title: The Unix Shell
subtitle: Loops
minutes: 20

The Unix Shell: Variables, Loops and Repeating Things

Learning Objectives

- Write a loop that applies one or more commands separately to each file in a set of files.
- Trace the values taken on by a loop variable during execution of the loop.
- Explain the difference between a variable's name and its value.
- Explain why spaces and some punctuation characters shouldn't be used in files' names.
- Demonstrate how to see what commands have recently been executed.
- Re-run recently executed commands without retyping them.

Wildcards and tab completion are two ways to reduce typing (and typing mistakes). Another is to tell the shell to do something over and over again.

Let's go back to our programming-fundamentals-master/data/new-york-times directory.

Before merging, let's say we'd like to make a backup directory called 'backup' and copy our articles in there, renaming each one `original-human-rights-2000.TXT` and `original-human-rights-2001.TXT`.

First we can make a directory and copy our files:

```
$ cd ~/programming-fundamentals-master/data/new-york-times
$ mkdir backup
$ cp *.txt backup
$ ls backup
```

So far so good. But when we try to rename the files, we can't use:

```
$ cd backup
$ mv *.TXT original-*.TXT
```

because that would expand to:

```
$ mv human-rights-2000.TXT human-rights-2001.TXT ... human-rights-2009.TXT
original-*.TXT
```

This wouldn't back up our files, instead we get an error

```
mv: target `original-*.TXT' is not a directory
```

This is because there are no files matching the wildcard `original-*.TXT`.
In this case, Bash passes the unexpanded wildcard as a parameter to the `mv` command.

Variables

In this case, we want to repeat the command on multiple files, but we don't want to write a new line of code with each different filename. Instead, we can create a **variable**, which we assign a value (e.g. a name), then use as a placeholder for that value in the operations we run. We can change the value assigned to the variable, in order to use the same lines of code to run the operations on different items or values.

First, we create a variable and assign it a value.

```
$ filename=human-rights-2000.TXT
```

Then we can run a command, using the variable as a placeholder for the value we stored in it. We retrieve the variable's value by putting $ in front of it.

```
$ wc -w $filename
```

```
63661 human-rights-2000.TXT
```

Notice what happens if we don't include the $: the computer tries to interpret `filename` as an actual file or folder, rather than a variable containing a real filename.

```
$ wc -w filename
```

```
wc: filename: open: No such file or directory
```

Loops

Now that we can refer to a specific thing using a variable as a placeholder, we can repeat the same operation on multiple things, using a **loop** to do the operation once for each thing in a list. Here's a simple example that displays the word counts of each file in turn:

```
$ for filename in human-rights-2000.TXT human-rights-2001.TXT
> do
>     wc -w $filename
> done
```

```
 63661 human-rights-2000.TXT
 56035 human-rights-2001.TXT
```

When the shell sees the keyword `for`, it knows it is supposed to repeat a command (or group of commands) once for each thing in a list. In this case, the list is the two filenames.

Each time through the loop, the name of the thing currently being operated on is assigned to the variable we created in the `for` line, called `filename`. Inside the loop, we get the variable's value by putting $ in front of it as before: `$filename` is `human-rights-2000.TXT` the first time through the loop, `human-rights-2001.TXT` the second, and so on.

The command that's actually being run is our old friend `wc` again,
so this loop prints out the word count of each data file in turn.

> **Follow the Prompt**
>
> The shell prompt changes from $ to > and back again as we were typing in our loop. The second prompt, >, is different to remind us that we haven't finished typing a complete command yet.

We have called the variable in this loop `filename` in order to make its purpose clearer to human readers. The shell itself doesn't care what the variable is called; if we wrote this loop as:

```
$ for x in human-rights-2000.TXT human-rights-2001.TXT
> do
>     wc -w $x
> done
```

or:

```
$ for temperature in human-rights-2000.TXT human-rights-2001.TXT
> do
>     wc -w $temperature
> done
```

it would work exactly the same way. *Don't do this.* Programs are only useful if people can understand them, so meaningless names (like x) or misleading names (like `temperature`) increase the odds that the program won't do what its readers think it does.

Let's use a for loop to backup all of our files in the new-york-times directory:

```
$ for filename in *.TXT
> do
>     mv $filename original-$filename
> done
```

More Loops

Here's a slightly more complicated loop:

```
for filename in africa*.txt
do
    echo $filename
    tail -1 $filename | wc -w
done
```

```
...
africa97.txt
     515
africa98.txt
     653
africa99.txt
     597
```

The shell starts by expanding `africa*.txt` to create the list of files it will process. The **loop body** then executes two commands for each of those files. The first, echo, just prints its command-line parameters to standard output. For example:

```
$ echo hello there
```

prints:

```
hello there
```

In this case, since the shell expands `$filename` to be the name of a file, `echo $filename` just prints the name of the file. Note that we can't write this as:

```
for filename in africa*.txt
do
    $filename
    tail -1 $filename | wc -w
done
```

because then the first time through the loop, when `$filename` expanded to `africa1.txt`, the shell would try to run `africa1.txt` as a program. Finally, the `tail` and wc combination computes the word count of the last line of each file.

Spaces in Names

Filename expansion in loops is another reason you should not use spaces in filenames. Suppose our data files are named:

```
basilisk.dat
red dragon.dat
unicorn.dat
```

If we try to process them using:

```
for filename in *.dat
do
    head -100 $filename | tail -20
done
```

then the shell will expand `*.dat` to create:

```
basilisk.dat red dragon.dat unicorn.dat
```

With older versions of Bash, or most other shells, `filename` will then be assigned the following values in turn:

```
basilisk.dat
red
dragon.dat
unicorn.dat
```

That's a problem: `head` can't read files called `red` and `dragon.dat` because they don't exist, and won't be asked to read the file `red dragon.dat`.

We can make our script a little bit more robust by **quoting** our use of the variable:

```
for filename in *.dat
do
    head -100 "$filename" | tail -20
done
```

but it's simpler just to avoid using spaces (or other special characters) in filenames.

A Loopy Solution

Going back to our original file renaming problem, we can solve it using this loop:

```
for filename in *.txt
do
    mv $filename original-$filename
done
```

This loop runs the mv command once for each filename. The first time, when $filename expands to africa1.txt, the shell executes:

```
mv africa1.txt original-africa1.txt
```

The second time, the command is:

```
mv africa2.txt original-africa2.txt
```

Measure Twice, Run Once

A loop is a way to do many things at once --- or to make many mistakes at once if it does the wrong thing. One way to check what a loop *would* do is to echo the commands it would run instead of actually running them. For example, we could write our file renaming loop like this:

```
for filename in *.txt
do
    echo mv $filename original-$filename
done
```

Instead of running mv, this loop runs echo, which prints out:

```
mv africa1.txt original-africa1.txt
mv africa2.txt original-africa2.txt
```

without actually running those commands. We can then use up-arrow to redisplay the loop, back-arrow to get to the word echo, delete it, and then press "enter" to run the loop with the actual mv commands. This isn't foolproof, but it's a handy way to see what's going to happen when you're still learning how loops work..

History

Another way to repeat previous work is to use the history command to get a list of the last few hundred commands that have been executed, and then to use !123 (where "123" is replaced by the command number) to repeat one of those commands. For example, if Rochelle types this:

```
$ history | tail -5
  474  for filename in *.txt; do sed -i".bak" '1d' $filename; done
  475  cat africa1.txt
  476  rm *.bak
  477  history | tail -5
```

then she can re-run cat africa1 simply by typing !475.

Exercises

Challenge 1

Suppose that ls initially displays:

```
fructose.dat    glucose.dat   sucrose.dat
```

What is the output of:

```
for datafile in *.dat
do
    ls *.dat
done
```

Now test your theory and redirect your answer to `programming-fundamentals/my_files/challenge_1.txt`!

Challenge 2

What is the effect of this loop if each .dat file contains only the word `sugar`?

```
for sugar in fructose.dat glucose.dat sucrose.dat
do
    echo $sugar
    cat $sugar > xylose.dat
done
```

Now test your theory and redirect the output to `programming-fundamentals/my_files/challenge_2.txt`. Then append the contents of `xylose.dat` to your `challenge_2.txt`. (HINT: use `cat` and redirection)

Challenge 3

The `expr` does simple arithmetic using command-line parameters:

```
$ expr 3 + 5
8
$ expr 30 / 5 - 2
4
```

Given this, what is the output of:

```
for left in 2 3
do
    for right in $left
    do
        expr $left + $right
    done
done
```

Now test your theory and redirect your answer to `programming-fundamentals/my_files/challenge_3.txt`!

`cd` into `test`, and type `. 1-4_test.sh` once you've completed the challenges.

Adapted from: [Software Carpentry (http://software-carpentry.org/v5/novice/shell/04-loop.html)](http://software-carpentry.org/v5/novice/shell/04-loop.html)