

# Unit V

- **What Is the dash Shell?**

- The **Debian dash shell** has had an interesting past. It's a direct descendant of the ash shell, a **simple copy of the original Bourne shell** available on Unix systems.
- **Kenneth Almquist** created a small-scale version of the **Bourne shell for Unix systems** and called it the **Almquist shell, which was then shortened to *ash***.
- This original version of the **ash shell was extremely small and fast but without many advanced features**, such as command line editing or history features, making it difficult to use as an interactive shell
- **The NetBSD developers** customized the ash shell by **adding several new features**, making it closer to the Bourne shell.
- The **Debian Linux distribution** created its own version of the ash shell (called Debian ash, or *dash*) for inclusion in its version of Linux. For **the most part, dash copies the features of the NetBSD version of the ash shell, providing the advanced command line editing capabilities**.

- **The dash Shell Features :**
- **The dash command line parameters**

**TABLE 23-1    The dash Command Line Parameters**

Parameter	Description
-a	Exports all variables assigned to the shell
-c	Reads commands from a specified command string
-e	If not interactive, exits immediately if any untested command fails
-f	Displays pathname wildcard characters
-n	If not interactive, reads commands but doesn't execute them
-u	Writes an error message to STDERR when attempting to expand a variable that is not set
-v	Writes input to STDERR as it is read
-x	Writes each command to STDERR as it is executed
-I	Ignores EOF characters from the input when in interactive mode
-i	Forces the shell to operate in interactive mode
-m	Turns on job control (enabled by default in interactive mode)
-s	Reads commands from STDIN (the default behavior if no file arguments are present)
-E	Enables the emacs command line editor
-V	Enables the vi command line editor

- **Positional parameters**

Here are the positional parameter variables available for use in the dash shell:

- `$0`: The name of the shell
- `$n`: The *n*th position parameter
- `$*`: A single value with the contents of all the parameters, separated by the first character in the IFS environment variable, or a space if IFS isn't defined
- `$@`: Expands to multiple arguments consisting of all the command line parameters
- `$#`: The number of positional parameters
- `$?`: The exit status of the most recent command
- `$-`: The current option flags
- `$$`: The process ID (PID) of the current shell
- `$_`: The process ID (PID) of the most recent background command

- **User-defined environment variables:**

- The dash shell also allows you to set your own environment variables. As with bash, you can define a new environment variable on the command line by using the assignment statement:

```
$ testing=10 ; export testing
```

```
$ echo $testing
```

```
10
```

```
$
```

**Without the `export` command, user-defined environment variables are visible only in the current shell or process.**

- The dash built-in commands

Command	Description
alias	Creates an alias string to represent a text string
bg	Continues specified job in background mode
cd	Switches to the specified directory
echo	Displays a text string and environment variables
eval	Concatenates all arguments with a space
exec	Replaces the shell process with the specified command
exit	Terminates the shell process
export	Exports the specified environment variable for use in all child shells
fg	Continues specified job in foreground mode
getopts	Obtains options and arguments from a list of parameters
hash	Maintains and retrieves a hash table of recent commands and their locations
pwd	Displays the value of the current working directory
read	Reads a line from STDIN and assign the value to a variable
readonly	Reads a line from STDIN to a variable that can't be changed
printf	Displays text and variables using a formatted string
set	Lists or sets option flags and environment variables
shift	Shifts the positional parameters a specified number of times
test	Evaluates an expression and returns 0 if true or 1 if false
times	Displays the accumulated user and system times for the shell and all shell processes
trap	Parses and executes an action when the shell receives a specified signal
type	Interprets the specified name and displays the resolution (alias, built-in, command, keyword)
ulimit	Queries or sets limits on processes
umask	Sets the value of the default file and directory permissions
unalias	Removes the specified alias

- **Scripting in dash**

## **Using arithmetic**

Three ways to express a mathematical operation in the bash shell script:

- **Using the `expr` command:** `expr operation`
- **Using square brackets:** `$( operation )`
- **Using double parentheses:** `$(( operation ))`

**The dash shell supports the `expr` command and the double parentheses method but doesn't support the square bracket method.** This can be a problem if you have lots of mathematical operations that use the square brackets.



The proper format for performing mathematical operations in dash shell scripts is to use the double parentheses method:

```
$ cat test5b
#!/bin/dash
# testing mathematical operations

value1=10
value2=15

value3=$(( $value1 * $value2 ))
echo "The answer is $value3"
$ ./test5b
The answer is 150
$
```

Now the shell can perform the calculation properly.



- **The test command**

However, the `test` command available in the dash shell doesn't recognize the `==` symbol for text comparisons. Instead, **it only recognizes the `=` symbol**. If you use the `==` symbol in your bash scripts, you need to change the text comparison symbol to just a single equal sign:

```
$ cat test7
#!/bin/dash
# testing the = comparison

test1=abcdef
test2=abcdef

if [ $test1 = $test2 ]
then
    echo "They're the same!"
else
    echo "They're different"
fi
$ ./test7
They're the same!
$
```

## • The function Command

The dash shell doesn't support the `function` statement. Instead, in the dash shell you must define a function using the function name with parentheses. If you're writing shell scripts that may be used in the dash environment, always define functions using the function name and not the `function()`

stat

```
$ cat test10
#!/bin/dash
# testing functions

func1() {
    echo "This is an example of a function"
}

count=1
while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))
done
echo "This is the end of the loop"
func1
echo "This is the end of the script"
$ ./test10
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is the end of the loop
This is an example of a function
This is the end of the script
$
```

- **The zsh Shell**

Another popular shell that you may run into is the Z shell (called zsh). The zsh shell is an open source Unix shell developed by **Paul Falstad**.

- The following are some of the features that make the zsh shell unique:
  - Improved shell option handling
  - Shell compatibility modes
  - Loadable modules
- A loadable module is the most advanced feature in shell design. **As you've seen in the bash and dash shells, each shell contains a set of built-in commands that are available without the need for external utility programs.**
- **The zsh shell provides a core set of built-in commands, plus the capability to add more *command modules*.**

## **Parts of the zsh Shell**

Shell options Most shells use command line parameters to define the behavior of the shell. The zsh shell

uses a few command line parameters to define the operation of the shell, but mostly it uses *options* to customize the behavior of the shell. You can set shell options either on the command line or within the shell itself using the `set` command.

**TABLE 23-3    The zsh Shell Command Line Parameters**

Parameter	Description
-c	Executes only the specified command and exits
-i	Starts as an interactive shell, providing a command line interface prompt
-s	Forces the shell to read commands from STDIN
-o	Specifies command line options

- **Built-in commands**

**TABLE 23-4 The zsh Core Built-In Commands**

Command	Description
alias	Defines an alternate name for a command and arguments
autoload	Preloads a shell function into memory for quicker access
bg	Executes a job in background mode
bindkey	Binds keyboard combinations to commands
builtin	Executes the specified built-in command instead of an executable file of the same name
bye	The same as exit
cd	Changes the current working directory
chdir	Changes the current working directory
command	Executes the specified command as an external file instead of a function or built-in command
declare	Sets the data type of a variable (same as typeset)
dirs	Displays the contents of the directory stack
disable	Temporarily disables the specified hash table elements
disown	Removes the specified job from the job table
echo	Displays variables and text
emulate	Sets zsh to emulate another shell, such as the Bourne, Korn, or C shells
enable	Enables the specified hash table elements
eval	Executes the specified command and arguments in the current shell



<code>exec</code>	Executes the specified command and arguments replacing the current shell process
<code>exit</code>	Exits the shell with the specified exit status. If none specified, uses the exit status of the last command
<code>export</code>	Allows the specified environment variable names and values to be used in child shell processes
<code>false</code>	Returns an exit status of 1
<code>fc</code>	Selects a range of commands from the history list
<code>fg</code>	Executes the specified job in foreground mode
<code>float</code>	Sets the specified variable for use as a floating point variable
<code>functions</code>	Sets the specified name as a function
<code>getln</code>	Reads the next value in the buffer stack and places it in the specified variable
<code>getopts</code>	Retrieves the next valid option in the command line arguments and places it in the specified variable
<code>hash</code>	Directly modifies the contents of the command hash table
<code>history</code>	Lists the commands contained in the history file
<code>integer</code>	Sets the specified variable for use as an integer value
<code>jobs</code>	Lists information about the specified job or all jobs assigned to the shell process
<code>kill</code>	Sends a signal (Default <code>SIGTERM</code> ) to the specified process or job
<code>let</code>	Evaluates a mathematical operation and assigns the result to a variable
<code>limit</code>	Sets or displays resource limits
<code>local</code>	Sets the data features for the specified variable
<code>log</code>	Displays all users currently logged in who are affected by the watch parameter
<code>logout</code>	Same as <code>exit</code> , but works only when the shell is a login shell
<code>popd</code>	Removes the next entry from the directory stack

<code>print</code>	Displays variables and text
<code>printf</code>	Displays variables and text using C-style format strings
<code>pushd</code>	Changes the current working directory and puts the previous directory in the directory stack
<code>pushln</code>	Places the specified arguments into the editing buffer stack
<code>pwd</code>	Displays the full pathname of the current working directory
<code>read</code>	Reads a line and assigns data fields to the specified variables using the IFS characters
<code>readonly</code>	Assigns a value to a variable that can't be changed
<code>rehash</code>	Rebuilds the command hash table
<code>set</code>	Sets options or positional parameters for the shell
<code>setopt</code>	Sets the options for a shell
<code>shift</code>	Reads and deletes the first positional parameter and shifts the remaining ones down one position
<hr/>	
<code>where</code>	Displays the pathname of the specified command if found by the shell
<code>Which</code>	Displays the pathname of the specified command using csh-style output
<code>zcompile</code>	Compiles the specified function or script for faster autoloading



Command	Description
source	Finds the specified file and copies its contents into the current location
suspend	Suspends the execution of the shell until it receives a SIGCONT signal
test	Returns an exit status of 0 if the specified condition is TRUE
times	Displays the cumulative user and system times for the shell and processes that run in the shell
trap	Blocks the specified signals from being processed by the shell and executes the specified commands if the signals are received
true	Returns a zero exit status
ttyctl	Locks and unlocks the display
type	Displays how the specified command would be interpreted by the shell
typeset	Sets or displays attributes of variables
ulimit	Sets or displays resource limits of the shell or processes running in the shell
umask	Sets or displays the default permissions for creating files and directories
unalias	Removes the specified command alias
unfunction	Removes the specified defined function
unhash	Removes the specified command from the hash table
unlimit	Removes the specified resource limit
unset	Removes the specified variable attribute.
unsetopt	Removes the specified shell option
wait	Waits for the specified job or process to complete
whence	Displays how the specified command would be interpreted by the shell

- **Add-in modules**

There's a long list of modules that provide additional built-in commands for the zsh shell, and the list continues to grow as resourceful programmers create new modules.

**TABLE 23-5    The zsh Modules**

Module	Description
zsh/datetime	Additional date and time commands and variables
zsh/files	Commands for basic file handling
zsh/mapfile	Access to external files via associative arrays
zsh/mathfunc	Additional scientific functions
zsh/pcre	The extended regular expression library
zsh/net/socket	Unix domain socket support
zsh/stat	Access to the stat system call to provide system statistics
zsh/system	Interface for various low-level system features
zsh/net/tcp	Access to TCP sockets
zsh/zftp	A specialized FTP client command
zsh/zselect	Blocks and returns when file descriptors are ready
zsh/zutil	Various shell utilities

- **Viewing and adding modules**

The `zmodload` command is the interface to the zsh modules. You use this command to view, add, and remove modules from the zsh shell session. Using the `zmodload` command without any command line parameters displays the currently installed modules in your zsh shell:

- ```
% zmodload
zsh/zutil
zsh/complete
zsh/main
zsh/terminfo
zsh/zle
zsh/parameter
```

```
%
```

Different zsh shell implementations include different modules by default. To add a new module, just specify the module name on the `zmodload` command line:

```
% zmodload zsh/zftp
```

```
%
```

- **Scripting with zsh**

### **Mathematical operations**

As you would expect, the zsh shell allows you to perform mathematical functions with ease. In the past, the Korn shell has led the way in supporting mathematical operations by providing support for floating-point numbers. The zsh shell has full support for floating point numbers in all its mathematical operations!

Performing calculations

The zsh shell supports two methods for performing mathematical operations:

- The `let` command
- Double parentheses

- When you use the `let` command, you **should enclose the operation in double quotation marks** to allow for spaces:

```
% let value1=" 4 * 5.1 / 3.2 "  
% echo $value1  
6.3750000000  
  
%
```

- The second method is to use the double parentheses. This method incorporates two techniques for defining the mathematical operation:

```
% value1=$(( 4 * 5.1 ))
% (( value2 = 4 * 5.1 ))
% printf "%6.3f\n" $value1 $value2
20.400
20.400

%
```

Notice that you can place the double parentheses either around just the operation (preceded by a dollar sign) or around the entire assignment statement. Both methods produce the same results.

## Mathematical functions

With the zsh shell, built-in mathematical functions are either feast or famine. The default zsh shell doesn't include any special mathematical function. However, **if you install the `zsh/mathfunc` module, you have more math functions than you'll most likely ever need:**

- ```
% value1=$(( sqrt(9) ))
zsh: unknown function: sqrt
% zmodload zsh/mathfunc
% value1=$(( sqrt(9) ))
% echo $value1
3.

%
```

- Structured commands

The zsh shell provides the usual set of structured commands for your shell scripts:

- `if-then-else` statements
- `for` loops (including the C-style)
- `while` loops
- `until` loops
- `select` statements
- `case` statements

- The zsh shell uses the same syntax for each of these structured commands that you're used to from the bash shell. The zsh shell also includes a different structured command called `repeat`. The `repeat` command uses this format:

```
repeat param
do
commands
done
```

```
% cat test1
#!/bin/zsh
# using the repeat command

value1=$(( 10 / 2 ))
repeat $value1
do
    echo "This is a test"
done
$ ./test1
This is a test
This is a test
This is a test
This is a test
This is a test
%
```

- **Functions**

- The zsh shell supports the creation of your own functions either using the `function` command or by defining the function name with parentheses:

```
% function functest1 {  
> echo "This is the test1 function"  
}  
% functest2() {  
> echo "This is the test2 function"  
}  
% functest1  
This is the test1 function  
% functest2  
This is the test2 function  
%
```



# Writing Simple Script Utilities

## •Automating backups

- Whether you're responsible for a Linux system in a business environment or just using it at home, the loss of data can be catastrophic.
- To help prevent bad things from happening, it's always a good idea to perform regular backups (or archives).
- However, what's a good idea and what's practical are often two separate things.
- Trying to arrange a backup schedule to store important files can be a challenge.

## •Archiving data files

- If you're using your Linux system to work on an important project, you can create a shell script that automatically takes snapshots of specific directories.
- Designating these directories in a configuration file allows you to change them when a particular project changes.
- This helps avoid a time consuming restore process from your main archive files.
- This section shows you how to create an automated shell script that can take snapshots of specified directories and keep an archive of your data's past versions.

- **Obtaining the required functions:**

- The workhorse for archiving data in the Linux world is the tar command .
- The tar command is used to archive entire directories into a single file. Here's an example of creating an archive file of a working directory using the tar command:

```
$ tar -cf archive.tar /home/Christine/Project/*.*
tar: Removing leading '/' from member names
$
$ ls -l archive.tar
-rw-rw-r--. 1 Christine Christine 51200 Aug 27 10:51 archive.tar
$
```

- The tar command responds with a warning message that it's removing the leading forward slash from the pathname to convert it from an absolute pathname to a relative pathname

- ❑ This allows you to extract the tar archived files anywhere you want in your filesystem.
- ❑ This allows you to extract the tar archived files anywhere you want in your filesystem.
- ❑ You can accomplish this by redirecting STDERR to the /dev/null file

```
$ tar -cf archive.tar /home/Christine/Project/*.* 2>/dev/null
$
$ ls -l archive.tar
-rw-rw-r--. 1 Christine Christine 51200 Aug 27 10:53 archive.tar
$
```

- ❑ Because a tar archive file can consume lots of disk space, it's a good idea to compress the file. You can do this by simply adding the -z option. This compresses the tar archive file into a gzipped tar file, which is called a tarball.

## Creating a daily archive location

- If you are just backing up a few files, it's fine to keep the archive in your personal directory.
- However, if several directories are being backed up, it is best to create a central repository archive directory

```
$ sudo mkdir /archive
[sudo] password for Christine:
$
$ ls -ld /archive
drwxr-xr-x. 2 root root 4096 Aug 27 14:10 /archive
$
```

- After you have your central repository archive directory created, you need to grant access to it for certain users. If you do not do this, trying to create files in this directory fails, as shown here:

```
$ mv Files_To_Backup /archive/
mv: cannot move 'Files_To_Backup' to
'/archive/Files_To_Backup': Permission denied
$
```

- You could grant the users needing to create files in this directory permission via `sudo` or create a user group. In this case, a special user group is created, `Archivers`:

```
$ sudo groupadd Archivers
$
$ sudo chgrp Archivers /archive
$
$ ls -ld /archive

drwxr-xr-x. 2 root Archivers 4096 Aug 27 14:10 /archive
$
$ sudo usermod -aG Archivers Christine
[sudo] password for Christine:
$
$ sudo chmod 775 /archive
$
$ ls -ld /archive

drwxrwxr-x. 2 root Archivers 4096 Aug 27 14:10 /archive
$
```

- After a user has been added to the Archivers group, the user must log out and log back in for the group membership to take effect. Now files can be created by this group's members without the use of super-user privileges:

```
$ mv Files_To_Backup /archive/  
$  
$ ls /archive  
Files_To_Backup  
$
```

## •Creating an hourly archive script

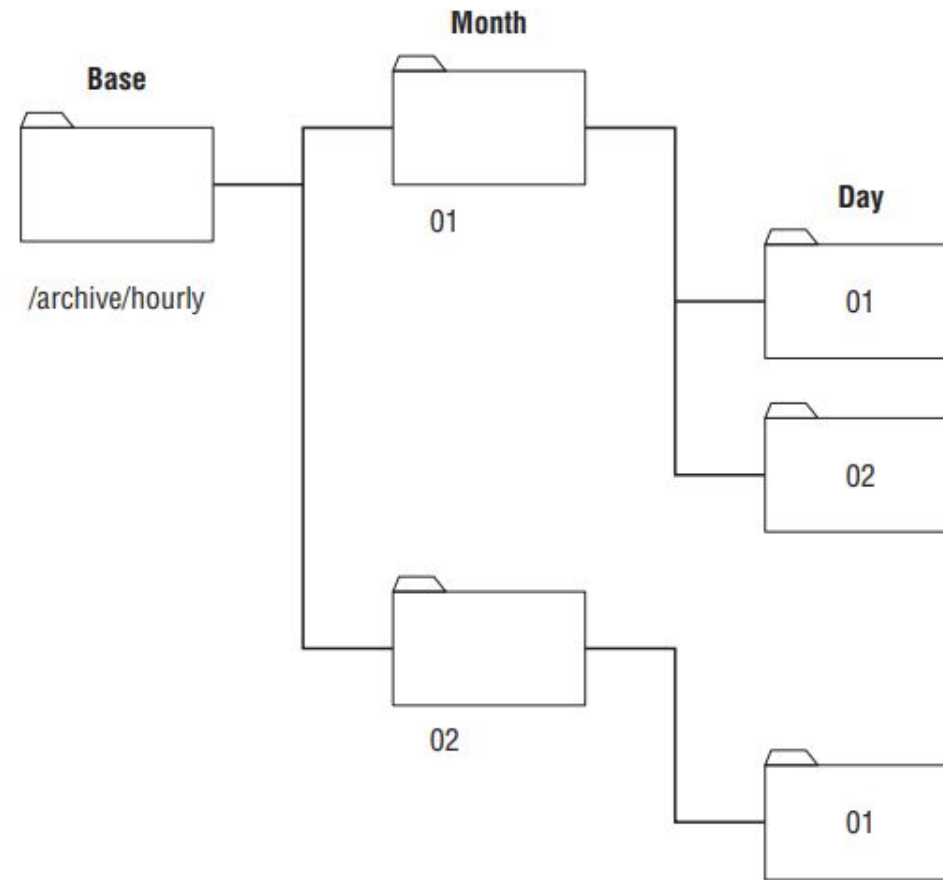
- If you are in a high-volume production environment where files are changing rapidly, a daily archive might not be good enough.
- If you want to increase the archiving frequency to hourly, you need to take another item into consideration



When backing up files hourly and trying to use the `date` command to timestamp each tarball, things can get pretty ugly pretty quickly. Sifting through a directory of tarballs with filenames looking like this is tedious:

`archive010211110233.tar.gz`

Instead of placing all the archive files in the same folder, you can create a directory hierarchy for your archived files.



- First, the new directory /archive/hourly must be created, along with the appropriate permissions set upon it.

```
$ sudo mkdir /archive/hourly
[sudo] password for Christine:
$

$ sudo chgrp Archivars /archive/hourly
$
$ ls -ld /archive/hourly/
drwxr-xr-x. 2 root Archivars 4096 Sep  2 09:24 /archive/hourly/
$
$ sudo chmod 775 /archive/hourly
$
$ ls -ld /archive/hourly
drwxrwxr-x. 2 root Archivars 4096 Sep  2 09:24 /archive/hourly
$
```

After the new directory is set up, the Files\_To\_Backup configuration file for the hourly archives can be moved to the new directory:

```
$ cat Files_To_Backup
/usr/local/Production/Machine_Errors
/home/Development/Simulation_Logs
$
$ mv Files_To_Backup /archive/hourly/
$
```

- **Running the hourly archive script**

As with the `Daily_Archive.sh` script, it's a good idea to test the `Hourly_Archive.sh` script before putting it in the `cron` table. Before the script is run, the permissions must be modified. Also, the hour and minute is checked via the `date` command. Having the current hour and minute allows the final archive filename to be verified for correctness:

```
$ chmod u+x Hourly_Archive.sh
```

```
$
```

```
$ date +%k%M
```

```
1011
```

```
$
```

```
$ ./Hourly_Archive.sh
```

```
Starting archive...
```

```
Archive completed
```

```
Resulting archive file is: /archive/hourly/09/02/archive1011.tar.gz
```

```
$
```

```
$ ls /archive/hourly/09/02/
```

```
archive1011.tar.gz
```

```
$
```

## ❑ **Managing User Accounts:**

Managing user accounts is much more than just adding, modifying, and deleting accounts.

- ❑ You must also consider security issues, the need to preserve work, and the accurate management of the accounts.
- ❑ This can be a time-consuming task.

- **Obtaining the required functions:**

Deleting an account is the more complicated accounts management task. When deleting an account, at least four separate actions are required:

1. Obtain the correct user account name to delete.
2. Kill any processes currently running on the system that belongs to that account.
3. Determine all files on the system belonging to the account.
4. Remove the user account.

## ❑ **Obtaining the required functions**

- ❑ Deleting an account is the more complicated accounts management task. When deleting an account, at least four separate actions are required:
  - 1. Obtain the correct user account name to delete.**
  - 2. Kill any processes currently running on the system that belongs to that account.**
  - 3. Determine all files on the system belonging to the account.**
  - 4. Remove the user account.**
- ❑ It's easy to miss a step. The shell script utility in this section helps you avoid making such mistakes.

- **Getting the correct account name :**

The first step in the account deletion process is the most important: **obtaining the correct user account name to delete**. Because this is an interactive script, you can use the `read` command to obtain the account name. you can use the **-t option on the read command and timeout after giving the script user 60 seconds** to answer the question:

- ```
echo "Please enter the username of the user "  
echo -e "account you wish to delete from system: \c"  
read -t 60 ANSWER
```

- **Creating a function to get the correct account name:**

- The first thing you need to do is declare the function's name, `get_answer`. Next, clear out any previous answers to questions your script user gave using the `unset` command

```
function get_answer {  
#  
unset ANSWER
```

- To ask the script user what account to delete, a few variables must be set and the `get_answer` function should be called. Using the new function makes the script code much simpler:
- ```
LINE1="Please enter the username of the user "  
LINE2="account you wish to delete from system:"  
get_answer  
USER_ACCOUNT=$ANSWER
```

## Verifying the entered account name

Because of potential typographical errors, the user account name that was entered should be verified. This is easy because the code is already in place to handle asking a question:

```
LINE1="Is $USER_ACCOUNT the user account "  
LINE2="you wish to delete from the system? [y/n]"  
get_answer
```

```
    case $ANSWER in  
        y|Y|YES|yes|Yes|yEs|yeS|YEs|yES )  
            #  
            ;;  
        *)  
            echo  
            echo "Because the account, $USER_ACCOUNT, is not "  
            echo "the one you wish to delete, we are leaving the script..."  
            echo  
            exit  
            ;;  
    esac
```



## ❑ **Determining whether the account exists**

- ❑ The user has given us the name of the account to delete and has verified it.
- ❑ Now is a good time to double-check that the user account really exists on the system.
- ❑ The `-w` option allows an exact word match for this particular user account: `USER_ACCOUNT_RECORD=$(cat /etc/passwd | grep -w $USER_ACCOUNT)`

# Removing any account processes

- The script has obtained and verified the correct name of the user account to be deleted.
- In order to remove the user account from the system, the account cannot own any processes currently running.
- Thus, the next step is to find and kill off those processes.
- Here the script can use the **ps** command and the **-u** option to locate any running processes owned by the account.
  - `ps -u $USER_ACCOUNT >/dev/null #Are user processes running?`

## •Finding account files :

- When a user account is deleted from the system, it is a good practice to archive all the files that belonged to that account.
- Along with that practice, it is also important to remove the files or assign their ownership to another account.

## □ Removing the account

- Finally, we get to the main purpose of our script, actually removing the user account from the system. Here the `userdel` command is used:

```
userdel $USER_ACCOUNT
```

## •Monitoring Disk Space:

- One of the biggest problems with multi-user Linux systems is the amount of available disk space.
- In some situations, such as in a file-sharing server, disk space can fill up almost immediately just because of one careless user.

## Obtaining the required functions:

- The first tool you need to use is the **du** command This command displays the **disk usage for individual files and directories**.
- The **-s** option lets you summarize totals at the directory level. This comes in handy when calculating the total disk space used by an individual user. Here's what it looks like to use the **du** command to summarize each user's \$HOME directory for the /home directory contents:

```
$ sudo du -s /home/*
```

```
[sudo] password for Christine:
```

```
4204    /home/Christine
56      /home/Consultant
52      /home/Development
4       /home/NoSuchUser
96      /home/Samantha
36      /home/Timothy
1024    /home/user1
$
```

- The **-s** option works well for users' `$HOME` directories, but what if we wanted to view disk consumption in a system directory such as `/var/log`?

```
$ sudo du -s /var/log/*
4      /var/log/anaconda.ifcfg.log
20     /var/log/anaconda.log
32     /var/log/anaconda.program.log
108    /var/log/anaconda.storage.log
40     /var/log/anaconda.syslog
56     /var/log/anaconda.xlog
116    /var/log/anaconda.yum.log
4392   /var/log/audit
4      /var/log/boot.log
[...]
$
```

- The listing quickly becomes too detailed. **The -S (capital S) option works better for our purposes here, providing a total for each directory and subdirectory individually.** This allows you to pinpoint problem areas quickly:

```
$ sudo du -S /var/log/  
4      /var/log/ppp  
4      /var/log/sss  
3020   /var/log/sa  
80     /var/log/prelink  
4      /var/log/samba/old  
4      /var/log/samba  
4      /var/log/ntpstats  
4      /var/log/cups  
4392   /var/log/audit  
420    /var/log/gdm  
4      /var/log/httpd  
152    /var/log/ConsoleKit  
2976   /var/log/  
$
```

- Because we are interested in the directories consuming the biggest chunks of disk space, the `sort` command is used on the listing produced by `du`: **The `-n` option allows you to sort numerically.** **The `-r` option lists the largest numbers first (reverse order).** This is perfect for finding the largest disk consumers

```
$ sudo du -S /var/log/ | sort -rn
```

```
4392    /var/log/audit
```

```
3020    /var/log/sa
```

```
2976    /var/log/
```

```
420     /var/log/gdm
```

```
152     /var/log/ConsoleKit
```

```
80      /var/log/prelink
```

```
4       /var/log/sss
```

```
4       /var/log/samba/old
```

```
4       /var/log/samba
```

```
4       /var/log/ppp
```

```
4       /var/log/ntpstats
```

```
4       /var/log/httpd
```

```
4       /var/log/cups
```

```
$
```



## ❑ Creating the script:

- ❑ To save time and effort, the script creates a report for multiple designated directories.
- ❑ A variable to accomplish this called CHECK\_DIRECTORIES is used. For our purposes here, the variable is set to just two directories:

CHECK\_DIRECTORIES=" /var/log /home"

- Each time the for loop iterates through the list of values in the variable CHECK\_DIRECTORIES, it assigns to the DIR\_CHECK variable the next value in the list:

```
for DIR_CHECK in $CHECK_DIRECTORIES
do
    [...]
    du -S $DIR_CHECK
    [...]
done
```

- **Running the script:**

- 

```
$ ls -l Big_Users.sh
-rw-r--r--. 1 Christine Christine 910 Sep  3 08:43 Big_Users.sh
$
$ sudo bash Big_Users.sh
[sudo] password for Christine:
$
$ ls disk_space*.rpt
disk_space_090314.rpt
$
$ cat disk_space_090314.rpt
Top Ten Disk Space Usage
```

```
for /var/log /home Directories
```

```
The /var/log Directory:
```

1:	4496	/var/log/audit
2:	3056	/var/log
3:	3032	/var/log/sa
4:	480	/var/log/gdm
5:	152	/var/log/ConsoleKit
6:	80	/var/log/prelink
7:	4	/var/log/sss
8:	4	/var/log/samba/old
9:	4	/var/log/samba
10:	4	/var/log/ppp

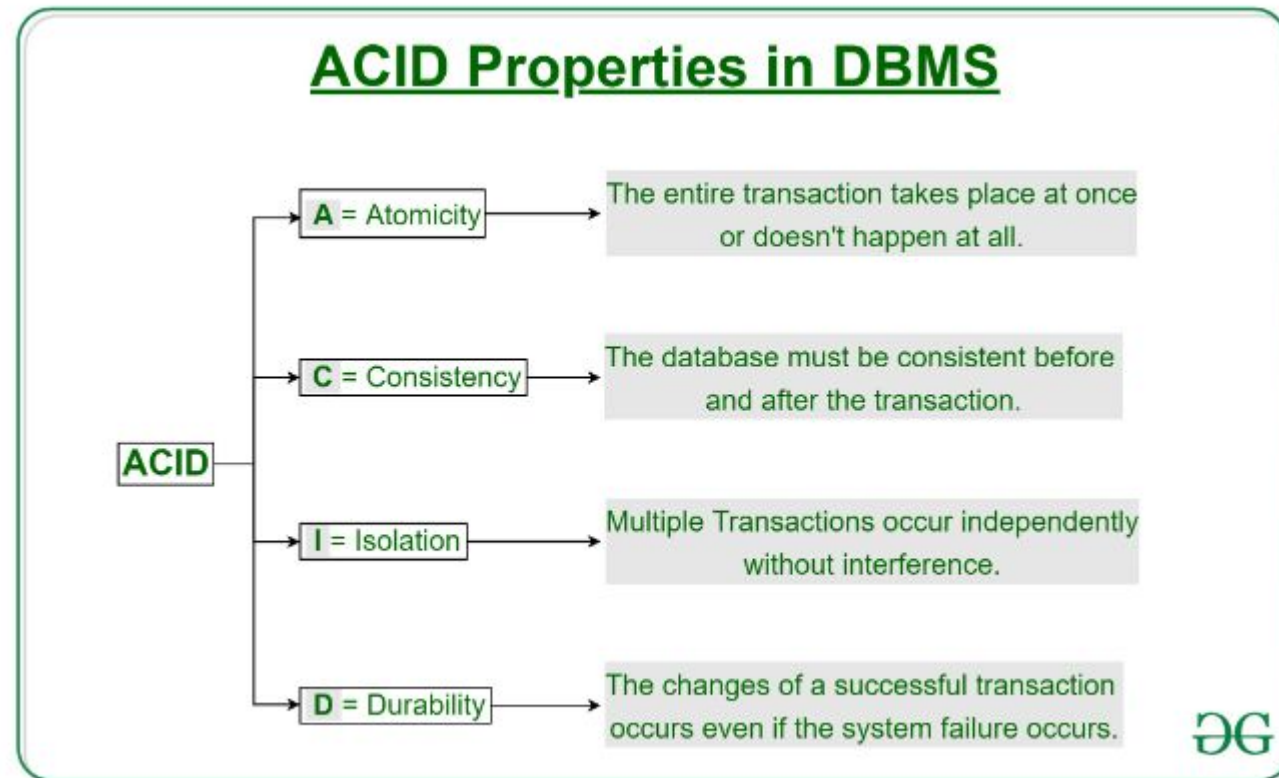
```
The /home Directory:
```

1:	34084	/home/Christine/Documents/temp/reports/archive
2:	14372	/home/Christine/Documents/temp/reports
3:	4440	/home/Timothy/Project__42/log/universe
4:	4440	/home/Timothy/Project_254/Old_Data/revision.56
5:	4440	/home/Christine/Documents/temp/reports/report.txt
6:	3012	/home/Timothy/Project__42/log
7:	3012	/home/Timothy/Project_254/Old_Data/data2039432
8:	2968	/home/Timothy/Project__42/log/answer
9:	2968	/home/Timothy/Project_254/Old_Data/data2039432/answer
10:	2968	/home/Christine/Documents/temp/reports/answer

```
$
```

# Producing Scripts for Database, Web, and E-Mail

- **Data base:**
- A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a [database management system \(D](#)



- **Atomicity:**

- By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

—**Abort:** If a transaction aborts, changes made to the database are not visible.

—**Commit:** If a transaction commits, changes made are visible.  
Atomicity is also known as the 'All or nothing rule'.

Before: X : 500	Y: 200
Transaction T	
<b>T1</b>	<b>T2</b>
Read (X) X: = X - 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

- **Consistency:**

- This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,

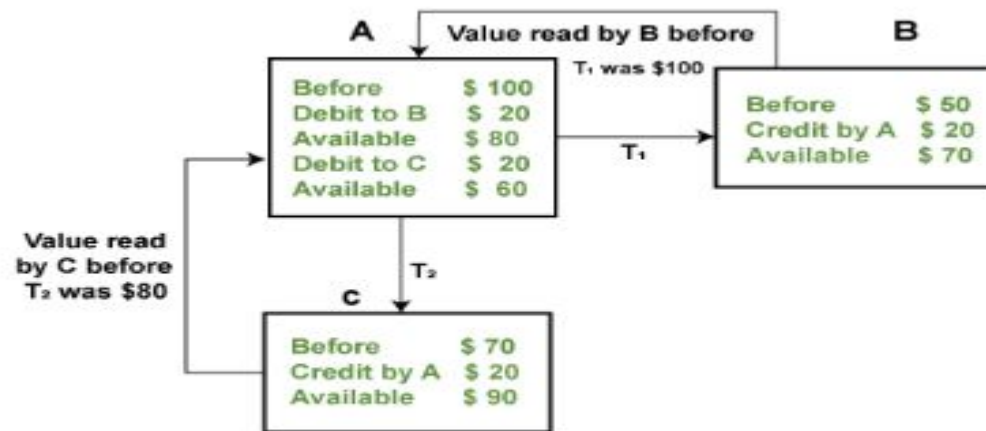
The total amount before and after the transaction must be maintained.

Total **before T** occurs =  $500 + 200 = 700$ .

Total **after T** occurs =  $400 + 300 = 700$ .

Therefore, the database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete.

- **Isolation:** The term 'isolation' means separation. In DBMS, Isolation is the property of a database where no data should affect the other one and may occur concurrently. In short, the operation on one database should begin when the operation on the first database gets complete. It means if two operations are being performed on two different databases, they may not affect the value of one another. In the case of transactions, when two or more transactions occur simultaneously, the consistency should remain maintained. Any changes that occur in any particular transaction will not be seen by other transactions until the change is not committed in the memory.



Isolation - Independent execution of T<sub>1</sub> & T<sub>2</sub> by A

- **Durability:** Durability ensures the permanency of something. In DBMS, the term durability ensures that the data after the successful execution of the operation becomes permanent in the database. The durability of the data should be so perfect that even if the system fails or leads to a crash, the database still survives.



- **Using a MySQL Database:**

```
system@system-virtual-machine:~/Riyaz$ sudo apt update
```

- ```
system@system-virtual-machine:~/Riyaz$ sudo apt install postgresql  
postgresql-contrib
```

```
system@system-virtual-machine:~/Riyaz$ sudo -i -u postgres
```

```
sudo] password for system:
```

```
postgres@system-virtual-machine:~$ psql
```

```
psql (14.5 (Ubuntu 14.5-0ubuntu0.22.04.1))
```

```
Type "help" for help.
```

- The mysql commands:

The `mysql` program uses two different types of commands:

- **Special `mysql` commands**

- **Standard SQL statements**

The `mysql` program uses its own set of commands that let you easily control the environment and retrieve information about the MySQL server. The `mysql` commands use either a full name (such as `status`) or a shortcut (such as `\s`).

```
mysql> \s
```

```
-----
```

```
mysql Ver 14.14 Distrib 5.5.38, for debian-linux-gnu (i686) using readline 6.3
```

```
Connection id:
```

```
Current database:
```

```
Current user: root@localhost
```

```
SSL: Not in use
```

```
Current pager: stdout
```

```
Using outfile: ''
```

```
Using delimiter: ;
```

```
Server version: 5.5.38-0ubuntu0.14.04.1 (Ubuntu)
```

```
Protocol version: 10
```

```
Connection: Localhost via UNIX socket
```

```
Server characterset: latin1
```

```
Db characterset: latin1
```

```
Client characterset: utf8
```

```
Conn. characterset: utf8
```

```
UNIX socket: /var/run/mysqld/mysqld.sock
```

```
Uptime: 2 min 24 sec
```

```
Threads: 1 Questions: 575 Slow queries: 0 Opens: 421 Flush tables: 1
```

```
Open tables: 41 Queries per second avg: 3.993
```

```
-----
```

```
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| mysql      |
+-----+
2 rows in set (0.04 sec)
```

```
mysql> USE mysql;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_mysql |
+-----+
| columns_priv |
| db            |
| func          |
| help_category |
| help_keyword  |
| help_relation |
```

```
| host
| proc
| procs_priv
| tables_priv
| time_zone
| time_zone_leap_second
| time_zone_name
| time_zone_transition
| time_zone_transition_type
| user
+-----+
17 rows in set (0.00 sec)
mysql>
```

- **Creating a database:**

The MySQL server organizes data into *databases*. A database usually holds the data for a single application, separating it from other applications that use the database server.

- Creating a separate database for each shell script application helps eliminate confusion and data mix-ups.

Here's the SQL statement required to create a new database:

```
CREATE DATABASE name;
```

- `mysql> CREATE DATABASE mytest;`

Query OK, 1 row affected (0.02 sec)

```
mysql> SHOW DATABASES;
```

|                    |
|--------------------|
| Database           |
| information_schema |
| mysql              |
| mytest             |

```
3 rows in set (0.01 sec)
```

- **Creating a table**

The MySQL server is considered a *relational* database. In a relational database, data is organized by *data fields*, *records*, and *tables*. A data field is a single piece of information, such as an employee's last name or a salary. A record is a collection of related data fields, such as the employee ID number, last name, first name, address, and salary. Each record indicates one set of the data fields.

```
mysql> CREATE TABLE employees (  
-> empid int not null,  
-> lastname varchar(30),  
-> firstname varchar(30),  
-> salary float,  
-> primary key (empid));  
Query OK, 0 rows affected (0.14 sec)
```

**TABLE 25-1 MySQL Data Types**

| Data Type | Description                                          |
|-----------|------------------------------------------------------|
| char      | A fixed-length string value                          |
| varchar   | A variable-length string value                       |
| int       | An integer value                                     |
| float     | A floating-point value                               |
| boolean   | A Boolean true/false value                           |
| date      | A date value in YYYY-MM-DD format                    |
| time      | A time value in HH:mm:ss format                      |
| timestamp | A date and time value together                       |
| text      | A long string value                                  |
| BLOB      | A large binary value, such as an image or video clip |

- The `empid` data field also specifies a *data constraint*. A data constraint restricts what type of data you can enter to create a valid record. The `not null` data constraint indicates that every record must have an `empid` value specified. Finally, the `primary key` defines a data field that uniquely identifies each individual

record. This means that each data record must have a unique `empid` value in the table.

## Inserting and deleting data

you use the `INSERT` SQL command to insert new data records into the table. Each `INSERT` command must specify the data field values for the MySQL server to accept the record. Here's the format of the `INSERT` SQL command:

```
INSERT INTO table VALUES (...)
```

```
mysql> INSERT INTO employees VALUES (1, 'Blum', 'Rich', 25000.00);
```

```
Query OK, 1 row affected (0.35 sec)
```

- ```
mysql> INSERT INTO employees VALUES (2, 'Blum', 'Barbara', 45000.00);
```

```
Query OK, 1 row affected (0.00 sec)
```

You should now have two data records in your table

- Here's the basic `DELETE` command format:

```
DELETE FROM table;
```

To just specify a single record or a group of records to delete, you must use the `WHERE` clause. The `WHERE` clause allows you to create a filter that identifies which records to remove. You use the `WHERE` clause like this:

```
DELETE FROM employees WHERE empid = 2;
```

This restricts the deletion process to all the records that have an `empid` value of 2. When you execute this command, the `mysql` program returns a message indicating how many records matched the filter:

- ```
mysql> DELETE FROM employees WHERE empid = 2;  
Query OK, 1 row affected (0.29 sec)
```

As expected, only one record matched the filter and was removed.

## Querying data

- Here's the basic format of a `SELECT` statement:

```
SELECT datafields FROM table
```



```
mysql> SELECT * FROM employees;
```

| empid | lastname | firstname  | salary |
|-------|----------|------------|--------|
| 1     | Blum     | Rich       | 25000  |
| 2     | Blum     | Barbara    | 45000  |
| 3     | Blum     | Katie Jane | 34500  |
| 4     | Blum     | Jessica    | 52340  |

```
4 rows in set (0.00 sec)
```

You can use one or more modifiers to define how the database server returns the data requested by the query. Here's a list of commonly used modifiers:

- **WHERE:** Displays a subset of records that meet a specific condition
- **ORDER BY:** Displays records in a specified order
- **LIMIT:** Displays only a subset of records

- Sending commands to the server After establishing the connection to the server, you'll want to send commands to interact with your database. There are two methods to do this:
  - Send a single command and exit.
  - Send multiple commands.
- To send a single command, you must include the command as part of the `mysql` command line. For the `mysql` command, you do this using the `-e` parameter:

```
$ cat mtest1
```

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

```
# send a command to the MySQL server
```

```
MYSQL=$(which mysql)
```

```
$MYSQL mytest -u test -e 'select * from employees'
```

```
$ ./mtest1
```

| empid | lastname | firstname  | salary |
|-------|----------|------------|--------|
| 1     | Blum     | Rich       | 25000  |
| 2     | Blum     | Barbara    | 45000  |
| 3     | Blum     | Katie Jane | 34500  |
| 4     | Blum     | Jessica    | 52340  |

- If you need to send more than one SQL command, you can use file redirection To redirect lines in the shell script, you must define an *end of file* string. The end of file string indicates the beginning and end of the redirected data.

This is an example of defining an end of file string, with data in it:

```
$ cat mtest2
#!/bin/bash
# sending multiple commands to MySQL

MYSQL=$(which mysql)
$MYSQL mytest -u test <<EOF
show tables;
select * from employees where salary > 40000;
EOF
$ ./mtest2
Tables_in_test
employees
empid    lastname    firstname    salary
2        Blum        Barbara      45000
4        Blum        Jessica      52340
$
```

- **Formatting data**

The standard output from the `mysql` command doesn't lend itself to data retrieval. If you need to actually do something with the data you retrieve, you need to do some fancy data manipulation. This section describes some of the tricks you can use to help extract data from your database reports.

- The `mysql` program also supports an additional popular format, called Extensible Markup Language (XML). This language uses HTML-like tags to identify data names and values. For the `mysql` program, you do this using the `-X` command line parameter:

```
$ mysql mytest -u test -X -e 'select * from employees where empid = 1'
<?xml version="1.0"?>

<resultset statement="select * from employees">
  <row>
    <field name="empid">1</field>
    <field name="lastname">Blum</field>
    <field name="firstname">Rich</field>
    <field name="salary">25000</field>
  </row>
</resultset>
$
```

- Using XML, you can easily identify individual rows of data, along with the individual data values in each record. You can then use standard Linux string handling functions to extract the data you need!

- **Using the Web:**

- Almost as old as the Internet itself, the Lynx program was **created in 1992 by students at the University of Kansas as a text-based browser**. Because it's text-based, the Lynx program allows you to browse websites directly from a terminal session, replacing the fancy graphics on web pages with HTML text tags.
- Lynx uses the standard keyboard keys to navigate around the web page. **Links appear as highlighted text within the web page. Using the right-arrow key allows you to follow a link to the next web page.**

- **Installing Lynx**

- Even though the **Lynx program is somewhat old**, it's still in active development. At the time of this writing, the **latest version of Lynx is version 2.8.8, released in June 2010**, with a new release in development. Because of its popularity among shell script programmers, many Linux distributions install the Lynx program in their default installations.

- **The lynx command line:**

The `lynx` command line command is extremely versatile in what information it can retrieve from the remote website. When you view a web page in your browser, you're only seeing part of the information that's transferred to your browser. Web pages consist of three types

of data elements:

- HTTP headers
- Cookies
- HTML content

- *HTTP headers* provide information about the type of data sent in the connection, the server sending the data, and the type of security used in the connection. If you're sending special types of data, such as video or audio clips, the server identifies that in the HTTP headers. The Lynx program allows you to view all the HTTP headers sent within a web page session.
- If you've done any type of web browsing, no doubt you're familiar with web page *cookies*. Websites use cookies to store data about your website visit for future use. Each individual site can store information, but it can only access the information it sets. The `lynx` command provides options for you to view cookies sent by web servers, as well as reject or accept specific cookies sent from servers.

- The Lynx program allows you to view the actual HTML content of the web page in three different formats:
  - In a **text-graphics display on the terminal session** using the curses graphical library
  - As a text file, **dumping the raw data** from the web page
  - As a text file, **dumping the raw HTML source code** from the web page

- **The Lynx configuration file**

The `lynx` command reads a configuration file for many of its parameter settings. By default, this file is located at `/usr/local/lib/lynx.cfg`, although you'll find that many Linux distributions change this to the `/etc` directory (`/etc/lynx.cfg`) (the Ubuntu distribution places the `lynx.cfg` file in the `/etc/lynx-cur` folder). The `lynx.cfg` configuration file groups related parameters into sections to make finding parameters easier. Here's the format of an entry in the configuration file:

- *PARAMETER:value*  
where *PARAMETER* is the full name of the parameter (often, but not always in uppercase letters) and *value* is the value associated with the parameter.  
such as the `ACCEPT_ALL_COOKIES` parameter,
- The most common configuration parameters that you can't set on the command line are for the *proxy servers*. **Some networks (especially corporate networks) use a proxy server as a middleman between the client's browser and the destination website server.** Instead of sending HTTP requests directly to the remote web server, client browsers must send their requests to the proxy server. The proxy server in turn sends the requests to the remote web server, retrieves the results, and forwards them back to the client browser.



- **Using E-Mail** :The main tool you have available for sending e-mail messages from your shell scripts is the Mailx program. Not only can you use it interactively to read and send messages, but you can also use the command line parameters to specify how to send a message.

The Mailx program sends the text from the `echo` command as the message body. This provides an easy way for you to send messages from your shell scripts. Here's a quick example:

```
$ cat factmail
#!/bin/bash
# mailing the answer to a factorial

MAIL=$(which mailx)

factorial=1
counter=1

read -p "Enter the number: " value
while [ $counter -le $value ]
do
    factorial=$((factorial * $counter))
    counter=$((counter + 1))
done
```

done

```
echo "The factorial of $value is $factorial" | $MAIL -s "Factorial
answer" $USER
echo "The result has been mailed to you."
```

- This script does not assume that the Mailx program is located in the standard location. It uses the `which` command to determine just where the `mail` program is. After calculating the result of the factorial function, the shell script uses the `mail` command to send the message to the user-defined `$USER` environment variable, which should be the person executing the script.

```
$ ./factmail
Enter the number: 5
The result has been mailed to you.
$
```

You just need to check your mail to see if the answer arrived:

```
$ mail
"/var/mail/rich": 1 message 1 new
>N  1 Rich Blum          Mon Sep  1 10:32  13/586  Factorial answer
?
Return-Path: <rich@rich-Parallels-Virtual-Platform>
X-Original-To: rich@rich-Parallels-Virtual-Platform
Delivered-To: rich@rich-Parallels-Virtual-Platform
Received: by rich-Parallels-Virtual-Platform (Postfix, from userid 1000)
        id B4A2A260081; Mon,  1 Sep 2014 10:32:24 -0500 (EST)
Subject: Factorial answer
To: <rich@rich-Parallels-Virtual-Platform>
X-Mailer: mail (GNU Mailutils 2.1)
Message-Id: <20101209153224.B4A2A260081@rich-Parallels-Virtual-Platform>
Date: Mon,  1 Sep 2014 10:32:24 -0500 (EST)
From: rich@rich-Parallels-Virtual-Platform (Rich Blum)

The factorial of 5 is 120
?
```

- **What is Python?**

- Python is an object-oriented interpreted language that is designed to be easy to use and to aid **Rapid Application Development**. This is achieved by the use of simplified semantics in the language.
- Python was **created at the end of the 1980s, towards the very end of December 1989**, by the **Dutch developer Guido van Rossum**. The majority of the design of the language aims for clarity and simplicity

If you are using another Linux distribution or Python 3 is not found for any reason, you can install it like this:

On RedHat based distributions:

**\$ sudo yum install python36**

On Debian based distributions:

**\$ sudo apt-get install python3.6**

- We can see that we are presented with >>> the prompt and this is known as the REPL console. We should emphasize that this is a scripting language and, like bash and Perl, we will normally execute code through the text files that we create. Those text files will normally be expected to have a QZ suffix to their name.

- While working with REPL, we can print the version independently by importing a module. In Perl, we will use the keyword; in bash we will use the command source; and in Python we use import:

```
>>>import sys
```

With the module loaded, we can now investigate the object-oriented nature of Python by printing the version:

```
>>> sys.version
```

```
>>> import sys
>>> sys.version
'3.2.3 (default, Mar  1 2013, 11:53:50) \n[GCC 4.6.3]'
>>> _
```

We will navigate to the TZT object within our namespace and call the version method from that object.

**Finally, to close the REPL, we will use *Ctrl* + *D* in Linux or *Ctrl* + *Z* in Windows.**

- **Saying Hello World the Python way:**

The Print function includes the newline and we do not need semicolons at the end of the line. We can see the edited version of **\$HOME/bin/hello.py** in the following example:

```
#!/usr/bin/python3
print("Hello World")
```

- We will still need to add the execute permission, but we can run the code as earlier using Chmod. This is shown in the following command but we should be a little used to this now:

**\$ chmod u+x \$HOME/bin/hello.py**

Finally, we can now execute the code to see our greeting. Similarly, you can run the file using the Python interpreter from the command line like this:

- **\$ python3 \$HOME/bin/hello.py**

- Or in some Linux distributions, you can run it like this:

**\$ python36 \$HOME/bin/hello.py**

- **Pythonic arguments:**

We should know by now that we will want to pass command-line arguments to Python and we can do this using the **BSHW** array. However, we are more like bash; with Python we combine the program name into the array with the other arguments.

Python also uses lowercase instead of uppercase in the object name:

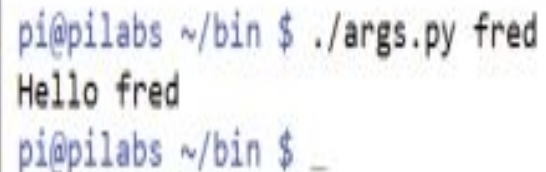
- The `argv` array is a part of the `sys` object
- `sys.argv[0]` is the script name
- `sys.argv[1]` is the first argument supplied to the script
- `sys.argv[2]` is the second supplied argument and so on
- The argument count will always be at least 1, so, keep this in mind when checking for supplied arguments

- Supplying arguments:

If we create the `$HOME/bin/args.py` file we can see this in action. The file should be created as follows and made executable:

```
#!/usr/bin/python3
import sys
print("Hello " + sys.argv[1])
```

If we run the script with a supplied argument, we should see something similar to the following screenshot:



```
pi@pilabs ~/bin $ ./args.py fred
Hello fred
pi@pilabs ~/bin $ _
```

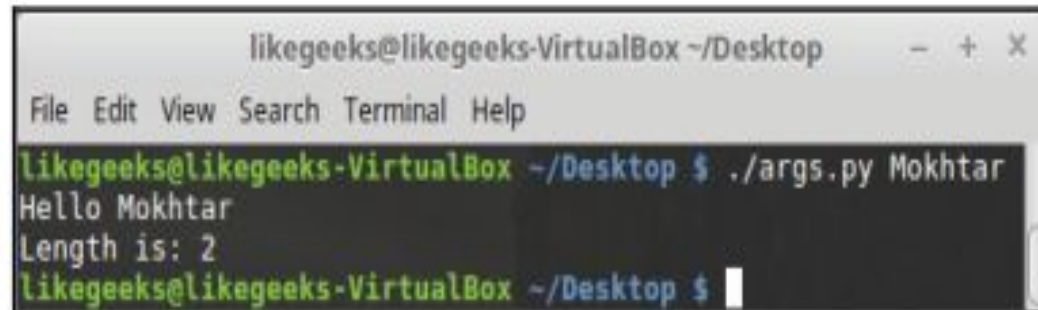


- **Counting arguments** : The script name is the first argument at index 0 of the array. So, if we try to count the arguments, then the count should always be at the very least 1.
- In other words, if we have not supplied arguments, the argument count will be 1. To count the items in an array, we can use the `len()` function.

If we edit the script to include a new line we will see this work, as follows:

```
#!/usr/bin/python3
import sys
print("Hello " + sys.argv[1])
print( "length is: " + str(len(sys.argv)) )
```

Executing the code as we have earlier, we can see that we have supplied two arguments—the script name and then the string Mokhtar:

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the execution of a Python script named 'args.py' with the argument 'Mokhtar'. The output of the script is 'Hello Mokhtar' followed by 'Length is: 2'. The prompt is ready for the next command.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./args.py Mokhtar
Hello Mokhtar
Length is: 2
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

- **Significant whitespace:**

- The indent level of your code defines the block of code to which it belongs. So far, we have not indented the code we have created past the start of the line. This means that all of the code is at the same indent level and belongs to the same code block.
- Rather than using brace brackets or the do and done keywords to define the code block, we use indents. If we indent with two or four spaces or even tabs, then we must stick to those spaces or tabs. When we return to the previous indent level, we return to the previous code block.

```
#!/usr/bin/python3
import sys
count = len(sys.argv)
if ( count > 1 ):
    print("Arguments supplied: " + str(count))
    print("Hello " + sys.argv[1])
print("Exiting " + sys.argv[0])
```

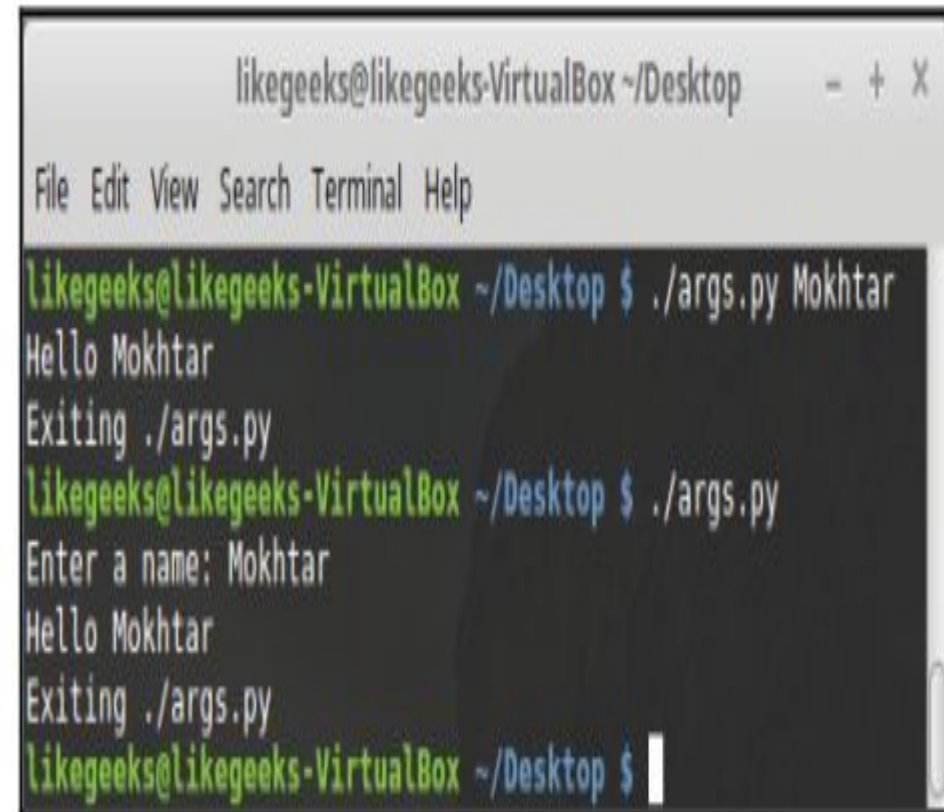
```
pi@pilabs ~/bin $ ./args.py
Exiting ./args.py
pi@pilabs ~/bin $ ./args.py fred
Arguments supplied: 2
Hello fred
Exiting ./args.py
```

- Reading user input :

```
#!/usr/bin/python3
import sys
count = len(sys.argv)
name = ''

if ( count == 1 ):
    name = input("Enter a name: ")
else:
    name = sys.argv[1]

print("Hello " + name)
print("Exiting " + sys.argv[0])
```



A terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the execution of a script named './args.py'. In the first run, the script is called with the argument 'Mokhtar', resulting in 'Hello Mokhtar' and 'Exiting ./args.py'. In the second run, the script is called without arguments, resulting in a prompt 'Enter a name: Mokhtar', followed by 'Hello Mokhtar' and 'Exiting ./args.py'. The prompt character is a dollar sign '\$'.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./args.py Mokhtar
Hello Mokhtar
Exiting ./args.py
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./args.py
Enter a name: Mokhtar
Hello Mokhtar
Exiting ./args.py
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

- **Using Python to write to files :**

We will start by making a copy of our existing **args.py** We will copy this **\$HOME/bin/file.py**. The new **file.py** should read similar to the following screenshot and have the execute permission set:

```
#!/usr/bin/python3
import sys
count = len(sys.argv)
name = ''

if ( count == 1 ):
    name = input("Enter a name: ")
else:
    name = sys.argv[1]

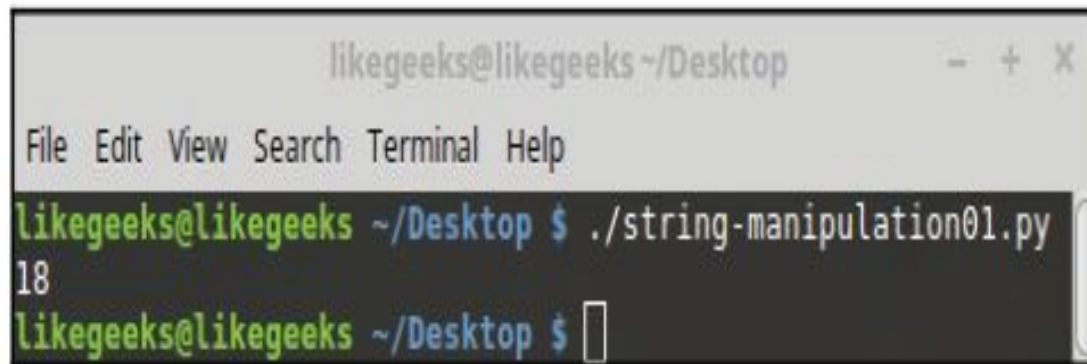
log = open("/tmp/script.log","a")
log.write("Hello " + name + "\n")
log.close()
```

- **String manipulation:**

Dealing with strings in Python is very simple: you can search, replace, change character case, and perform other manipulations with ease:

To search for a string, you can use the find method like this:

```
#!/usr/bin/python3
str = "Welcome to Python scripting world"
print(str.find("scripting"))
```

A terminal window titled 'likegeeks@likegeeks ~/Desktop' with standard window controls. The menu bar includes 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the command './string-manipulation01.py' being executed, which outputs the number '18'. The prompt then returns to the shell.

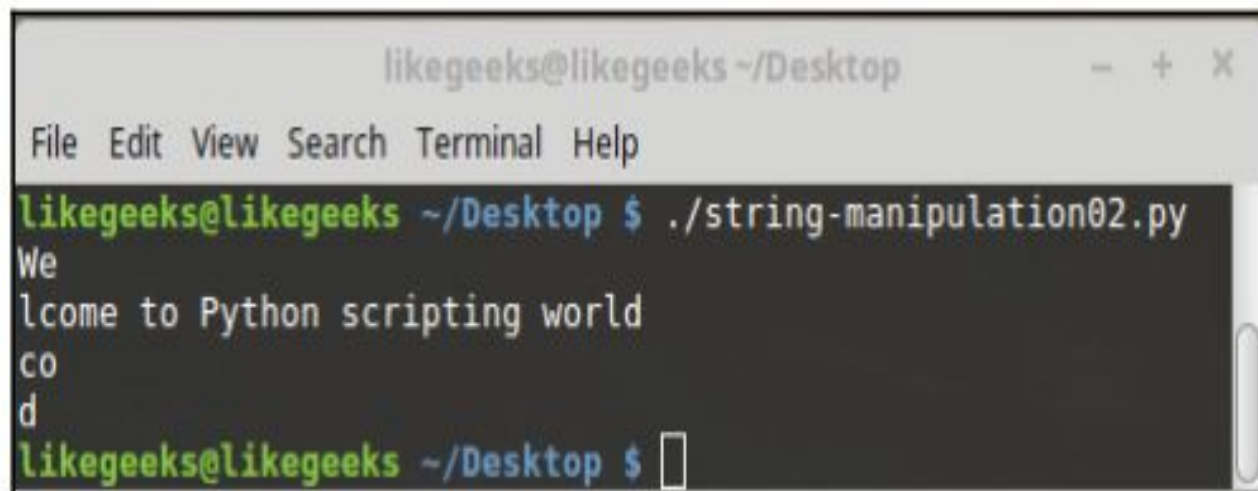
```
likegeeks@likegeeks ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks ~/Desktop $ ./string-manipulation01.py
18
likegeeks@likegeeks ~/Desktop $
```

The string count in Python starts from zero too, so the position of the word scripting is at 18.



You can get a specific substring using square brackets like this:

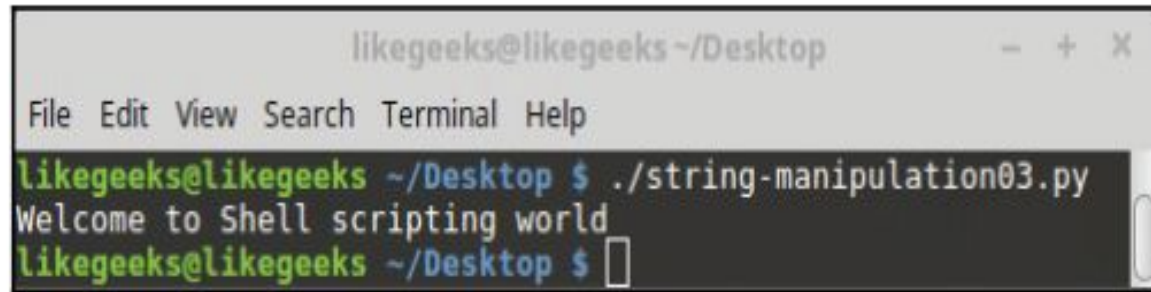
```
#!/usr/bin/python3
str = "Welcome to Python scripting world"
print(str[:2]) # Get the first 2 letters (zero based)
print(str[2:]) # Start from the second letter
print(str[3:5]) # from the third to fifth letter
print(str[-1]) # -1 means the last letter if you don't know the length
```

A terminal window titled 'likegeeks@likegeeks ~/Desktop' with standard window controls. The menu bar includes 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the command './string-manipulation02.py' being executed, which outputs the text 'Welcome to Python scripting world' across four lines. The prompt 'likegeeks@likegeeks ~/Desktop \$' is visible at the bottom with a cursor.

```
likegeeks@likegeeks ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks ~/Desktop $ ./string-manipulation02.py
We
lcome to Python scripting world
co
d
likegeeks@likegeeks ~/Desktop $
```

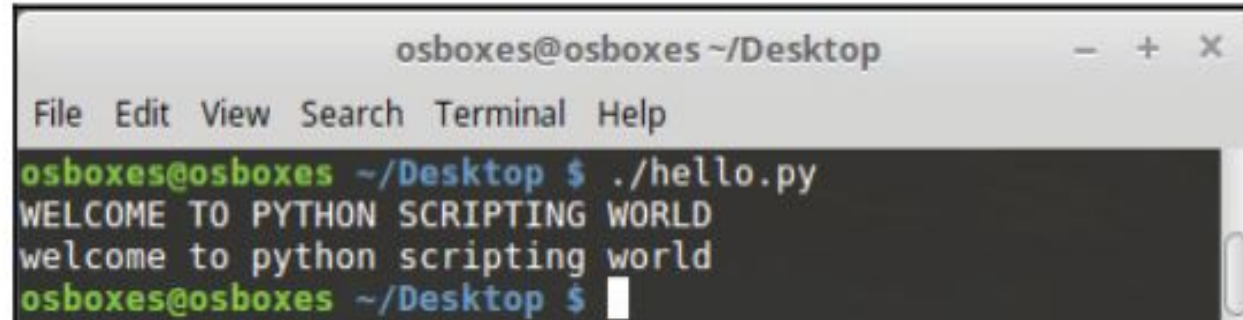
To replace a string, you can use the replace method like this:

```
#!/usr/bin/python3
str = "Welcome to Python scripting world"
str2 = str.replace("Python", "Shell")
print(str2)
```



A terminal window titled 'likegeeks@likegeeks ~/Desktop' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'likegeeks@likegeeks ~/Desktop \$'. The command './string-manipulation03.py' is entered and executed, resulting in the output 'Welcome to Shell scripting world'. The prompt is now 'likegeeks@likegeeks ~/Desktop \$' with a cursor.

To change the character case, you can use upper () and lower () functions:



A terminal window titled 'osboxes@osboxes ~/Desktop' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'osboxes@osboxes ~/Desktop \$'. The command './hello.py' is entered and executed, resulting in the output 'WELCOME TO PYTHON SCRIPTING WORLD' followed by 'welcome to python scripting world'. The prompt is now 'osboxes@osboxes ~/Desktop \$' with a cursor.