More Bash

Environment Variables

The bash shell uses a feature called **environment variables** to store information about the shell session and the working environment. This feature allows you to store data in memory that can be easily accessed by any program or script running from the shell.

There are two environment variable types in the bash shell:

- Global variables
- Local variables

Global variables are visible from the shell session and any other subshells. Local variables are available only in the shell where they were created.

Global Environment Variables

To see the global environment variables use the `env` or `printenv` command:

```
printenv
```

`printenv` will return a long list of global variables.

Some important bash shell variables:

- `CDPATH`: a colon-separated list of directories used as a search path for the `cd` command
- `HOME`: the current user's home directory
- `PATH`: a colon-separated list of directories where the shell looks for commands
- `BASH`: the full pathname to execute the current instance of the bash shell
- `PS1`: the primary shell command line interface prompt string
- `PS2`: the secondary shell command line interface prompt string

To display a particular variable's use `printenv` followed by the name of the variable:

```
printenv HOME
```

You can also use the `echo` command to display the value of a variable. In this case you must place a dollar sign `$` before the variable name:

```
echo $HOME
```

The dollar sign before a variable name allows the variable to be passed as a command parameter:

```
ls $HOME
```

Local Environment Variables

In addition to the global environment variables, there's also the local variables. These variables can be seen only in the local process in which they are defined.

You can define your own local vaariables, better known as *user-defined* local variables.

How do you create your own variables? You can assign either a numeric or a string value to an environment variable using the equal sign `=`:

```
my_variable="Hello"

echo $my_variable
```

After `my_variable` has been created, you can reference it by the name `$my_variable`.

```
# another example
stat243="Introduction to Statistical Computing"

echo $stat243
```

It is important to not have spaces between the variable's name and the value and its assigned value. If you leave spaces, the shell will interpret the name of the variable as a command:

```
# incorrect way to set a variable
failure = "oops"
```

Setting and Removing global environment variables

The method to create a global environment variable is to first create a local variable and then export it to the global environment. This is done by using the `export` command and the variable name minus the dollar sign:

```
# setting a global variable
stat243="Introduction to Statistical Computing"

export stat243

echo $stat243

# start a child shell and testing stat243 as a global variable
bash
echo $stat243
exit

echo $stat243
```

To remove an existing environment variable you can use the `unset` command:

```
echo $stat243

unset stat243

echo $stat243
```

Setting the PATH environment variable

One of the most important variables is `PATH`. When you enter an external command, the shell must search the system to find the program. The `PATH` environment variable defines the directories it searches looking for commands and programs.

```
echo $PATH
```

If a command's or program's location is not included in the `PATH` variable, the shell cannot find it without an absolute directory reference. If the shell cannot find the command or program, it produces an error message. Try typing this and see what happens:

```
myscript
```

Often, programs and applications place their executable programs in directories that are not in the `PATH` environment variable. But you can always add (and remove) new search directories to the existing `PATH` variable. All you need to do is reference the original `PATH` value and add any new directories to the string:

```
echo $PATH

# adding more directories to PATH
PATH=$PATH:/Users/john/scripts

# check PATH again
echo $PATH
```

By adding the directory to the `PATH` environment variable, you can now execute your program from anywhere in the directory structure.

File Permissions

Working with an Operating System requires some form of security. There must be a mechanism available to protect files from unauthorized viewing or modificaiton. Unix has a method of file permissions, allowing individual users and group access to files based on a set of secutiry settings for each file and directory.

The `ls` command allows you to see the contents (directories and files) of the specified directory (e.g. current working directory by default). Adding the `-l` option, you get a list in long format of direcotires and files:

```
ls -l
```

The first field in the output describes the permissions for the files and directories. The first character in the field defines the type of the object:

- `-` for files
- `d` for directories
- `l` for links

After the first character, you see three sets of three characters. Each set of three characters defines an access permission triplet:

- `r` for read permission for the object
- `w` for write permission for the object
- `x` for execute permission for the object
- `-` if a permission is denied a dash appears in the location

The three sets relate the three levels of security for the object:

- first triplet: the owner of the object
- second triplet: the group owner
- third triplet: everyone else on the system

Permission Codes

Permissions	Octal	Description
`---`	0	No permissions
`--x`	1	Execute-only permission
`-w-`	2	Write-only permission
`-wx`	3	Write and execute permissions
`r--`	4	Read-only permission
`r-x`	5	Read and execute permissions
`rw-`	6	Read and write permissions
`rwx`	7	Read, write, and execute permissions

Changing Security Settings

The `chmod` command allows you to change the security settings for files and directories. The format of the `chmod` command is:

```
chmod options mode file
```

The `mode` parameter allows you to set the security settings. The

For instance,

```
mkdir newdir
cd newdir

# create a new file
touch newfile.txt

ls -l
```

As you can tell, `newfile.txt` has permissions `-rw-r--r--`. To change the permissions of `newfile.txt` to `rwx` at the owner level, `rw-` at the group level, and `---` to everyone else, you use the `chmod` command with the `mode` option set to octal numbers `760`

```
chmod 760 newfile.txt

ls -l
```

Instead of using the normal string of three sets of characters, the `chmod` command takes a different approach using a *symbolic* mode:

```
[ugoa...] [+-=] [rwxXstugo]
```

The first group of characters to whom the new permissions apply:

- `u` for the user
- `g` for the group
- `o` for other users (everyone else)
- `a` for all of the above

Next, a symbol is used:

- `+` add the permission to the existing permissions
- `-` subtract the permission from the existing permissions
- `=` set the permission to the value

Finally, the third symbol is the permission used for the setting:

- `X` assigns execute permissions only if the object is a directory or if it already had execute permissions
- `s` sets the UID or GID execution
- `t` saves program text
- `u` sets the permissions to the owner's permissions
- `g` sets the permissions to the group's permissions
- `o` sets the permissions to the other's permissions

Here's how to use it:

```
rm -f newfile.txt

touch newfile.txt
ls -l

chmod u+x newfile.txt
ls -l
```

Basic Shell Scripts

The power of the shell is the ability to enter multiple commands and process the results from each command, even possibly passing the results of one command to another. There are several ways in which the shell allows you to chain commands together into a single step.

One way to run commands together is by entering them on the sme prompt line, seperated with a semicolon:

```
pwd; date; who
```

Using this technique is fine for small scripts. Instead of having to manually enter the commands onto a command line, you can combine the commands into a simple text file. When you need to run the commands, just simply run the text file.

To place the commands in a text file, first you need to use a text editor to create a file and then enter the commands into the file.

When creating a shell script file, you must specify the shell you are using in the first line of the file:

```
#!/bin/bash
```

If you are not sure where the bash shell is located in your computer, then find out with:

```
echo $BASH

# or also with
which bash
```

In a normal shell script line, the has symbol `#` is used as a comment. However, the first line of a shell script file is a special case, and the hash symbol followed by the exclamation point, also known as *shebang*, tells the shell what shell to run the script under.

Open a text editor and type the following commands:

```
#!/bin/bash
# this script displays the date and who's logged on
date
who
```

Save your script in a file calles `myscript1.txt`. Before running your new shell script file you need to do a couple of things.

If you try to run the file now, you'll get an error message:

```
myscript1.txt
```

The first obstacle you need to pass is getting the bash shell to find your script file. Remember that the shell uses the `PATH` environment variable to find commands. To get the shell to find the `myscript1.txt` script, you need to do one of two things:

- Add the directory where your shell script file is located to the `PATH` environment variable.
- Or use an absolute or relative file path to reference your script file in the prompt.

For this example, we'll use the second method to tell the shell exactly where the script file is located. To reference a file in the current directory, you can use the single dot operator:

```
./myscript1.txt
-bash: ./myscript1.txt: Permission denied
```

Now there's another problem. The shell found the script file just fine, but you don;t have permission to execute the file. Take a quick look at the permissions:

```
ls -l
```

you should be able to see that `myscript1.txt` file has permissions `-rwxr--r--`; in other words, it is not executable.

The next step is to change the file permissions so you can execute it. Use the command `chmod` and then try to execute it:

```
chmod u+x myscript1.txt

ls -l

./myscript1.txt
```

Displaying Messages in Scripts

Many time you will want to add your own text messages to help the script user know what is happening. You can do this with the `echo` command. You can add `echo` statements anywhere in your shell scripts where you need to display additional information. Modify the content of the `myscript1.txt` by adding some `echo` statements:

```
#!/bin/bash
# this script displays the date and who's logged on
echo "The time and date are:"
date
echo "Who's logged into the system"
who
```

Using variables in shell scripts

Often, you will want to incorporate other data in your shell commands to process information. You can do this by using **variables**. Variables allow you to temporarily store information within the shell script.

We've already talked about environment variables. You can use these variables in your scripts. Here's an example with the `HOME` variable (remember to use the `$` before the name of the variable):

```
#!/bin/bash
# this script displays the date and who's logged on
echo "The home directory is:"
echo $HOME
echo
echo "The time and date are:"
date
echo
echo "Who's logged into the system"
who
```

Create a new script `script2.txt` with the following variables and `echo` statement:

```
#!/bin/bash
var1=price
var2=cappuccino
var3=3

echo "The $var1 of a $var2 is $var3 dollars"
```

Change the permissions of `script2.txt` and then execute it:

```
chmod u+x script2.txt

./script2.txt
```