whole program:

```
# This prints the first N prime numbers,
# where the value of N is supplied by the user.
# The output is printed in C columns.

def main():
    N = eval(input("How many prime numbers do you want? "))
    C = eval(input("How many columns of output do you want? "))
    number = 2
    primeCount = 0
    lineCount = 0
    while primeCount < N:
        isPrime = True
        for factor in range(2, number):
            if number % factor == 0:
                isPrime = False

        if isPrime:
            print( "%7d " % number, end="")
            primeCount = primeCount + 1
            lineCount = lineCount + 1
            if lineCount == C:
                print( )
                lineCount = 0

        number = number + 1
main()
```

Program 3.4.5: Final version