# How to determine Shell

You can get the name of your shell prompt, with following command :

**Syntax:**

echo $SHELL

# Shell Scripting She-bang

The sign #**!** is called she-bang and is written at top of the script. It passes instruction to program **/bin/sh.**

To run your script in a certain shell (shell should be supported by your system), start your script with #! followed by the shell name.

**myscript.sh**

**Example:**

#!/bin/bash

echo Hello World

#!/bin/ksh

echo Hello World

Typographical Conventions Used in This Tutorial

Significant words will be written in *italics* when mentioned for the first time.

Code segments and script output will be displayed as monospaced text.  
Command-line entries will be preceded by the Dollar sign ($). If your prompt is different, enter the command:

PS1="$ " ; export PS1

Then your interactions should match the examples given (such as **./my-script.sh** below).   
Script output (such as "Hello World" below) is displayed at the start of the line.

$ cat > newfile.sh

echo “hello”

$ echo '#!/bin/sh' > my-script.sh

$ echo 'echo Hello World' >> my-script.sh

$ chmod 755 my-script.sh

$ ./my-script.sh

Hello World

$

my-script.sh

#!/bin/sh

# This is a comment!

echo Hello World # This is a comment, too!

Note that to make a file executable, you must set the eXecutable bit, and for a shell script, the Readable bit must also be set:

Permission: rwx = 7

$ chmod a+rx my-script.sh

$ ./my-script.sh

Shell script programming has a bit of a bad press amongst some Unix systems administrators. This is normally because of one of two things:

* The speed at which an interpreted program will run as compared to a C program, or even an interpreted Perl program.
* Since it is easy to write a simple batch-job type shell script, there are a lot of poor quality shell scripts around.

It is partly due to this that there is a certain machismo associated with creating *good* shell scripts. Scripts which can be used as CGI programs, for example, without losing out too much in speed to Perl (though both would lose to C, in many cases, were speed the only criterion).   
There are a number of factors which can go into good, clean, quick, shell scripts.

* The most important criteria must be a clear, readable layout.
* Second is avoiding unnecessary commands.

A clear layout makes the difference between a shell script appearing as "black magic" and one which is easily maintained and understood.   
You may be forgiven for thinking that with a simple script, this is not too significant a problem, but two things here are worth bearing in mind.

1. First, a simple script will, more often than anticipated, grow into a large, complex one.
2. Secondly, if nobody else can understand how it works, you will be lumbered with maintaining it yourself for the rest of your life!

Something about shell scripts seems to make them particularly likely to be badly indented, and since the main control structures are if/then/else and loops, indentation is critical for understanding what a script does.

One weakness in many shell scripts is lines such as:

cat /tmp/myfile | grep "mystring"

which would run much faster as:

grep "mystring" /tmp/myfile

Not much, you may consider; the OS has to load up the **/bin/grep** executable, which is a reasonably small 75600 bytes on my system, open a **pipe** in memory for the transfer, load and run the **/bin/cat** executable, which is an even smaller 9528 bytes on my system, attach it to the input of the pipe, and let it run.

Of course, this kind of thing is what the OS is there for, and it's normally pretty efficient at doing it. But if this command were in a loop being run many times over, the saving of not locating and loading the **cat** executable, setting up and releasing the pipe, can make some difference, especially in, say, a CGI environment where there are enough other factors to slow things down without the script itself being too much of a hurdle. Some Unices are more efficient than others at what they call "building up and tearing down processes" - i.e., loading them up, executing them, and clearing them away again. But however good your flavour of Unix is at doing this, it'd rather not have to do it at all.

As a result of this, you may hear mention of the Useless Use of Cat Award (UUoC), also known in some circles as **The Award For The Most Gratuitous Use Of The Word Cat In A Serious Shell Script** being bandied about on the **comp.unix.shell** newsgroup from time to time. This is purely a way of peers keeping each other in check, and making sure that things are done right.

# A First Script

For our first shell script, we'll just write a script which says "Hello World". We will then try to get more out of a Hello World program than any other tutorial you've ever read :-)   
Create a file (first.sh) as follows:

first.sh

#!/bin/sh

# This is a comment!

echo Hello World # This is a comment, too!

# Variables - Part I

Just about every programming language in existence has the concept of *variables* - a symbolic name for a chunk of memory to which we can assign values, read and manipulate its contents. The Bourne shell is no exception, and this section introduces that idea. This is taken further in Variables - Part II which looks into variables which are set for us by the environment.   
Let's look back at our first Hello World example. This could be done using variables (though it's such a simple example that it doesn't really warrant it!)   
Note that there must be no spaces around the "=" sign: **VAR=value** works; **VAR = value** doesn't work. In the first case, the shell sees the "=" symbol and treats the command as a variable assignment. In the second case, the shell assumes that VAR must be the name of a command and tries to execute it.  
If you think about it, this makes sense - how else could you tell it to run the command VAR with its first argument being "=" and its second argument being "value"?   
Enter the following code into var.sh:

var.sh

#!/bin/sh  
MY\_MESSAGE="Hello World"

N1 = 111

N2 =333

echo $MY\_MESSAGE

echo $N1

echo $N2

echo $N1 + $N2

The shell does not care about types of variables; they may store strings, integers, real numbers - anything you like.   
People used to Perl may be quite happy with this; if you've grown up with C, Pascal, or worse yet Ada, this may seem quite strange.   
In truth, these are all stored as strings, but routines which expect a number can treat them as such.   
If you assign a string to a variable then try to add 1 to it, you will not get away with it:

$ x="hello"

A = $N1 + $N2   
$ expr $x + 1  
expr: non-numeric argument  
$

This is because the external program **expr** only expects numbers. But there is no syntactic difference between:

MY\_MESSAGE="Hello World"  
MY\_SHORT\_MESSAGE=hi  
MY\_NUMBER=1  
MY\_PI=3.142  
MY\_OTHER\_PI="3.142"  
MY\_MIXED=123abc

var2.sh

#!/bin/sh  
echo What is your FIRST name?  
read MY\_FNAME

echo What is your LAST name?

read MY\_LNAME

echo "Hello $MY\_FNAME $MY\_LNAME - hope you're well."

Mario Bacinsky kindly pointed out to me that I had originally missed out the double-quotes in line 3, which meant that the single-quote in the word "you're" was unmatched, causing an error. It is this kind of thing which can drive a shell programmer crazy, so watch out for them!

This is using the shell-builtin command **read** which reads a line from standard input into the variable supplied.   
Note that even if you give it your full name and don't use double quotes around the **echo** command, it still outputs correctly. How is this done? With the **MY\_MESSAGE** variable earlier we had to put double quotes around it to set it.   
What happens, is that the **read** command automatically places quotes around its input, so that spaces are treated correctly. (You will need to quote the output, of course - e.g. **echo "$MY\_MESSAGE"**).

## Scope of Variables

Variables in the Bourne shell do not have to be declared, as they do in languages like C. But if you try to read an undeclared variable, the result is the empty string. You get no warnings or errors. This can cause some subtle bugs - if you assign   
**MY\_OBFUSCATED\_VARIABLE=Hello**  
and then   
**echo $MY\_OSFUCATED\_VARIABLE**   
Then you will get nothing (as the second OBFUSCATED is mis-spelled).

There is a command called **export** which has a fundamental effect on the scope of variables. In order to really know what's going on with your variables, you will need to understand something about how this is used.

Create a small shell script, **myvar2.sh**:

myvar2.sh

#!/bin/sh  
echo "MYVAR is: $MYVAR"  
MYVAR="hi there"  
echo "MYVAR is: $MYVAR"

Now run the script:

$ ./myvar2.sh  
MYVAR is:  
MYVAR is: hi there

MYVAR hasn't been set to any value, so it's blank. Then we give it a value, and it has the expected result.   
Now run:

$ MYVAR=hello  
$ ./myvar2.sh  
MYVAR is:  
MYVAR is: hi there

It's still not been set! What's going on?!   
When you call **myvar2.sh** from your interactive shell, a new shell is spawned to run the script. This is partly because of the **#!/bin/sh** line at the start of the script, which we discussed earlier.   
We need to **export** the variable for it to be inherited by another program - including a shell script. Type:

$ export MYVAR  
$ ./myvar2.sh  
MYVAR is: hello  
MYVAR is: hi there

Now look at line 3 of the script: this is changing the value of **MYVAR**. But there is no way that this will be passed back to your interactive shell. Try reading the value of **MYVAR**:

$ echo $MYVAR  
hello  
$

Once the shell script exits, its environment is destroyed. But **MYVAR** keeps its value of **hello** within your interactive shell.   
In order to receive environment changes back from the script, we must *source* the script - this effectively runs the script within our own interactive shell, instead of spawning another shell to run it.   
We can source a script via the "." (dot) command:

$ MYVAR=hello  
$ echo $MYVAR  
hello  
$ . ./myvar2.sh  
MYVAR is: hello  
MYVAR is: hi there  
$ echo $MYVAR  
hi there

The change has now made it out into our shell again! This is how your **.profile** or **.bash\_profile** file works, for example.  
Note that in this case, we don't need to **export MYVAR**.   
Thanks to *sway* for pointing out that I'd originally said **echo MYVAR** above, not **echo $MYVAR** as it should be. Another example of an easy mistake to make with shell scripts. One other thing worth mentioning at this point about variables, is to consider the following shell script:

#!/bin/sh  
echo "What is your name?"  
read USER\_NAME  
echo "Hello $USER\_NAME"  
echo "I will create you a file called $USER\_NAME\_file"  
touch $USER\_NAME\_file

Think about what result you would expect. For example, if you enter "steve" as your USER\_NAME, should the script create **steve\_file**?  
Actually, no. This will cause an error unless there is a variable called **USER\_NAME\_file**. The shell does not know where the variable ends and the rest starts. How can we define this?   
The answer is, that we enclose the variable itself in *curly brackets*:

user.sh

#!/bin/sh  
echo "What is your name?"  
read USER\_NAME  
echo "Hello $USER\_NAME"  
echo "I will create you a file called ${USER\_NAME}\_file"  
touch "${USER\_NAME}\_file"

# Wildcards

Wildcards are really nothing new if you have used Unix at all before.   
It is not necessarily obvious how they are useful in shell scripts though. This section is really just to get the old grey cells thinking how things look when you're in a shell script - predicting what the effect of using different syntaxes are. This will be used later on, particularly in the Loops section.   
Think first how you would copy all the files from **/tmp/a** into **/tmp/b**. All the .txt files? All the .html files?   
Hopefully you will have come up with:

$ cp /tmp/a/\* /tmp/b/  
$ cp /tmp/a/\*.txt /tmp/b/  
$ cp /tmp/a/\*.html /tmp/b/

Now how would you list the files in **/tmp/a/** without using **ls /tmp/a/**?   
How about **echo /tmp/a/\***? What are the two key differences between this and the **ls** output? How can this be useful? Or a hinderance?   
How could you rename all .txt files to .bak? Note that

$ mv \*.txt \*.bak

will not have the desired effect; think about how this gets expanded by the shell before it is passed to **mv**. Try this using **echo** instead of **mv** if this helps.   
We will look into this further later on, as it uses a few concepts not yet covered.

# Escape Characters

Certain characters are significant to the shell; we have seen, for example, that the use of double quotes (") characters affect how spaces and TAB characters are treated, for example:

$ echo Hello World

Hello World

$ echo "Hello World"

Hello World

So how do we display: **Hello    "World"** ?

$ echo "Hello \"World\""

The first and last " characters wrap the whole lot into one parameter passed to **echo** so that the spacing between the two words is kept as is. But the code:

$ echo "Hello " World ""

would be interpreted as three parameters:

1. "Hello   "
2. World
3. ""

So the output would be

Hello World

Note that we lose the quotes entirely. This is because the first and second quotes mark off the Hello and following spaces; the second argument is an unquoted "World" and the third argument is the empty string; "".

Thanks to Patrick for pointing out that this:

$ echo "Hello "World""

is actually only one parameter (no spaces between the quoted parameters), and that you can test this by replacing the **echo** command with (for example) **ls**.

Most characters (**\***, **'**, etc) are not interpreted (ie, they are taken literally) by means of placing them in double quotes (""). They are taken as is and passed on to the command being called. An example using the asterisk (\*) goes:

$ echo \*

case.shtml escape.shtml first.shtml

functions.shtml hints.shtml index.shtml

ip-primer.txt raid1+0.txt

$ echo \*txt

ip-primer.txt raid1+0.txt

$ echo "\*"

\*

$ echo "\*txt"

\*txt

In the first example, \* is expanded to mean all files in the current directory.  
In the second example, \*txt means all files ending in **txt**.  
In the third, we put the \* in double quotes, and it is interpreted literally.  
In the fourth example, the same applies, but we have appended **txt** to the string.

However, **"**, **$**, **`**, and **\** are still interpreted by the shell, even when they're in double quotes.   
The backslash (\) character is used to mark these special characters so that they are not interpreted by the shell, but passed on to the command being run (for example, **echo**).   
So to output the string: (Assuming that the value of **$X** is 5):

A quote is ", backslash is \, backtick is `.

A few spaces are and dollar is $. $X is 5.

we would have to write:

$ echo "A quote is \", backslash is \\, backtick is \`."

A quote is ", backslash is \, backtick is `.

$ echo "A few spaces are ; dollar is \$. \$X is ${X}."

A few spaces are ; dollar is $. $X is 5.

We have seen why the " is special for preserving spacing. Dollar (**$**) is special because it marks a variable, so **$X** is replaced by the shell with the contents of the variable **X**. Backslash (**\**) is special because it is itself used to mark other characters off; we need the following options for a complete shell:

$ echo "This is \\ a backslash"

This is \ a backslash

$ echo "This is \" a quote and this is \\ a backslash"

This is " a quote and this is \ a backslash

So backslash itself must be escaped to show that it is to be taken literally. The other special character, the backtick, is discussed later in Chapter

# Loops

Most languages have the concept of loops: If we want to repeat a task twenty times, we don't want to have to type in the code twenty times, with maybe a slight change each time.   
As a result, we have **for** and **while** loops in the Bourne shell. This is somewhat fewer features than other languages, but nobody claimed that shell programming has the power of C.

## For Loops

**for** loops iterate through a set of values until the list is exhausted:

for.sh

#!/bin/sh

for i in 1 2 3 4 5

do

echo "Looping ... number $i"

done

Try this code and see what it does. Note that the values can be anything at all:

for2.sh

#!/bin/sh

for i in hello 1 \* 2 goodbye

do

echo "Looping ... i is set to $i"

done

This is well worth trying. Make sure that you understand what is happening here. Try it without the **\*** and grasp the idea, then re-read the Wildcards section and try it again with the **\*** in place. Try it also in different directories, and with the **\*** surrounded by double quotes, and try it preceded by a backslash (**\***)

In case you don't have access to a shell at the moment (it is very useful to have a shell to hand whilst reading this tutorial), the results of the above two scripts are:

Looping .... number 1

Looping .... number 2

Looping .... number 3

Looping .... number 4

Looping .... number 5

and, for the second example:

Looping ... i is set to hello

Looping ... i is set to 1

Looping ... i is set to (name of first file in current directory)

... etc ...

Looping ... i is set to (name of last file in current directory)

Looping ... i is set to 2

Looping ... i is set to goodbye

So, as you can see, **for** simply loops through whatever input it is given, until it runs out of input.

Advertise Here

## While Loops

**while** loops can be much more fun! (depending on your idea of fun, and how often you get out of the house... )

while.sh

#!/bin/sh

INPUT\_STRING=yes

while [ "$INPUT\_STRING" != "no" ]

do

echo "Please type something in (no to quit)"

read INPUT\_STRING

echo “enter data”

read data

echo "You typed: $data"

done

What happens here, is that the echo and read statements will run indefinitely until you type "bye" when prompted.   
Review Variables - Part I to see why we set **INPUT\_STRING=hello** before testing it. This makes it a repeat loop, not a traditional while loop.

The colon (**:**) always evaluates to true; whilst using this can be necessary sometimes, it is often preferable to use a real exit condition. Compare quitting the above loop with the one below; see which is the more elegant. Also think of some situations in which each one would be more useful than the other:

while2.sh

#!/bin/sh

while :

do

echo "Please type something in (^C to quit)"

read INPUT\_STRING

echo "You typed: $INPUT\_STRING"

done

Another useful trick is the **while read f** loop. This example uses the case statement, which we'll cover later. It reads from the file **myfile**, and for each line, tells you what language it thinks is being used. Each line must end with a LF (newline) - if **cat myfile** doesn't end with a blank line, that final line will not be processed.

while3a.sh

#!/bin/sh

while read f

do

case $f in

hello) echo English ;;

howdy) echo American ;;

gday) echo Australian ;;

bonjour) echo French ;;

"guten tag") echo German ;;

\*) echo Unknown Language: $f

;;

esac

done < myfile

On many Unix systems, this can also be done as:

while3b.sh

#!/bin/sh

while f=`line`

do

.. process f ..

done < myfile

But since the **while read f** works with any \*nix, and doesn't depend on the external program **line**, the former is preferable. See External Programs to see why this method uses the backtick (`).  
Had I referred to **$i** (not **$f**) in the default ("Unknown Language") case above, there would have been no warnings or errors, even though **$i** has not been declared or defined. For example:

$ i=THIS\_IS\_A\_BUG

$ export i

$ ./while3.sh something

Unknown Language: THIS\_IS\_A\_BUG

$

So make sure that you avoid typos. This is also another good reason for using **${x}** and not just **$x** - if **x="A"**and you want to say "A1", you need **echo ${x}1**, as **echo $x1** will try to use the variable **x1**, which may not exist, or may be set to "**B2**," or anything else unexpected.

I recently found an old thread on Usenet which I had been involved in, where I actually learned more ... Google has it here..

A handy Bash (but not Bourne Shell) tip I learned recently from the Linux From Scratch project is:

mkdir rc{0,1,2,3,4,5,6,S}.d

instead of the more cumbersome:

mkdir rc{a,b,c,3,4,5,6,S}.d

for i in 0 1 2 3 4 5 6 S

do

mkdir rc${i}.d

done

And this can be done recursively, too:

$ cd /

$ ls -ld {,usr,usr/local}/{bin,sbin,lib}

drwxr-xr-x 2 root root 4096 Oct 26 01:00 /bin

drwxr-xr-x 6 root root 4096 Jan 16 17:09 /lib

drwxr-xr-x 2 root root 4096 Oct 27 00:02 /sbin

drwxr-xr-x 2 root root 40960 Jan 16 19:35 usr/bin

drwxr-xr-x 83 root root 49152 Jan 16 17:23 usr/lib

drwxr-xr-x 2 root root 4096 Jan 16 22:22 usr/local/bin

drwxr-xr-x 3 root root 4096 Jan 16 19:17 usr/local/lib

drwxr-xr-x 2 root root 4096 Dec 28 00:44 usr/local/sbin

drwxr-xr-x 2 root root 8192 Dec 27 02:10 usr/sbin