Loops

Overview

Teaching: 15 min
Exercises: 0 min
Questions

- How can I perform the same actions on many different files?

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.

Loops are key to productivity improvements through automation as they allow us to execute commands repetitively. Similar to wildcards and tab completion, using loops also reduces the amount of typing (and typing mistakes). 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 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 -n 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. A semicolon, `;`, can be used to separate two commands written on a single line.

📌 Same Symbols, Different Meanings

Here we see `>` being used a shell prompt, whereas `>` is also used to redirect output. Similarly, `$` is used as a shell prompt, but, as we saw earler, it is also used to ask the shell to get the value of a variable.

If the *shell* prints `>` or `$` then it expects you to type something, and the symbol is a prompt.

If *you* type `>` or `$` yourself, it is an instruction from you that the shell to redirect output or get the value of a variable.

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 -n 3 $x
done
```

or:

```
for temperature in basilisk.dat unicorn.dat
do
    head -n 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 -n 100 $filename | tail -n 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 -n 100 $filename | tail -n 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

Whitespace is used to separate the elements on the list that we are going to loop over. If on the list we have elements with whitespace we need to quote those elements and our variable when using it. Suppose our data files are named:

```
red dragon.dat
purple unicorn.dat
```

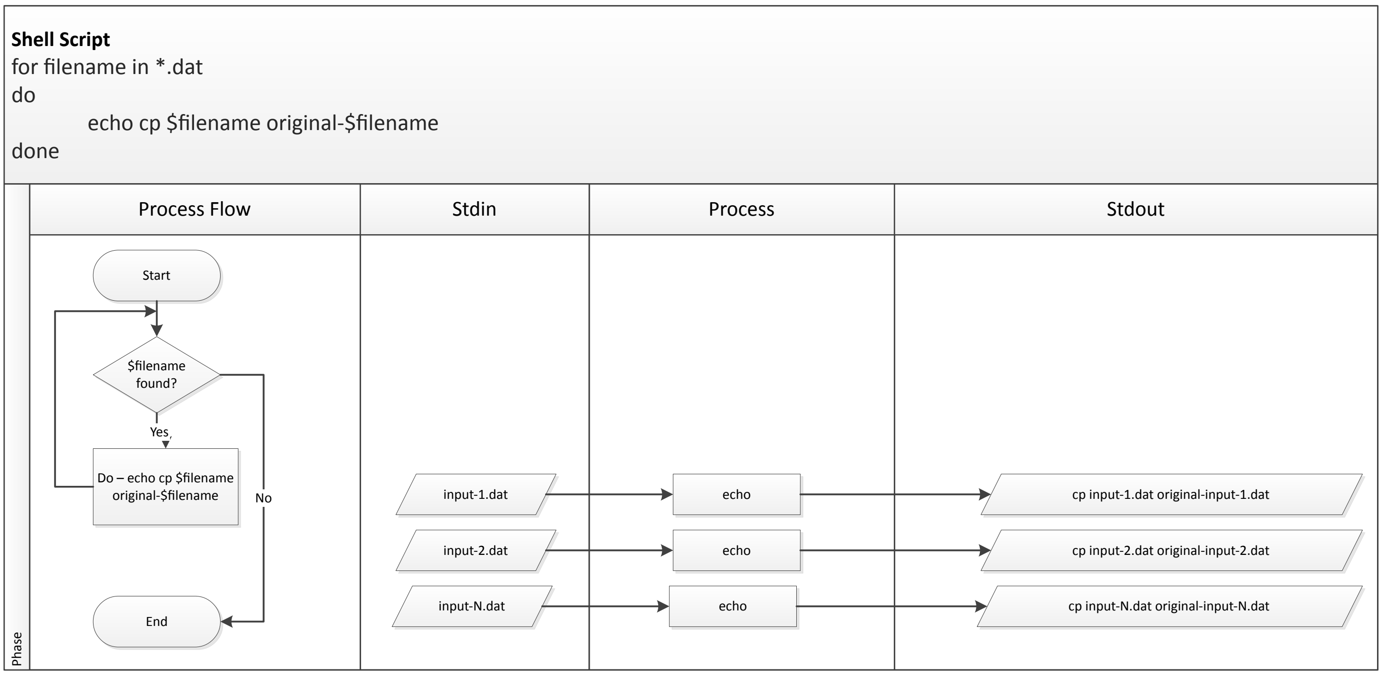
We need to use

```
for filename in "red dragon.dat" "purple unicorn.dat"
do
    head -n 100 "$filename" | tail -n 20
done
```

Is simpler just to avoid using whitespaces (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
```


Shell Script

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

Process Flow
Stdin
Process
Stdout
Phase
Start
$filename found?
Yes
No
Do – echo cp $filename original-$filename
End
input-1.dat
input-2.dat
input-N.dat
echo
echo
echo
cp input-1.dat original-input-1.dat
cp input-2.dat original-input-2.dat
cp input-N.dat original-input-N.dat

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
```

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 `bash 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 running command by typing `Ctrl-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; don
e
```

📌 Beginning and End

We can move to the beginning of a line in the shell by typing `Ctrl-A` and to the end using `Ctrl-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 -n 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`.

📌 Other History Commands

There are a number of other shortcut commands for getting at the history. Two of the more useful are `!!`, which retrieves the immediately preceding command (you may or may not find this more convenient than plain up-arrow), and `!$`, which retrieves the last word of the last command. That's useful more often than you might expect: after `bash goostats NENE01729B.txt stats-NENE01729B.txt`, you can type `less !$` to look at the file `stats-NENE01729B.txt`, which is quicker than doing up-arrow and editing the command-line.

✏ Variables in Loops

✏ Saving to a File in a Loop - Part One

✏ Saving to a File in a Loop - Part Two

Limiting Sets of Files

Doing a Dry Run

Nested Loops

Key Points

- A `for` loop repeats commands once for every thing in a list.
- Every `for` loop needs a variable to refer to the thing it is currently operating on.
- Use `$name` to expand a variable (i.e., get its value). `${name}` can also be used.
- Do not use spaces, quotes, or wildcard characters such as ‘*’ or ‘?’ in filenames, as it complicates variable expansion.
- Give files consistent names that are easy to match with wildcard patterns to make it easy to select them for looping.
- Use the up-arrow key to scroll up through previous commands to edit and repeat them.
- Use `Ctrl-R` to search through the previously entered commands.
- Use `history` to display recent commands, and `!number` to repeat a command by number.