**Shell programming:**

The shell is a program that takes commands from the keyboard and gives them to the operating system to perform. In the old days, it was the only user interface available on a Unix-like system such as Linux. Nowadays, we have graphical user interfaces (GUIs) in addition to command line interfaces (CLIs) such as the shell.

On most Linux systems a program called [bash](http://linuxcommand.org/lc3_man_pages/bash1.html) (which stands for Bourne Again SHell, an enhanced version of the original Unix shell program, sh, written by Steve Bourne) acts as the shell program. Besides bash, there are other shell programs that can be installed in a Linux system. These include: ksh, tcsh and zsh.

**Terminal:** It's a program called a terminal emulator. This is a program that opens a window and lets you interact with the shell. There are a bunch of different terminal emulators you can use. Most Linux distributions supply several, such as: gnome-terminal, konsole, xterm, rxvt, kvt, nxterm, and eterm.

**/ :** The root directory where the file system begins. In most cases the root directory only contains subdirectories.

**/boot :** This is where the Linux kernel and boot loader files are kept. The kernel is a file called Vmlinuz

**/bin , /usr/bin :** These two directories contain most of the programs for the system. The /bin directory has the essential programs that the system requires to operate, while /usr/bin contains applications for the system's users.

**/home :** /home is where users keep their personal work. In general, this is the only place users are allowed to write files. This keeps things nice and clean

**/root :** this is the super user’s home directory.

the shell maintains a list of directories where executable files (programs) are kept, and only searches the directories in that list. If it does not find the program after searching each directory in the list, it will issue the famous command not found error message.

This list of directories is called your path. You can view the list of directories with the following command:

echo $PATH

This will return a colon separated list of directories that will be searched if a specific path name is not given when a command is attempted. In our first attempt to execute your new script, we specified a pathname ("./") to the file.

You can add directories to your path with the following command, where directory is the name of the directory you want to add:

export PATH=$PATH : directory

for each user has a specific directory to store the script files. This directory is called bin and it is a subdirectory of our home directory.

the system is holding a number of facts about the system in its memory. This information is called the environment. The environment contains such things as your path, your user name, the name of the file where your mail is delivered, and much more. You can see a complete list of what is in your environment with the [set](http://linuxcommand.org/lc3_man_pages/seth.html) command.

When you log on to the system, the bash program starts, and reads a series of configuration scripts called startup files. These define the default environment shared by all users. This is followed by more startup files in your home directory that define your personal environment. The exact sequence depends on the type of shell session being started. There are two kinds: a login shell session and a non-login shell session. A login shell session is one in which we are prompted for our user name and password; when we start a virtual console session, for example. A non-login shell session typically occurs when we launch a terminal session in the GUI.

Login shells read one or more files as shown below.

/etc/profile : a global configuration script that applies to all users.

~/.bash\_profile : a user’s personal startup file

~/.bash\_login : if ~/.bash\_profile is not found, bash attempts to read this file.

~/.profile : : if ~/.bash\_profile and ~/.bash\_login are not found, bash attempts to read this file.

Non login shells read the following startup files.

/etc/bash.bashrc : a global configuration script that applies to all users.

~/.bashrc : a user’s personal startup file.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line.  To create a shell script, you use a text editor. A text editor is a program, like a word processor, that reads and writes ASCII text files. View is an editor for writing scripts.  #!/bin/bash  # My first script  echo "Hello World!"  The first line of the script is important. This is a special clue, called a shebang, given to the shell indicating what program is used to interpret the script. In this case, it is /bin/bash. Other scripting languages such as Perl, awk, tcl, Tk, and python also use this mechanism.  The second line is a comment. Everything that appears after a "#" symbol is ignored by bash.  The most basic shell script is a list of commands exactly as could be typed interactively, prefaced by the #! magic header. All the parsing rules, filename wildcards, $PATH searches etc., which were summarized above, apply.  In addition:  # as the first non-whitespace character on a line flags the line as a comment, and the rest of the line is completely ignored. Use comments liberally in your scripts, as in all other forms of programming.  \ as the last character on a line causes the following line to be logically joined before interpretation. This allows single very long commands to be entered in the script in a more readable fashion. You can continue the line as many times as needed.   ; as a separator between words on a line is interpreted as a newline. It allows you to put multiple commands on a single line. There are few occasions when you must do this, but often it is used to improve the layout of compound commands.  **Exit status**  Every command (program) has a value or exit status which it returns to the calling program. This is separate from any output generated. The exit status of a shell script can be explicitly set using [exit](http://www.freebsd.org/cgi/man.cgi?query=exit&apropos=0&sektion=1&manpath=Red+Hat+Linux%2Fi386+9&format=html) N, or it defaults to the value of the last command run.  Note: The exit status is an integer 0-255. Conventionally 0=success and any other value indicates a problem. Think of it as only one way for everything to work, but many possible ways to fail. If the command was terminated by a signal, the value is 128 plus the signal value.  **Permission for execution:**  we have to do is give the shell permission to execute your script through chmod command.  The "755" will give you read, write, and execute permission. Everybody else will get only read and execute permission. If you want your script to be private (i.e., only you can read and execute), use "700" instead.  Now run the script  $ ./hello\_world or ksh hello\_world  **Why use Shell Scripts**   * Combine lengthy and repetitive sequences of commands into a single, simple command. * Generalize a sequence of operations on one set of data, into a procedure that can be applied to any similar set of data. * Create new commands using combinations of utilities in ways the original authors never thought of. * Simple shell scripts might be written as shell aliases, but the script can be made available to all users and all processes. Shell aliases apply only to the current shell. * Wrap programs over which you have no control inside an environment that you can control. * Create customized datasets on the fly, and call applications (e.g. matlab, sas, idl, gnuplot) to work on them, or create customized application commands/procedures. * Rapid prototyping (but avoid letting prototypes become production)   **Typical uses**   * System boot scripts (/etc/init.d) * System administrators, for automating many aspects of computer maintenance, user account creation etc. * Application package installation tools * Application startup scripts, especially unattended applications (e.g. started from [cron](http://www.freebsd.org/cgi/man.cgi?query=cron&apropos=0&sektion=1&manpath=Red+Hat+Linux%2Fi386+9&format=html) or [at](http://www.freebsd.org/cgi/man.cgi?query=at&apropos=0&sektion=1&manpath=Red+Hat+Linux%2Fi386+9&format=html)) * Any user needing to automate the process of setting up and running commercial applications, or their own code.   **Variables:** Variables are areas of memory that can be used to store information and are referred to by a name  Whenever the shell sees a word that begins with a "$", it tries to find out what was assigned to the variable and substitutes it.  To create a variable, put a line in your script that contains the name of the variable followed immediately by an equal sign ("="). No spaces are allowed. After the equal sign, assign the information you wish to store. Note that no spaces are allowed on either side of the equal sign.  **Rules for creating variables:**   * It must start with a letter and it is a case sensitive. * It must not contain embedded spaces. Use underscores instead. * Don't use punctuation marks. * Don't use a name that is already a word understood by bash. These are called reserved words and should not be used as variable names.   **Defining variables:** variables are defined as follows.  Syntax: Variable\_name=variable\_value  Example:  ename=”david”  esal=2000  the above variables are called scalar variables. It holds only one value at a time.  Accessing Variables: To access the value stored in a variable, prefix its name with the dollar sign ( $):    **Example:**  **1. cat emp.ksh**  #!/bin/ksh  #sample script  ename=”ram”  esal=2000  echo $ename  echo $esal  **run:**  ksh emp.ksh  **o/p:** ram  2000  echo {ename} and echo $ename both are same but littile difference.  Note: If the variable is not surrounded by whitespace (or other characters that can't be in a name), the name must be surrounded by "{}" braces so that the shell knows what characters you intend to be part of the name.  **Ex: cat emp.ksh**  #!/bin/ksh  #sample script  ename=”ram”  esal=2000  echo ${ename}Prasad # here ename variable is appended to string.  echo $esal  run:  **ksh emp.ksh**  **o/p:** ramprasad  2000  **Defining Default values to variables:**  1. If we are not providing any value to variable then it will print blank space.  To avoid this we can provide default values to variables.  **{ename:-default\_value}**  **Ex: cat emp.ksh**  #!/bin/ksh  #sample script  read ename  esal=2000  echo ${ename:-david}  echo $esal  **run:**  ksh emp.ksh  # omitting input value  **o/p:**  david # it taken default value  2. if the variable has not been defined then it will take default value if we have provided.  **Ex:** **cat emp.ksh**  #!/bin/ksh  #sample script  echo ${ename:-david} # ${ename=david} is also correct.  echo $esal  **run:**  ksh emp.ksh  **o/p:**  david # it taken default value  3. if we are need take default value even if we are providing value to variable.  **Ex: cat emp.ksh**  #!/bin/ksh  #sample script  read ename  echo ${ename+david}  **run:**  ksh emp.ksh  ram  **o/p:**  david # default value is printed.  **Read only Variables:**  The shell provides a way to mark variables as read-only by using the readonly command. After a variable is marked read-only, its value cannot be changed.  **Ex: cat emp.ksh**  #!/bin/ksh  # readonly variables  ename=”david”  echo $ename  readonly ename  ename=”ram”  echo $ename  **run:**  ksh emp.ksh  **o/p:**  david  ename: is read only # it will throw an error msg.  **Unsetting Variables:**  Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you would not be able to access stored value in the variable.  Syntax: unset variable\_name  **Ex: cat emp.ksh**  #!/bin/ksh  # readonly variables  ename=”david”  unset ename  echo $ename  **run:**  ksh emp.ksh  **o/p:** # no output bcoz of ename is unset.    **Variable Types:**  When a shell is running, three main types of variables are present:  **Local Variables:** A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.  **Environment Variables:** An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.  **Shell Variables:** A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.  Several special variables are used by the system -- you can use these, but may not be able to change them. The special variables use uppercase names, or punctuation characters. Some variables are set by the login process and inherited by the shell (e.g. **$USER**), while others are used only by the shell.  **$USER, $LOGNAME**  Preset to the currently logged-in username.  **$PATH :** The list of directories that will be searched for external commands. You can change this in a script to make sure you get the programs you intend, and don't accidentally get other versions which might have been installed.  **$TERM :** The terminal type in which the shell session is currently executing. Usually "xterm" or "vt100". Many programs need to know this to figure out what special character sequences to send to achieve special effects.  **$PAGER :** If set, this contains the name of the program which the user prefers to use for text file viewing. Usually set to "more" or "less" or something similar. Many programs which need to present multipage information to the user will respect this setting (e.g. [**man**](http://www.freebsd.org/cgi/man.cgi?query=man&apropos=0&sektion=1&manpath=Red+Hat+Linux%2Fi386+9&format=html)). This isn't actually used by the shell itself, but shell scripts should honour it if they need to page output to the user.  **$EDITOR :** If set, this contains the name of the program which the user prefers to use for text file editing. A program which needs to have the user manually edit a file might choose to start up this program instead of some built-in default (e.g. "crontab -e". This also determines the default command-line-editing behaviour in interactive shells.  **Shell internal settings**  **$PWD :** Always set the current working directory (readonly)  **$OLDPWD**: The previous directory (before the most recent [**cd**](http://www.freebsd.org/cgi/man.cgi?query=cd&apropos=0&sektion=1&manpath=Red+Hat+Linux%2Fi386+9&format=html) command). However, changing directories in a script is often dangerous.  **$?** (readonly): Set to the exit status of the last command run, so you can test success or failure. Every command resets this so it must be saved immediately if you want to use it later.  **$- :**Set to the currently set options flags.  **$IFS:**Internal Field Separators: the set of characters (normally space and tab) which are used to parse a command line into separate arguments. This may be set by the user for special purposes, but things get very confusing if it isn't changed back.  **Process ID variables :**  **$$** (readonly) : Set to the process ID of the current shell - useful in making unique temporary files.  **$PPID** (readonly) : Set to the process ID of the parent process of this shell - useful for discovering how the script was called.  **$!** (readonly) :Set to the process ID of the last command started in background - useful for checking on background processes.  **Filename Wildcards:**  The following characters are interpreted by the shell as filename wildcards, and any word containing them is replaced by a sorted list of all the matching files.   \* : Match zero or more characters.  ? : Match any single character  [...] : Match any single character from the bracketed set. A range of  characters can be specified with [ - ]  [!...] : Match any single character NOT in the bracketed set.   * An initial "." in a filename does not match a wildcard unless explicitly given in the pattern. In this sense filenames starting with "." are hidden. A "." elsewhere in the filename is not special. * Pattern operators can be combined   **Command line (positional) arguments:**  To customize the behaviour of a script at run time, you can give it any number of arguments on the command line.   |  |  | | --- | --- | | **Variable** | **Description** | | $0 | The name of the current script. | | $n | These variables correspond to the arguments with which a script was invoked ( $1, $2...... $9) | | ${10}, ${11}... | Positional arguments greater than 9 are set by ksh and bash. Remember to use braces to refer to them. | | $# | The number of arguments supplied to a script. | | $\* | contains all of the arguments in a single string, with one space separating them. | | [$@](mailto:$@) | same as $\*, but its arguments are individualy quoted. | | $? | The exit status of the last command executed. | | $$ | The process number of the current shell. For shell scripts, this is the process ID under which they are executing. | | $! | The process number of the last background command. | | shift | discard $1 and renumber all the other variables. "shift N" will shift N arguments at once. |   **Shell options:** Startup options.  Syntax:  ksh -options scriptname  Options:  -x : echo line to stderr before executing it  -n : read commands and check for syntax errors, but do not execute.  -a :all variables are automatically exported  -f : disable wildcard filename expansion (globbing)  set –x : Set an option within a shell script  $- : contains the currently set option letters  **Ex: cat pos.ksh**  #!/bin/ksh  #positional parameters script  echo "employee name: $1"  echo " employess address : $2"  echo "employee mobile: $3"  echo "employee salaryL: $4"  echo " script name : $0"  echo " no of arguments : $#"  echo " list of arguments: $\* "  echo " list of values: $@"  **run:**  ksh pos.ksh ram ogl 8128825400 25000  **o/p:**    employee name : ram  employess address : ogl  employee mobile : 8128825400  employee salary : 25000  script name : pos.ksh  no of arguments : 4  list of arguments : ram ogl 8128825400 25000  list of values : ram ogl 8128825400 25000  **Command substitution:**  **Syntax : `command`**  A command (plus optional arguments) enclosed in backticks is executed and the standard output of that command is substituted. If the command produces multiline output, the newlines are retained.  **Or**  **Syntax: $(command)**  This syntax is functionally the same as backticks, but commands can be more easily nested.  **I/O Redirection and Pipelines:**  Any simple command (or shell function, or compound command) may have its input and output redirected using the following operators.  **Output redirection :**  **> filename :** Standard ouput is redirected to the named file. The file is overwritten unless the **noclobber** option is set. The file is created if it does not exist.   **>> filename :** Standard ouput is appended to the named file. The file is created if it does not exist.  **>| filename :** Output redirect, and override the noclobber option, if set.  **Input redirection:**  **< filename :** Standard input is redirected to the named file. The file must already exist.  **Command pipelines:**  command | command [ | command ...]  Pipe combines multiple commands together. The standard output of the first command becomes the standard input of the second command. All commands run simultaneously, and data transfer happens via memory buffers. This is one of the most powerful constructs in Unix.  **Input and output in shell script:**  Shell scripts can generate output directly or read input into variables using the following commands:  **Script output:**  **[echo](http://www.freebsd.org/cgi/man.cgi?query=echo&apropos=0&sektion=1&manpath=Red+Hat+Linux%2Fi386+9&format=html" \t "manpage) :** Print arguments, separated by spaces, and terminated by a newline, to stdout. Use quotes to preserve spacing. Echo also understands C-like escape conventions.   -n : suppress newline  [**print**](http://www.freebsd.org/cgi/man.cgi?query=print&apropos=0&sektion=1&manpath=Red+Hat+Linux%2Fi386+9&format=html)**(ksh internal)** : Print arguments, separated by spaces, and terminated by a newline, to stdout. Print observes the same escape conventions as echo.  -n :suppress newline  -r : raw mode - ignore \-escape conventions  -R : raw mode - ignore \-escape conventions and -options except -n.    **Script input:**  [read](http://www.freebsd.org/cgi/man.cgi?query=read&apropos=0&sektion=1&manpath=Red+Hat+Linux%2Fi386+9&format=html" \t "manpage) **var1 var2 rest**  read a line from stdin, parsing by $IFS, and placing the words into the named variables. Any left over words all go into the last variable. A '\' as the last character on a line removes significance of the newline, and input continues with the following line.  -r : raw mode - ignore \-escape conventions  Ex:  **1. $ cat exmp1.ksh**  #!/bin/ksh  # input and output script  echo "value of a: "  read a  echo "value of b:"  read b  c=`expr $a + $b|bc` # bc means basic calculator  echo "value of c: $c"  **run:**  ksh exmp1.ksh  value of a: 10  value of b : 20  o/p: value of c : 30  **2. $ cat exmp2.ksh**  #!/bin/ksh  # simple script  echo " read two strings: "  read estring  read estring1  echo "first: $estring \n second : $estring1"  **run:** ksh exmp2.ksh  read two strings:  first : ram  second: hari  **o/p:**  ram  hari  **3. $ cat exmp3.ksh**  #!/bin/ksh  # simple script  echo " read multiline string"  read estring  read estring1  print -n "first: $estring \n second : $estring1"  **run:** ksh exmp2.ksh  read two strings:  ram  hari  **o/p:**  first : ram  second: hari $ # new line is suppressed.  **4. $ cat exmp4.ksh**  #!/bin/ksh  # simple script  echo " read two strings"  read estring  read estring1  print -r "first: $estring \n second : $estring1"  **run:** ksh exmp2.ksh  read two strings:  ram  hari  **o/p:**  first :ram \n second : hari  **5. cat exmp5.ksh**  #!/bin/ksh  # sample arithmetic operation  echo " read a , b values "  read a b  c=`expr $a + $b `  echo "total : $c"  **run:** ksh exmp5.ksh  read a , b values  2 4  **o/p:**  total : 6  **operators:**  There are various operators supported by each shell. Our tutorial is based on default shell (Bourne) so we are going to cover all the important Bourne Shell operators in the tutorial.  There are following operators which we are going to discuss:   * Arithmetic Operators. * Relational Operators. * Boolean Operators. * String Operators. * File Test Operators.   **Ex:cat hari1.ksh**  #!/bin/ksh  # sample arithmetic operation  echo " read a , b values "  read a b  c=`expr 2 + 3 `  echo "total : $c"  **run:** ksh hari1.ksh  **o/p:** total : 5  **note:**   * There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2. * Complete expression should be enclosed between **``**, called inverted commas.   **Arithmetic operators:**  Assume a=10, b=20.   |  |  |  | | --- | --- | --- | | operator | Description | Example | | + | Addition - Adds values on either side of the operator | `expr $a + $b` will give 30 | | - | Subtraction - Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 | | \* | Multiplication - Multiplies values on either side of the operator | `expr $a \\* $b` will give 200 | | / | Division - Divides left hand operand by right hand operand | `expr $b / $a` will give 2 | | % | Modulus - Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 | | = | Assignment - Assign right operand in left operand | a=$b would assign value of b into a | | == | Equality - Compares two numbers, if both are same then returns true. | [ $a == $b ] would return false. | | != | Not Equality - Compares two numbers, if both are different then returns true. | [ $a != $b ] would return true. |   Note: [ $a == $b ] is correct one. [$a == $b] is wrong.  **Relational Operators:**   |  |  |  | | --- | --- | --- | | operator | Description | Example | | -eq | Checks if the value of two operands are equal or not, if yes then condition becomes true. | [ $a -eq $b ] is not true. | | -ne | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | [ $a -ne $b ] is true. | | -gt | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | [ $a -gt $b ] is not true. | | -lt | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | [ $a -lt $b ] is true. | | -ge | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | [ $a -ge $b ] is not true. | | -le | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | [ $a -le $b ] is true. |   **Boolean operators:**   |  |  |  | | --- | --- | --- | | operator | Description | Example | | ! | This is logical negation. This inverts a true condition into false and vice versa. | [ ! false ] is true. | | -o | This is logical OR. If one of the operands is true then condition would be true. | [ $a -lt 20 -o $b -gt 100 ] is true. | | -a | This is logical AND. If both the operands are true then condition would be true otherwise it would be false. | [ $a -lt 20 -a $b -gt 100 ] is false. |   **String operators:**   |  |  |  | | --- | --- | --- | | operator | Description | Example | | = | Checks if the value of two operands are equal or not, if yes then condition becomes true. | [ $a = $b ] is not true. | | != | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | [ $a != $b ] is true. | | -z | Checks if the given string operand size is zero. If it is zero length then it returns true. | [ -z $a ] is not true. | | -n | Checks if the given string operand size is non-zero. If it is non-zero length then it returns true. | [ -z $a ] is not false. | | Str | Check if str is not the empty string. If it is empty then it returns false. | [ $a ] is not false. |   **File operators:**   |  |  |  | | --- | --- | --- | | **operator** | **description** | **example** | | -c file | Checks if file is a character special file if yes then condition becomes true. | [ -b $file ] is false. | | -d file | Check if file is a directory if yes then condition becomes true. | [ -d $file ] is not true. | | -f file | Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true. | [ -f $file ] is true. | | -r file | Checks if file is readable if yes then condition becomes true. | [ -r $file ] is true. | | -w file | Check if file is writable if yes then condition becomes true. | [ -w $file ] is true. | | -x file | Check if file is execute if yes then condition becomes true. | [ -x $file ] is true. | | -s file | Check if file has size greater than 0 if yes then condition becomes true. | [ -s $file ] is true. | | -e file | Check if file exists. Is true even if file is a directory but exists. | [ -e $file ] is true. |   **Conditional statements:-**  Unix Shell supports conditional statements which are used to perform different actions based on different conditions. Here we will explain following two decision making statements:   * 1. The **if...else** statements * 2. The **case...esac** statement   **IF statement:**  Syntax: if command  Then  Success\_Statements  Else  statements  fi  or  if var –eq val # in place of –eq we can use relational  then operators  true statements  else  false statements  fi  or  if [ condition ]  then  true satatements  else  false statements  fi  the command is executed successfully then success\_statement part excuted otherwise else part will execute.  **Else if:**  if *condition1*  then  *statement1*  *statement2*  ..........  elif *condition2*  then  *statement3*  *statement4*  ........  elif *condition3*  then  *statement5*  *statement6*  ........  fi  **Examples: $ cat sam2.ksh**  #!/bin/ksh  # simple program  a=$1  if [ $a = ram ]  then  echo "ram"  else  echo "not ram"  fi  **Run:** ksh sam2.ksh ram  **o/p:** ram  ksh sam2.ksh hari  o/p: not ram.  **2. $ cat sam3.ksh**  #!/bin/ksh  # simple program  a=$1  if [ $a -eq 10 ] # if [ $a == 10 ] is also correct.  then  echo " value is $1"  else  echo " value is not 10 "  fi  **Run:**  Ksh sam3.ksh 10  o/p: value is 10.  **3. cat sam2.ksh** |  |
|  |  |
|  |  |

# !/bin/ksh

# sample shell program

a=$1

If [ “${a}” = “david” ]

Then

Echo “ name is david”

Else

Echo “ name is $a”

fi

**run:** ksh sam2.ksh david

**o/p:** name is david

**4. find biggest among three in shell script**

**cat big3.ksh**

# !/bin/sh

# biggest among three numbers

If [ $1 –gt $2 ]

then

if [ $1 –gt $3 ]

then

echo “ a is big value”

else

echo “ c is big value”

fi

else

if [ $2 –gt $3 ]

then

echo “ b value is big “

else

echo “c is big”

fi

fi

**run:** ksh big3.ksh 10 20 15 # a=10, b=20 c=15

**o/p:** b is big.

**5. script for the named file exists or not**

# !/bin/sh

Dir=/home/ram

If [ -f $dir/$1 ]

then

echo “ file $1 exists”

else

echo “file $1 does not exists”

fi

**6. input is file or directory checking.**

# !/bin/ksh

If [ “$1” ]

then

if [ -f $1 ]

then

echo “ $1 file name exists”

else

echo “$1 is a directory”

fi

else

echo “ argument is not correct, supply correct argument”

fi

**Loops:**

Loops are a powerful programming tool that enable you to execute a set of commands repeatedly.

* [The while loop](http://www.tutorialspoint.com/unix/while-loop.htm)
* [The for loop](http://www.tutorialspoint.com/unix/for-loop.htm)
* [The until loop](http://www.tutorialspoint.com/unix/until-loop.htm)
* [The select loop](http://www.tutorialspoint.com/unix/select-loop.htm)

**While loop:**

While loop would execute given commands until given condition remains true where as until loop would execute until a given condition becomes true.

Syntax: while [ condition ]

do

statements # it will execute when condition is true.

done

**7. $ cat prime.ksh**

#!/bin/ksh

# given value is prime number or not

echo " enter value"

read num

i=2

while [ $i -lt $num ]

do

num1=`expr $num % $i`

if [ $num1 -eq 0 ]

then

echo " $num is not a prime number"

exit

fi

i=`expr $i + 1` # let i++ also correct for incrementing.

done

echo " $num is a prime number"

**run:** ksh prime.ksh 7

**o/p:** 7 is a prime number

**8. $ cat slogic.ksh**

#!/bin/ksh

# simple logic script

a=0

while [ $a -lt 10 ]

do

b="$a"

while [ $b -ge 0 ]

do

echo -n "$b "

b=`expr $b - 1`

done

echo

a=`expr $a + 1`

done

**For Loop:**

There are two types of bash for loops available. One using the “in” keyword with list of values, another using the C programming like syntax.

**Method 1:**

**Syntax:**

for varname in list

do

command1

command2

...........

Done

In the above syntax:

* for, in, do and done are keywords
* “list” contains list of values. The list can be a variable that contains several words separated by spaces. If list is missing in the for statement, then it takes the positional parameter that were passed into the shell.
* varname is any Ksh variable name.

 the for statement executes the commands enclosed in a body, once for each

item in the list.

**Method 2:**

**Syntax:** in this method we have three expressions.

for (( expr1; expr2; expr3 ))

do

command1

command2

.........

done

In the above bash for command syntax,

* Before the first iteration, expr1 is evaluated. This is usually used to initialize variables for the loop.
* All the statements between do and done are executed repeatedly until the value of expr2 is TRUE.
* After each iteration of the loop, expr3 is evaluated. This is usually used to increment a loop counter.

**Examples:**

**1. $ cat for1.ksh**

#!/bin/ksh

# simple for loop script

i=1

for day in mon tue wed thu fri # shell treats as multiple values

do

echo "weekday $((i++)) : $day"

done

o/p:

weekday 1 : mon

weekday 2 : tue

weekday 3 : wed

weekday 4 : thu

weekday 5 : fri

**2.$ cat for2.ksh**

#!/bin/ksh

# simple for loop script

i=1

for day in " mon tue wed thu fri " # shell treats as single value

do

echo "weekday $((i++)) : $day"

done

**o/p:**

weekday 1 : mon tue wed thu fri

**3. $ cat for1.ksh**

#!/bin/ksh

# simple for loop script

i=1

weekdays=" mon tue wed thu fri "

for day in $weekdays

do

echo "weekday $((i++)) : $day"

done

**o/p:**

weekday 1 : mon

weekday 2 : tue

weekday 3 : wed

weekday 4 : thu

weekday 5 : fri

**4. $ cat for1.ksh** // here we are passing positional parameters

#!/bin/ksh

# simple for loop script

i=1

for day

do

echo "weekday $((i++)) : $day"

done

**run:** ksh for1.ksh mon tue wed thu fri

o/p:

weekday 1 : mon

weekday 2 : tue

weekday 3 : wed

weekday 4 : thu

weekday 5 : fri

**4. $ cat for3.ksh**

#!/bin/ksh

# simple for loop script

i=1

for day in 1 2 3 4 5 6 7 8 9 10

do

echo "number : $day"

done

**o/p:**

number : 1

number : 2

number : 3

number : 4

number : 5

number : 6

number : 7

number : 8

number : 9

number : 10

6. it will display all the files and sub directories in current directory.

**$ cat for3.ksh**

#!/bin/ksh

# reading files and sub dirctories in curent directory

for i in \*

do

echo " file : $i"

done

**o/p:**

**7. it will list the files and directories**

**$ cat for4.ksh**

#!/bin/ksh

# reading files and sub dirctories in curent directory

for i in \*

do

if [ -f $i ]

then

echo " file : $i"

else

echo " directory : $i "

fi

done

**8. display numbers 1 to 20 using for loop**

#!/bin/ksh

for i in $( seq 1 20 )

do

echo "Welcome $i times..."

done

**echo command options:**

echo has three options :

-e :  enables interpretation of backslash escapes.

-E :  to disable interpretation of backslash escapes (default).

-n :  to disable insertion of new line.

The following escape sequences which can be used in echo command:

|  |  |
| --- | --- |
| **Escape** | **Description** |
| **\\** | backslash |
| **\a** | alert (BEL) |
| **\b** | backspace |
| **\c** | suppress trailing newline |
| **\f** | form feed |
| **\n** | new line |
| **\r** | carriage return |
| **\t** | horizontal tab |
| **\v** | vertical tab |

Examples:

**1. $cat exec1.ksh**

#!/bin/ksh

# echo options

a=ram

echo " my name is $a \n prasad"

**run:** $ ksh exec1.ksh

my name is ram

prasad

**2 $cat exec1.ksh**

#!/bin/ksh

# echo options

a=ram

echo -e " my name is $a \n prasad"

**run:**

$ ksh exec1.ksh

my name is ram

Prasad

**3. $ cat exec1.ksh**

#!/bin/ksh

# echo options

a=ram

echo -e " my name is $a \n prasad"

$ sh exec1.ksh

my name is ram

prasad

**4**.**$cat exec1.ksh**

#!/bin/ksh

# echo options

a=ram

echo " my name is $a \n prasad"

**run:** $ sh exec1.ksh

my name is ram \n prasad

**5. $ cat exec2.ksh**

#!/bin/ksh

#simple script

echo hi \; how r u

**run:** $ ksh exec2.ksh

hi ; how r u

$ sh exec2.ksh

hi ; how r u

**Special characters:**

The following are the special characters in shell.

\* ? [ ] ' " \ $ ; & ( ) | ^ < > new-line space tab

|  |  |
| --- | --- |
| **Quoting** | **Description** |
| **Single quote** | All special characters between these quotes lose their special meaning. |
| **Double quote** | Most special characters between these quotes lose their special meaning with these exceptions:   * $ * ` * \$ * \' * \" * \\ |
| **Backslash** | Any character immediately following the backslash loses its special meaning. |
| **Back Quote** | Anything in between back quotes would be treated as a command and would be executed. |

**1. $ cat exec3.ksh**

#!/bin/ksh

#simple script

echo 'hi ; how r u'

**run:** $ ksh exec3.ksh

hi ; how r u

**2.$ cat exec3.ksh**

#!/bin/ksh

#simple script

a=10

echo 'value of a is: $a '

**run:** $ ksh exec3.ksh

value of a is: $a

**3.$ cat exec3.ksh**

#!/bin/ksh

#simple script

a=10

echo “value of a is: $a “

**run:** $ ksh exec3.ksh

value of a is: 10

**4.$ cat exec3.ksh**

#!/bin/ksh

#simple script

a=10

echo " value \\ of a is: $a"

**run:** $ ksh exec3.ksh

value \ of a is: 10

**5.$ cat exec3.ksh**

#!/bin/ksh

#simple script

a=10

echo ' value \" of a is: $a'

**run:** ksh exec3.ksh

value \" of a is: $a

**6.$ cat exec3.ksh**

#!/bin/ksh

#simple script

a=10

echo “ value \" of a is: $a”

**run:** ksh exec3.ksh

value " of a is: 10

**7.$ cat exec4.ksh**

#!/bin/ksh

# back quotes shell program

a=`date`

echo " curremt date is : $a "

**run:** $ ksh exec4.ksh

curremt date is : Tue Nov 12 09:50:34 CET 2013

**8.$ cat exec4.ksh**

#!/bin/ksh

# back quotes shell program

echo " curremt date is : `date` "

**run:** $ ksh exec4.ksh

curremt date is : Tue Nov 12 09:52:12 CET 2013

**9.$ cat exec4.ksh**

#!/bin/ksh

# back quotes shell program

echo curremt date is : `date`

**run:** ksh exec4.ksh

curremt date is : Tue Nov 12 09:56:22 CET 2013

**Case statement:**

simple employee salary calculation program.

# !/bin/ksh

# simple shell script for employee salary calculation

echo “name is $1”

case “ $2 “ in

-lt ‘20000’ )

Hra=`expr $2 \\* 0.25`

gs=`expr $2 + $hra`

echo “ update salary is $gs”

;;

-lt ‘50000’ )

Hra=`expr $2 \\* 0.5`

gs=`expr $2 + $hra`

echo “ update salary is $gs”

;;

\*)

Hra=`expr $2 \\* 0.75`

gs=`expr $2 + $hra`

echo “ update salary is $gs”

;;

Esac

EMP salary calculation using if statement:

echo enter basic salary

read bs

if [ $bs -lt 20000 ]

then

hra=`echo $bs\\*10/100|bc`

da=`echo $bs\\*90/100|bc`

else

hra=500

da=`echo $bs\\*98/100|bc`

echo dearness allovance is=$da

fi

gs=`echo $bs+$hra+$da|bc`

echo Gross salary=$gs

**using case statement:**

$ cat emp.ksh

#!/bin/ksh

# employee salary calculation script

#echo "emp name is $1"

bs=$1

case $bs in

[0-9] | [0-9][0-9] | [0-9][0-9][0-9] | [0-9][0-9][0-9][0-9] | [0-2][0-9][0-9][0-9][0-9] )

hra=`echo $bs \\* 20/100|bc`

gs=`echo $bs + $hra|bc`

;;

[3-5][0-9][0-9][0-9][0-9] )

hra=`echo $bs \\* 50/100|bc`

gs=`echo $bs + $hra|bc`

;;

\*)

hra=`echo $bs \\* 75/100|bc`

gs=`echo $bs+$hra|bc`

;;

esac

echo "employee basic salary $bs"

echo " employee hra is $hra"

echo " emp updated salary is $gs"

**functions:**

The Function or Subroutine is one of the most useful features of the shell programming languages.

By definition a function must return one thing but sometimes these shell functions return more than one by using stdout as well as the *return value*s.

1. A function encapsulates a small piece of code that will be used often.
2. You only need to define it once.
3. A function will execute faster than the equivalent code in the script.
4. Parameters are passed into the function via numbered arguments just like scripts.
5. The *special* shell variables are also set up when a function is called (*$#, $@,*etc.) but in this case they relate to the called function not the calling script. Their scope is local to the function.
6. Output can be in the form of *stdout* or a *return code* value or *both*. You cannot stop the function returning a value via its *return code*, but you can exercise control over the *return code*. If you choose not to, then the function will return the value of the last executed sub command from within the function.
7. Variables used and set inside a function *can* be visible from the calling script
8. Variables set within the environment or calling script *can* be visible in the function. This relies on the use of the export command in both directions.
9. Functions can be nested to any depth - well I haven't found a limit.
10. No amount of nesting will ever disguise the *$0* parameter. The *$0* will never be set to the function name, it is only ever set to the name of the calling script.
11. With care and shift, any number of parameters can be handled.

**Examples:**

**1.** **$ cat fun1.ksh**

#!/bin/ksh

# this is first program for using functions in script

hello() {

echo " this is sample script by using function"

}

Hello

**Run: Ksh fun1.ksh**

this is sample script by using function

**2. $ cat fun1.ksh**

#!/bin/ksh

# this is first program for using functions in script

hello() {

echo " first parameter: $1 "

echo " second parameter: $2 "

}

hello $1 $2

**run:** ksh fun1.ksh hi hello1

first parameter: hi

second parameter: hello1

**3.$ cat fun1.ksh**

#!/bin/ksh

# this is first program for using functions in script

a=$1

b=$2

hello() {

echo " first parameter: $a "

echo " second parameter: $b "

}

hello a b

**run:** ksh fun1.ksh hi hello1

first parameter: hi

second parameter: hello1

**4.$ cat fun1.ksh**

#!/bin/ksh

# this is first program for using functions in script

a=$1

b=$2

hello() {

a=100

b=200

echo " first parameter: $a "

echo " second parameter: $b "

}

hello a b

**run:** $ ksh fun1.ksh 10 20

first parameter: 100

second parameter: 200

**5. $ cat fun1.ksh**

#!/bin/ksh

# this is first program for using functions in script

a=$1

b=$2

hello() {

a=100

b=200

echo " a value in function: $a "

echo " b value in function: $b "

}

hello a b

echo " after function a value: $a "

echo " after function b value: $b "

**run:**$ ksh fun1.ksh 10 20

a value in function: 100

b value in function: 200

after function a value: 100

after function b value: 200

**6. $ cat fun1.ksh**

#!/bin/ksh

# this is first program for using functions in script

a=$1

b=$2

hello() {

local a=100

local b=200

echo " a value in function: $a "

echo " b value in function: $b "

}

hello a b

echo " after function a value: $a "

echo " after function b value: $b "

**run:**$ ksh fun1.ksh 10 20

a value in function: 100

b value in function: 200

after function a value: 10

after function b value: 20

**7. $ cat fun1.ksh**

#!/bin/ksh

# this is first program for using functions in script

hello() {

echo " hi we are returning value 10 "

return 10

}

hello

re=$?

echo " returned value is : $re"

**run:** $ ksh fun1.ksh

hi we are returning value 10

returned value is : 10

**8. $ cat fun2.ksh**

#!/bin/ksh

# date sample script

sdate(){

echo current date : `date +%Y%m%d`

}

sdate

run: **$ ksh fun2.ksh**

current date : 20131114

**9. $ cat fun2.ksh**

#!/bin/ksh

# date sample script

sdate(){

echo current date : `date +%Y%h%d`

echo current date : `date +%Y-%h-%d`

}

sdate

**run:** **$ ksh fun2.ksh**

current date : 2013Nov14

current date : 2013-Nov-14

**10.$ cat fun3.ksh**

#!/bin/ksh

# calculating no of days in month

Month\_length(){

if [ $1 -lt 1 ] || [ $1 -gt 12 ]

then

echo " month number is not correct: please pass correct value"

exit

fi

lengths="312831303130313130313031"

cut2=`expr $1 + $1`

cut1=`expr $cut2 - 1`

echo -n " $1 month days are: "

echo $lengths | cut -c$cut1-$cut2

}

month\_length $1

**run: $ ksh fun3.ksh 8**

8 month days are: 31

**11. $ cat exec7.ksh**

#!/bin/ksh

#sample program for reading line by line from file // using while loop.

sfile=$1

count=0

while read LINE

do

let count++

echo " $count : $LINE "

done < $sfile

echo "total no of lines :$count "

run: ksh exec7.ksh sample.dat

o/p:

1 : hi how r u

2 : hi how is ur mom

3 : hi

4 : how

5 : hi

total no of lines :5

**12.$ cat exec8.ksh**

#!/bin/ksh

# read line by line form file using awk

filename=$1

awk '{kount++;print kount,$0} END {print "\n total " kount "lines read"}' $filename

**run: $ ksh exec8.ksh sample.dat**

1 hi how r u

2 hi how is ur mom

3 hi

4 how

5 hi

total 5lines read

**reading fields from file. First and second fileds, delimeter is space.**

1. $ awk -F" " '{print $1 "\t" $2}' sample.dat

**2. $ cat exec9.ksh**

#!/bin/ksh

# read data from file using wc and while loop

sfile=$1

lines=` wc -l < $sfile`

count=0

while [ $count -lt $lines ]

do

let count++

Line=` head -n $count $sfile | tail -1`

echo "$count $Line"

done

echo "\n total $count lines read"

**run:** ksh exec9.ksh sample.dat

**3. $ cat exec10.ksh**

#!/bin/ksh

# word count, lines count in a file, convert lower to upper and vice versa,find

particular word, display content in reverse

echo " enter file name"

read fname

echo " enter choices"

echo " 1. no of characters, words and lines"

echo " 2. reverse the words"

echo " 3. convert upper to lower "

echo " 4. convert lower to upper"

echo " 5. search a word in file"

echo " enter choice: "

read ch

case $ch in

1) echo " total characters:"

wc -c $fname

echo " total words "

wc -w $fname

echo " total lines "

wc -l $fname

;;

2) echo " reverse the word:"

rev $fname

;;

3) echo " upper case letters:"

tr '[a-z]' '[A-Z]' < $fname

cat $fname

;;

4) echo " lower case letters :"

tr '[A-Z]' '[a-z]' < $fname

;;

5) echo " enter search word"

read w

grep -c "$w" $fname

;;

\*) echo " wrong choice"

esac

**Arrays:**

**1.** **$ cat exec11.ksh**

#!/bin/ksh

# simple array creation

set -A sa 1 2 3 4 5

i=0

echo " array values are: "

for s in ${sa[\*]}

do

let i++

echo " array $i value : $s"

done

**2.$ cat exec12.ksh**

#!/bin/ksh

# linear search in shell script

set -A larray 12 13 2 7 1 34 23

echo " enter search value in : ${larray[\*]}"

read num

pos=1

for i in ${larray[\*]}

do

if [ $num -eq $i ]

then

echo " value is found : index = $pos"

exit

fi

let pos++

done

echo " search value is not found "

**3.$ cat exec13.ksh**

#!/bin/ksh

# fibonacy series

if [ $# -eq 1 ]

then

num1=$1

else

echo "read a number"

read num1

fi

f1=0

f2=1

echo "fibonacy series:"

for(( i=1;i<=num1;i++ ))

do

echo -n "$f1 "

fn=$((f1+f2))

f1=$f2

f2=$fn

done

**4.$ cat exec14.ksh**

#!/bin/ksh

# check the armstrong number

echo " enter the start value"

read start

ec

ho "enter the ending value"

read ending

while [ $start -le $ending ]

do

num=$start

length=${#num}

sum=0

onum=$num

while [ $num -ne 0 ]

do

rem=$(( num%10 ))

num=$(( num/10 ))

power=$(echo "$rem ^ $length" |bc)

sum=$(( sum + power ))

done

if [ $sum == $onum ]

then

echo -n " $onum "

fi

let start++

done

# Writing UNIX Scripts

## Introduction

In UNIX, commands are submitted to the Operating System via a *shell*. A shell is an environment which allows commands to be issued, and also includes facilities to control input and output, and programming facilities to allow complex sets of actions to be performed. Whenever you type commands at the prompt in Unix, you are actually communicating interactively with a shell.

In addition to typing commands directly to the shell, you can place them in a file (which must be given execute permission), and then run the file. A file containing UNIX shell commands is known as a *script* (or shell script). Executing the script is equivalent to typing the commands within it.

The Unix shells are actually quite sophisticated programming languages in their own right: there are entire books devoted to programming in them. Together with the large number of special utility programs which are provided as standard, scripts make Unix an extremely powerful operating system.

Scripts are *interpreted* rather than *compiled*, which means that the computer must translate them each time it runs them: they are thus less efficient than programs written in (for example) C. However, they are extremely good where you want to use Operating System facilities; for example, when processing files in some fashion.

There are actually no less than three different types of scripts supported in Unix: Bourne shell, C shell, and Korn shell. Bourne is the most common, Korn the most powerful, and C the most C-like (handy for C programmers). This tutorial will concentrate on the simplest of the three: the Bourne shell.

### A simple Bourne-shell script

If you simply type Unix commands into a file, and make it executable, then run it, Unix will assume that the commands in it are written in whatever shell language you happen to be using at the time (in your case, this is probably the C shell). To make sure that the correct shell is run, the first line of the script should always indicate which one is required. Below is illustrated a simple Bourne-shell script: it just outputs the message "hello world":

#!/bin/sh

echo "hello world"

Use the text editor to create a file with the lines above in it. Save the file, calling it *hello*. Then make the *hello* file executable by typing:

chmod 755 hello

You can then run the script by simply typing *hello*. When you type hello, UNIX realises that this is a script, and runs the UNIX commands inside it (in this case, just the *echo* command).

Naming Shell Script Files

You can give your shell scripts almost any name you like. Be careful, however! If the name you use happens to already be an existing UNIX command, when you try to run your script you will end up running the system-defined command instead. Some fairly obvious names can cause you this problem; for example, *test*. To make sure that your intended name is O.K., you should check whether it exists before you start editing the new command. You can do this by using the *which* command, which will locate a command, if it exists, and tell you that it can't locate it if it doesn't. For example, to find out whether there are commands called *dc* and *dac*, type:

which dc

which dac

Most UNIX commands are stored in the */bin* directory, so you can get a list of most of them by typing:

ls /bin

### Comments and Commands

In the Bourne shell, any line beginning with a hash '#' character in the first column is taken to be a comment and is ignored. The only exception is the first line in the file, where the comment is used to indicate which shell should be used (**/bin/sh** is the Bourne shell). It is always good practice to use comments to indicate what a script does; both in a header section at the top, and line by line if the code is at all complicated. There aren't many comments in the examples in this document, to reduce the amount of paper used in printing it. Your scripts should be far more liberally commented.

Lines not preceded by a hash '#' character are taken to be Unix commands and are executed. Any Unix command can be used. For example, the script below displays the current directory name using the **pwd** command, and then lists the directory contents using the **ls** command.

#!/bin/sh

# purpose: print out current directory name and contents

pwd

ls

### Shell Variables

Like every programming language, shells support variables. Shell variables may be assigned values, manipulated, and used. Some variables are automatically assigned for use by the shell.

The script below shows how to assign and use a variable. In general, shell variables are all treated as *strings* (i.e. bits of text). Shells are extremely fussy about getting the syntax exactly right; in the assignment there must be no space between the variable name and the equals sign, or the equals sign and the value. To use the variable, it is prefixed by a dollar '$' character.

#!/bin/sh

# name is a variable

name="fred"

echo "The name is $name"

The special variables $1-$9 correspond to the arguments passed to the script when it is invoked. For example, if we rewrite the script above as shown below, calling the script **name**, and then invoke the command **name Dave Smith**, the message "Your name is Dave Smith" will be printed out:

#!/bin/sh

echo "Your name is $1 $2"

Shell scripts can also do arithmetic, although this does not come particularly naturally to them. The script below adds one to the number passed to it as an argument. To do this, it must use the **expr**command, enclosed in back-quote characters. Once again, precise syntax is critical. You must use the correct type of speech marks, as they have a special meaning (this will be explained later), and the arguments of the **expr** command (**$1**, **+** and **1**) must be separated by spaces.

#!/bin/sh

result=`expr $1 + 1`

echo "Result is $result"

### Conditionals in Shell Scripts

All third-generation programming languages share a number of critical features: sequential execution, use of variables, logical and arithmetic operators, conditional branching and looping. This section is concerned with conditional execution.

Conditionals are used where an action is appropriate only under certain circumstances. The most frequently used conditional operator is the *if-statement*. For example, the shell below displays the contents of a file on the screen using **cat**, but lists the contents of a directory using **ls**.

#!/bin/sh

# show script

if [ -d $1 ]

then

ls $1

else

cat $1

fi

Here, we notice a number of points:

* The if-statement begins with the keyword **if**, and ends with the keyword **fi** (**if**, reversed).
* The **if** keyword is followed by a *condition*, which is enclosed in square brackets. In this case, the condition **-d $1** may be read as: *if $1 is a directory*.
* The line after the **if** keyword contains the keyword **then**.
* Optionally, you may include an **else** keyword.

If the condition is satisfied (in this case, if **$1** is a directory) then the commands between the **then** and **else** keywords are executed; if the condition isn't satisfied then the commands between the**else** and **fi** keywords are executed. If an **else** keyword isn't included, then the commands between the **then** and **fi** keywords are executed if the condition is true; otherwise the whole section is skipped.

There are a number of conditions supported by shell scripts; for a complete list, use the on-line manual on the **test** command (**man test**). Some examples are: **-d** (is a directory?), **-f** (is a file?), **=**(are two strings the same?), **-r** (is string set?), **-eq** (are two numbers equal?), **-gt** (is first number greater than second?). You can also test whether a variable is set to anything, simply by enclosing it in quotes in the condition part of the if-statement. The script below gives an example:

#!/bin/sh

# Script to check that the user enters one argument, "fred"

if [ "$1" ]

then

echo "Found an argument to this script"

if [ $1 = "fred" ]

then

echo "The argument was fred!"

else

echo "The argument was not fred!"

fi

else

echo "This script needs one argument"

fi

The above script illustrates another important feature of if-statements, and this is true of all the other constructs covered in this Guide. It is possible to *nest* constructs, which means to put them inside one another. Here, there is an outer if-statement and an inner one. The inner one checks whether *$1* is "fred", and says whether it is or not. The outer one checks whether *$1* has been given at all, and only goes on to check whether it is "fred" if it does exist. Note that each if-statement has its own corresponding condition, *then*, *else* and *fi* part. The inner if-statement is wholly contained between the *then* and *else* parts of the outer one, which means that it happens only when the first condition is passed.

The **if** condition is suitable if a single possibility, or at most a small number of possibilities, are to be tested. However, it is often the case that we need to check the value of a variable against a number of possibilities. The **case** statement is used to handle this situation. The script below reacts differently, depending on which name is given to it as an argument.

#!/bin/sh

case "$1" in

fred)

echo "Hi fred. Nice to see you"

;;

joe)

echo "Oh! Its you, is it, joe?"

;;

harry)

echo "Clear off!"

;;

\*)

echo "Who are you?"

;;

esac

The case-statement compares the string given to it (in this case "$1", the first argument passed to the script) with the various strings, each of which is followed by a closing bracket. Once a match is found, the statements up to the double semi-colon (;;) are executed, and the case-statement ends. The asterix \* character matches anything, so having this as the last case provides a default case handler (that is, what to do if none of the other cases are matched). The keywords are **case**, **in** and **esac** (end of case).

### Examples

#!/bin/sh

# join command - joins two files together to create a third

# Three parameters must be passed: two to join, the third to create

# If $3 doesn't exist, then the user can't have given all three

if [ "$3" ]

then

# this cat command will write out $1 and $2; the > operator redirects

# the output into the file $3 (otherwise it would appear on the screen)

cat $1 $2 > $3

else

echo "Need three parameters: two input and one output. Sorry."

fi

#!/bin/sh

# An alternative version of the join command

# This time we check that $# is exactly three. $# is a special

# variable which indicates how many parameters were given to

# the script by the user.

if [ $# -eq 3 ]

then

cat $1 $2 > $3

else

echo "Need exactly three parameters, sorry."

fi

#!/bin/sh

# checks whether a named file exists in a special directory (stored in

# the dir variable). If it does, prints out the top of the file using

# the head command.

# N.B. establish your own dir directory if you copy this!

dir=/home/cs0ahu/safe

if [ -f $dir/$1 ]

then

head $dir/$1

fi

### Exercises

1) Write a script called **exists**, which indicates whether a named file exists or not.

2) Write a modified version of the **show** example script (the first example of the if-statement given above), which prints out the message "File does not exist" if the user gives a name which isn't a file or directory, and the message "You must give an argument" if the user doesn't give an argument to the program.

3) Write a script called **save** which copies a file into a special directory, and another called **recover** which copies a file back out of the special directory. The user of the script should not be aware of the location of the special directory (obviously the script will be).

4) Alter your scripts so that, if you try to save a file which already exists in the special directory, the script refuses to save the file and prints out a message to that effect.

### Quoting in Scripts

Confusingly, in Shell scripts no less than three different types of quotes are used, all of which have special meanings. We have already met two of these, and will now consider all three in detail.

Two types of quotes are basically designed to allow you to construct messages and strings. The simplest type of quotes are single quotes; anything between the two quote marks is treated as a simple string. The shell will not attempt to execute or otherwise interpret any words within the string.

The script below simply prints out the message: "your name is fred."

#!/bin/sh

echo 'Your name is fred'

What happens if, rather than always using the name "fred," we want to make the name controlled by a variable? We might then try writing a script like this:

#!/bin/sh

name=fred

echo 'Your name is $name'

However, this will **not** do what we want! It will actually output the message "Your name is $name", because anything between the quote marks is treated as literal text - and that includes $name.

For this reason, shells also understand double quotes. The text between double quotes marks is also interpreted as literal text, except that any variables in it are interpreted. If we change the above script to use double quotes, then it will do what we want:

#!/bin/sh

name=fred

echo "Your name is $name"

The above script writes out the message: "Your name is fred." Double quotes are so useful that we normally use them rather than single quotes, which are only really needed on the rate occasions when you actually want to print out a message with variable names in it.

The third type of quotes are called back-quotes, and we have already seen them in action with the **expr** command. Back-quotes cause the Shell to treat whatever is between the quotes as a command, which is executed, then to substitute the output of the command in its place. This is the main way to get the results of commands into your script for further manipulation. Use of back-quotes is best described by an example:

#!/bin/sh

today=date

echo "Today is $today"

The **date** command prints out today's date. The above script attempts to use it to print out today's date. However, it does not work! The message printed out is "Today is date". The reason for this is that the assignment **today=date** simply puts the string "date" into the variable today. What we actually want to do is to execute the **date** command, and place the *output* of that command into the**today** variable. We do this using back-quotes:

#/bin/sh

today=`date`

echo "Today is $today"

Back-quotes have innumerable uses. Here is another example. This uses the grep command to check whether a file includes the word "and."

#!/bin/sh

# Check for the word "and" in a file

result=`grep and $1`

if [ "$result" ]

then

echo "The file $1 includes the word and"

fi

The *grep* command will output any lines in the file which do include the word "and." We assign the results of the *grep* command to the variable *result*, by using the back-quotes; so if the file does include any lines with the word "and" in them, *result* will end up with some text in it, but if the file doesn't include any lines with the word "and" in them, *result* will end up empty. The if-statement then checks whether result has actually got any text in it.

### Exercises.

5) Write a script which checks whether a given file contains a given word. If it does, the script should output the message "The file contains the word"; if not, it should output the message "The file doesn't contain the word."

6) **[Optional - tricky!]** Write a script which checks whether a given file was created today (Hint: the **ls -l** command includes a listing of the date when a file was created).

#!/bin/sh

today=`date`

result=`grep $today $1`

if [ "result" ]

then

echo "The file $1 was created today"

else

echo "The file $1 was not created today"

fi

### Looping Commands

Whereas conditional statements allow programs to make choices about what to do, looping commands support repetition. Many scripts are written precisely because some repetitious processing of many files is required, so looping commands are extremely important.

The simplest looping command is the **while** command. An example is given below:

#!/bin/sh

# Start at month 1

month=1

while [ $month -le 12 ]

do

# Print out the month number

echo "Month no. $month"

# Add one to the month number

month=`expr $month + 1`

done

echo "Finished"

The above script repeats the while-loop twelve times; with the month number stepping through from 1 to 12. The body of the loop is enclosed between the **do** and **done** commands. Every time the**while** command is executed, it checks whether the condition in the square brackets is true. If it is, then the body of the while-loop is executed, and the computer "loops back" to the **while** statement again. If it isn't, then the body of the loop is skipped.

If a while-loop is ever to end, something must occur to make the condition become untrue. The above example is a typical example of how a loop can end. Here, the month variable is initially set to one. Each time through the loop it is incremented (i.e. has one added to it); once it reaches 12, the condition fails and the loop ends. This is the standard technique for repeating something a set number of times.

Occasionally, it can actually be useful to loop unconditionally, but to break out of the loop when something happens. You can do this using a **while** command with a piece of text as the condition (since the piece of text is always there), and a **break** command to break out of the loop. The computer will go round and round the loop continuously, until such time as it gets to the **break**statement; it will then go to the end of the loop. The break statement is issued from within an if-statement, so that it only happens when you want to loop to end. The example below loops continuously until the user guesses the right word. If you get inadvertently stuck in such a loop, you can always press Ctrl-C to break out.

This example also demonstrates how a shell script can get input from the user using the **read** command. The script loops continuously around the while-loop, asking the user for the password and placing their answer in the **answer** variable. If the **answer** variable is the same as the **password** variable, then the **break** command breaks out of the loop.

#!/bin/sh

password="open"

answer=""

# Loop around forever (until the break statement is used)

while [ "forever" ]

do

# Ask the user for the password

echo "Guess the password to quit the program> \c"

# Read in what they type, and put in it $answer

read answer

# If the answer is the password, break out of the while loop

if [ "$answer" = "$password" ]

then

break

fi

done

# If they get to here, they must've guessed the password,

# because otherwise it would just keep looping

echo "Good guess!"

Another form of looping command, which is useful in other circumstances, is the **for** command. The **for** command sets a variable to each of the values in a list, and executes the body of the command once for each value. A simple example is given below:

#!/bin/sh

for name in fred joe harry

do

echo "Hello $name"

done

The script above prints out the messages "Hello fred," "Hello joe," and "Hello harry." The command consists of the keyword **for**, followed by the name of a variable (in this case, **$name**, but you don't use the dollar in the for-statement itself), followed by the keyword **in**, followed by a list of values. The variable is set to each value in turn, and the code between the do and done keywords is executed once for each value.

The for-loop is most successful when combined with the ability to use wildcards to match file names in the current directory. The for-loop below uses the \* wildcard to match all files and sub-directories in the current directory. Thus, the loop below is executed once for each file or directory, with **$file** set to each one's name. This script checks whether each one is a directory, using the -d option, and only writes out the name if it is. The effect is to list all the sub-directories, but not the files, in the current directory.

#!/bin/sh

for file in \*

do

if [ -d "$file" ]

then

echo "$file"

fi

done

### Exercises

6) Alter your **save** script so that, if a file has previously been saved, the user is asked whether it should be overwritten (Hint: use the **read** command to get the user's decision).

7) Write a script which lists all files in the current directory which have also been stored in your special directory by the **save** script.

8) Adapt your answer to exercise five so that you list all files in your current directory which include a given word.

9) Write an interactive script which allows the user to repeatedly apply the **save** and **restore** scripts. It should continuously prompt the user for commands, which can be either of the form **save <file>**, **restore <file>**, or **quit**. Hint: use **while**, **read** and **case**.

10) Right a script which searches all the sub-directories of your current directory for files with a given name. Hint: use **for**, **if**, **cd**.

### More Sample Scripts

Below is a more complex script, which acts as a DOS command interpreter. DOS uses the commands **cd, dir, type, del, ren** and **copy** to do the same functions as the UNIX commands **cd, ls, cat, rm, mv** and **cp**. This script loops continuously, allowing the user to type in DOS commands, which are stored in the variables **$command**, **$arg1** and **$arg2**. The command is considered by the case statement, which executes an appropriate UNIX command, depending on which DOS command has been give.

#!/bin/sh

# DOS interpreter. Impersonates DOS as follows:

# DOS command UNIX equivalent Action

# cd cd Change directory

# dir ls List directory contents

# type cat List file contents

# del rm Delete a file

# ren mv Rename a file

# copy cp Copy a file

echo "Welcome to the DOS interpreter"

echo "Type Ctrl-C to exit"

# Infinite loop

while [ "forever" ]

do

# Show DOS prompt; \c stops a new line from being issued

echo "DOS> \c"

# Read in user's command

read command arg1 arg2

# Do a UNIX command corresponding to the DOS command

case $command in

cd)

cd $arg1

;;

dir)

ls

;;

type)

cat $arg1

;;

del)

rm $arg1

;;

ren)

mv $arg1 $arg2

;;

copy)

cp $arg1 $arg2

;;

\*)

echo "DOS does not recognise the command $command"

;;

esac

done

## Bourne Shell Summary

### Variables

Variables are assigned using the equals sign, no spaces. They are used by preceding them with a dollar character.

### Arguments

Arguments are labelled $1, $2, ..., $9. $# indicates how many arguments there are. **shift** moves all the arguments down, so that $2 becomes $1, $3 becomes $2, etc. This allows more than nine arguments to be accessed, if necessary.

### Quotes

Single quotes ( 'string' ) form string literals. No interpretation is performed on the string.

Double quotes ( "string" ) form string literals with limited substitution: variables are replaced with their value, and back-quoted commands are replaced with the results of their execution. A backslash '**\**' at the end of a line allows strings to be stretched over a number of lines.

Back quotes ( `string` ) execute the string in a sub-shell, and substitute in the results of the execution.

### Conditionals

**if [ <condition> ] then ... elif ... else ... fi**. **elif** and **else** are optional.

**case <string> in <case1>) ... ;; <case2>) ... ;;; esac**. The case **\*)** acts as a default for any value not matched by one of the earlier cases.

### Looping

**for <variable> in <list> do ... done**

**while [ <condition> ] do ... done**

You can break out of a loop using the **break** command; you can jump to the beginning of the loop and begin its next execution using the **continue** command.

### Expressions

The **expr** command will do calculations. It usually needs to be enclosed in back-quotes, as the result of the calculation will be assigned to some variable. The arguments to the **expr** command *must*be separated by spaces. The value of **`expr 3 + 1`** is "**4**", whereas the value of **`expr 3+1`** (no spaces) is the string "**3+1**".

### Input

User-input can be solicited using the **read** command, which places what the user types into variables. If users are to type several words, **read** can be given a number of arguments.

* Related Searches:
* [**San Luis Obispo Hotels**](http://www.calpoly.edu/~rasplund/script.html)
* [**Academic Support**](http://www.calpoly.edu/~rasplund/script.html)
* [**San Luis Obispo County**](http://www.calpoly.edu/~rasplund/script.html)
* [**California State University Sacramento**](http://www.calpoly.edu/~rasplund/script.html)
* [**Student Success**](http://www.calpoly.edu/~rasplund/script.html)
* [**California State University Campuses**](http://www.calpoly.edu/~rasplund/script.html)
* [**Cal Poly SLO**](http://www.calpoly.edu/~rasplund/script.html)
* [**Academic Excellence Scholarship**](http://www.calpoly.edu/~rasplund/script.html)
* [**Student Scholarships**](http://www.calpoly.edu/~rasplund/script.html)