Operations in a loop

tsh73, May - June 2013

Table of Contents

Operations in a loop

Terms

Filter

Index, counter, flag

Index

Counter

Flag

Sum, product, concatenation

Sum

Concatenation

Product

Average

Arithmetic mean

Geometric mean

Min, max

Minimum

argMin

Maximum (and argMax)

Mixing it together

Common operations in a FOR loop.
After all, loops are ubiquitous thing in programming, and having standard loop tricks at your fingertips is a must.

Terms

First, let's recall our terms.
We will consider single FOR loop (that is, no nesting). It has header (loop operator), and a body ("payload"). Body executes several times (well, that was the idea about loops). Each pass through the loop called "iteration". In Basic, FOR loop has also word NEXT marking where payload ends:

```
FOR i=1 TO 10 STEP 4
'payload
NEXT i
```

Variable in the header (i) is called loop control variable, loop variable, loop counter, or index. It goes from start value (here, just 1) to end value (here 10) with given step. Actually, loop executes until index value gets bigger then end value – and like in our example, it might not exactly hit it. (Put "print i" as a payload and see.)
STEP is optional – without it, index incremented by 1.
Naming loop variable after NEXT optional too.

Let's make a loop that does something.
A loop that prints pairs (x, f(x)) for say sin(x) over interval 0..20 with step 2:

```
for x = 0 to 20 step 2
    y=sin(x)
    print x, y
next
```

Filter

First concept is *filter*. Our loop produces some data; by setting a condition, we could filter it, selecting only things we like.
For example, let's print only positive values of a function.

```
for x = 0 to 20 step 2
    y=sin(x)
    if y>0 then
```

```
        print x, y
    end if
next
```

So filter is a condition put in a loop, which checks each iteration and allow for processing only things we like.
Of course you could nest filters. Or use “else” part, if you need to. After all, it’s just conditions.

Index, counter, flag

Index

We already agreed that loop index is just loop variable. Here, x. It runs from initial value till the bitter end with given step.

Counter

If we have a loop that goes from 1 to 10 with step 1 (or STEP omitted), we might be pretty sure loop executes exactly 10 times. If we have no EXIT FOR somewhere inside, that is.
But if start value, end value, step turned to be variables, when it might get tricky. (No, we cannot just suppose N=int((endValue-startValue)/step)+1. Just try it on `FOR x=1 TO 2 STEP 0.1`, then run that loop and see for yourself.)
Counter is the answer. We take a variable; set it to 0 before start of the loop; and add 1 to it on each pass. Like this.

```
counter=0
FOR x=0 TO 1 STEP 0.1
    counter = counter+1
next
print "counter =";counter
```

Construction “take some variable, add a thing and put it back” happens in a programs quite often. It is said that variable *accumulates* something. Here it accumulates counter.
So it really counts stuff that happened. And if we cut it short with EXIT FOR, it will still have correct number.
We don’t have to count *ALL* iterations. Here we are counting only positive values of y:

```
counter=0
for x = 0 to 20 step 2
```

```
    y=sin(x)
    if y>0 then
        counter = counter+1
        print x, y
    end if
next
print "counter =";counter
```

Wait! It's a counter inside a filter! Yep, true ;).

Flag

Sometimes, we don't need to know how much times something happened. We just interested to know does it happen at all - or not (do we already have that guy in an array? Are there any enemy pieces left? Etc.). We can use a counter and check if it became greater then zero; it will work.
Common way to handle this – much for clarity sake - usually to peruse special Boolean variable, called *flag*. Boolean variable can hold only two values – true and false (happened or not); in BASIC, folks generally use 1 (or -1) and 0, 0 being false.
So before loop, we set flag to false (0); on each iteration, we check condition, and if it holds, set flag to true. After loop, we check flag value to gather information if our event happened or not.
Note: since it is Boolean, it could be checked for true/false without explicit compare operator.
Let's check if we have any negatives in our loop:

```
flag=0  'false
for x = 0 to 20 step 2
    y=sin(x)
    if y<0 then flag=1  'true
    print x, y
next
if flag then print "We have negatives"
```

If you recall that logical operations (comparisons) return Boolean result (0 or 1 in LB/JB), you might be tempted to write string that sets flag without IF, like this:

```
flag = (y<0)
```

Don't do that, it will not work. Flag needs to change only "one way", from false to true, and stay that way. Without IF, it could change back and forth on any iteration.

Sum, product, concatenation

Sum

If we start from zero and add 1 on each loop, we'll get loop counter.
If we start from zero and add some value (like, value of a function), we'll get a *sum* of these values - we accumulate sum.
Here's we calculate sum of our function, y=sin(x):

```
sum=0
for x = 0 to 20 step 2
    y=sin(x)
    sum=sum+y
    print x, y
next
print "sum= ";sum
```

Concatenation

Now, we can accumulate not only sum (numeric) but also strings. Let's build string containing alphabet:

```
a$=""   'start with empty
FOR i=asc("A") TO asc("Z")
    a$=a$;chr$(i)   'accumulate in a string
next
print a$
```

I used string concatenation operator, (;) to emphasize that we work on strings, but you can use (+) all the same:

```
a$=a$+chr$(i)
```

But you better have in mind that contrary to numerical (+), result of string concatenation depends on order. So with this line instead

```
a$=chr$(i)+a$
```

we'll have our alphabet reversed.

Product

Similar pattern holds for calculating a *product*: we start with initial value, multiply our variable by a value, put result back to that variable, and get product of all values then loop ends. Obviously, we should not use 0 as initial value. But 1 will do.
Let's calculate n!, defined as 1*2*3…*(n-1)*n:

```
n=5
f=1   'start with 1 for product
FOR i=1 TO n
    f=f*i   'accumulate factorial
next
print "Factorial of "; n; " equals to "; f
```

Average

Arithmetic mean

Average, or *arithmetic mean*, defined as sum of values divided by their number. Or, on our terms, sum / counter. We have already seen both, and could calculate them in a single loop.
So, average of all values y=sin(x) in our loop:

```
sum=0
counter=0
for x = 0 to 20 step 2
    y=sin(x)
    sum=sum+y
    counter=counter+1
    print x, y
next
print "average= ";sum/counter
```

Geometric mean

If you remember some mean other then arithmetic, it likely involves counter and some operation in a loop. For example, *geometric mean* is defined as N-th root of a product (has sense only for positive values), where N is counter.

```
product=1   'initial for product always 1
counter=0
for x = 0 to 20 step 2
    y=sin(x)
```

```
    if y>0 then
        product=product*y
        counter=counter+1
    end if
    print x, y
next
print "geometric mean for positives= ";product^(1/counter)
```

Min, max

Minimum

Let's have a take on *minimum*.
Reasonable approach is:

- assign first value to minimum variable
- for all other values,
    - compare value to minimum variable
    - If it is less then current minimum, we change current minimum to value.
    - So minimum variable always has smallest of values encountered so far.
- After the loop, this gives us smallest of all values.

That would look something like this:

```
x0=0: xEnd=20: dx=2
minimum = sin(x0)   'first value
for x = x0+dx to xEnd step dx  'start from second value
    y=sin(x)
    if y < minimum then minimum = y
    print x, y
next
print "minimum= ";minimum
```

It could be simplified a bit, at cost of computer working slightly more.
First, if loop start from first value, all we have is extra calculating of sin(x0). Resulting value of minimum will not change. This leaves us with line

```
for x = x0 to xEnd step dx
```

As a bonus, having all values processed in the same manner is beneficial – you likely could reuse same loop for some other task (as we did, printing function table).
Second simplification gets rid of separate calculating first value.

Instead, we will set minimum initially to impossible value, so that first check sure change it to real one. Usual choice is something obviously big, like 1e10.
This came with a bonus, too. If we insert a filter, we cannot be sure beforehand what first value passes through filter. Getting rid of separately calculating first value solves this problem.
So our program ends up like this:

```
x0=0: xEnd=20: dx=2
minimum = 1e10   'impossible value
for x = x0 to xEnd step dx
    y=sin(x)
    if y < minimum then minimum = y
    print x, y
next
print "minimum= ";minimum
```

In LB, you can write

```
minimum = min (minimum, y)
```

instead of “if y

argMin

Pretty often you need to find not only minimum value, but an argument (or loop index) that provides that value. In math, this thing called *argMin* (from *argument minimum*). Getting it pretty straightforward: at the line where we change current minimum, we store current index:

```
if y < minimum then minimum = y: argMin = x
```

Interesting thing happens if we have several equal minimal values. Minimum found would be one and the same; but there are several possible argMins.
Which one we’ll get, depends on our test condition. If we test for “