Coding on Linux

Linux is such a versatile operating system that's both flexible and powerful, while offering the programmer a perfect foundation on which to build their skills. While all the popular and mainstream programming languages are available on Linux, as they are on Windows and macOS, Linux also utilises its own coding language, called scripting.

Bash scripting on Linux can be used to create a wealth of useful, realworld programs that interact with the user, or simply work in the background based on a predefined schedule. Scripting is a powerful interface to the Linux system, so we've crafted this section to help you get to grips with how everything fits together, and how to make some amazing Linux scripts.

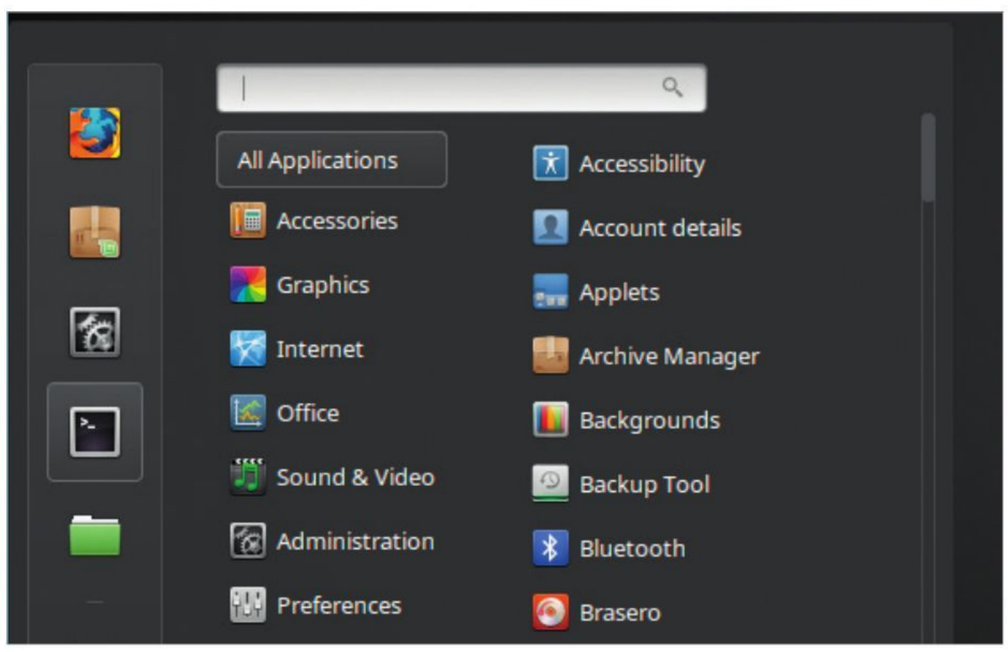
Getting Ready to Code in Linux

Operating systems use two kinds of interface: the GUI, which is the desktop that Windows etc. boots into and the command line, which is what we're going to be using here. In this section we're using a distro called Linux Mint, found at www.linuxmint.com/.

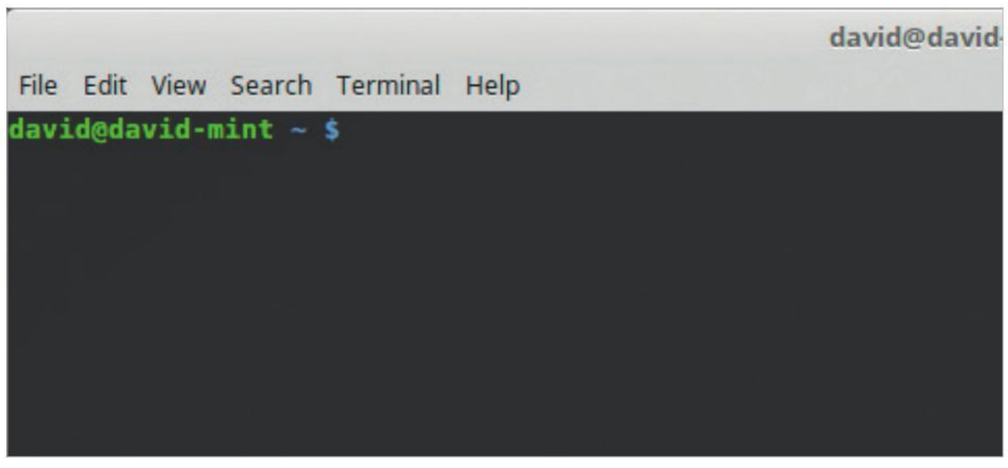
TAKING COMMAND

Linux Mint is a great distribution that's easy to use and exceedingly powerful. For more information check out BDM's Linux titles: https://bdmpublications.com/?s=Linux&post_type=product.

STEP 1 We won't go into the installation of Linux Mint here but as it's Debian-based, the following section will also work with Ubuntu, Raspbian OS and other Linux distributions. The Terminal is where you're going to start scripting in Mint. It can be accessed by clicking on the Terminal icon in the Panel or by opening the Mint Menu and selecting it from the left-strip.

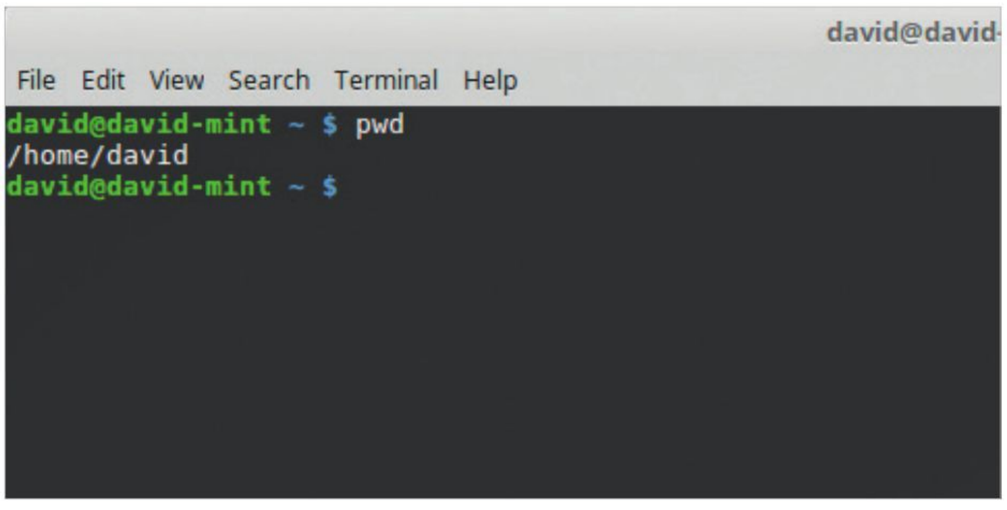


STEP 2 The Terminal will give you access to the Linux Mint Shell, called BASH; this gives you access to the underlying operating system, which is why scripting is such a powerful language to learn and use. Everything in Mint, and Linux as a whole, including the desktop and GUI, is a module running from the command line.

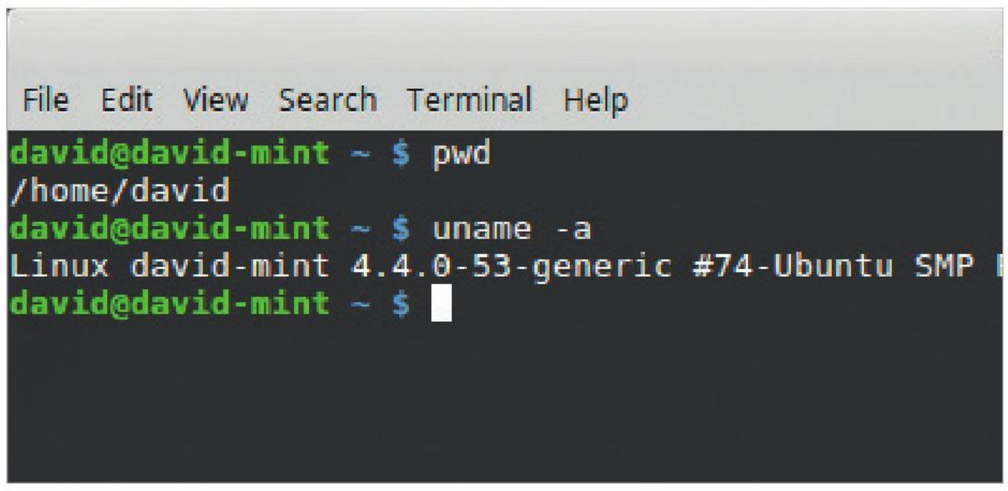


STEP 3 What you currently see in the Terminal is your login name followed by the name of the computer, as you named it when you first installed the OS on to the computer. The line then ends with the current folder name; at first this is just a tilde (~), which means your Home folder.

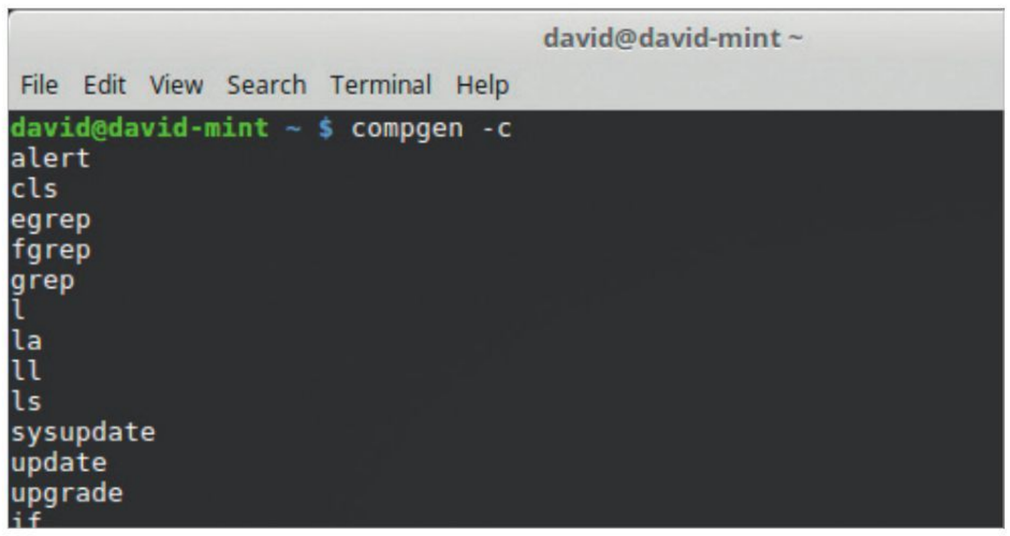
STEP 4 The flashing cursor at the very end of the line is where your text-based commands will be entered. You can begin to experiment with a simple command, Print Working Directory (pwd), which will output to the screen the current folder you're in. Type: pwd and press Enter.



STEP 5 All the commands you enter will work in the same manner. You enter the command, include any parameters to extend the use of the command and press Enter to execute the command line you've entered. Now type: `uname -a` and press Enter. This will display information regarding Linux Mint. In scripting, you can use all the Linux command-line commands within your own scripts.



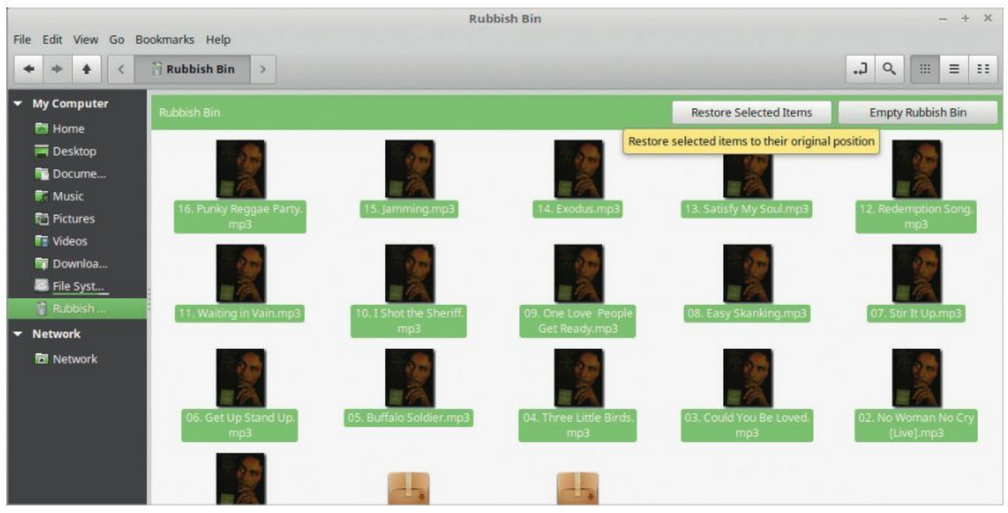
STEP 6 The list of available Linux commands is vast, with some simply returning the current working directory, while others are capable of deleting the entire system in an instant. Getting to know the commands is part of learning how to script. By using the wrong command, you could end up wiping your computer. Type `compgen -c` to view the available commands.



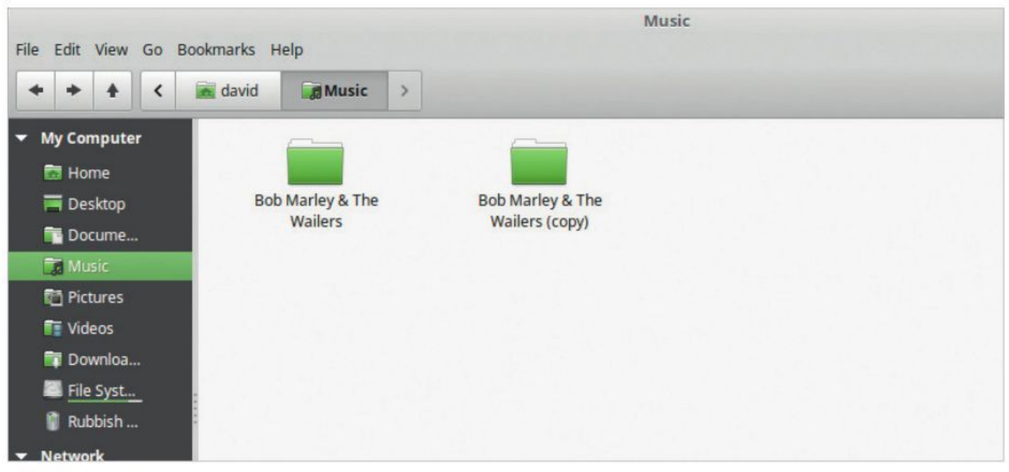
HERE BE DRAGONS!

There's a story on the Internet that an employee at Disney Pixar nearly ruined the animated movie Toy Story by inadvertently entering the wrong Linux command and deleting the entire system the film was stored on.

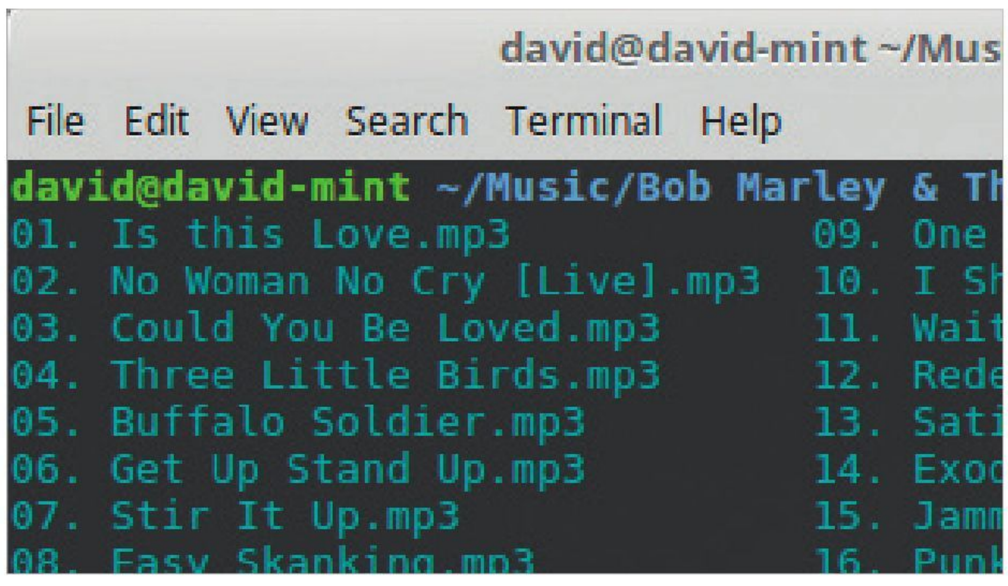
STEP 1 Having access to the Terminal means you're bypassing the GUI desktop method of working with the system. The Terminal is a far more powerful environment than the desktop, which has several safeguards in place in case you accidentally delete all your work, such as Rubbish Bin to recover deleted files.



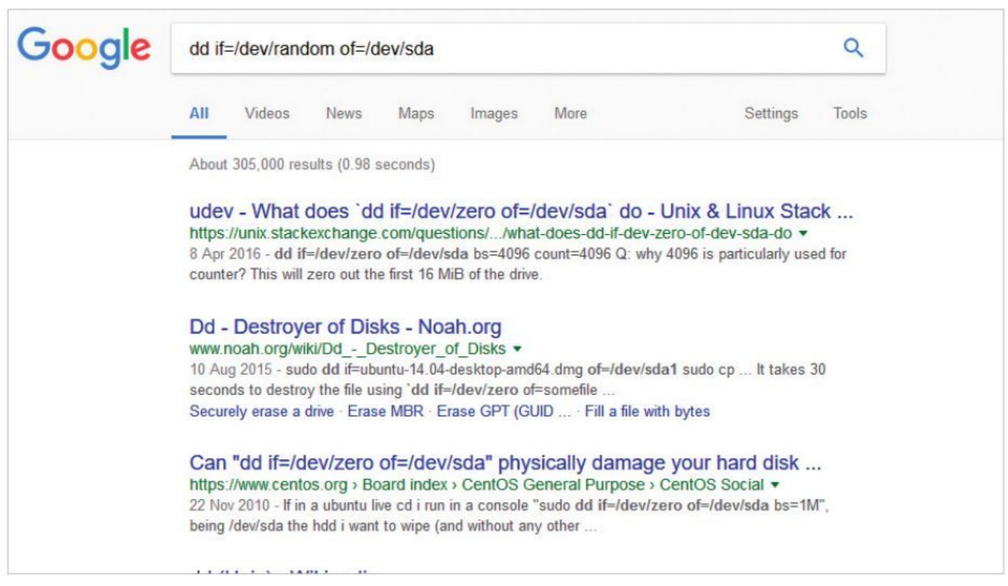
STEP 3 Therefore it's always a good idea to work in the Terminal using a two-pronged approach. First, use the desktop to make regular backups of the folders you're working in when in the Terminal. This way, should anything go wrong, there's a quick and handy backup waiting for you.



STEP 2 However, the Terminal doesn't offer that luxury. If you were to access a folder with files within via the Terminal and then enter the command: `rm *.*`, all the files in that folder would be instantly deleted. They won't appear in the Rubbish Bin either, they're gone for good.



STEP 4 Second, research before blindly entering a command you've seen on the Internet. If you see the command: `sudo dd if=/dev/random of=/dev/sda` and use it in a script, you'll soon come to regret the action as the command will wipe the entire hard drive and fill it with random data. Take a moment to Google the command and see what it does.



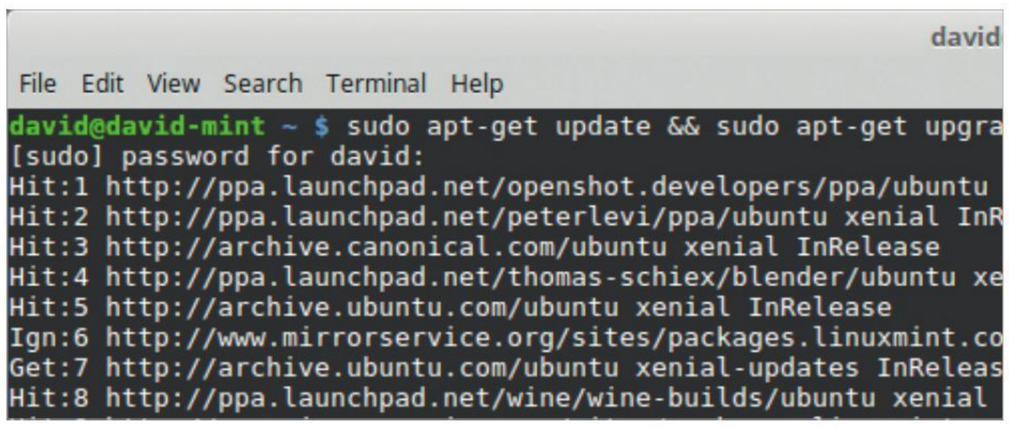
Creating Bash Scripts – Part 1

Eventually, as you advance with Linux Mint, you'll want to start creating your own automated tasks and programs. These are essentially scripts, Bash Shell scripts to be exact, and they work in the same way as a DOS Batch file does, or any other programming language.

GET SCRIPTING

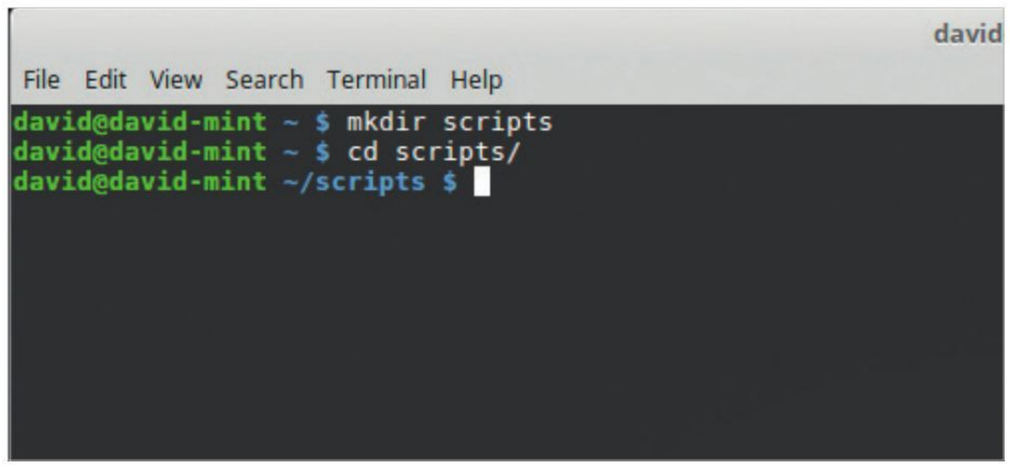
A Bash script is simply a series of commands that Mint will run through to complete a certain task. They can be simple or remarkably complex, it all depends on the situation.

STEP 1 You'll be working within the Terminal and with a text editor throughout the coming pages. There are alternatives to the text editor, which we'll look at in a moment but for the sake of ease, we'll be doing our examples in Xed. Before you begin, however, run through the customary update check: `sudo apt-get update && sudo apt-get upgrade`.

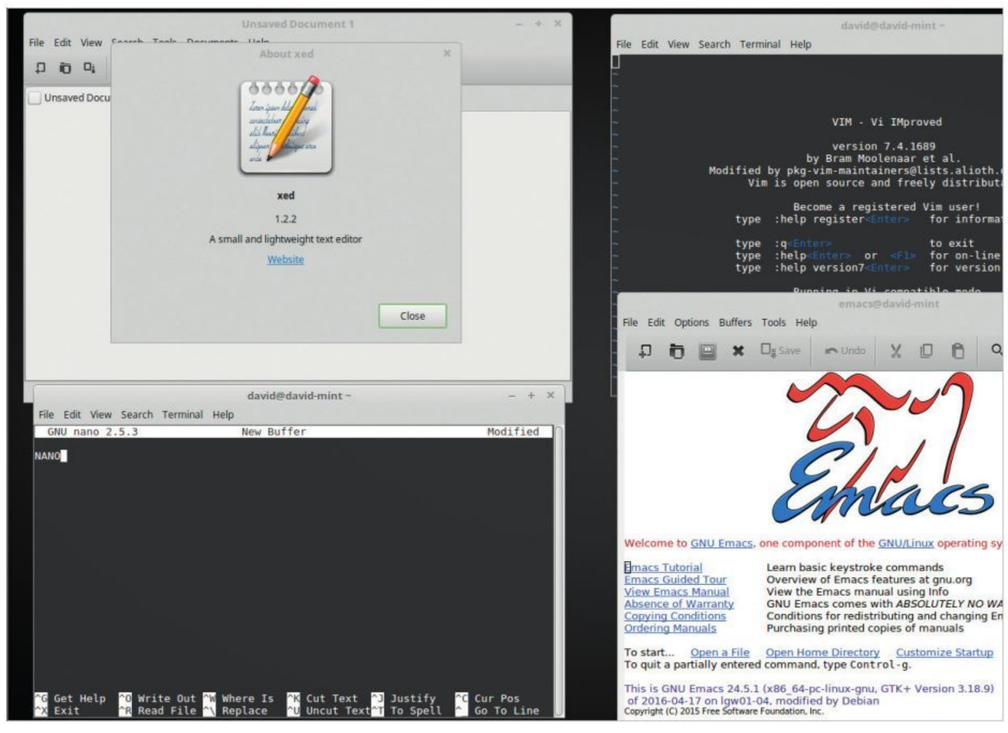


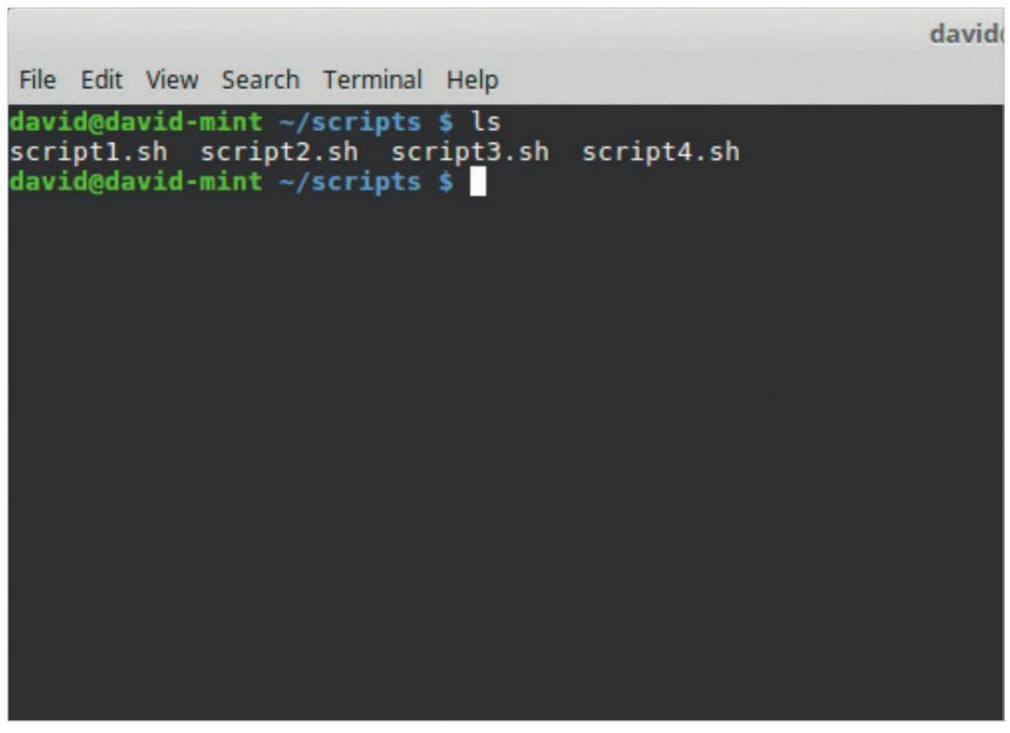
STEP 2 There are several text editors we can use to create a Bash script: Xed, Vi, Nano, Vim, GNU Emacs and so on. In the end it all comes down to personal preference. Our use of Xed is purely due to making it easier to read the script in the screenshots you see below.

STEP 3 To begin with, and before you start to write any scripts, you need to create a folder where you can put all our scripts into. Start with `mkdir scripts`, and enter the folder `cd scripts/`. This will be our working folder and from here you can create sub-folders if you want of each script you create.

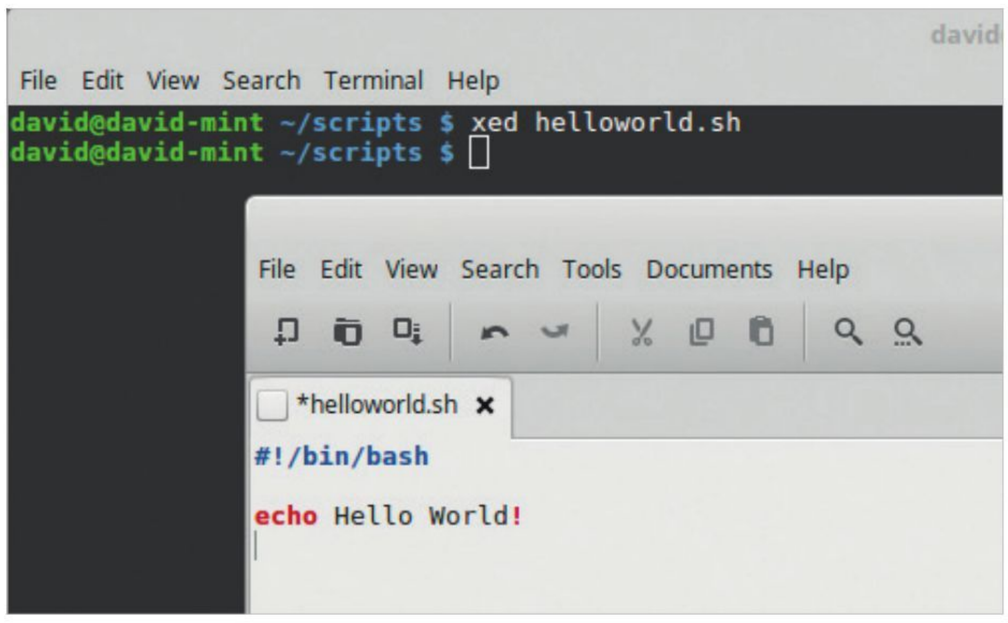


STEP 4 Windows users will be aware that in order for a batch file to work, as in be executed and follow the programming within it, it needs to have a .BAT file extension. Linux is an extension-less operating system but the convention is to give scripts a .sh extension.

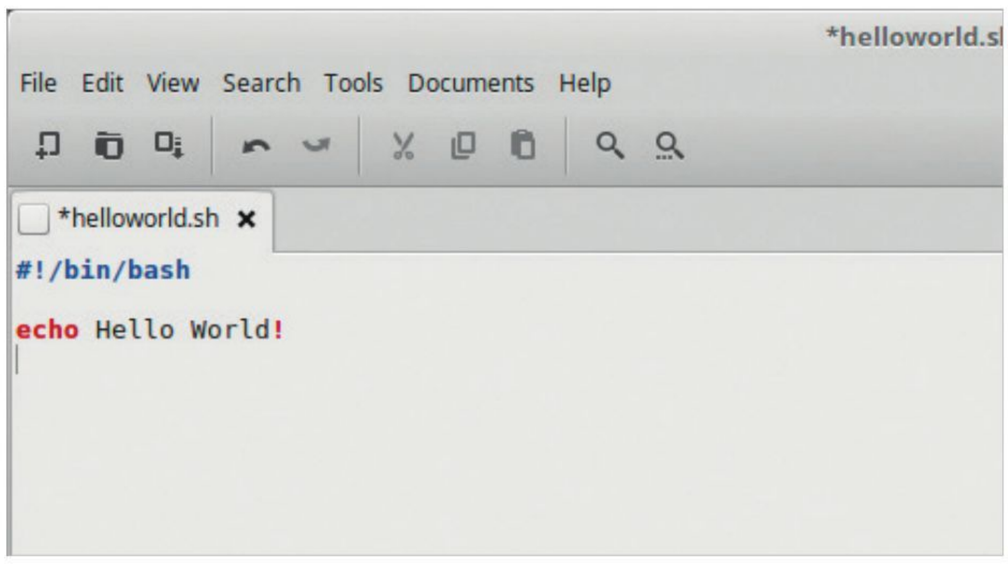




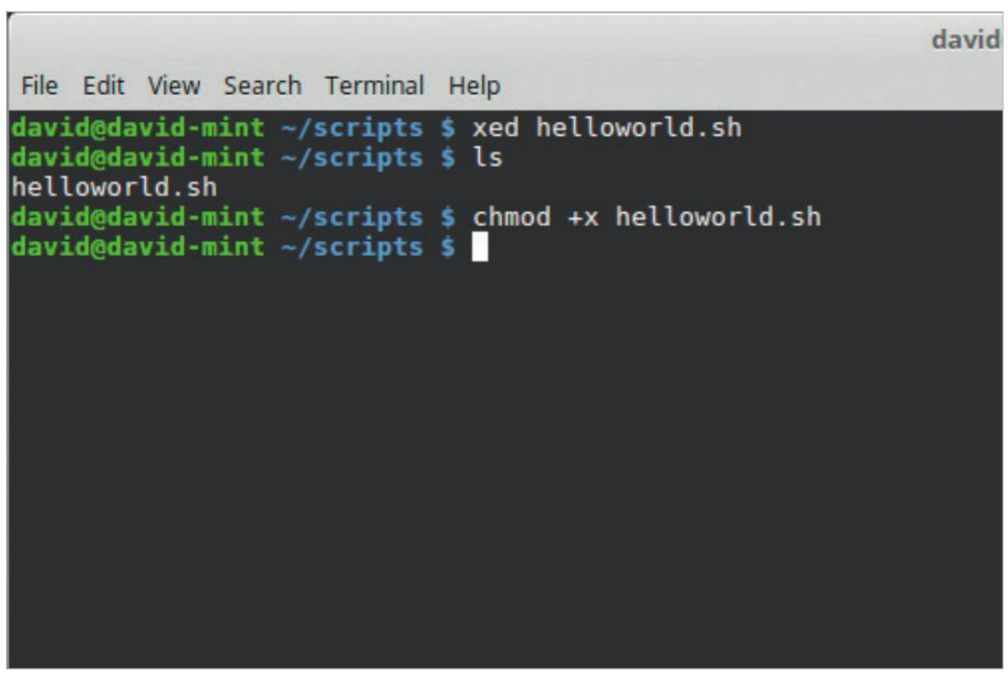
STEP 5 Let's start with a simple script to output something to the Terminal. Enter `xed helloworld.sh`. This will launch Xed and create a file called helloworld.sh. In Xed, enter the following: `#!/bin/bash`, then on a new line: `echo Hello World!`.



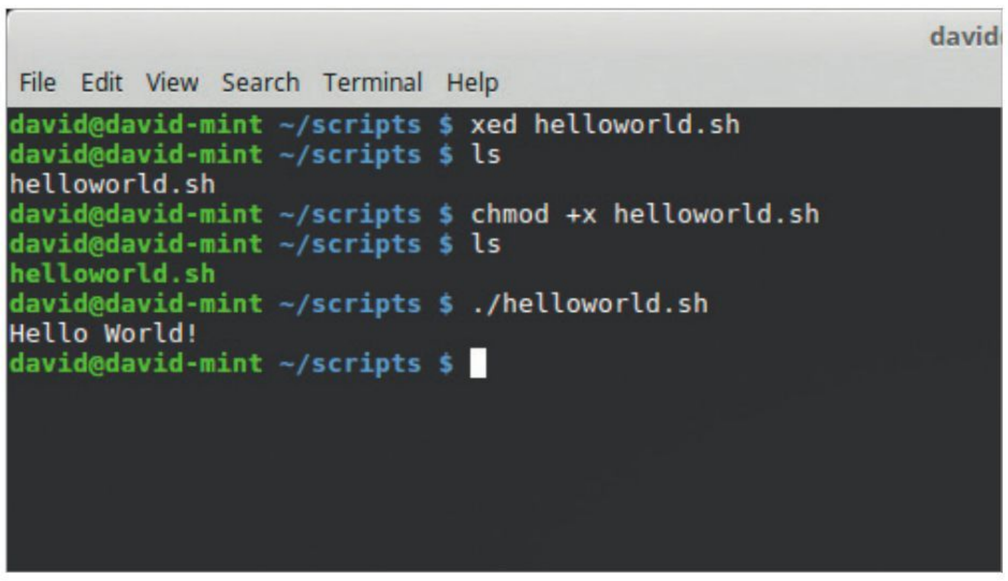
STEP 6 The `#!/bin/bash` line tells the system what Shell you're going to be using, in this case Bash. The hash (#) denotes a comment line, one that is ignored by the system, the exclamation mark (!) means that the comment is bypassed and will force the script to execute the line as a command. This is also known as a Hash-Bang.



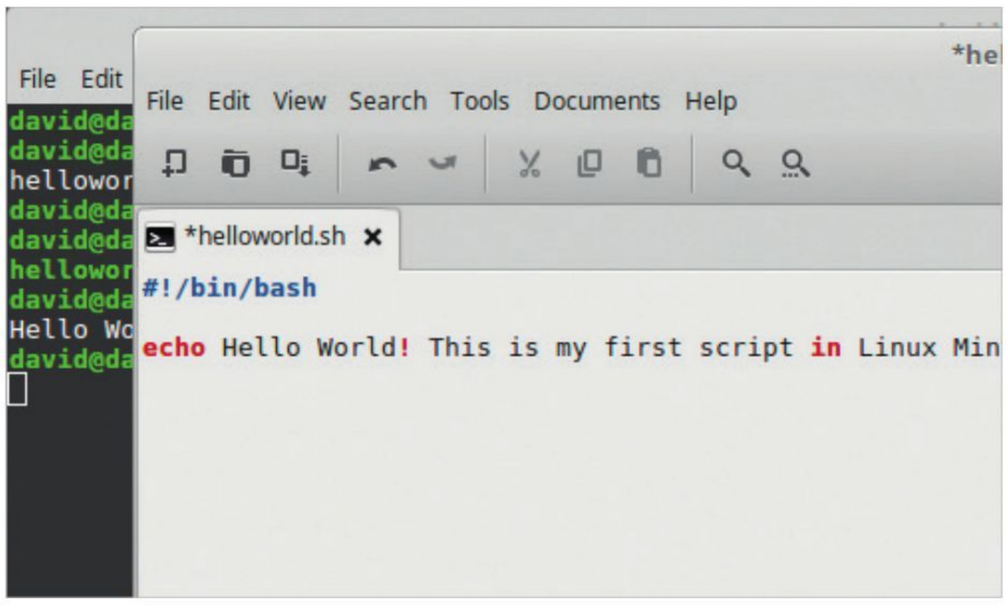
STEP 7 You can save this file, clicking File > Save, and exit back to the Terminal. Entering `ls`, will reveal the script in the folder. To make any script executable, and able to run, you need to modify its permissions. Do this with `chmod +x helloworld.sh`. You need to do this with every script you create.



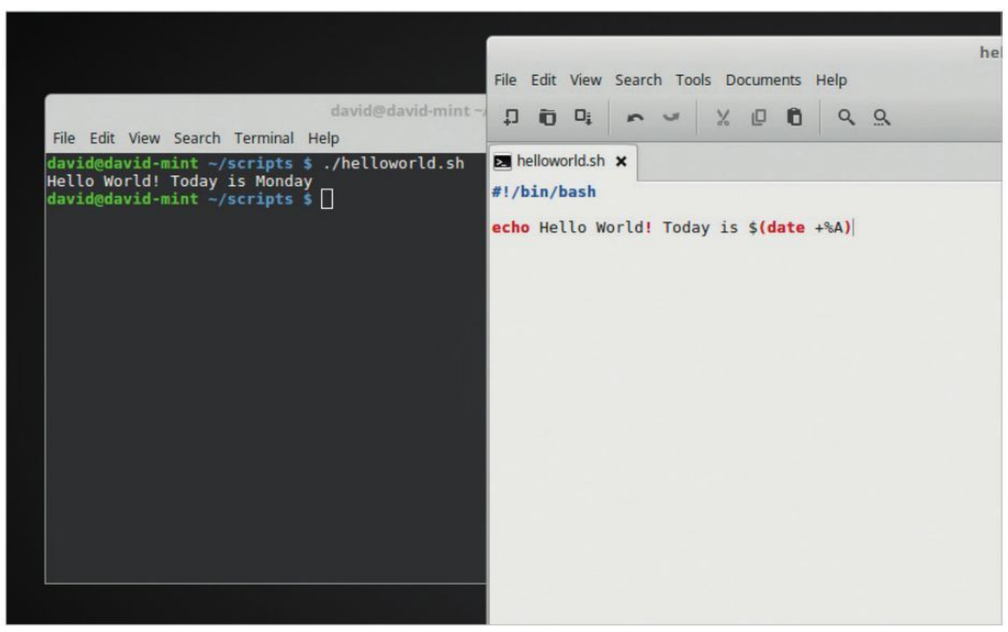
STEP 8 When you enter `ls` again, you can see that the helloworld.sh script has now turned from being white to green, meaning that it's now an executable file. To run the script, in other words make it do the things you've typed into it, enter: `./helloworld.sh`.



STEP 9 Although it's not terribly exciting, the words 'Hello World!' should now be displayed in the Terminal. The echo command is responsible for outputting the words after it in the Terminal, as we move on you can make the echo command output to other sources.



STEP 10 Think of echo as the old BASIC Print command. It displays either text, numbers or any variables that are stored in the system, such as the current system date. Try this example: `echo Hello World! Today is $(date +%A)`. The $(date +%A) is calling the system variable that stores the current day of the week.



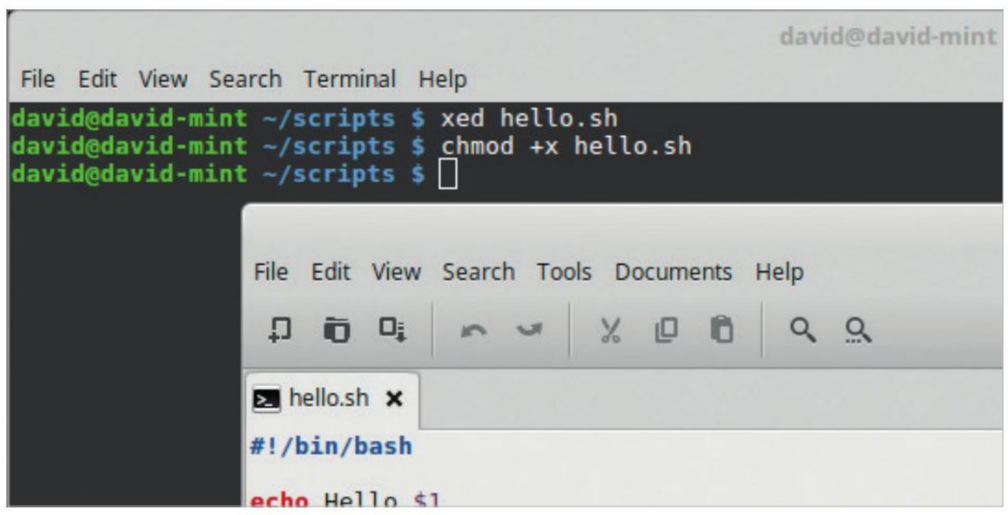
Creating Bash Scripts – Part 2

Previously we looked at creating your first Bash script, Hello World, and adding a system variable. Now you can expand these and see what you can do when you start to play around with creating your own unique variables.

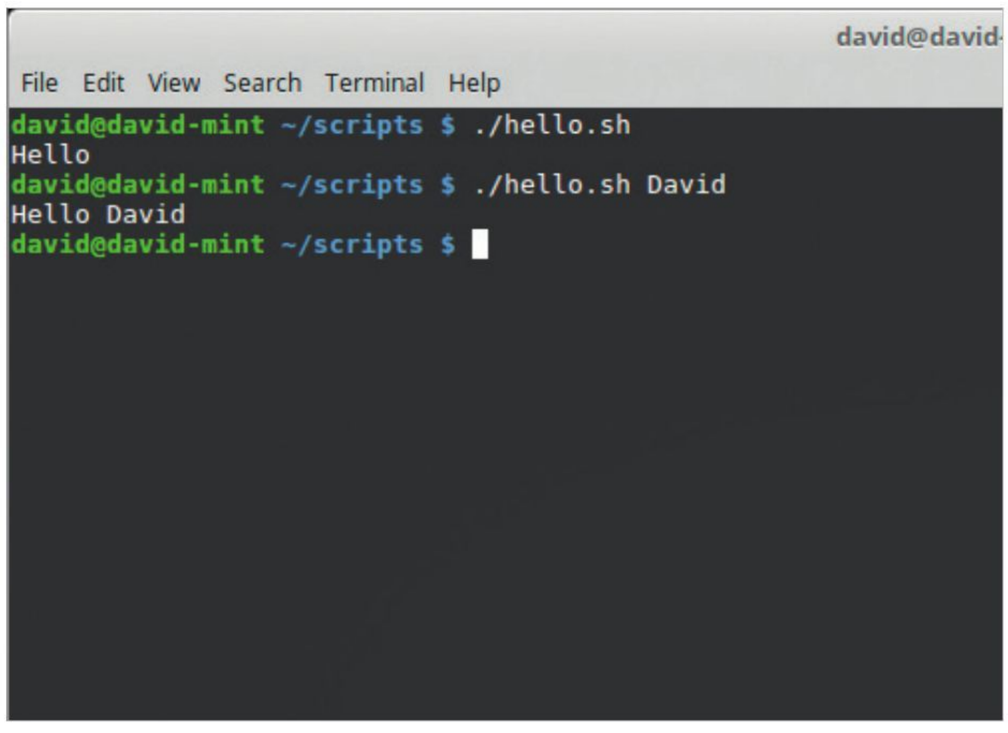
VARIABLES

Just as in every other programming language a Bash script can store and call certain variables from the system, either generic or user created.

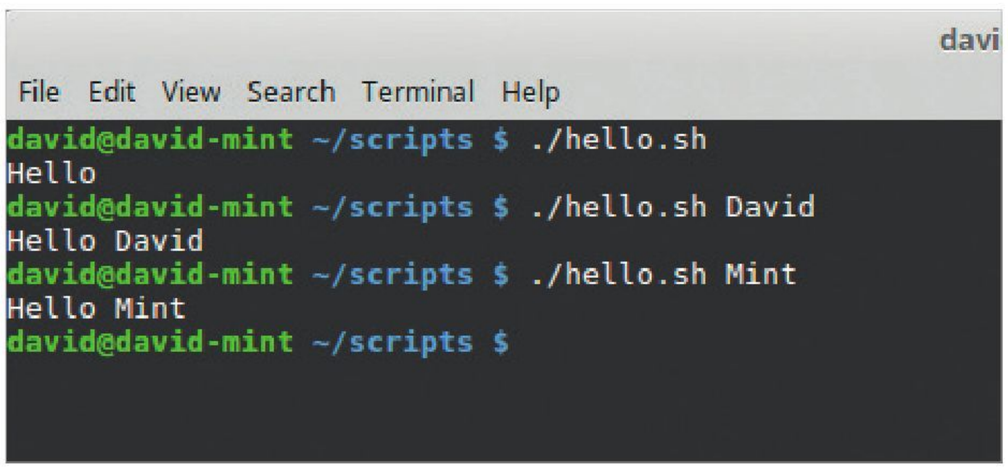
STEP 1 Let's start by creating a new script called hello.sh; `xed hello.sh`. In it enter: `#!/bin/bash`, then, `echo Hello $1`. Save the file and exit Xed. Back in the Terminal make the script executable with: `chmod +x hello.sh`.



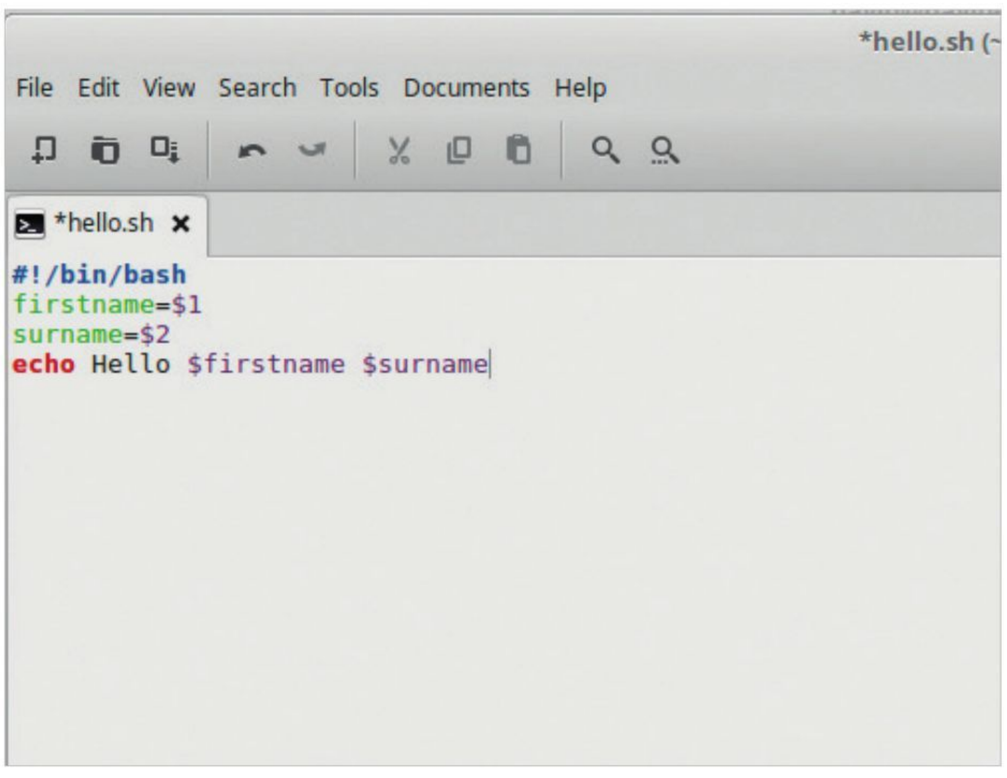
STEP 2 As the script is now executable, run it with `./hello.sh`. Now, as you probably expected a simple 'Hello' is displayed in the Terminal. However, if you then issue the command with a variable, it begins to get interesting. For example, try `./hello.sh David`.



STEP 3 The output now will be Hello David. This is because Bash automatically assigns variables for the user, which are then held and passed to the script. So the variable '$1' now holds 'David'. You can change the variable by entering something different: `./hello.sh Mint`.



STEP 4 You can even rename variables. Modify the hello.sh script with the following: `firstname=$1`, `surname=$2`, `echo Hello $firstname $surname`. Putting each statement on a new line. Save the script and exit back into the Terminal.



STEP 5 When you run the script now you can use two custom variables: `./hello.sh David Hayward`. Naturally change the two variables with your own name; unless you're also called David Hayward. At the moment we're just printing the contents, so let's expand the two-variable use a little.

```
File  Edit  View  Search  Terminal  Help
david@david-mint ~/scripts $ ./hello.sh David Hayward
Hello David Hayward
david@david-mint ~/scripts $ ./hello.sh Linux Mint
Hello Linux Mint
david@david-mint ~/scripts $
```

STEP 6 Create a new script called `addition.sh`, using the same format as the hello.sh script, but changing the variable names. Here we've added `firstnumber` and `secondnumber`, and used the echo command to output some simple arithmetic by placing an integer expression, `echo The sum is $(($firstnumber +$secondnumber))`. Save the script, and make it executable (`chmod +x addition.sh`).

addition.sh

File Edit View Search Tools Documents Help

addition.sh

```
#!/bin/bash
firstnumber=$1
secondnumber=$2
echo The sum is $(($firstnumber+$secondnumber))
```

STEP 7 When you now run the addition.sh script we can enter two numbers: `./addition.sh 1 2`. The result will hopefully be 3, with the Terminal displaying 'The sum is 3'. Try it with a few different numbers and see what happens. See also if you can alter the script and rename it do multiplication, and subtraction.

```
File  Edit  View  Search  Terminal  Help
david@david-mint ~/scripts $ ./addition.sh 1 2
The sum is 3
david@david-mint ~/scripts $ ./addition.sh 34 45
The sum is 79
david@david-mint ~/scripts $ ./addition.sh 65756 1456
The sum is 67212
david@david-mint ~/scripts $ ./multiplication.sh 2 8
The sum is 16
david@david-mint ~/scripts $
```

STEP 8 Let's expand things further. Create a new script called `greetings.sh`. Enter the scripting as below in the screenshot, save it and make it executable with the `chmod` command. You can see that there are a few new additions to the script now.

greetings.sh

File Edit View Search Tools Documents Help

greetings.sh

```
#!/bin/bash

echo -n "Hello, what is your name? "
read firstname
echo -n "Thank you, and what is your surname? "
read surname
clear
echo Hello $firstname $surname, how are you today?
```

STEP 9 We've added a `–n` to the echo command here which will leave the cursor on the same line as the question, instead of a new line. The `read` command stores the users' input as the variables firstname and surname, to then read back later in the last `echo` line. And the `clear` command clears the screen.

```
File  Edit  View  Search  Terminal  Help
Hello David Hayward, how are you today?
david@david-mint ~/scripts $
```

STEP 10 As a final addition, let's include the date variable we used in the last section. Amend the last line of the script to read: `echo Hello $firstname $surname, how are you on this fine $(date +%A)?`. The output should display the current day of the week, calling it from a system variable.

greetings.sh

File Edit View Search Tools Documents Help

greetings.sh

```
#!/bin/bash

echo -n "Hello, what is your name? "
read firstname
echo -n "Thank you, and what is your surname? "
read surname
clear
echo Hello $firstname $surname, how are you on this fine $(dat
```

Creating Bash Scripts – Part 3

In the previous pages we looked at some very basic Bash scripting, which involved outputting text to the screen, getting a user's input, storing it and outputting that to the screen, as well as including a system variable (which was the current day) using the Date command.

IF, THEN, ELSE

With most programming structures there will come a time where you need to loop through the commands you've entered to create better functionality, and ultimately a better program.

STEP 1 Let's look at the If, Then and Else statements now, which when executed correctly, compare a set of instructions and simply work out that IF something is present, THEN do something, ELSE do something different. Create a new script called `greeting2.sh` and enter the text in the screenshot below into it.

*greetings.sh

File Edit View Search Tools Documents Help

*greetings.sh

```
#!/bin/bash

echo -n "Hello, what is your name? "
read firstname
echo -n "Thank you, and what is your surname? "
read surname
clear
if [ "$firstname" == "David" ]
then echo "Awesome name, " $firstname
else echo Hello $firstname $surname, how are you on this fine
fi
```

STEP 2 Greeting2.sh is a copy of greeting.sh but with a slight difference. Here we've added a loop starting at the `if` statement. This means, IF the variable entered is equal to David the next line, THEN, is the reaction to what happens, in this case it will output to the screen 'Awesome name,' followed by the variable (which is David).

david@david

File Edit View Search Terminal Help

```
Awesome name,  David
david@david-mint ~/scripts $
```

STEP 3 The next line, ELSE, is what happens if the variable doesn't equal 'David'. In this case it simply outputs to the screen the now familiar 'Hello…'. The last line, the FI statement, is the command that will end the loop. If you have an IF command without a Fi command, then you get an error.

david@david-

File Edit View Search Terminal Help

```
Hello Pink Floyd, how are you on this fine Wednesday?
david@david-mint ~/scripts $
```

STEP 4 You can obviously play around with the script a little, changing the name variable that triggers a response; or maybe even issuing a response where the first name and surname variables match a specific variable.

greetings2.sh

File Edit View Search Tools Documents Help

greetings2.sh

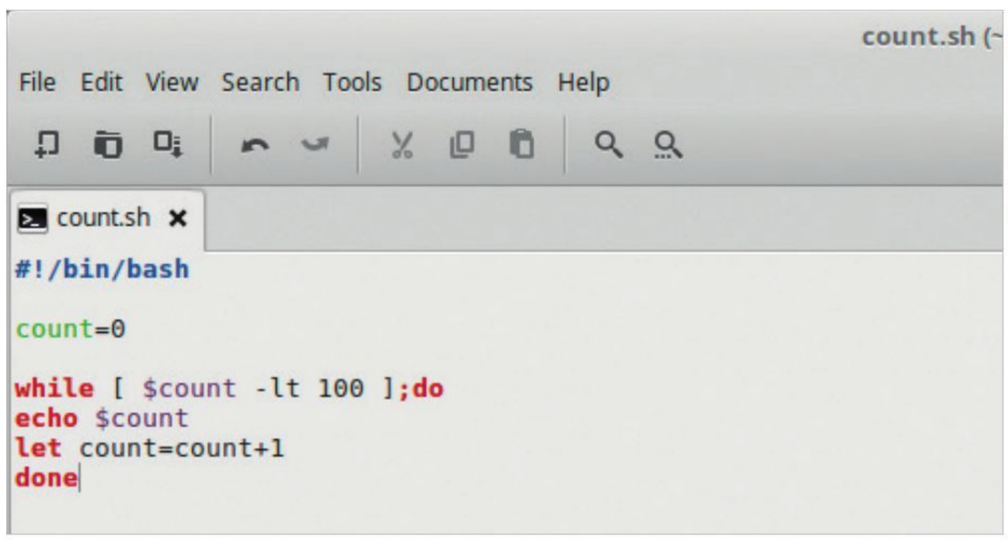
```
#!/bin/bash

echo -n "Hello, what is your name? "
read firstname
echo -n "Thank you, and what is your surname? "
read surname
clear
if [ "$firstname" == "David" ] && [ "$surname" == "Hayward" ]
then echo "Awesome name, " $firstname $surname
else echo Hello $firstname $surname, how are you on this fine
fi
```

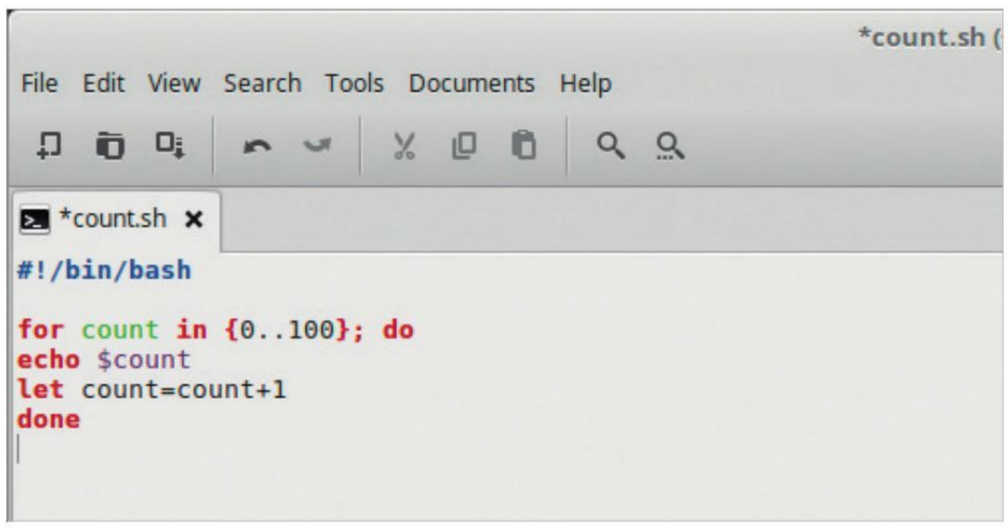
MORE LOOPING

You can loop over data using the FOR, WHILE and UNTIL statements. These can be handy if you're batch naming, copying or running a script where a counter is needed.

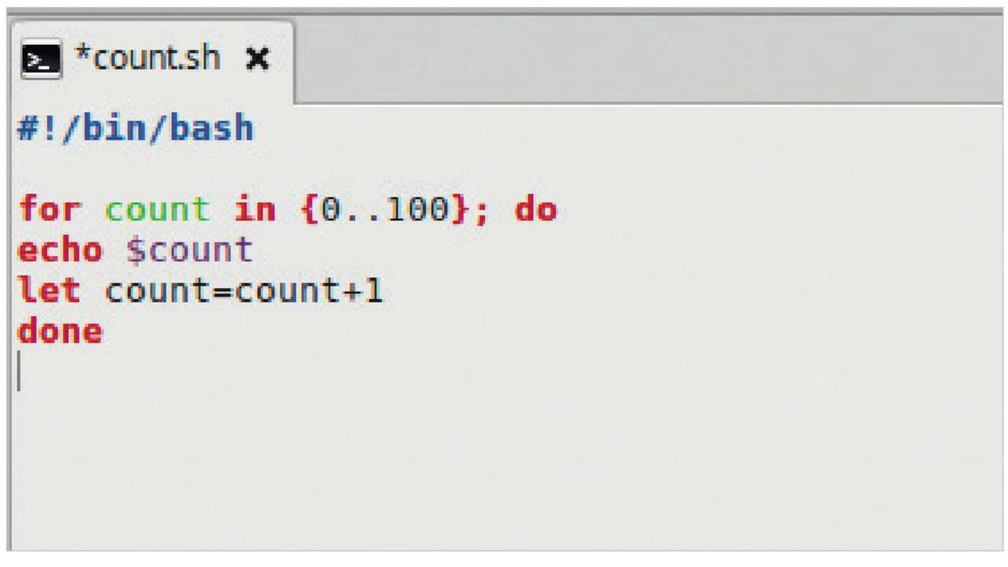
STEP 1 Create a new script called `count.sh`. Enter the text in the screenshot below, save it and make it executable. This creates the variable 'count' which at the beginning of the script equals zero. Then start the WHILE loop, which WHILE count is less than (the LT part) 100 will print the current value of count in the echo command.



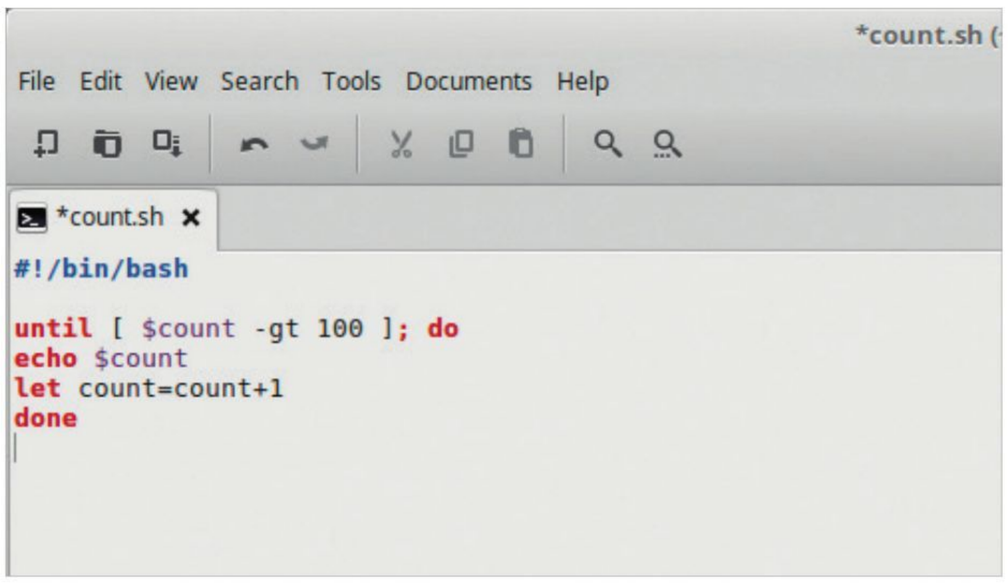
STEP 2 Executing the count.sh script will result in the numbers 0 to 99 listing down the Terminal screen; when it reaches 100 the script will end. Modifying the script with the FOR statement, makes it work in much the same way. To use it in our script, enter the text from the screenshot into the count.sh script.



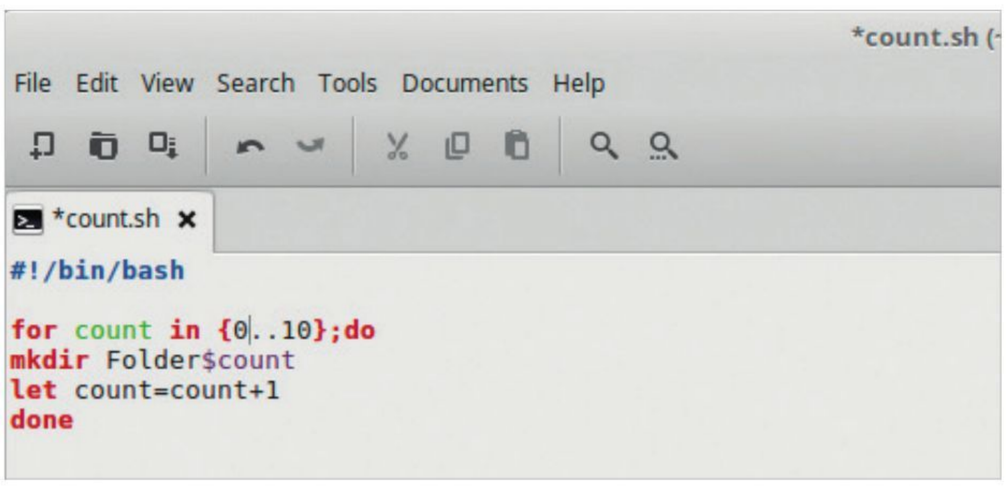
STEP 3 The addition we have here is: `for count in {0..100}; do`. Which means: FOR the variable 'count' IN the numbers from zero to one hundred, then start the loop. The rest of the script is the same. Run this script, and the same output should appear in the Terminal.



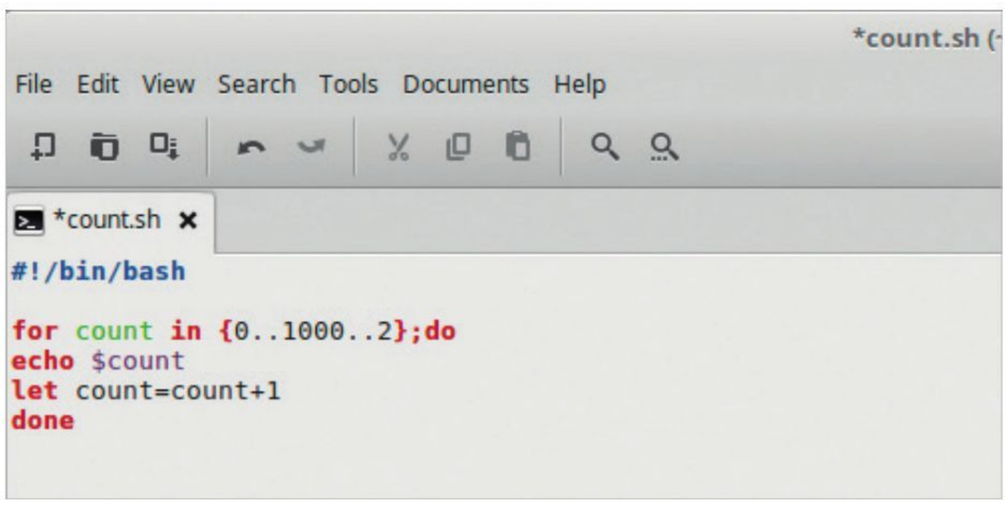
STEP 4 The UNTIL loop works much the same way as the WHILE loop only, more often than not, in reverse. So our counting to a hundred, using `UNTIL, would be: until [ $count -gt 100 ]; do`. The difference being, UNTIL count is greater than (the gt part) one hundred, keep on looping.



STEP 5 You're not limited to numbers zero to one hundred. You can, within the loop, have whatever set of commands you like and execute them as many times as you want the loop to run for. Renaming a million files, creating fifty folders etc. For example, this script will create ten folders named folder1 through to folder10 using the FOR loop.



STEP 6 Using the FOR statement once more, we can execute the counting sequence by manipulating the {0..100} part. This section of the code actually means {START..END..INCREMENT}, if there's no increment then it's just a single digit up to the END. For example, we could get the loops to count up to 1000 in two's with: `for count in {0..1000..2}; do`.



Creating Bash Scripts – Part 4

You've encountered user interaction with your scripts, asking what the user's name is and so on. You've also looked at creating loops within the script to either count or simply do something several times. Let's combine and expand some more.

CHOICES AND LOOPS

Let's bring in another command, CHOICE, along with some nested IF and ELSE statements. Start by creating a new script called `mychoice.sh`.

STEP 1

The mychoice.sh script is beginning to look a lot more complex. What we have here is a list of four choices, with three possible options. The options: Mint, Is, and Awesome will be displayed if the user presses the correct option key. If not, then the menu will reappear, the fourth choice.

mychoic

File Edit View Search Tools Documents Help

mychoice.sh

```
#!/bin/bash

choice=4

echo "1. Mint"
echo "2. Is"
echo "3. Awesome"
echo -n "Please choose an option (1, 2 or 3) "

while [ $choice -eq 4 ]; do

read choice

if [ $choice -eq 1 ] ; then
    echo "You have chosen: Mint"
else
    if [ $choice -eq 2 ] ; then
        echo "You have chosen: Is"
    else
        if [ $choice -eq 3 ] ; then
            echo "You have chosen: Awesome"
        else
            echo "Please make a choice between 1 to 3"
            echo "1. Mint"
            echo "2. Is"
            echo "3. Awesome"
            echo -n "Please choose an option (1, 2 or 3) "
            choice=4
        fi
    fi
fi
done
```

STEP 2

If you follow the script through you soon get the hang of what's going on, based on what we've already covered. WHILE, IF, and ELSE, with the FI closing loop statement will run through the options and bring you back to the start if you pick the wrong option.

david@david

File Edit View Search Terminal Help

```
david@david-mint ~/scripts $ ./mychoice.sh
1. Mint
2. Is
3. Awesome
Please choose an option (1, 2 or 3) 1
You have chosen: Mint
david@david-mint ~/scripts $ ./mychoice.sh
1. Mint
2. Is
3. Awesome
Please choose an option (1, 2 or 3) 2
You have chosen: Is
david@david-mint ~/scripts $ ./mychoice.sh
1. Mint
2. Is
3. Awesome
```

STEP 3

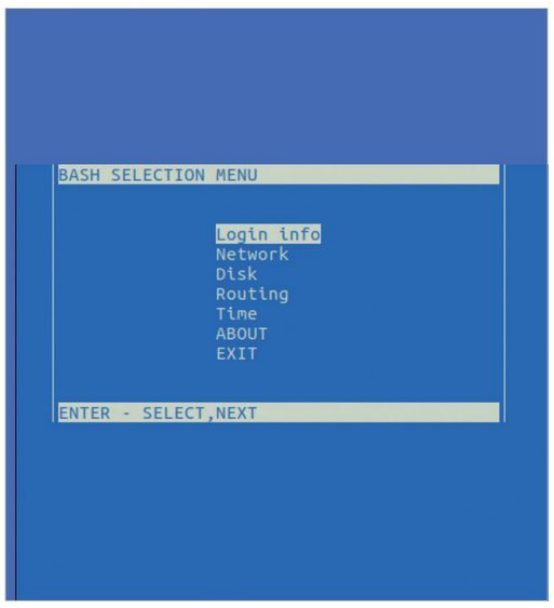
You can, of course, increase the number of choices but you need to make sure that you match the number of choices to the number of IF statements. The script can quickly become a very busy screen to look at. This lengthy script is another way of displaying a menu, this time with a fancy colour scheme too.

```
#/bin/bash

     E='echo -e';e='echo -en';trap "R;exit" 2
   ESC=$( $e "\e")
  TPUT(){ $e "\e[${1};${2}H";}
 CLEAR(){ $e "\ec";}
 CIVIS(){ $e "\e[?25l";}
  DRAW(){ $e "\e%@\e(0";}
 WRITE(){ $e "\e(B";}
  MARK(){ $e "\e[7m";}
UNMARK(){ $e "\e[27m";}
     R(){ CLEAR ;stty sane;$e "\ec\e[37;44m\e[J";};
  HEAD(){ DRAW
          for each in $(seq 1 13);do
          $E "   x                                    x"
          done
          WRITE;MARK;TPUT 1 5
          $E "BASH SELECTION MENU                    ";UNMARK;}
  FOOT(){ MARK;TPUT 13 5
          printf "ENTER - SELECT,NEXT                 ";UNMARK;}
 ARROW(){ read -s -n3 key 2>/dev/null >&2
          if [[ $key = $ESC[A ]];then echo up;fi
          if [[ $key = $ESC[B ]];then echo dn;fi;}
    M0(){ TPUT  4 20; $e "Login info";}
    M1(){ TPUT  5 20; $e "Network";}
    M2(){ TPUT  6 20; $e "Disk";}
    M3(){ TPUT  7 20; $e "Routing";}
    M4(){ TPUT  8 20; $e "Time";}
    M5(){ TPUT  9 20; $e "ABOUT  ";}
    M6(){ TPUT 10 20; $e "EXIT   ";}
     LM=6
  MENU(){ for each in $(seq 0 $LM);do M${each};done;}
   POS(){ if [[ $cur == up ]];then ((i--));fi
          if [[ $cur == dn ]];then ((i++));fi
          if [[ $i -lt 0   ]];then i=$LM;fi
          if [[ $i -gt $LM ]];then i=0;fi;}
REFRESH(){ after=$((i+1)); before=$((i-1))
          if [[ $before -lt 0  ]];then before=$LM;fi
          if [[ $after -gt $LM ]];then after=0;fi
          if [[ $j -lt $i      ]];then UNMARK;M$before;else UNMARK;M$after;fi
          if [[ $after -eq 0 ]] || [ $before -eq $LM ];then
          UNMARK; M$before; M$after;fi;j=$i;UNMARK;M$before;M$after;}
  INIT(){ R;HEAD;FOOT;MENU;}
    SC(){ REFRESH;MARK;$S;$b;cur=`ARROW`;}
    ES(){ MARK;$e "ENTER = main menu ";$b;read;INIT;};INIT
 while [[ "$O" != " " ]]; do case $i in
       0) S=M0;SC;if [[ $cur == "" ]];then R;$e "\n$(w        )\n";ES;fi;;
       1) S=M1;SC;if [[ $cur == "" ]];then R;$e "\n$(ifconfig )\n";ES;fi;;
       2) S=M2;SC;if [[ $cur == "" ]];then R;$e "\n$(df -h    )\n";ES;fi;;
       3) S=M3;SC;if [[ $cur == "" ]];then R;$e "\n$(route -n )\n";ES;fi;;
```

STEP 4

You can use the arrow keys and Enter in the menu setup in the script. Each choice is an external command that feeds back various information. Play around with the commands and choices, and see what you can come up with. It's a bit beyond what we've looked at but it gives a good idea of what can be achieved.



CREATING A BACKUP TASK SCRIPT

One of the most well used examples of Bash scripting is the creation of a backup routine, one that automates the task as well as adding some customisations along the way.

STEP 1

A very basic backup script would look something along the lines of: `#!/bin/bash`, then, `tar cvfz ~/backups/my-backup.tgz ~/Documents/`. This will create a compressed file backup of the ~/Documents folder, with everything in it, and put it in a folder called /backups with the name my-backup.tgz.

backup1.sh (~/scripts)

File Edit View Search Tools Documents Help

backup1.sh

```
#!/bin/bash

tar cvfz ~/backups/my-backup.tgz ~/Documents/
```

STEP 2

While perfectly fine, we can make the simple script a lot more interactive. Let's begin with defining some variables. Enter the text in the screenshot into a new `backup.sh` script. Notice that we've misspelt 'source' as 'sauce', this is because there's already a built-in command called 'source' hence the different spelling on our part.

backup1.sh

```
#!/bin/bash

clear
# Time stamp
day=$(date +%A)
month=$(date +%B)
year=$(date +%Y)

# Folders
dest=~/backups
sauce=~/Documents
```

STEP 3

The previous script entries allowed you to create a Time Stamp, so you know when the backup was taken. You also created a 'dest' variable, which is the folder where the backup file will be created (~/backups). You can now add a section of code to first check if the ~/backups folder exits, if not, then it creates one.

backup1.sh (~/scripts)

File Edit View Search Tools Documents Help

backup1.sh

```
#!/bin/bash

clear
# Time stamp
day=$(date +%A)
month=$(date +%B)
year=$(date +%Y)

# Folders
dest=~/backups
sauce=~/Documents

if [ -d $dest ]; then
echo "Backup folder exists"
else
echo "Backup folder does not exist! I'm now creating it..."; (mkdir -p $dest)
echo
fi
read -p "Press any key to continue.." -n1 -s
```

STEP 4

Once the ~/backups folder is created, we can now create a new subfolder within it based on the Time Stamp variables you set up at the beginning. Add `mkdir -p $dest/"$day $month $year"`. It's in here that you put the backup file relevant to that day/month/year.

backup1.sh

```
#!/bin/bash

clear
# Time stamp
day=$(date +%A)
month=$(date +%B)
year=$(date +%Y)

# Folders
dest=~/backups
sauce=~/Documents

if [ -d $dest ]; then
echo "Backup folder exists"
else
echo "Backup folder does not exist! I'm now creating it..."; (mkdir -p $dest)
echo
fi
read -p "Press any key to continue.." -n1 -s
mkdir -p $dest/"$day $month $year"
```

STEP 5

With everything in place, you can now enter the actual backup routine, based on the Tar command from Step 5. Combined with the variables, you have: `tar cvfz $dest/"$day $month $year"/DocumentsBackup.tgz $sauce`. In the screenshot, we added a handy "Now backing up..." echo command.

backup1.sh

```
#!/bin/bash

clear
# Time stamp
day=$(date +%A)
month=$(date +%B)
year=$(date +%Y)

# Folders
dest=~/backups
sauce=~/Documents

if [ -d $dest ]; then
echo "Backup folder exists"
else
echo "Backup folder does not exist! I'm now creating it..."; (mkdir -p $dest)
echo
fi
read -p "Press any key to continue.." -n1 -s
mkdir -p $dest/"$day $month $year"

clear
echo "Now backing up. Please wait..."
tar cvfz $dest/"$day $month $year"/DocumentsBackup.tgz $sauce
```

STEP 6

Finally, you can add a friendly message: `echo "Backup complete. All done..."`. The completed script isn't too over-complex and it can be easily customised to include any folder within your Home area, as well as the entire Home area itself.

backup1.sh

```
#!/bin/bash

clear
# Time stamp
day=$(date +%A)
month=$(date +%B)
year=$(date +%Y)

# Folders
dest=~/backups
sauce=~/Documents

if [ -d $dest ]; then
echo "Backup folder exists"
else
echo "Backup folder does not exist! I'm now creating it..."; (mkdir -p $dest)
echo
fi
read -p "Press any key to continue.." -n1 -s
mkdir -p $dest/"$day $month $year"

clear
echo "Now backing up. Please wait..."
tar cvfz $dest/"$day $month $year"/DocumentsBackup.tgz $sauce

clear
echo
echo "Backup complete. All done..."
```

Creating Bash Scripts – Part 5

The backup script we looked at previously can be further amended to incorporate choices, user-interaction with regards to where the backup file will be copied to and so on. Automating tasks is one of the main benefits of Bash scripting, a simple script can help you out in many ways.

EASY AUTOMATION AND HANDY SCRIPTS

Entering line after line of commands to retrieve system information, find a file or rename a batch of files? A script is a better answer.

STEP 1 Let's start by creating a script to help display the Mint system information; always a handy thing to have. Create a new script called `sysinfo.sh` and enter the following into Xed, or the text editor of your choice.

sysinfo.sh

```
#!/bin/bash

# -Hostname information:
echo -e "\e[31;43m***** HOSTNAME INFORMATION *****\e[0m"
hostnamectl
echo ""

# -File system disk space usage:
echo -e "\e[31;43m***** FILE SYSTEM DISK SPACE USE *****\e[0m"
df -h
echo ""

# -Free and used memory:
echo -e "\e[31;43m***** FREE AND USED MEMORY *****\e[0m"
free
echo ""

# -System uptime and performance load:
echo -e "\e[31;43m***** SYSTEM UPTIME AND LOAD *****\e[0m"
uptime
echo ""

# -Users currently logged in:
echo -e "\e[31;43m***** CURRENT USERS *****\e[0m"
who
echo ""

# -Top five processes being used by the system:
echo -e "\e[31;43m***** TOP 5 MEMORY CONSUMING PROCESSES *****\e[0m"
```

STEP 2 We've included a couple of extra commands in this script. The first is the `-e` extension for echo, this means it'll enable echo interpretation of additional instances of a new line, as well as other special characters. The proceeding '31;43m' element enables colour for foreground and background.

```
david@david
File  Edit  View  Search  Terminal  Help

david@david-mint ~/scripts $ ./sysinfo.sh
***** HOSTNAME INFORMATION *****
   Static hostname: david-mint
         Icon name: computer-vm
           Chassis: vm
        Machine ID: 5ab3c275b7304ed3b8aeef9ffcc37eb4
           Boot ID: 61ce1baadf934f649cf5ac809abe7e18
    Virtualization: oracle
  Operating System: Linux Mint 18.1
            Kernel: Linux 4.4.0-53-generic
      Architecture: x86-64

***** FILE SYSTEM DISK SPACE USE *****
Filesystem      Size  Used Avail Use% Mounted on
udev            980M     0  980M   0% /dev
```

STEP 3 Each of the sections runs a different Terminal command, outputting the results under the appropriate heading. You can include a lot more, such as the current aliases being used in the system, the current time and date and so on. Plus, you could also pipe all that information into a handy HTML file, ready to be viewed in a browser.

```
david@david-mint ~/scripts
File  Edit  View  Search  Terminal  Help
david@david-mint ~/scripts $ ./sysinfo.sh > sysinfo.html
david@david-mint ~/scripts $
```

STEP 4 Although there are simple Terminal commands to help you look for a particular file or folder, it's often more fun to create a script to help you. Plus, you can use that script for other non-technical users. Create a new script called `look4.sh`, entering the content from the screenshot below.

look4.sh (~/s

File Edit View Search Tools Documents Help

look4.sh

```
#!/bin/bash

target=~/

read name

output=$( find "$target" -iname "*.$name" 2> /dev/null )

if [[ -n "$output" ]]; then
    echo "$output"
else
    echo "No match found"
fi
```

STEP 5 When executed the script waits for input from the user, in this case the file extension, such as jpg, mp4 and so on. It's not very friendly though. Let's make it a little friendlier. Add an echo, with: `echo -n "Please enter the extension of the file you're looking for: "`, just before the read command.

*look4.sh

```
#!/bin/bash

target=~/

echo -n "Please enter the extensions of the file you're looking for: "
read name

output=$( find "$target" -iname "*.$name" 2> /dev/null )

if [[ -n "$output" ]]; then
    echo "$output"
else
    echo "No match found"
fi
```

STEP 6 Here's an interesting, fun kind of script using the app espeak. Install espeak with `sudo apt-get install espeak`, then enter the text below into a new script called `speak.sh`. As you can see it's a rehash of the first greeting script we ran. Only this time, it uses the variables in the espeak output.

speak.sh

```
#!/bin/bash

echo -n "Hello, what is your first name? "
read firstname
echo -n "Thank you, and what is your surname? "
read surname
clear
espeak "Hello $firstname $surname, how are you on this fine $(date +%A)?"
```

STEP 7 We briefly looked at putting some colours in the output for our scripts. Whilst it's too long to dig a little deeper into the colour options, here's a script that outputs what's available. Create a new script called `colours.sh` and enter the text (see below) into it.

colours.sh

```
#!/bin/bash

clear
echo -e "Normal \e[1mBold"
echo -e "Normal \e[2mDim"
echo -e "Normal \e[4mUnderlined"
echo -e "Normal \e[5mBlink"
echo -e "Normal \e[7minverted"
echo -e "Normal \e[8mHidden"
echo
echo -e "\e[0mNormal Text"
echo

echo -e "Default \e[39mDefault"
echo -e "Default \e[30mBlack"
echo -e "Default \e[31mRed"
echo -e "Default \e[32mGreen"
echo -e "Default \e[33mYellow"
echo -e "Default \e[34mBlue"
echo -e "Default \e[35mMagenta"
echo -e "Default \e[36mCyan"
echo -e "Default \e[37mLight gray"
echo -e "Default \e[90mDark gray"
echo -e "Default \e[91mLight red"
echo -e "Default \e[92mLight green"
echo -e "Default \e[93mLight yellow"
echo -e "Default \e[94mLight blue"
echo -e "Default \e[95mLight magenta"
echo -e "Default \e[96mLight cyan"
echo -e "Default \e[97mWhite"
echo
echo -e "Default \e[49mDefault"
echo -e "Default \e[40mBlack"
echo -e "Default \e[41mRed"
echo -e "Default \e[42mGreen"
echo -e "Default \e[43mYellow"
echo -e "Default \e[44mBlue"
echo -e "Default \e[45mMagenta"
echo -e "Default \e[46mCyan"
echo -e "Default \e[47mLight gray"
echo -e "Default \e[100mDark gray"
echo -e "Default \e[101mLight red"
echo -e "Default \e[102mLight green"
echo -e "Default \e[103mLight yellow"
echo -e "Default \e[104mLight blue"
echo -e "Default \e[105mLight magenta"
echo -e "Default \e[106mLight cyan"
```

STEP 8 The output from colours.sh can, of course, be mixed together, bringing different effects depending on what you want to the output to say. For example, white text in a red background flashing (or blinking). Sadly the blinking effect doesn't work on all Terminals, so you may need to change to a different Terminal.

```
Normal Bold
Normal Dim
Normal Underlined
Normal Blink
Normal inverted
Normal

Normal Text

Default Default
Default Black
Default Red
Default Green
Default Yellow
Default Blue
Default Magenta
Default Cyan
Default Light gray
Default Dark gray
Default Light red
Default Light green
Default Light yellow
Default Light blue
Default Light magenta
Default Light cyan
Default White

Default Default
Default Black
Default Red
Default Green
Default Yellow
Default Blue
Default Magenta
Default Cyan
Default Light gray
Default Dark gray
Default Light red
Default Light green
Default Light yellow
Default Light blue
Default Light magenta
Default Light cyan
Default
```

STEP 9 Whilst we're on making fancy scripts, how about using Zenity to output a graphical interface? Enter what you see below into a new script, `mmenu.sh`. Make it executable and then run it. You should have a couple of dialogue boxes appear, followed by a final message.

mmenu.sh

```
#!/bin/bash

firstname=$(zenity --entry --title="Your Name" --text="What is your first name?")

surname=$(zenity --entry --title="Your Name" --text="What is your first surname?")

zenity --info --title="Hello!" --text="Welcome to Linux Mint \n\n Have fun, $firstname $surname."
```

STEP 10 While gaming in a Bash script isn't something that's often touched upon, it is entirely possible, albeit, a little basic. If you fancy playing a game, enter `wget http://bruxy.regnet.cz/linux/housenka/housenka.sh`, make the script executable and run it. It's in Polish, written by Martin Bruchanov but we're sure you can modify it. Hint: the title screen is in Base64.