Bash variables and command substitution

Using variables to refer to data, including the results of a command.

An essential feature of programming is the ability to use a name or a label to refer to some other quantity: such as a value, or a command. This is commonly referred to as **variables**.

Variables can be used, at the very least, to make code more readable for humans:

```
domain='http://www.whitehouse.gov'
path='/some/path'
base_url="$domain$path"
page='index.html'
# download http://www.whitehouse.gov/some/path/index.html
# and save to 'downloads/index.html'
curl "$base_url/$page" -o "downloads/$page"
```

However, variables really come into use in more advanced programming, when we're in a situation in which the actual values aren't known before executing a program. A variable acts as a **placeholder** that gets resolved upon actual execution time.

For example, imagine that `websites.txt` contains a list of website addresses. The following routine reads each line (via `cat`, which isn't best practice...but will do for now) into a `for` loop, which then downloads each URL:

```
for url in $(cat websites.txt); do
  curl $url > megapage.html
done
```

Variables

Basic variable usage and syntax

Setting a variable

The following command assigns `Hello World` to the variable named `var_a`, and `42` to `another_var`

```
user@host:~$ var_a="Hello World"
user@host:~$ another_var=42
```

Unlike most modern languages, Bash is pretty picky about the syntax for setting variables. In particular, no whitespace is allowed between the variable name, the equals sign, and the value.

All of these examples would cause Bash to throw an error:

```
var_a= "Hello World"
var_a = "Hello World"
var_a ="Hello World"
```

Referencing the value of a variable

Whenever Bash encounters a **dollar-sign**, immediately followed by a word, within a command or in a double-quoted string, it will attempt to replace that token with the value of the named variable. This is sometimes referred to as **expanding the variable**, or parameter substitution (http://tldp.org/LDP/abs/html/parameter-substitution.html):

```
user@host:~$ var_a="Hello World"
user@host:~$ another_var=42
user@host:~$ echo $var_a
Hello World
user@host:~$ echo $another_var
42
user@host:~$ echo $var_a$another_var
Hello World42
```

Failure to dereference

When a dollar-sign *doesn't* precede a variable name, or a variable reference is within **single-quotes**, Bash will interpret the string *literally*:

```
user@host:~$ var_a="Hello World"
user@host:~$ another_var=42
user@host:~$ echo var_a
var_a
user@host:~$ echo '$another_var'
$another_var
user@host:~$ echo "$var_a$another_var"
Hello World42
user@host:~$ echo '$var_a$another_var'
$var_a$another_var
```

Concatenating strings

Variables can be very useful for text-patterns that will be repeatedly used:

```
user@host:~$ wh_domain='http://www.whitehouse.gov'
user@host:~$ wh_path='/briefing-room/press-briefings?page='
user@host:~$ wh_base_url="$wh_domain$wh_path"
user@host:~$ curl -so 10.html "$wh_base_url=10"
user@host:~$ curl -so 20.html "$wh_base_url=20"
user@host:~$ curl -so 30.html "$wh_base_url=30"
```

If your variable name *butts* up against a literal alphanumeric character, you can use this more verbose form, involving **curly braces**, to reference a variable's value:

```
user@host:~$ BASE_BOT='R2'
user@host:~$ echo "$BASE_BOTD2"
# nothing gets printed, because $BASE_BOTD2 is interpreted
# as a variable named BASE_BOTD2, which has not been set
user@host:~$ echo "${BASE_BOT}D2"
R2D2
```

Valid variable names

Variable names can contain a sequence of alphanumeric characters and underscores. For variables created by you, the user, they should start with either an alphabetical letter or an underscore (i.e. not a number):

Valid variable names:

- `hey`
- `x9`
- `GRATUITOUSLY_LONG_NAME`
- `_secret`

When we write functions and shell scripts, in which arguments are passed in to be processed, the arguments will be passed int numerically-named variables, e.g. `$1`, `$2`, `$3`

For example:

```
bash my_script.sh Hello 42 World
```

Inside `my_script.sh`, commands will use `$1` to refer to `Hello`, `$2` to `42`, and `$3` for `World`

The variable reference, `$0`, will expand to the current script's name, e.g. `my_script.sh`

Command substitution

The standard output of a command can be encapsulated, much like a value can be stored in a value, and then expanded by the shell.

This is known as **command substitution**. From the Bash documentation (http://www.gnu.org/software/bash/manual/bashref.html#Command-Substitution):

> Command substitution allows the output of a command to replace the command itself. Bash performs the expansion by executing command and replacing the command substitution with the standard output of the command, with any trailing newlines deleted. Embedded newlines are not deleted, but they may be removed during word splitting.

As an example, consider the **seq** command, which will print a sequence of numbers starting from the first argument to the second argument:

```
user@host~:$ seq 1 5
1
2
3
4
5
```

With command substitution, we can encapsulate the result of `seq 1 5` into a variable by enclosing the command with `$(` and `)`, and pass it as an argument to another command:

```
user@host~:$ echo $(seq 1 5)
1 2 3 4 5
# Or, to create 5 new directories:
user@host~:$ mkdir $(seq 1 5)
```

As a GIF:

Variables and command expansion

When a command is replaced by its standard output, that output, presumably just text, can be assigned to a variable like any other value:

```
user@host~:$ a=$(echo 'hello' | tr '[:lower:]' '[:upper:]')
user@host~:$ b=$(echo 'WORLD' | tr '[:upper:]' '[:lower:]')
user@host~:$ echo "$a, $b"
HELLO, world
```

The loss of newlines in command substitution

Earlier, I quoted from the Bash documentation on command expansion. Here's an emphasized version of the excerpt:

> Command substitution allows the output of a command to replace the command itself. Bash performs the expansion by executing command and replacing the command substitution with the standard output of the command, **with any trailing newlines deleted. Embedded newlines are not deleted, but they may be removed during word splitting.**

What does that mean? Consider `seq 1 5` being called normally, and then, via command substitution, and note the change in formatting:

```
user@host:~$ seq 1 5
1
2
3
4
5
user@host:~$ echo $(seq 1 5)
1 2 3 4 5
```

Why do the newlines get removed during the command expansion? It's something we'll encounter later (and there's a section on it at the end of this tutorial) and deals with how Bash interprets space and newline characters during expansion. Anyway, it's worth noting the behavior for now, as it may be new to you if you're coming from another programming language.

Arithmetic expansion

To do basic calculations, you can enclose an expression inside `$(( ))`:

```
user@host:~$ echo "42 - 10 is...$(( 42 - 10))"
42 - 10 is...32
```

Check the Bash documentation for the full set of arithmetic operators (http://www.gnu.org/software/bash/manual/bashref.html#Shell-Arithmetic). Math at the command-line can be a bit clunky so we won't be focusing too much on it.

The bc utility

An aside: if you want to do more advanced math from the command line, use **bc**, which reads in from stdout and evaluates the expression:

```
user@host:~$ echo "9.45 / 2.327" | bc
4
user@host:~$ echo "9.45 / 2.327" | bc -l
4.06102277610657498925
```

Word-splitting in the wild

This section covers more technical details of how Bash handles space characters when it does an exapansion. It's not necessary to memorize for the specific assignments in this class. However, as many of you are wont to copy and paste code directly from things you've seen on the Internet, it might be worth knowing all the different ways you could accidentally harm yourself, due to the way Bash handles spaces and newline characters.

Here's the Bash documentation for the concept known as "word-splitting" (http://www.gnu.org/software/bash/manual/bashref.html#Word-Splitting)

The internal field separator

The global variable `IFS` is what Bash uses to split a string of expanded into separate *words*...think of it as how Excel knows to split a CSV (comma-separated-values) text file into a spreadsheet: it assumes the **commas** separate the columns.

Let's pretend that `IFS` has been set to something arbitrary, like a capital `Z`. When **Bash** expands a variable that happens to contain a `Z`, the value of that variable will be split into separate **words** (and the literal `Z` will disappear):

```
user@host:~$ IFS=Z
user@host:~$ story="The man named Zorro rides a Zebra"
user@host:~$ echo '>>' $story '<<'
>> The man named  orro rides a  ebra <<
```

By default, the `IFS` variable is set to three characters: **newline**, **space**, and the **tab**. If you `echo $IFS`, you won't see anything because those characters...well, how do you *see* a **space** character if there aren't any visible characters?

The upshot is that you may see code snippets online in which the `IFS` variable is changed to something like `$'\n'` (which stands for the **newline** character).

Imagine a textfile that contains a bunch of lines of text that, for example, may refer to filenames:

```
rough draft.txt
draft 1.txt
draft 2.txt
final draft.txt
```

When Bash reads each line of the file, the default value of `IFS`, which includes a **space character**, will cause Bash to treat the file named `rough draft.txt` as two files, `rough` and `draft.txt`, because the space character is used to **split** words.

With `IFS` set to just the **newline** character, `rough draft.txt` is treated as a single filename.

This concept will make sense when it comes to reading text files and operating on each line. I don't expect you to fully understand this, but only to be *aware* of it, just in case you are haphazardly copy-pasting code from the Internet.

The dangers of unquoted variables

In an ideal world, everyone would keep their string values short and without space/newline, or any other special characters. In that ideal world, the following unquoted variable reference would work just fine:

```
user@host:~$ file_to_kill='whatsup.txt'
usr@host:~$ rm $file_to_kill   # delete the file named whatsup.txt
```

But when people start adding special characters to filenames, such as spaces, expanding variables, *without the use of double quotes*, can be *dangerous*.

In the following example, the programmer intends the file named `Junk Final.docx` to be deleted:

```
user@host:~$ file_to_kill='Junk Final.docx'
```

Unexpected word-splitting

However, when referenced *without double-quotes*, Bash sees `file_to_kill` as containing two separate values, `Junk` and `Final.docx`. The subsequent `rm` command will attempt to delete *those two files*, and not `Junk Final.docx`:

```
user@host:~$ file_to_kill='Junk Final.docx'
user@host:~$ rm $file_to_kill
rm: cannot remove 'Junk': No such file or directory
rm: cannot remove 'Final.docx': No such file or directory
```

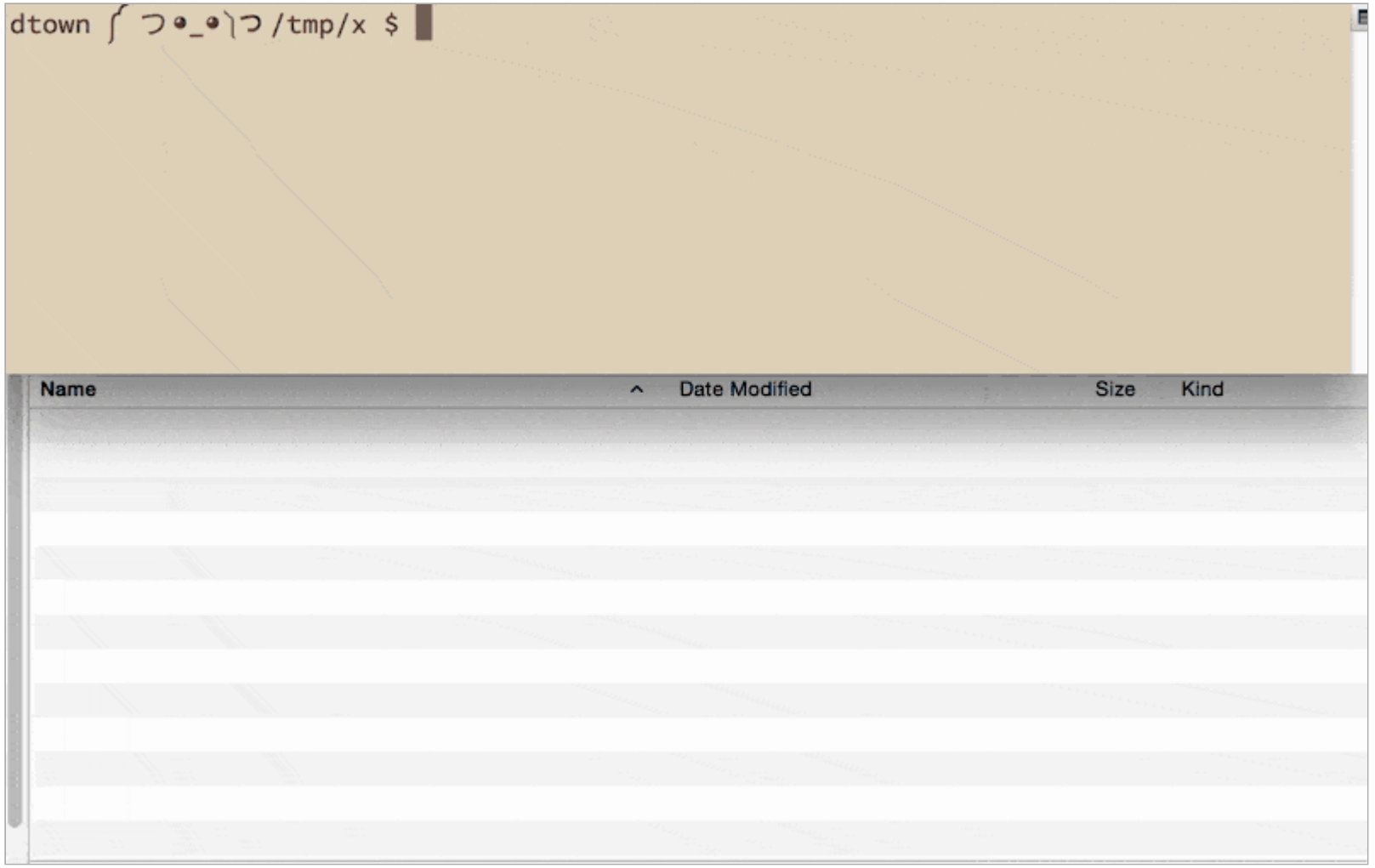
Unexpected special characters in filenames

Ah, *no harm done*, you say, because those files didn't exist in the first place. OK, but what happens when someone puts a **star** (i.e. **asterisk**) into a filename? You're aware of what happens when you do `grep *` and `rm *` – the **star** acts as a wildcard, grabbing *every* file.

```
user@host:~$ file_to_kill='Junk * Final.docx'
user@host:~$ rm $file_to_kill
```

So you'll see the previous errors, since `Junk` and `Final.docx` don't exist. But in between those attempted deletions, `rm` will run on `*` ...so say bye-bye to every file in that directory.

Here's the animated GIF version:



Notice how `rm "$filename"` affects *only* the file that is named, `* LOL BYE FILES.`

So the main takeaway here is: **double-quote your variable references whenever possible**.

To reiterate

Expanding a variable can lead to unexpected and sometimes catastrophic results if the variable contains special characters:

```
user@host:~$ do_something $some_variable
```

Expanding a variable within double-quotes can prevent such problems:

```
user@host:~$ do_something "$some_variable"
```

Who would do such a thing?

You might think, *Who the hell puts star characters in their filenames?* Well, besides people who really enjoy star-shaped symbols, malicious hackers and pranksters. And variables usually aren't just manually assigned by the result of human typing. As you've read above, sometimes the result of commands are stored in a variable. And if such commands are processing raw data, it's not unimaginable that the raw data, quite innocently, contains special characters that are destructive to certain Bash programs.

For the purposes of the CompCiv course, the assignments will try to stay far from untrusted sources of data. But keep in mind the dangers of just pasting in seemingly safe-looking code. Bash's syntax and behavior in handling strings is hard to fully comprehend, which is why developers use other languages for more complex applications.

You can read more about quoting variables (http://tldp.org/LDP/abs/html/quotingvar.html at TLDP.org). There's a lot of minutiae, but the main takeaway, besides general *safety*, is to have a general understanding how Bash, and any other programming environment, uses certain conventions and syntax rules to deal with the myriad ways that users want to pass around values in their programs.

Search

Custom Search

Quick Links

- Curriculum (/curriculum)
- Homework (/homework)
- Bash guide (/bash-guide)
- Unix tools (/unix-tools)
- Stanford Computational Journalism Lab (http://cjlab.stanford.edu)
- Software-Carpentry's guide to the Unix Shell (http://software-carpentry.org/v5/novice/shell/index.html)
- Data Science at the Command Line (http://datascienceatthecommandline.com/)

About Computational Methods in the Civic Sphere

CompCiv (/) is a Stanford Journalism course (http://journalism.stanford.edu/) taught by Dan Nguyen (http://danwin.com/)