Printing Shell and Environmental Variables

Each shell session keeps track of its own shell and environmental variables. We can access these in a few different ways.

We can see a list of all of our environmental variables by using the `env` or `printenv` commands. In their default state, they should function exactly the same:

```
printenv
```

```
SHELL=/bin/bash
TERM=xterm
USER=demouser
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40
MAIL=/var/mail/demouser
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
PWD=/home/demouser
LANG=en_US.UTF-8
SHLVL=1
HOME=/home/demouser
LOGNAME=demouser
LESSOPEN=| /usr/bin/lesspipe %s
LESSCLOSE=/usr/bin/lesspipe %s %s
_=/usr/bin/printenv
```

This is fairly typical of the output of both `printenv` and `env`. The difference between the two commands is only apparent in their more specific functionality. For instance, with `printenv`, you can requests the values of individual variables:

```
printenv SHELL
```

```
/bin/bash
```

On the other hand, `env` let's you modify the environment that programs run in by passing a set of variable definitions into a command like this:

```
env VAR1="blahblah" command_to_run command_options
```

Since, as we learned above, child processes typically inherit the environmental variables of the parent process, this gives you the opportunity to override values or add additional variables for the child.

As you can see from the output of our `printenv` command, there are quite a few environmental variables set up through our system files and processes without our input.

These show the environmental variables, but how do we see shell variables?

The `set` command can be used for this. If we type `set` without any additional parameters, we will get a list of all shell variables, environmental variables, local variables, and shell functions:

```
set
```

```
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extglob:extquote:force_fignore:histappend:interact
BASH_ALIASES=()
```

```
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
. . .
```

This is usually a huge list. You probably want to pipe it into a pager program to deal with the amount of output easily:

```
set | less
```

The amount of additional information that we receive back is a bit overwhelming. We probably do not need to know all of the bash functions that are defined, for instance.

We can clean up the output by specifying that `set` should operate in POSIX mode, which won't print the shell functions. We can execute this in a sub-shell so that it does not change our current environment:

```
(set -o posix; set)
```

This will list all of the environmental and shell variables that are defined.

We can attempt to compare this output with the output of the `env` or `printenv` commands to try to get a list of only shell variables, but this will be imperfect due to the different ways that these commands output information:

```
comm -23 <(set -o posix; set | sort) <(env | sort)
```

This will likely still include a few environmental variables, due to the fact that the `set` command outputs quoted values, while the `printenv` and `env` commands do not quote the values of strings.

This should still give you a good idea of the environmental and shell variables that are set in your session.

These variables are used for all sorts of things. They provide an alternative way of setting persistent values for the session between processes, without writing changes to a file.

Common Environmental and Shell Variables

Some environmental and shell variables are very useful and are referenced fairly often.
Here are some common environmental variables that you will come across:

- **SHELL**: This describes the shell that will be interpreting any commands you type in. In most cases, this will be bash by default, but other values can be set if you prefer other options.
- **TERM**: This specifies the type of terminal to emulate when running the shell. Different hardware terminals can be emulated for different operating requirements. You usually won't need to worry about this though.
- **USER**: The current logged in user.
- **PWD**: The current working directory.
- **OLDPWD**: The previous working directory. This is kept by the shell in order to switch back to your previous directory by running `cd -`.
- **LS_COLORS**: This defines color codes that are used to optionally add colored output to the `ls` command. This is used to distinguish different file types and provide more info to the user at a glance.
- **MAIL**: The path to the current user's mailbox.

- **PATH**: A list of directories that the system will check when looking for commands. When a user types in a command, the system will check directories in this order for the executable.
- **LANG**: The current language and localization settings, including character encoding.
- **HOME**: The current user's home directory.
- **_**: The most recent previously executed command.

In addition to these environmental variables, some shell variables that you'll often see are:

- **BASHOPTS**: The list of options that were used when bash was executed. This can be useful for finding out if the shell environment will operate in the way you want it to.
- **BASH_VERSION**: The version of bash being executed, in human-readable form.
- **BASH_VERSINFO**: The version of bash, in machine-readable output.
- **COLUMNS**: The number of columns wide that are being used to draw output on the screen.
- **DIRSTACK**: The stack of directories that are available with the `pushd` and `popd` commands.
- **HISTFILESIZE**: Number of lines of command history stored to a file.
- **HISTSIZE**: Number of lines of command history allowed in memory.
- **HOSTNAME**: The hostname of the computer at this time.
- **IFS**: The internal field separator to separate input on the command line. By default, this is a space.
- **PS1**: The primary command prompt definition. This is used to define what your prompt looks like when you start a shell session. The `PS2` is used to declare secondary prompts for when a command spans multiple lines.
- **SHELLOPTS**: Shell options that can be set with the `set` option.
- **UID**: The UID of the current user.

Setting Shell and Environmental Variables

To better understand the difference between shell and environmental variables, and to introduce the syntax for setting these variables, we will do a small demonstration.

Creating Shell Variables

We will begin by defining a shell variable within our current session. This is easy to accomplish; we only need to specify a name and a value. We'll adhere to the convention of keeping all caps for the variable name, and set it to a simple string.

```
TEST_VAR='Hello World!'
```

Here, we've used quotations since the value of our variable contains a space. Furthermore, we've used single quotes because the exclamation point is a special character in the bash shell that normally expands to the bash history if it is not escaped or put into single quotes.

We now have a shell variable. This variable is available in our current session, but will not be passed down to child processes.

We can see this by grepping for our new variable within the `set` output:

```
set | grep TEST_VAR
```

```
TEST_VAR='Hello World!'
```

We can verify that this is not an environmental variable by trying the same thing with `printenv`:

```
printenv | grep TEST_VAR
```

No output should be returned.

Let's take this as an opportunity to demonstrate a way of accessing the value of any shell or environmental variable.

```
echo $TEST_VAR
```

```
Hello World!
```

As you can see, reference the value of a variable by preceding it with a `$` sign. The shell takes this to mean that it should substitute the value of the variable when it comes across this.

So now we have a shell variable. It shouldn't be passed on to any child processes. We can spawn a *new* bash shell from within our current one to demonstrate:

```
bash
echo $TEST_VAR
```

If we type `bash` to spawn a child shell, and then try to access the contents of the variable, nothing will be returned. This is what we expected.

Get back to our original shell by typing `exit`:

```
exit
```

Creating Environmental Variables

Now, let's turn our shell variable into an environmental variable. We can do this by *exporting* the variable. The command to do so is appropriately named:

```
export TEST_VAR
```

This will change our variable into an environmental variable. We can check this by checking our environmental listing again:

```
printenv | grep TEST_VAR
```

```
TEST_VAR=Hello World!
```

This time, our variable shows up. Let's try our experiment with our child shell again:

```
bash
echo $TEST_VAR
```

```
Hello World!
```

Great! Our child shell has received the variable set by its parent. Before we exit this child shell, let's try to export another variable. We can set environmental variables in a single step like this:

```
export NEW_VAR="Testing export"
```

Test that it's exported as an environmental variable:

```
printenv | grep NEW_VAR
```

```
NEW_VAR=Testing export
```

Now, let's exit back into our original shell:

```
exit
```

Let's see if our new variable is available:

```
echo $NEW_VAR
```

Nothing is returned.

This is because environmental variables are only passed to child processes. There isn't a built-in way of setting environmental variables of the parent shell. This is good in most cases and prevents programs from affecting the operating environment from which they were called.

The `NEW_VAR` variable was set as an environmental variable in our child shell. This variable would be available to itself and any of **its** child shells and processes. When we exited back into our main shell, that environment was destroyed.

Demoting and Unsetting Variables

We still have our `TEST_VAR` variable defined as an environmental variable. We can change it back into a shell variable by typing:

```
export -n TEST_VAR
```

It is no longer an environmental variable:

```
printenv | grep TEST_VAR
```

However, it is still a shell variable:

```
set | grep TEST_VAR
```

```
TEST_VAR='Hello World!'
```

If we want to completely unset a variable, either shell or environmental, we can do so with the `unset` command:

```
unset TEST_VAR
```

We can verify that it is no longer set:

```
echo $TEST_VAR
```

Nothing is returned because the variable has been unset.

Setting Environmental Variables at Login

We've already mentioned that many programs use environmental variables to decide the specifics of how to operate. We do not want to have to set important variables up every time we start a new shell session, and we have already seen how many variables are already set upon login, so how do we make and define variables automatically?

This is actually a more complex problem than it initially seems, due to the numerous configuration files that the bash shell reads depending on how it is started.

The Difference between Login, Non-Login, Interactive, and Non-Interactive Shell Sessions

The bash shell reads different configuration files depending on how the session is started.

One distinction between different sessions is whether the shell is being spawned as a "login" or "non-login" session.

A **login** shell is a shell session that begins by authenticating the user. If you are signing into a terminal session or through SSH and authenticate, your shell session will be set as a "login" shell.

If you start a new shell session from within your authenticated session, like we did by calling the `bash` command from the terminal, a **non-login** shell session is started. You were were not asked for your authentication details when you started your child shell.

Another distinction that can be made is whether a shell session is interactive, or non-interactive.

An **interactive** shell session is a shell session that is attached to a terminal. A **non-interactive** shell session is one is not attached to a terminal session.

So each shell session is classified as either login or non-login and interactive or non-interactive.

A normal session that begins with SSH is usually an interactive login shell. A script run from the command line is usually run in a non-interactive, non-login shell. A terminal session can be any combination of these two properties.

Whether a shell session is classified as a login or non-login shell has implications on which files are read to initialize the shell session.

A session started as a login session will read configuration details from the `/etc/profile` file first. It will then look for the first login shell configuration file in the user's home directory to get user-specific configuration details.

It reads the first file that it can find out of `~/.bash_profile`, `~/.bash_login`, and `~/.profile` and does not read any further files.

In contrast, a session defined as a non-login shell will read `/etc/bash.bashrc` and then the user-specific `~/.bashrc` file to build its environment.

Non-interactive shells read the environmental variable called `BASH_ENV` and read the file specified to define the new environment.

Implementing Environmental Variables

As you can see, there are a variety of different files that we would usually need to look at for placing our settings.

This provides a lot of flexibility that can help in specific situations where we want certain settings in a login shell, and other settings in a non-login shell. However, most of the time we will want the same settings in both situations.

Fortunately, most Linux distributions configure the login configuration files to source the non-login configuration files. This means that you can define environmental variables that you want in both inside the non-login configuration files. They will then be read in both scenarios.

We will usually be setting user-specific environmental variables, and we usually will want our settings to be available in both login and non-login shells. This means that the place to define these variables is in the `~/.bashrc` file.

Open this file now:

```
nano ~/.bashrc
```

This will most likely contain quite a bit of data already. Most of the definitions here are for setting bash options, which are unrelated to environmental variables. You can set environmental variables just like you would from the command line:

```
export VARNAME=value
```

Any new environmental variables can be added anywhere in the `~/.bashrc` file, as long as they aren't placed in the middle of another command or for loop. We can then save and close the file. The next time you start a shell session, your environmental variable declaration will be read and passed on to the shell environment. You can force your current session to read the file now by typing:

```
source ~/.bashrc
```

If you need to set system-wide variables, you may want to think about adding them to `/etc/profile`, `/etc/bash.bashrc`, or `/etc/environment`.