The Unix Shell

(index.html)

Loops

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 file 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. Suppose we have several hundred genome data files named `basilisk.dat`, `unicorn.dat`, and so on. In this example, we'll use the `creatures` directory which only has two example files, but the principles can be applied to many many more files at once. We would like to modify these files, but also save a version of the original files, naming the copies `original-basilisk.dat` and `original-unicorn.dat`. We can't use:

```
$ cp *.dat original-*.dat
```

because that would expand to:

```
$ cp basilisk.dat unicorn.dat original-*.dat
```

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

```
cp: target `original-*.dat' is not a directory
```

This a problem arises when `cp` receives more than two inputs. When this happens, it expects the last input to be a directory where it can copy all the files it was passed. Since there is no directory named `original-*.dat` in the `creatures` directory we get an error.

Instead, we can use a loop to do some operation once for each thing in a list. Here's a simple example that displays the first three lines of each file in turn:

```
$ for filename in basilisk.dat unicorn.dat
> do
>     head -3 $filename
> done
```

```
COMMON NAME: basilisk
CLASSIFICATION: basiliscus vulgaris
UPDATED: 1745-05-02
COMMON NAME: unicorn
CLASSIFICATION: equus monoceros
UPDATED: 1738-11-24
```

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 called `filename`. Inside the loop, we get the variable's value by putting `$` in front of it: `$filename` is `basilisk.dat` the first time through the loop, `unicorn.dat` the second, and so on.

By using the dollar sign we are telling the shell interpreter to treat `filename` as a variable name and substitute its value on its place, but not as some text or external command. When using variables it is also possible to put the names into curly braces to clearly delimit the variable name: `$filename` is equivalent to `${filename}`, but is different from `${file}name`. You may find this notation in other people's programs.

Finally, the command that's actually being run is our old friend `head`, so this loop prints out the first three lines 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 basilisk.dat unicorn.dat
do
    head -3 $x
done
```

or:

```
for temperature in basilisk.dat unicorn.dat
do
    head -3 $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.

Here's a slightly more complicated loop:

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

The shell starts by expanding `*.dat` 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 *.dat
do
    $filename
    head -100 $filename | tail -20
done
```

because then the first time through the loop, when `$filename` expanded to `basilisk.dat`, the shell would try to run `basilisk.dat` as a program. Finally, the `head` and `tail` combination selects lines 81-100 from whatever file is being processed.

📌 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.

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

```
for filename in *.dat
do
    cp $filename original-$filename
done
```

This loop runs the `cp` command once for each filename. The first time, when `$filename` expands to `basilisk.dat`, the shell executes:

```
cp basilisk.dat original-basilisk.dat
```

The second time, the command is:

```
cp unicorn.dat original-unicorn.dat
```

📌 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 copying loop like this:

```
for filename in *.dat
do
    echo cp $filename original-$filename
done
```

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

```
cp basilisk.dat original-basilisk.dat
cp unicorn.dat original-unicorn.dat
```

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 `cp` 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.

Nelle's Pipeline: Processing Files

Nelle is now ready to process her data files. Since she's still learning how to use the shell, she decides to build up the required commands in stages. Her first step is to make sure that she can select the right files — remember, these are ones whose names end in 'A' or 'B', rather than 'Z'. Starting from her home directory, Nelle types:

```
$ cd north-pacific-gyre/2012-07-03
$ for datafile in *[AB].txt
> do
>     echo $datafile
> done
```

```
NENE01729A.txt
NENE01729B.txt
NENE01736A.txt
...
NENE02043A.txt
NENE02043B.txt
```

Her next step is to decide what to call the files that the `goostats` analysis program will create. Prefixing each input file's name with "stats" seems simple, so she modifies her loop to do that:

```
$ for datafile in *[AB].txt
> do
>     echo $datafile stats-$datafile
> done
```

```
NENE01729A.txt stats-NENE01729A.txt
NENE01729B.txt stats-NENE01729B.txt
NENE01736A.txt stats-NENE01736A.txt
...
NENE02043A.txt stats-NENE02043A.txt
NENE02043B.txt stats-NENE02043B.txt
```

She hasn't actually run `goostats` yet, but now she's sure she can select the right files and generate the right output filenames.

Typing in commands over and over again is becoming tedious, though, and Nelle is worried about making mistakes, so instead of re-entering her loop, she presses the up arrow. In response, the shell redisplays the whole loop on one line (using semi-colons to separate the pieces):

```
$ for datafile in *[AB].txt; do echo $datafile stats-$datafile; done
```

Using the left arrow key, Nelle backs up and changes the command `echo` to `goostats`:

```
$ for datafile in *[AB].txt; do bash goostats $datafile stats-$datafile; done
```

When she presses enter, the shell runs the modified command. However, nothing appears to happen — there is no output. After a moment, Nelle realizes that since her script doesn't print anything to the screen any longer, she has no idea whether it is running, much less how quickly. She kills the job by typing Control-C, uses up-arrow to repeat the command, and edits it to read:

```
$ for datafile in *[AB].txt; do echo $datafile; bash goostats $datafile stats-$datafile;
done
```

📌 Beginning and End

We can move to the beginning of a line in the shell by typing `^A` (which means Control-A) and to the end using `^E`.

When she runs her program now, it produces one line of output every five seconds or so:

```
NENE01729A.txt
NENE01729B.txt
NENE01736A.txt
...
```

1518 times 5 seconds, divided by 60, tells her that her script will take about two hours to run. As a final check, she opens another terminal window, goes into `north-pacific-gyre/2012-07-03`, and uses `cat stats-NENE01729B.txt` to examine one of the output files. It looks good, so she decides to get some coffee and catch up on her reading.

📌 Those Who Know History Can Choose to Repeat It

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 Nelle types this:

```
$ history | tail -5
  456  ls -l NENE0*.txt
  457  rm stats-NENE01729B.txt.txt
  458  bash goostats NENE01729B.txt stats-NENE01729B.txt
  459  ls -l NENE0*.txt
  460  history
```

then she can re-run `goostats` on `NENE01729B.txt` simply by typing `!458`.

Variables in Loops

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, what is the output of:

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

Why do these two loops give you different outputs?

Saving to a File in a Loop - Part One

In the same directory, what is the effect of this loop?

```
for sugar in *.dat
do
    echo $sugar
    cat $sugar > xylose.dat
done
```

1. Prints `fructose.dat`, `glucose.dat`, and `sucrose.dat`, and the text from `sucrose.dat` will be saved to a file called `xylose.dat`.
2. Prints `fructose.dat`, `glucose.dat`, and `sucrose.dat`, and the text from all three files would be concatenated and saved to a file called `xylose.dat`.
3. Prints `fructose.dat`, `glucose.dat`, `sucrose.dat`, and `xylose.dat`, and the text from `sucrose.dat` will be saved to a file called `xylose.dat`.
4. None of the above.

Saving to a File in a Loop - Part Two

In another directory, where `ls` returns:

```
fructose.dat    glucose.dat   sucrose.dat   maltose.txt
```

What would be the output of the following loop?

```
for datafile in *.dat
do
    cat $datafile >> sugar.dat
done
```

1. All of the text from `fructose.dat`, `glucose.dat` and `sucrose.dat` would be concatenated and saved to a file called `sugar.dat`.
2. The text from `sucrose.dat` will be saved to a file called `sugar.dat`.
3. All of the text from `fructose.dat`, `glucose.dat`, `sucrose.dat` and `maltose.txt` would be concatenated and saved to a file called `sugar.dat`.
4. All of the text from `fructose.dat`, `glucose.dat` and `sucrose.dat` would be printed to the screen and saved to a file called `sugar.dat`

Doing a Dry Run

Suppose we want to preview the commands the following loop will execute without actually running those commands:

```
for file in *.dat
do
  analyze $file > analyzed-$file
done
```

What is the difference between the two loops below, and which one would we want to run?

```
# Version 1
for file in *.dat
do
  echo analyze $file > analyzed-$file
done
```

```
# Version 2
for file in *.dat
do
  echo "analyze $file > analyzed-$file"
done
```

Nested Loops and Command-Line Expressions

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
```