The Internal Field Separator

In this lesson we will cover how and when to use the 'IFS' variable to protect your scripts from bugs, as well as other techniques available to achieve similar ends.

WE'LL COVER THE FOLLOWING

How Important is this Lesson?

Learning this concept will save you a great deal of time trying to figure out why your for loop is not working as expected, and will help you write more correct bash scripts that won't fail.

Files With Spaces

Create a couple of files with spaces in their names:

```
echo file1 created > "Spaces in filename1.txt"
cat "Spaces in filename1.txt"
echo file2 created > "Spaces in filename2.txt"
cat "Spaces in filename2.txt"
```

Type the above code into the terminal in this lesson.

Terminal

Note that you have to quote the filename to get the spaces in. Without the quotes, bash treats the space as a token separator. This means that it would treat the redirection as going to the file `Spaces` and not know what to do with the `in` and `filenameN.txt` tokens.

Now if you write a `for` loop over these files that just runs `ls` on each file in turn, you might not get what you expect. Type this in and see what happens:

```
for f in $(ls)
do
    ls $f
done
```

Type the above code into the terminal in this lesson.

Hmmm. The for loop has treated every word in the filenames as a separate token to look for with `ls`.

In other words, bash has treated each space as a ‘field separator’. Normally this is fine, as our for loops have items separated by spaces, like this:

```
for f in 1 2 3 4
do
    echo $f
done
```

Type the above code into the terminal in this lesson.

However, here we want the spaces to be ignored. And we can control this by setting the `IFS` shell variable.

The IFS Shell Variable

```
echo $IFS
```

Type the above code into the terminal in this lesson.

If you retained the default, then you will have seen nothing in the output, which isn’t very helpful. To see how it’s really set up, we can use `set`:

```
set | grep IFS
```

Type the above code into the terminal in this lesson.

You should see output like this:

```
    IFS=$' \t\n'
```

Recall that the `$` before the single quotes means that the variable is showing you special characters with the backlash escape notation. Above, the `IFS` variable is set to: space, tab, and newline.

Bash takes the characters in that variable and will treat any of them as a field separator, which means that the original for loop you wrote above will create from these files:

```
    Spaces in filename1.txt
    Spaces in filename2.txt
```

these list of items:

```
    Spaces
    in
    filename1.txt
    Spaces
    in
    filename2.txt
```

What we want to do is treat the spaces like any other character. We can do this by altering the `IFS` variable to remove the space and re-running the for loop:

```
IFS=$'\t\n'
for f in $(ls)
do
    ls $f
done
```

Type the above code into the terminal in this lesson.

This might sound obscure but it’s quite common in bash to perform operations over bunches of files like this:

```
find . -type f | xargs -n1 grep somestring
```

and in these cases you may have files lying around that have spaces in their names. Then your scripts can fail in ways that it can be difficult to debug when you don't know about the `IFS` variable.

Code Explanation

The `find` program used above (as the name suggests) helps you find files.

- If you just give it a folder (as above with the '.' local directory) then it will return all files and folders under that folder
- The `-type f` flag tells find to just return files, not folders
- The `xargs` command runs the command given against the files piped in
- The `-n1` flag tells xargs to run the command once per field piped in

The Null Byte as Separator

While we're on the subject, it's worth mentioning quickly a pattern that's very commonly-used to get around the above scenario.

```
find . -type f -print0 | xargs -0 -n1 grep somestring
```

By adding the `-print0` flag, find no longer uses a new line as a field separator. It uses what's called a `NUL` byte as the separator. The `NUL` byte is literally a byte of value zero. It doesn't get displayed on the screen, but can be read by `xargs` as the separator if it's given the `-0` flag. This bypasses all sorts of challenges you might get fiddling with the `IFS` variable or dealing with spaces or other odd characters in filenames.

What You Learned

- What the `IFS` shell variable does
- How to deal with filenames containing spaces and other unusual characters
- What the `find` and `xargs` programs do
- What a `NUL` byte is

What Next?

Well done! You’ve made it to the end of the ‘Scripting Bash’ section of the course. Now you are fully equipped to write and read useful shell scripts.

The next part takes what you’ve learned so far and extends it to understanding in a more advanced way how bash is used day to day on the command line.

Exercises

1) Set the `IFS` variable to various characters and demonstrate how field separation works

2) Run the output of `find . -type f` through hexdump and pick out the `NUL` bytes in the stream