(../07-aliases/)

Extra Unix Shell Material (../)

(../)

Shell Variables

Overview

Teaching: 10 min
Exercises: 0 min
Questions

- How to change shell variables

Objectives

- Understanding shell variables

The shell is just a program, and like other programs, it has variables. Those variables control its execution, so by changing their values you can change how the shell and other programs behave.

Let's start by running the command `set` and looking at some of the variables in a typical shell session:

```
$ set
```

```
COMPUTERNAME=TURING
HOME=/home/vlad
HOMEDRIVE=C:
HOSTNAME=TURING
HOSTTYPE=i686
NUMBER_OF_PROCESSORS=4
OS=Windows_NT
PATH=/Users/vlad/bin:/usr/local/git/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin
PWD=/home/vlad
UID=1000
USERNAME=vlad
...
```

As you can see, there are quite a few—in fact, four or five times more than what's shown here. And yes, using `set` to *show* things might seem a little strange, even for Unix, but if you don't give it any arguments, it might as well show you things you *could* set.

Every variable has a name. By convention, variables that are always present are given upper-case names. All shell variables' values are strings, even those (like `UID`) that look like numbers. It's up to programs to convert these strings to other types when necessary. For example, if a program wanted to find out how many processors the computer had, it would convert the value of the `NUMBER_OF_PROCESSORS` variable from a string to an integer.

Similarly, some variables (like `PATH`) store lists of values. In this case, the convention is to use a colon ':' as a separator. If a program wants the individual elements of such a list, it's the program's responsibility to split the variable's string value into pieces.

The `PATH` Variable

Let's have a closer look at that `PATH` variable. Its value defines the shell's search path (../reference/#search-path), i.e., the list of directories that the shell looks in for runnable programs when you type in a program name without specifying what directory it is in.

For example, when we type a command like `analyze`, the shell needs to decide whether to run `./analyze` or `/bin/analyze`. The rule it uses is simple: the shell checks each directory in the `PATH` variable in turn, looking for a program with the requested name in that directory. As soon as it finds a match, it stops searching and runs the program.

To show how this works, here are the components of `PATH` listed one per line:

```
/Users/vlad/bin
/usr/local/git/bin
/usr/bin
/bin
/usr/sbin
/sbin
/usr/local/bin
```

On our computer, there are actually three programs called `analyze` in three different directories: `/bin/analyze`, `/usr/local/bin/analyze`, and `/users/vlad/analyze`. Since the shell searches the directories in the order they're listed in `PATH`, it finds `/bin/analyze` first and runs that. Notice that it will *never* find the program `/users/vlad/analyze` unless we type in the full path to the program, since the directory `/users/vlad` isn't in `PATH`.

Showing the Value of a Variable

Let's show the value of the variable `HOME`:

```
$ echo HOME
```

```
HOME
```

That just prints "HOME", which isn't what we wanted (though it is what we actually asked for). Let's try this instead:

```
$ echo $HOME
```

```
/home/vlad
```

The dollar sign tells the shell that we want the *value* of the variable rather than its name. This works just like wildcards: the shell does the replacement *before* running the program we've asked for. Thanks to this expansion, what we actually run is `echo /home/vlad`, which displays the right thing.

Creating and Changing Variables

Creating a variable is easy—we just assign a value to a name using “=”:

```
$ SECRET_IDENTITY=Dracula
$ echo $SECRET_IDENTITY
```

```
Dracula
```

To change the value, just assign a new one:

```
$ SECRET_IDENTITY=Camilla
$ echo $SECRET_IDENTITY
```

```
Camilla
```

If we want to set some variables automatically every time we run a shell, we can put commands to do this in a file called `.bashrc` in our home directory. (The ‘.’ character at the front prevents `ls` from listing this file unless we specifically ask it to using `-a` : we normally don’t want to worry about it. The “rc” at the end is an abbreviation for “run control”, which meant something really important decades ago, and is now just a convention everyone follows without understanding why.)

For example, here are two lines in `/home/vlad/.bashrc` :

```
export SECRET_IDENTITY=Dracula
export TEMP_DIR=/tmp
export BACKUP_DIR=$TEMP_DIR/backup
```

These three lines create the variables `SECRET_IDENTITY` , `TEMP_DIR` , and `BACKUP_DIR` , and export them so that any programs the shell runs can see them as well. Notice that `BACKUP_DIR` ’s definition relies on the value of `TEMP_DIR` , so that if we change where we put temporary files, our backups will be relocated automatically.

While we’re here, it’s also common to use the `alias` command to create shortcuts for things we frequently type. For example, we can define the alias `backup` to run `/bin/zback` with a specific set of arguments:

```
alias backup=/bin/zback -v --nostir -R 20000 $HOME $BACKUP_DIR
```

As you can see, aliases can save us a lot of typing, and hence a lot of typing mistakes. You can find interesting suggestions for other aliases and other bash tricks by searching for “sample bashrc” in your favorite search engine.

Key Points

- FIXME

‹ (../07-aliases/)

^ (../)

Edit on GitHub (https://github.com/swcarpentry/shell-extras/edit/gh-pages/_episodes/08-environment-variables.md) / Source (https://github.com/swcarpentry/shell-extras/) / Contributing (../contributing/) / Cite (../citation/) / Contact (mailto:lessons@software-carpentry.org)