Continuing with Bash Scripting

Updated Thursday, March 9, 2023, by Mihalis Tsoukalos

In the previous guide of this series, Getting Started with Bash Scripting, you learr variables, getting user input, using environment variables, and more. In this guid already learned and put together more complex Bash scripts for common operat administrators like creating interactive Bash scripts with menu options, scripts th data, and scripts that work with files and directories. Each section will provide a l commands with a few examples that you can run to better understand its functic

In this guide, you will learn about:

- Standard Streams
- Creating menus with the select statement
- Using the printf command to format the output of your scripts
- Using file and directory test operators to control the flow of your scripts
- Reading files and searching directories in your scripts
- Bash exit codes

Before You Begin

1. All example scripts in this guide are run from the `bin` directory in the user's h `/bin/`. If you do not have a `bin` directory in your home directory, create one

```
cd ~ && mkdir bin && cd bin
```

2. Verify that the `bin` directory is in your system PATH (i.e. `/home/username/bi`

```
echo $PATH
```

3. If it is not, add the new `bin` directory to your system's PATH:

```
PATH=$PATH:$HOME/bin/
```

> **Note**
>
> Ensure all scripts throughout this guide are executable. To add execute pe

following command:

```
chmod +x my-script.sh
```

Standard Streams

A *standard stream* is a communication mechanism used between a computer pr operating system contains three types of standard streams, *standard input* (stdin *standard error* (stderr). These three streams are represented by three files, `/dev/` `/dev/stderr` . Since these three files are always open, you can *redirect* their stre when you use the output from one source, a file, program, script, or command ar the context of Bash scripting, you can access stdin, stdout, and stderr using file d

Reading from Standard Input

Bash scripts very often make use of standard input. The example script `input.sh` is not available in the expected location, it tries to read standard input (`/dev/std`

File: input.sh

```
#!/bin/bash

file=$1

if [[ "$file" == "" || (! -f "$file") ]]
then
    echo Using standard input!
    file="/dev/stdin"
fi

while read -r line
do
    echo "$line"
done < "${file}"
```

The script reads the first value passed as a command line argument, represer script will read and output each line of text.

If a no command line argument is passed or if the file does not exist, standar This will prompt you to enter text and will output to the terminal screen what of your stdin input type **CTRL+D**

1. Using your preferred text editor create an example file for the `input.sh` scrip

```
echo -e 'Ultimately, literature is nothing but carpentry
```

2. Run the script and pass `marquez.txt` as a command line argument:

```
./input.sh marquez.txt
```

```
Ultimately, literature is nothing but carpentry.
With both you are working with reality, a material just a
```

3. Run the script without a command line argument:

```
./input.sh
```

Enter some text after the prompt followed by *enter* and you will see it echoec
CTRL+D to end the script.

Create Menus with the Select Statement

You can use the `select` statement to create menu systems in your bash scripts t
combine `select` with the `case` statement you can create more sophisticated m
three examples that use `select` to create menus. If you are not familiar with the
Getting Started with Bash Shell Scripting guide.

The general format for the `select` statement is the following:

File: bash

```
select WORD [in LIST];
do COMMANDS;
done
```

Create a Basic Menu

The `simple-menu.sh` script expands on the skeleton example to create a basic m
favorite color, print out the value of any valid menu selection, and then break ou

File: simple-menu.sh

```
#!/bin/bash

echo "Enter the number corresponding to your favorite c

select COLOR in blue yellow red green
do
    echo "Your selection is: $COLOR"
```

```
    break
done
```

1. Copy and paste the contents of `simple-menu.sh` into a new file and save it.
2. Run the script:

```
./simple-menu.sh
```

Your output will resemble the following, but may vary depending on the men

```
Enter the number corresponding to your favorite color:
1) blue
2) yellow
3) red
4) green
#? 2
Your selection is: yellow
```

Create a Menu Using the Case Statement

The second example script, `computing-terms.sh`, improves on the previous exa statement and by explicitly providing a way for the user to exit the script. By addi script can execute separate tasks based on what the user selects. The reserved B with `select` statements to provide a custom prompt to the user. This script will cloud related terms and return its corresponding definition when selected.

File: computing-terms.sh

```
#!/bin/bash

echo "This script shows you how to create select menus
echo "Enter a number corresponding to the term whose d
PS3="My selection is:"

select TERM in cloud-computing virtual-machine object-
do
    case $TERM in
        cloud-computing)
            echo "Cloud Computing: A combined system o
            ;;
        virtual-machine)
            echo "Virtual Machine: The emulating of a
            ;;
        object-storage)
```

```
            echo "Object Storage: stores data, called
            ;;
        exit)
            echo "You are now exiting this script."
                break
            ;;
        *)
            echo "Please make a selection from the pro
    esac
done
```

1. Copy and paste the contents of `computing-terms.sh` into a new file and save
2. Run the script:

```
./computing-terms.sh
```

Your output will resemble the following, but may vary depending on the men

```
This script shows you how to create select menus in your
Enter a number corresponding to the term whose definition
1) cloud-computing
2) virtual-machine
3) object-storage
4) exit
My selection is:3
Object Storage: stores data, called objects, in container
My selection is:4
You are now exiting this script.
```

Create a Menu that Includes a Submenu

The third example, `submenu.sh`, uses all the previously covered concepts and en with a new series of options for the user to select. The script will read all files in t display them to the user as selectable options. Once the user selects a file, a subı to select an action to perform on the previously selected file. The submenu allow file's contents, or to simply exit the script.

File: submenu.sh

```
#!/bin/bash

echo "Use this script to manipulate files in your curr
echo "----------------------------------------------
echo "Here is a list of all your files. Select a file
```

```
6   echo "Available file actions:"
7
8   select FILE in * exit;
9   do
10      case $FILE in
11      exit)
12          echo "Exiting script ..."
13          break
14          ;;
15      *)
16          select ACTION in delete view exit;
17          do
18              case $ACTION in
19              delete)
20                  echo "You've chose to delete your file
21                  rm -i "$FILE"
22                  echo "File ""$FILE" "has been deleted"
23                  echo "Exiting script ..."
24                  break
25                  ;;
26              view)
27                  echo "Your selected file's contents wi
28                  cat "$FILE"
29                  echo "------------------------"
30                  echo "Exiting script ..."
31                  break
32                  ;;
33              exit)
34                  echo "Exiting script ..."
35                  break
36                  ;;
37              esac
38          done
39          break
40          ;;
41      esac
42  done
```

1. Copy and paste the contents of `submenu.sh` into a new file and save it.
2. Run the script:

```
./submenu.sh
```

> **Note**
>
> Ensure that the directory you are executing your script from contains at le
> the full demo of the `submenu.sh` script.

Your output will resemble the following, but may vary depending on the men

```
Use this script to manipulate files in your current worki
--------------------------------------------------------
Here is a list of all your files. Select a file to access
available file actions:
1) example-file-1.txt
2) example-file-2.txt
3) exit
#? 2
1) delete
2) view
3) exit
#? 2
Your selected file's contents will be printed to the term
Lorem ipsum lorem ipsum
------------------------
Script is exiting ...
```

Introduction to the printf Command

The bash scripting language supports the `printf` command, which allows you t your scripts. Its roots are in the C programming language. You can read about C's operating system's manual pages with the following command: `man 3 printf`.

The general syntax for `printf` is a *format string* followed by *arguments* that will format and then inserted into the final output. A format string specifies where an

```
printf FORMAT [ARGUMENT]...
```

You can use variables as arguments to your `printf` commands. This is a powerfu will display varied output based on variable values. For example, the following `p` output by adding line breaks, defining an output color, and replacing part of the variable value. `$PWD` is an environment variable that stores your current working

```
printf "Your current working directory is: \x1b[32m\n %s\n"
```

```
Your current working directory is:
 /home/user
```

Format Strings

Format strings accept regular characters, which are unchanged in their output ar
where and how a string will be presented in the output.

Below is a list of common format specifiers:

`%s` : formatting an argument as a string

```
printf "%s\n" $OSTYPE
```

```
linux-gnu
```

`%d` : printing a value as an integer

```
printf "%d\n" "0xF9"
```

```
249
```

`%x` : printing a value as a hexadecimal number with lower case `a-f` . You cou
print the hexadecimal value with upper case `A-F`

```
printf "%x\n" "2000000"
```

```
1e8480
```

`%f` : printing floating point values

```
printf "%f\n" "0.01" "0.99"
```

```
0.010000
0.990000
```

> **Note**
>
> The `-v var` option causes the output of `printf` to be assigned to a variable i
> standard output. In the example below, the result of the `printf` format speci
> variable named `myvar` . To view the result, the example echoes the value of `$m`
>
> ```
> printf -v myvar "%d\n" "0xF9"
> echo $myvar
> ```

```
249
```

Use printf in a Script

The example script below makes use of `printf` to create a readable and nicely f of numbers. A `for` loop is used with the `seq` command to generate the number uses different format specifiers to provide slightly varying information from each the format specifiers used in the script that have not yet been covered:

`%04d` tells `printf` to print a decimal number using up to 4 digits. If the num added to the front of the number to fulfill the criteria.

The `%.10s` format string tells `printf` to print a string using no more than 10 string will be truncated.

`\t` and `\n` are used for printing tabs and newlines, respectively.

File: printf.sh

```
#!/bin/bash

for i in $( seq 1 10 )
do
    printf "%04d\t" "$i"
done
echo

for i in $( seq 1 10 )
do
    printf "%x\t" "$i"
done
echo

for i in $( seq 1 10 )
do
    printf "%X\t" "$i"
done
echo

for i in $( seq 10 15 )
do
    printf "%04d\t is %X\t in HEX.\n" "$i" "$i"
done

for i in $( seq 5 10 )
do
```

```
28      printf "%.10s is %X in HEX.\n" "$i.....
29  done
```

1. Copy and paste the contents of `printf.sh` into a new file and save it.
2. Run the script:

```
./printf.sh
```

The output of `printf.sh` will resemble the following:

```
0001    0002    0003    0004    0005    0006    0007    0
1   2   3   4   5   6   7   8   9   a
1   2   3   4   5   6   7   8   9   A
0010     is A    in HEX.
0011     is B    in HEX.
0012     is C    in HEX.
0013     is D    in HEX.
0014     is E    in HEX.
0015     is F    in HEX.
5......... is 5 in HEX.
6......... is 6 in HEX.
7......... is 7 in HEX.
8......... is 8 in HEX.
9......... is 9 in HEX.
10........ is A in HEX.
```

File and Directory Test Operators

Bash offers file and directory test operators that return a boolean value based on
These operators can be used in your Bash scripts to present different behaviors c
directory. A list of all test operators is included in the expandable note, “File and

The general format for file and directory test operators is the following:

```
test -[OPERATOR] [FILE]
```

The example below tests if your `/etc/passwd` file exists. If the file exists, you will
output. If the file does not exist, the first part of the command, `test -a /etc/pa`
(the exit value will not print as output) and the second part of the command, `ec`
execute.

```
test -a /etc/passwd && echo "Yes, it exists!"
```

> File and Directory Test Operators

Use File and Directory Test Operators in a Script

The example script, `file-operator.sh`, takes file or directory locations as argun each type of file that is passed to it. The script makes use of file and directory tes information. The first `if` statement tests to ensure you have passed the script ar to test if the arguments are files that actually exist and then continues through a directory for other criteria.

> **Note**
>
> You can use `[]` and `[[]]` commands instead of using the `if` conditional sta script makes use of this format on lines 26 - 40.

File: file-operator.sh

```
1  #!/bin/bash
2
3  if [[ $# -le 0 ]]
4  then
5      echo "You did not pass any files as arguments to t
6      echo "Usage:" "$0" "my-file-1 my-file-2"
7      exit
8  fi
9
10 for arg in "$@"
11 do
12     # Does it actually exist?
13     if [[ ! -e "$arg" ]]
14     then
15         echo "* Skipping ${arg}"
16         continue
17     fi
18
19     # Is it a regular file?
20     if [ -f "$arg" ]
21     then
22         echo "* $arg is a regular file!"
23     else
24         echo "* $arg is not a regular file!"
25     fi
26
27     [ -b "$arg" ] && echo "* $arg is a block device."
28     [ -d "$arg" ] && echo "* $arg is a directory."
```

```
29      [ ! -d "$arg" ] && echo "* $arg is not a directory
30
31      [ -x "$arg" ] && echo "* $arg is executable."
32      [ ! -x "$arg" ] && echo "* $arg is not executable.
33
34      [[ -h "$arg" ]] && echo "* $arg is a symbolic link
35      [ ! -h "$arg" ] && echo "* $arg is not a symbolic
36
37      [[ -s "$arg" ]] && echo "* $arg has nonzero size."
38      [ ! -s "$arg" ] && echo "* $arg has zero size."
39
40      [[ -r "$arg" && -d "$arg" ]] && echo "* $arg is a
41      [[ -r "$arg" && -f "$arg" ]] && echo "* $arg is a
42  done
```

1. Copy and paste the contents of `file-operator.sh` into a new file and save it
2. Run the script and pass it a file location as an argument:

   ```
   ./file-operator.sh /dev/fd/2
   ```

   Your output will resemble the following:

   ```
   * /dev/fd/2 is not a regular file!
   * /dev/fd/2 is not a directory.
   * /dev/fd/2 is not executable.
   * /dev/fd/2 is not a symbolic link.
   * /dev/fd/2 has zero size.
   ```

3. Run the script and pass it a directory location as an argument:

   ```
   ./file-operator.sh /var/log
   ```

   Your output will resemble the following:

   ```
   * /var/log is not a regular file!
   * /var/log is a directory.
   * /var/log is executable.
   * /var/log is not a symbolic link.
   * /var/log has nonzero size.
   * /var/log is a readable directory.
   ```

Read Files and Searching Directories

This section will present a few utility scripts that can be adopted and expanded c files and directories, like reading the contents of a text file by line, word, or chara of the concepts and techniques covered in this guide and in the Getting Started v

Read a File Line by Line

The example file, `line-by-line.sh` , expects a file passed to it as an argument. It line by line. The `IFS` variable (internal field separator) is a built-in Bash variable word boundaries when splitting words. The script sets `IFS` to the null string to p space within your text file.

File: line-by-line.sh

```
#!/bin/bash

if [[ $# -le 0 ]]
then
    echo "You did not pass any files as arguments to t
    echo "Usage:" "$0" "my-file"
    exit
fi

file=$1

if [ ! -f "$file" ]
then
    echo "File does not exist!"
fi

while IFS='' read -r line || [[ -n "$line" ]]; do
    echo "$line"
done < "${file}"
```

1. Copy and paste the contents of `line-by-line.sh` into a new file and save it.
2. Create an example file for the `line-by-line.sh` script to read. The leading w intentional.

```
echo -e '    Ultimately, literature is nothing but carp
```

3. Run the script and pass it a file location as an argument:

```
./line-by-line.sh marquez.txt
```

Your output will resemble the following:

```
Ultimately, literature is nothing but carpentry. Wit
```

Read a File Word by Word

The example bash script, `word-by-word.sh` expects a file to be passed as an argu argument has been passed to the script and that it is a file. It then uses a for loop word in the file to your output. The default value of the `IFS` variable separates a no need to change its value.

File: word-by-word.sh

```
#!/bin/bash

if [[ $# -le 0 ]]
then
    echo "You did not pass any files as arguments to t
    echo "Usage:" "$0" "my-file"
    exit
fi

file=$1

if [ ! -f "$file" ]
then
    echo "File does not exist!"
fi

for word in $(cat "${file}")
do
    echo "$word"
done
```

1. Copy and paste the contents of `word-by-word.sh` into a new file and save it.
2. Create an example file for the `word-by-word.sh` script to read.

```
echo -e 'Ultimately, literature is nothing but carpentry
```

3. Run the script and pass it a file location as an argument:

```
./word-by-word.sh marquez.txt
```

Your output will resemble the following:

```
Ultimately,
```

```
literature
is
nothing
but
carpentry.
With
both
you
are
working
with
reality,
a
material
just
as
hard
as
wood.
```

Read a File Character by Character

The example bash script, `char-by-char.sh` expects a file to be passed as an argu argument has been passed to the script and that it is a file. It then uses a while lo each character in the file to your shell's output. The `-n1` flag is added to the stan specify the number of characters to read at a time, which in this case is `1`.

File: char-by-char.sh

```
#!/bin/bash

if [[ $# -le 0 ]]
then
    echo "You did not pass any files as arguments to t
    echo "Usage:" "$0" "my-file"
    exit
fi

file=$1

if [ ! -f "$file" ]
then
    echo "File does not exist!"
fi

while read -r -n1 char; do
```

```
18      echo "$char"
19  done < "${file}"
```

1. Copy and paste the contents of `char-by-char.sh` into a new file and save it.
2. Create an example file for the `char-by-char.sh` script to read. The leading w intentional.

```
echo -e 'Linode' > linode.txt
```

3. Run the script and pass it a file location as an argument:

```
./char-by-char.sh linode.txt
```

Your output will resemble the following:

```
L
i
n
o
d
e
```

Search Directories

The bash script, `search.sh` will search a directory for files and directories that b command line argument. All matching regular files and directories will be presen search string as the first argument and a directory location as the second argume command for searching a directory and looks for everything that begins with ma

File: search.sh

```
1   #!/bin/bash
2
3   if [[ $# -le 1 ]]
4   then
5       echo "You did not pass any files as arguments to t
6       echo "Usage:" "$1" "my-file"
7       exit
8   fi
9
10  dir=$2
11  string=$1
12
13  if [ ! -d "$dir" ]
```

```
13  if [ ! -d "$dir" ]
14  then
15      echo "Directory" "$dir" "does not exist!"
16      exit
17  fi
18
19  for i in $(find "$dir" -name "$string*");
20  do
21      if [ -d "$i" ]
22      then
23         echo "$i" "[Directory]"
24      elif [ -f "$i" ]
25      then
26          echo "$i" "[File]"
27      fi
28  done
```

1. Copy and paste the contents of `search.sh` into a new file and save it.
2. Move to your home directory and create an example file and directory for the

```
cd ~ && echo -e 'Ultimately, literature is nothing but c
```

3. Run the script and pass it a string and directory location as arguments. Ensur `/home/user/` with your own home directory.

```
./bin/search.sh mar /home/user/
```

Your output will resemble the following:

```
/home/user/marketing [Directory]
/home/user/marquez.txt [File]
```

Bash Exit Codes

An *exit code* is the code returned to a parent process after executing a command Bash scripts allows the script to modify its behavior based on the success or failu codes range between `0 - 255`. An exit code of `0` indicates success, while any nc section will provide an introduction to Bash exit codes and a few examples on hc

> **Exit Codes**

Learning the Exit Code of a Shell Command

You can understand whether a bash command was executed successfully or not l
command. The built-in Bash variable `$?` stores the exit (return) status of the pre
example below issues the long format list files (`ls -l`) command against your `/`
output and standard error to `/dev/null` in order to suppress any output. Withou
knowing if the command executed successfully or failed. To circumvent this scen
variable to view the command's exit status.

1. Execute the following example command. You should not see any output, ho
executed successfully.

```
ls -l /tmp 2>/dev/null 1>/dev/null
```

2. Find the value of `$?` to determine if your command executed successfully or

```
echo "$?"
```

The exit code status should output `0` if the command was successful:

```
0
```

3. Issue the long form list files command against a directory that does not exist

```
ls -l /doesNotExist 2>/dev/null 1>/dev/null
```

4. Find the value of `$?` to determine if your command executed successfully or

```
echo "$?"
```

The exit code status should output `1` if the command failed:

```
1
```

> **Note**
>
> After you execute `echo $?` , the value of `$?` will always be `0` because `ech`

Using set -e

The `set` command is used to set or unset different shell options or positional pa
be set with this command is the `-e` option, which causes a bash script to exit if a
non-zero exit code. This option is useful, because it works globally on all comma
have to test the return status of each command that is executed.

The example script, `set-example.sh` , tries to create a file at the specified path. I
created, the script will immediately exit and none of the remaining commands w
zero exit code, you should not expect to see the last line execute the `echo "Scri`

File: set-example.sh

```
#!/bin/bash

set -e

if [[ $# -le 0 ]]
then
    echo "You did not pass any file paths as arguments
    echo "Usage:" "$0" "my-new-file-path"
    exit
fi

fpath=$1

echo "About to create file: " "$fpath"

if [ -e "$fpath" ]
then
    echo "${fpath}" "already exists!"
    exit
fi

echo "Creating and writing to the file: " "$fpath"
echo "Test" >> "$fpath"

echo "Script is exiting"
```

1. Copy and paste the contents of `set-example.sh` into a new file and save it.
2. Run the script and pass it a file location as an argument.

```
./set-example.sh /tmp/new-file
```

Creating a file in this location should be successful and your output will reser

```
About to create file:  /tmp/new-file
About to create and write to the file:  /tmp/new-file
Script is exiting
```

3. Now, run the script and pass it a file location that you likely do not have eleva

```
./set-example.sh /dev/new-file
```

Creating a file in this location should not be successful and your script will ex
command:

```
About to create file:  /dev/new-file
About to create and write to the file:  /dev/new-file
./set-e.sh: line 23: /dev/new-file: Permission denied
```

Using set -x

Another handy way to use the `set` command is by enabling the `-x` option. This
arguments before they're executed, which makes this a great option for debuggi

> **Note**
>
> Any output generated by the `set -x` execution trace will be preceded by a `+`
> built-in variable, `PS4` .

The example script below, `debug-set-example.sh` , contains identical code to th
however, it makes use of `set -x` in order to print out all commands before they'

File: debug-set-example.sh

```
1  #!/bin/bash
2
3  set -xe
4
5  if [[ $# -le 0 ]]
6  then
7      echo "You did not pass any file paths as arguments
8      echo "Usage:" "$0" "my-new-file-path"
9      exit
10 fi
11
12 fpath=$1
13
14 echo "About to create file: " "$fpath"
15
16 if [ -e "$fpath" ]
17 then
18     echo "${fpath}" "already exists!"
19     exit
20 fi
```

```
21
22    echo "Creating and writing to the file: " "$fpath"
23    echo "Test" >> "$fpath"
24
25    echo "Script is exiting"
```

1. Copy and paste the contents of `debug-set-example.sh` into a new file and sa
2. Run the script and pass it a file location as an argument.

```
./debug-set-example.sh /dev/new-file
```

Creating a file in this location should not be successful and your script will ex command. However, since you also have the `set -x` option enabled, you wi command the script exited.

```
+ [[ 1 -le 0 ]]
+ fpath=/dev/new-file
+ echo 'About to create file: ' /dev/new-file
About to create file:  /dev/new-file
+ '[' -e /dev/new-file ']'
+ echo 'About to create and write to the file: ' /dev/new
About to create and write to the file:  /dev/new-file
+ echo Test
./set-e.sh: line 23: /dev/new-file: Permission denied
```

More Information

You may wish to consult the following resources for additional information on th the hope that they will be useful, please note that we cannot vouch for the accur materials.

- GNU Bash

This page was originally published on Tuesday, November 5, 2019.

Your Feedback Is Important

Let us know if this guide was helpful to you

Provide Feedback

Join the conversation.

Read other comments or post your own below. Comments must be respectful, cc
the guide. Do not post external links or advertisements. Before posting, consider
addressed by contacting our Support team or asking on our Community Site.

Site Map
Support
Partners
Careers

Press Center
Legal Center
System Status