CS303E: Elements of Computers and Programming

Loops

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

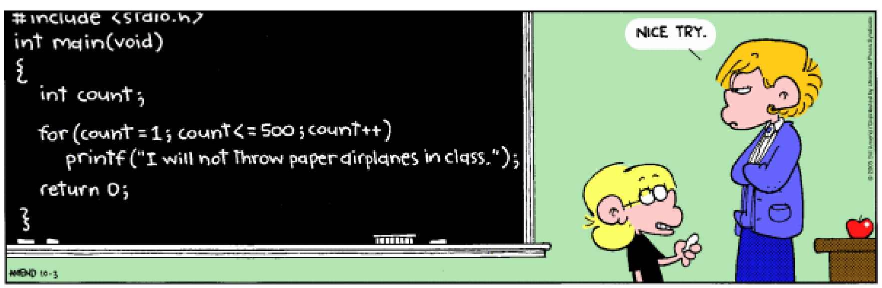
Last updated: August 27, 2024 at 14:25

Repetitive Activity

Often we need to do some (program) activity numerous times:

Using Loops

So you might as well use cleverness to do it. *That's what loops are for.*



It doesn't have to be the exact same thing over and over.

While Loop

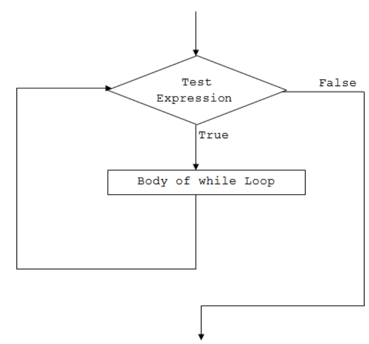
One way is to use a while loop.

General form:

```
while condition:
    statement(s)
```

Meaning: as long as the condition remains True, execute the statements.

As usual, all of the statements in the body must be indented the same amount.



While Loop

In file `WhileExample.py`:

```
COUNT = 500
STRING = "I will not throw paper airplanes in class."

def main():
    """ Print STRING COUNT times. """
    i = 0
    while (i < COUNT):
        print(STRING)
        i += 1

main()
```

```
> python WhileExample.py
I will not throw paper airplanes in class.
I will not throw paper airplanes in class.
   ...
I will not throw paper airplanes in class.
```


Another While Example

Compute $N! = 1 \times 2 \times \ldots N$, the factorial of N.

In file `fact2.py`:

```
def factorial():
    """ Compute the factorial of a number supplied
    by the user. """
    num = int(input("Compute factorial of: "))
    ans = 1
    i = 1
    # Do we know that this loop terminates?
    while ( i <= num ):
        ans *= i
        i += 1
    print("Factorial of", num, "is", ans)

factorial()
```

```
> python fact2.py
Compute factorial of: 17
Factorial of 17 is 355687428096000
>
```


Infinite Loops

You *have* to do something in the loop to ensure that you eventually exit; otherwise, you'll be in an *infinite loop*.

Either:

- change some variable so that the test eventually becomes False, or
- break out of the loop on some condition that eventually occurs.

Another Example: Sum to N

```
def sumToN():
    # Accept input from the user until a positive integer
    # is entered.
    while True:
        n = int(input("Sum to what positive integer: "))
        if n < 1:
            print("That's not positive. Try again!")
        else:
            # This will exit the loop
            break
    # What must be true here?
    # Sum the numbers up to n
    sum = 0
    i = n
    while i > 0:
        sum += i
        i -= 1
    print("The numbers to", n, "sum to", sum)

sumToN()
```

Do we know that both loops terminate? Why or Why not?

Another Example: Sum to N

Here's running our program:

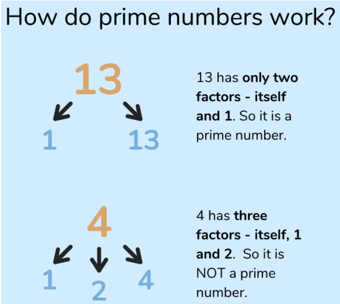
```
> python sumToN.py
Sum to what positive integer: -4
That's not positive. Try again!
Sum to what positive integer: 0
That's not positive. Try again!
Sum to what positive integer: 10
The numbers to 10 sum to 55
>
```

Would this program work if the user entered a float?

While Loop Example: Test Primality

An integer is prime if it has no positive integer divisors except 1 and itself.

To test whether an arbitrary integer `n` is prime, see if any number in `[2 ... n-1]`, divides it.



You couldn't do that in *straight line* code without knowing `n` in advance. Why not?

Even then it would be *really* tedious if `n` is very large.

isPrime Loop Example

In file `IsPrime.py`:

```
def main():
   """ See if an integer entered is prime. """
   # Can you spot the inefficiencies in this?
   num = int( input("Enter an integer: ") )
   isPrime = True
   if ( num < 2 ):
      isPrime = False
   elif ( num == 2 ):
      isPrime = True
   else:
      divisor = 2
      while ( divisor < num ):
         # Keep repeating this block until condition
         # becomes false (or we break out of the loop).
         if ( num % divisor == 0 ):
            isPrime = False
            break
         else:
            divisor += 1
   print(num, "is prime" if isPrime else "is not prime")
```


isPrime Loop Example

```
> python IsPrime.py
Enter an integer: 53
53 is  prime
> python IsPrime.py
Enter an integer: 54
54 is not  prime
```

It works, though it's pretty inefficient. If a number is prime, we test every possible divisor in `[2 ... n-1]`.

- We don't actually need the special test for 2. *Think about why that is.*
- There's no need to test any even divisor except 2. Why not?
- If n is not prime, it will have a divisor less than or equal to $\sqrt{n}$.

A Better Version: IsPrime2.py

In file IsPrime2.py:

```
import math

def main():
   """ See if an integer entered is prime. """
   num = int( input("Enter an integer: ") )

   isPrime = True
   if ( num % 2 == 0 ):
      # If num is even, then it's prime only if (num == 2)
      isPrime = ( num == 2 )
   else:
      divisor = 3               # Why 3?
      while ( divisor <= math.sqrt( num )):
         if ( num % divisor == 0 ):
            isPrime = False
            break               # exit from loop
         else:
            divisor += 2        # Why 2?
   print(num, "is", "prime" if isPrime else "not prime")
```


The Better isPrime Version

```
> python IsPrime2.py
Enter an integer: 2
2 is  prime
> python IsPrime2.py
Enter an integer: 53
53 is  prime
> python IsPrime2.py
Enter an integer: 54
54  is not prime
> python IsPrime2.py
Enter an integer: 997
997 is  prime
```

Notice that IsPrime does 995 divisions to discover that 997 is prime. IsPrime2 only does 16. Why?

Example While Loop: Approximate Square Root

Approximate the square root of a positive integer as follows:

In file GuessSqrt.py:

```
def main():
   """Approximate the square root of a positive integer."""
   num = 0
   while (num <= 0):
      num = int( input("Enter a positive integer: ") )
      if (num <= 0):
         print( "Try again" )

   # Iterate by increments of 0.1 until we find an
   # approximate square root (within 0.1).
   guess = 0.1
   while ( guess ** 2 < num ):
      guess += 0.1

   sqrt = guess
   print( "The square root of ", num, "is approximately", \
          format( sqrt, "4.1f") )

main()
```


Running the Example

```
> python GuessSqrt.py
Enter a positive integer: -20
Try again
Enter a positive integer: 20
The square root of  20 is approximately  4.5
> python GuessSqrt.py
Enter a positive integer: 1024
The square root of  1024 is approximately 32.0
> python GuessSqrt.py
Enter a positive integer: 100
The square root of  100 is approximately 10.1
```

Notice that the last one isn't quite right. The square root of 100 is exactly 10.0. Foiled again by the approximate nature of floating point arithmetic.

How would you change the code to get an approximation within 0.01?

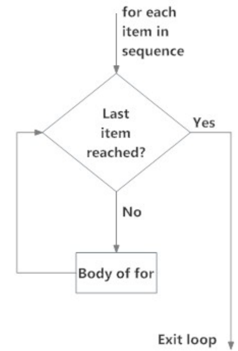
Let's Take a Break

For Loop

In a `for` loop, you typically know how many times you'll execute.

General form:

```
for var in sequence:
  statement(s)
```



Meaning: assign each element of sequence in turn to var and execute the statements.

As usual, all of the statements in the body must be indented the same amount.

Loop Variable

```
for var in sequence:
  statement(s)
```

`var` is called the *loop variable* or sometimes the *indicial variable*. It takes on successive values from the sequence in successive iterations of the loop.

```
>>> for i in [1, 2, 4, 8, 16, 32, 64]:
...     print(i)
...
1
2
4
8
16
32
64
>>>
```


What's a Sequence?

A Python `sequence` holds multiple items stored one after another.

```
>>> seq = [2, 3, 5, 7, 11, 13]   # a list
```

The `range` function is a good way to generate a sequence.

`range(a, b)` : denotes the sequence `a, a+1, ..., b-1`.

`range(b)` : is the same as `range(0, b)`.

`range(a, b, c)` : generates `a, a+c, a+2c, ...., b'`, where b' is the last value < b.

Actually, `range()` doesn't really return a sequence, but rather a special type of immutable object that supplies a value on demand. *Don't worry about this!*

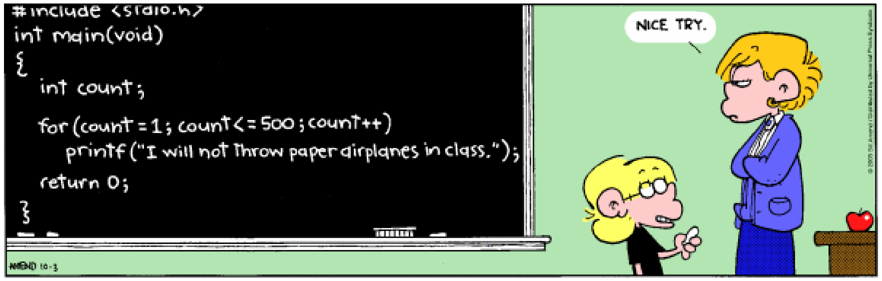
Range Examples

```
>>> for i in range(3, 6): print(i, end=" ")
...
3 4 5 >>> for i in range(3): print(i, end=" ")
...
0 1 2 >>> for i in range(0, 11, 3): print(i, end=" ")
...
0 3 6 9 >>> for i in range(11, 0, -3): print(i, end=" ")
...
11 8 5 2 >>>
```

Why is it printing strangely?

Loop Example

Remember this one?



How would you do this with a for loop in Python?

Loop Example

In file ForExample.py:

```
COUNT = 500
STRING = "I will not throw paper airplanes in class."

def main():
    for i in range(COUNT):
        print( STRING )

main()
```

```
> python ForExample.py
I will not throw paper airplanes in class.
I will not throw paper airplanes in class.
   ...
I will not throw paper airplanes in class.
```

Notice that the variable i isn't used in the loop body; it's only for counting *in this example*. Does it print the right number of lines?

Another For Loop Example

Suppose you want to print a table of the powers of 2 up to 2^n.

In file PowersOf2.py:

```
def main():
   """ Print a table of powers of 2 up to 2**n,
       where n is entered by the user.  """
   num = int( input("Enter an integer: ") )

   for power in range (num + 1):
      print( format( power, "3d"), \
             format( 2 ** power, "8d" ) )
```

Why does the range go to num + 1?

For Loop Example

```
> python PowersOf2.py
Enter an integer: 15
  0        1
  1        2
  2        4
  3        8
  4       16
  5       32
  6       64
  7      128
  8      256
  9      512
 10     1024
 11     2048
 12     4096
 13     8192
 14    16384
 15    32768
```


Break and Continue

Two useful commands in loops (`while` or `for`) are:

- break: exit the loop (but continue the program);
- continue: exit the current iteration, but continue with the loop.

```
while (True):
   value = float( input( "Enter a number, or 0 to exit: " ))
   if ( value == 0 ):
       break
   # When will the following happen?
   < process value >
```

```
while (True):
   value = int( input( "Enter a non-negative integer: " ))
   if (value < 0):
       continue
   # When will the following happen?
   < process value >
```

What's the problem with this loop?

Nested Loops

The body of `while` loops and `for` loops contain arbitrary statements, including other loops.

Suppose we want to compute and print out a multiplication table like the following:

```
           Multiplication Table
      |   1   2   3   4   5   6   7   8   9
-------------------------------------------
   1  |   1   2   3   4   5   6   7   8   9
   2  |   2   4   6   8  10  12  14  16  18
   3  |   3   6   9  12  15  18  21  24  27
   4  |   4   8  12  16  20  24  28  32  36
   5  |   5  10  15  20  25  30  35  40  45
   6  |   6  12  18  24  30  36  42  48  54
   7  |   7  14  21  28  35  42  49  56  63
   8  |   8  16  24  32  40  48  56  64  72
   9  |   9  18  27  36  45  54  63  72  81
```


Multiplication Table

```
           Multiplication Table
      |   1   2   3   4   5   6   7   8   9
-------------------------------------------
   1  |   1   2   3   4   5   6   7   8   9
   2  |   2   4   6   8  10  12  14  16  18
                     ....
   9  |   9  18  27  36  45  54  63  72  81
```

Here's an algorithm to do this:

1. How many columns/rows in the table?
2. Print the header information.
3. For each row i:
   1. Print i.
   2. For each column j: compute and print (i * j).
   3. Go to the next row.

This is easily coded using nested `for` loops.

Nested Loops

Print the header:

```
                Multiplication Table
    |   1   2   3   4   5   6   7   8   9
------------------------------------------
```

In file MultiplicationTable.py:

```
# Defines the size of the table + 1.
LIMIT = 10

def main():
    """ Print a multiplication table to LIMIT - 1. """
    print("          Multiplication Table")
    # Display the column headers.
    print("    |", end = "")
    for j in range(1, LIMIT):
        print(format(j, "4d"), end = "")
    print()    # jump to a new line
    # Print line to separate header from body of the table.
    print("------------------------------------------")
```


Nested Loops

This continues our multiplication example.

```
  1 |   1   2   3   4   5   6   7   8   9
  2 |   2   4   6   8  10  12  14  16  18
               ....
  9 |   9  18  27  36  45  54  63  72  81
```

```
    # Display table body
    for row in range(1, LIMIT):
        print( format(row, "3d"), "|", end = "")
        for col in range(1, LIMIT):
            # Display the product and align properly
            print( format( row*col, "4d"), end = "")
        print()

main()
```


Nested Loops Example

```
> python MultiplicationTable.py
          Multiplication Table
    |   1   2   3   4   5   6   7   8   9
------------------------------------------
  1 |   1   2   3   4   5   6   7   8   9
  2 |   2   4   6   8  10  12  14  16  18
  3 |   3   6   9  12  15  18  21  24  27
  4 |   4   8  12  16  20  24  28  32  36
  5 |   5  10  15  20  25  30  35  40  45
  6 |   6  12  18  24  30  36  42  48  54
  7 |   7  14  21  28  35  42  49  56  63
  8 |   8  16  24  32  40  48  56  64  72
  9 |   9  18  27  36  45  54  63  72  81
```

Notice that if you want a bigger or smaller table, you only have to change LIMIT in the code. But what would be wrong?

Nested Loops Example

Notice that if you want a bigger or smaller table, you only have to change LIMIT in the code. But what would be wrong?

Suppose we set `LIMIT = 5`?

```
> python MultiplicationTable.py
          Multiplication Table
    |   1   2   3   4
------------------------------------------
  1 |   1   2   3   4
  2 |   2   4   6   8
  3 |   3   6   9  12
  4 |   4   8  12  16
```

The header is no longer right. We'll see how to fix this later.

Next stop: Functions.