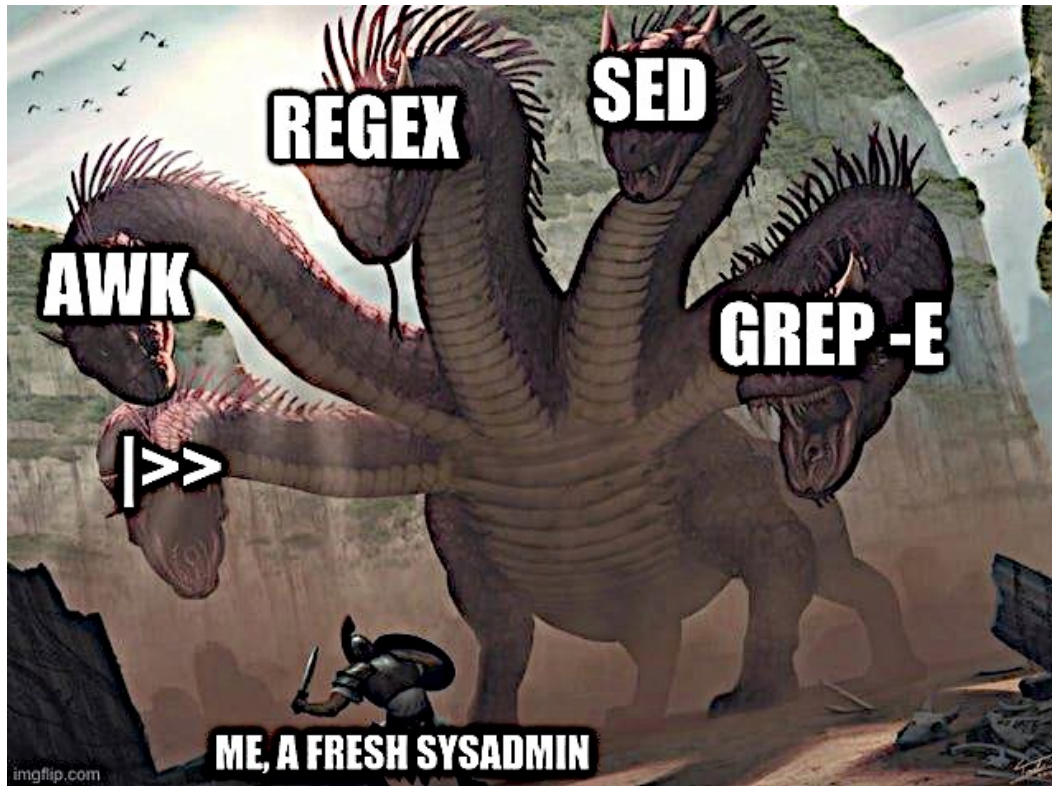


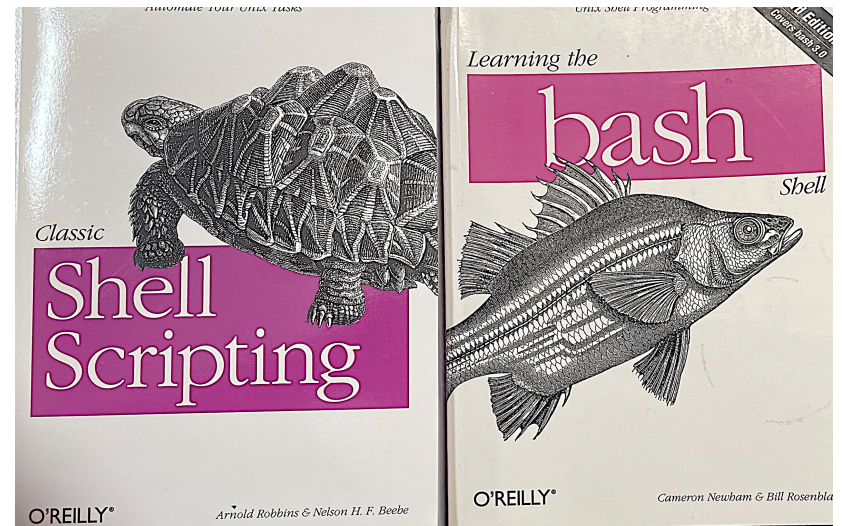
Lecture 5: Bash Language Part V

Photo of a new
Linux user learning
the command line



Suggested Literature

Both books are available at
<https://learning.oreilly.com>



RECAP: Printing: **echo** vs **printf**

echo: powerful and, in most cases, adequate command.

However, sometimes we need formatted output: **printf**

```
whovian@phy504:~/host$ printf "Hello World\n"
Hello World
whovian@phy504:~/host$ printf "Hello %s\n" world
Hello world
whovian@phy504:~/host$ printf "|%10s|\n" Hello
|      Hello|
whovian@phy504:~/host$ printf "|%-10s|\n" Hello
|Hello      |
whovian@phy504:~/host$
whovian@phy504:~/host$ printf "%d + %d = %d\n" 2 2 4
2 + 2 = 4
```

RECAP: Defining variables in Bash

We can define variables that can hold values different than their names

```
whovian@phy504:~$ myvar=this_is_a_long_string_that_does_not_mean_much
whovian@phy504:~$ echo -e '\n' $myvar '\n'

this_is_a_long_string_that_does_not_mean_much
```

Variables can hold integers. They can also hold floats

```
whovian@phy504:~$ myvar=10
whovian@phy504:~$
whovian@phy504:~$ echo -e '\n' $myvar '\n'

10
```

However, there isn't float arithmetic in pure bash

RECAP: Parameter expansion

Many times, it is important to provide default values / behavior

Operator	Substitution
<code>\${ varname := word}</code>	If <i>varname</i> exists and isn't null, return its value; otherwise set it to <i>word</i> and then return its value. Positional and special parameters cannot be assigned this way. Purpose: Setting a variable to a default value if it is undefined. Example: <code>\${count:=0}</code> sets count to 0 if it is undefined.

```
whovian@phy504:~/host$ echo ${MYVAR}
```

```
whovian@phy504:~/host$ MYVAR2=${MYVAR:=10}
```

```
whovian@phy504:~/host$ echo ${MYVAR2}
```

```
10
```

```
whovian@phy504:~/host$ echo ${MYVAR}
```

```
10
```

RECAP: Removing substrings

String manipulation w/ pattern matching is powerful in bash

Syntax:

`${parameter#word}` (short match pattern)

`${parameter##word}` (long match pattern)

Example:

```
whovian@phy504:~$ text="abcXYZdefXYZghi"
whovian@phy504:~$ short_remove="${text#abc*XYZ}"
whovian@phy504:~$ echo "Using #: $short_remove"
Using #: defXYZghi
whovian@phy504:~$ long_remove="${text##abc*XYZ}"
whovian@phy504:~$ echo "Using ##: $long_remove"
Using ##: ghi
```

What is the meaning of **`abc*XYZ`**?

- “**`*`**” = wildcard. Expand to **None**, **One** or **more** chars.

RECAP: Removing substrings

There are more wildcard operators in pattern matching

?(*pattern-list*)

Matches zero or one occurrence of the given patterns.

***(*pattern-list*)**

Matches zero or more occurrences of the given patterns.

+(*pattern-list*)

Matches one or more occurrences of the given patterns.

@(*pattern-list*)

Matches one of the given patterns.

!(*pattern-list*)

Matches anything except one of the given patterns.

Mastering them takes time and practice.

This is where AI can help you in practice

Example:

abc3567XYZ1defXYZomgXYZghi

and you want the output to be

omgXYZghi

Precision on pattern matching is everything. Diff. logic

```
whovian@phy504:~$ echo -e ${text#abc+([0-9])XYZ+([a-zA-Z])XYZ}
abc3567XYZ1defXYZomgXYZghi
```

```
whovian@phy504:~$ █ With the missing [0-9], our pattern doesn't match anything
```

RECAP: Removing/replacing substrings

String manipulation w/ pattern matching is powerful in bash

Syntax: (need this line) **shopt -s extglob**

`${parameter/pattern/string}.` (first match)

`${parameter//pattern/string}.` (all match)

`${parameter/#pattern/string}` (beginning of line)

`${parameter/%pattern/string}` (end of line)

```
whovian@phy504:~$ NAME="Vivian Miranda Miranda"
whovian@phy504:~$ echo -e ${NAME/#"Miranda"/"Elisa"}
Vivian Miranda Miranda
whovian@phy504:~$ NAME="Miranda Vivian Miranda"
whovian@phy504:~$ echo -e ${NAME/#"Miranda"/"Elisa"}
Elisa Vivian Miranda
whovian@phy504:~$ echo -e ${NAME/%"Miranda"/"Elisa"}
Miranda Vivian Elisa
```

Positional Parameters inside a script (.sh)

Built-in variables. They have names 1, 2, 3...

Unique variables “@” and “*”. Positional parameter: 0

The number of positional parameters: #

shift moves parameters (and delete the first)

```
#!/bin/bash

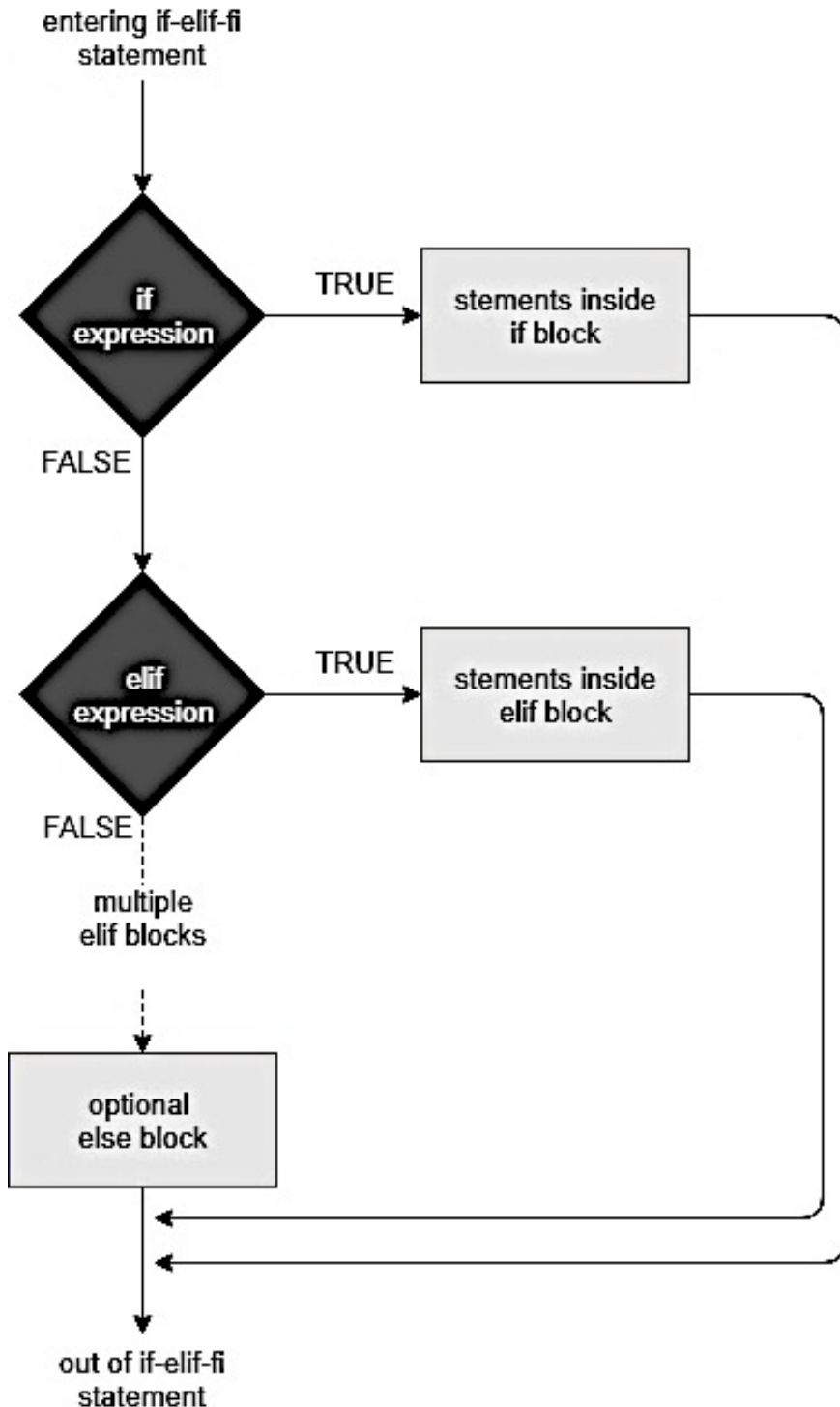
echo -en "Show all parameters: $@ \n"
echo -en "Show first parameter: $1 \n"
echo -en "Show the number of parameters $# \n"

echo -e "\nThis is going to be interesting now \n"

echo -en "Show first parameter: $1. Number of parameters $# \n"
shift
echo -en "Show second parameter: $1. Number of parameters $# \n"
shift
echo -en "Show third parameter: $1. Number of parameters $# \n"
```

Loops

if/elif/else statements



```
GNU nano 4.8
#!/bin/bash

n=2

if [ $n -eq 1 ]; then
    echo value of n is 1
elif [ $n -eq 2 ]; then
    echo value of n is 2
else
    echo value of n is other than 1 and 2
fi
```

Comparing operator for integers

```
whovian@phy504:~/host$ bash ifelse.sh
value of n is 2
```

Comparing Strings

How can we know if two strings are the same (case sensitive)

```
GNU nano 4.8
#!/bin/bash
s1="HELLO WORLD"
s2="Hello World"
if [ "$s1" = "$s2" ]; then
    echo "s1 and s2 are the same"
else
    echo "Not Same"
fi
```

Question with
T/F possible
answers

This runs if
True

This is the first time
we also introduce
if/else statements

Be careful

a = 2

is assignment while

"\$a" = "\$b"

can be string
comparison

```
whovian@phy504:~/host$ bash sequa1.sh
Not Same
```

HW: use the operator **!=** which returns T if strings are not equal

Comparing Strings

How can we know if one strings is “less than” than other

```
GNU nano 4.8
#!/bin/bash
s1="HELLO WORLD"
s2="HELLO"

if [[ "$s1" < "$s2" ]]; then
    echo "s1 is less than s2"
elif [[ "$s1" > "$s2" ]]; then
    echo "s2 is less than s1"
else
    echo "s1 and s2 are equal"
fi
```

Many interesting commands in this script

- **if/elif/else** statements
- Lexicographically comparisons (dictionary)
- double brackets [[]] vs []

```
whovian@phy504:~/host$ bash sless.sh
s2 is less than s1
```



Behavior differences

422

Some differences on Bash 4.3.11:



- POSIX vs Bash extension:
 - `[]` is POSIX
 - `[[` is a Bash extension inspired from [KornShell](#)
 - regular command vs magic
 - `[]` is just a regular command with a weird name.
 - `]` is just the last argument of `[]`.

[Ubuntu 16.04](#) actually has an executable for it at `/usr/bin/[]` provided by [coreutils](#), but the Bash built-in version takes precedence.

Nothing is altered in the way that Bash parses the command.

In particular, `<` is redirection, `&&` and `||` concatenate multiple commands, `()` generates subshells unless escaped by `\`, and word expansion happens as usual.
 - `[[ X ]]` is a single construct that makes `X` be parsed magically. `<`, `&&`, `||` and `()` are treated specially, and word splitting rules are different.
- There are also further differences like `=` and `=~`.
- In Bashese: `[]` is a built-in command, and `[[` is a keyword: [What's the difference between shell builtin and shell keyword?](#)
- `<`
 - `[[ a < b ]]`: lexicographical comparison
 - `[ a \< b ]`: Same as above. `\` required or else does redirection like for any other command. Bash extension.
 - `expr x"$x" \< x"$y" > /dev/null` or `[ "$(expr x"$x" \< x"$y")" = 1 ]`: POSIX equivalents, see: [How to test strings for lexicographic less than or equal in Bash?](#)
 - `&&` and `||`
 - `[[ a = a && b = b ]]`: true, logical *and*
 - `[ a = a && b = b ]`: syntax error, `&&` parsed as an AND command separator `cmd1 && cmd2`
 - `[ a = a ] && [ b = b ]`: POSIX reliable equivalent
 - `[ a = a -a b = b ]`: almost equivalent, but deprecated by POSIX because it is insane and fails for some values of `a` or `b` like `!` or `()` which would be interpreted as logical operations

[...] VS [[...]]

Usually, POSIX is better (**not** in this case).

What is POSIX?

Portable Operating System Interface

When things are not POSIX compliant, code can break when running in different systems.

POSIX is conservative!

GNU nano 4.8

#!/bin/bash

Code from my research

```
if [ -z "${ROOTDIR}" ]; then
    echo 'ERROR ROOTDIR not defined'
    return
fi
if [ -z "${CXX_COMPILER}" ]; then
    echo 'ERROR CXX_COMPILER not defined'
    return
fi
if [ -z "${C_COMPILER}" ]; then
    echo 'ERROR C_COMPILER not defined'
    return
fi
if [ -z "${MAKE_NUM_THREADS}" ]; then
    echo 'ERROR MAKE_NUM_THREADS not defined'
    return
fi

# ----- COMPILE COSMOLIKE -----

cd $ROOTDIR/projects/des_y3/interface
```

To be, or not to be, that is the question

The **-z** and **-n** comparison operators test if a string is **null**.

If your script depends on a key being set, use **if/else** with **-n** and **-z** operators to check if string is **null**

Comparing Strings: summary

Table 5-1. String comparison operators

Operator	True if...
<code>str1 = str2</code> ^[4]	<code>str1</code> matches <code>str2</code>
<code>str1 != str2</code>	<code>str1</code> does not match <code>str2</code>
<code>str1 < str2</code>	<code>str1</code> is less than <code>str2</code>
<code>str1 > str2</code>	<code>str1</code> is greater than <code>str2</code>
<code>-n str1</code>	<code>str1</code> is not null (has length greater than 0)
<code>-z str1</code>	<code>str1</code> is null (has length 0)

1. Digits
2. Uppercase letters
3. Lowercase letters

< and > are dictionary comparison

if/elif/else statements

if / elif / else with file attribute checking (super useful)

Operator	True if...
-a <i>file</i>	<i>file</i> exists
-d <i>file</i>	<i>file</i> exists and is a directory
-e <i>file</i>	<i>file</i> exists; same as -a
-f <i>file</i>	<i>file</i> exists and is a regular file (i.e., not a directory or other special type of file)
-r <i>file</i>	You have read permission on <i>file</i>
-s <i>file</i>	<i>file</i> exists and is not empty
-w <i>file</i>	You have write permission on <i>file</i>

```
whovian@phy504:~/host$ bash ifelse2.sh nofile.sh
file nofile.sh does not exist.
```

```
#!/bin/bash
if [ ! -e "$1" ]; then
    echo "file $1 does not exist."
    exit 1
fi
```

Red arrows point from the text "opposite positional parameters" to the `!` and `$1` in the script.

Built-in positional parameters. They have names **1, 2, 3...**

if/elif/else statements

Integer comparison has different operators

Table 5-3. Arithmetic test operators

Test	Comparison	-eq	Equal	-gt	Greater than
-lt	Less than	-ge	Greater than or equal	-ne	Not equal
-le	Less than or equal				

```
GNU nano 4.8
#!/bin/bash

if [[ 6 > 57 ]]; then
    echo "Dictionary order"
fi

if [[ 57 -gt 6 ]]; then
    echo "Arithmetic order"
fi
```

```
whovian@phy504:~/host$ bash ifelse.sh
Dictionary order
Arithmetic order
```

if/elif/else statements

if/elif/else with logical operators (**&&** equals and), (**||** equals or)

The syntax *statement1 && statement2* means, “execute *statement1*, and if its exit status is **0**, execute *statement2*.”

The syntax *statement1 || statement2* is the converse: it means, “execute *statement1*, and if its exit status is *not* **0**, execute *statement2*.”

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

```
if [[ $1 -gt 3 && $1 -gt 4 ]]
then
```

```
    echo "Yes (&&): $1"
```

```
fi
```

```
if [[ $1 -gt 3 || $1 -gt 5 ]]
then
```

```
    echo "Yes (||): $1"
```

```
fi
```

```
whovian@phy504:~/host$ bash ifelse3.sh 4
Yes (||): 4
```

```
whovian@phy504:~/host$ bash ifelse3.sh 5
Yes (&&): 5
Yes (||): 5
```

Loops: arithmetic **for** loops

The form of an arithmetic for loop is similar to those found in C/C++

```
for (( initialisation ; ending condition ; update ))  
do  
    statements...  
done
```

```
GNU nano 4.8  
#!/bin/bash  
  
for (( i=1; i <= 5; i++ ))  
do  
    for (( j=1; j <=5; j++ ))  
    do  
        echo -ne "$(( j*i ))\t"  
    done  
  
    echo  
done
```

Arithmetic for loop useful when
dealing with arrays

```
whovian@phy504:~/host$ bash forloop.sh  
1      2      3      4      5  
2      4      6      8      10  
3      6      9      12     15  
4      8      12     16     20  
5      10     15     20     25
```

Loops: arithmetic **for** loops x2

Loop through a range of numbers with a step or a simple manual list

```
GNU nano 4.8
#!/bin/bash

for i in {0..10..2}; do
    echo $i
done
```



```
GNU nano 4.8
#!/bin/bash

for i in 1 2 3 4 5; do
    echo $i
done
```

```
whovian@phy504:~/host$ bash forloop4.sh
0
2
4
6
8
10
```

```
whovian@phy504:~/host$ bash forloop6.sh
1
2
3
4
5
```

The range and list can be a list of strings as well!

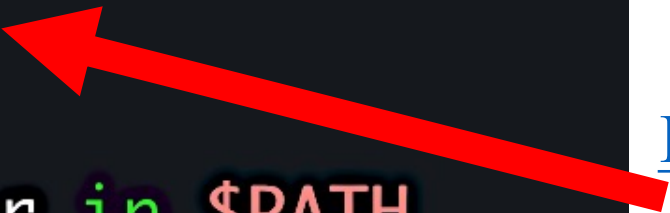
Loops: **for loops** (iteration)

You can loop over files in a directory

```
whovian@phy504:~/host$ echo $PATH
/opt/conda/bin /usr/local/nvidia/bin /usr/local/cuda/bin
/usr/local/sbin /usr/local/bin /usr/sbin /usr/bin /sbin
/bin
```

```
GNU nano 4.8
#!/bin/bash

IFS=:
for dir in $PATH
do
    ls -ld $dir
done
```



```
for name [in list]
do
    statements that can use
        $name...
done
```

IFS determines how Bash recognizes word boundaries while splitting a sequence of character strings

The default value of **IFS** is a three-character string comprising a space, tab, and newline:

Loops: **for loops** (iteration)

You can loop over files in a directory

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

```
for file in ~/host/*; do
    if [ -f "$file" ]; then
        echo "$file is a regular file"
    fi
    if [ -d "$file" ]; then
        echo "$file is a directory"
    fi
done
```

wildcard
(means every file)



-f checks if the
file is a regular
file (not a
symbolic link)

```
whovian@phy504:~/host$ bash forloop3.sh
/home/whovian/host/2 is a regular file
/home/whovian/host/address_list_backup.txt is a regular file
/home/whovian/host/address_list.txt is a regular file
/home/whovian/host/C is a directory
```

Loops: **for loops** (iteration)

You can loop over files in a directory recursively using find

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

```
x=$(find . -type f)
```

**command
substitution**



```
for file in $x; do
    if [ -f "$file" ]; then
        echo "$file is a regular file"
    fi
done
```

```
echo -e "\n\nAnother way to write the same loop\n\n"
```

```
for file in $(find . -type f); do
    if [ -f "$file" ]; then
        echo "$file is a regular file"
    fi
done
```

while loops

while loops run until certain condition is **True**

```
while [ condition ]  
do  
    commands  
done
```

```
GNU nano 4.8  
#!/bin/bash  
  
# Initialize the counter  
n=1  
while [ $n -le 5 ]  
do  
    echo "Running $n time"  
    # Increment the value of n by 1  
    (( n++ ))  
done
```

```
whovian@phy504:~/host$ bash whileloop.sh  
Running 1 time  
Running 2 time  
Running 3 time  
Running 4 time  
Running 5 time
```

while loops x2

```
#!/bin/bash

# Initialize the counter
n=1
# Iterate the loop for 10 times
while [ $n -le 10 ]
do
    if [[ $n == 6 ]]
    then
        echo "terminated"
        break
    fi
    echo "Position: $n"
    (( n++ ))
done
```

```
#
    echo "terminated"
    break
    (( n++ ))
    continue
```

Warning: **while [True]**
never stops
(unless we **break** it!)
How **break** works?

```
whovian@phy504:~/host$ bash whileloop2.sh
Position: 1
Position: 2
Position: 3
Position: 4
Position: 5
terminated
```

break vs **continue**



```
Position: 5
terminated
Position: 7
```

while loops x3 – nested while loops

```
#!/bin/bash

n=1
while [ $n -le 3 ]
do
    echo "Position: n=$n"
    m=1
    while [ $m -le 3 ]
    do
        echo -ne "Position: m=$m \t"
        if [[ $m == 2 ]]
        then
            echo -ne "terminated \n"
            break
        fi
        (( m++ ))
    done
    (( n++ ))
done
```

How **break** works?

break in the inner loop
exit only the inner loop

```
whovian@phy504:~/host$ bash whileloop3.sh
Position: n=1
Position: m=1    Position: m=2    terminated
Position: n=2
Position: m=1    Position: m=2    terminated
Position: n=3
Position: m=1    Position: m=2    terminated
```

case loops

```
#!/bin/bash

echo -n "Enter the name of a country: "
read COUNTRY

echo -n "The official language of $COUNTRY is "

case $COUNTRY in
    Lithuania)
        echo "Lithuanian"
        ;;
    Romania | Moldova)
        echo "Romanian"
        ;;
    Italy | "San Marino" | "Vatican City")
        echo "Italian"
        ;;
    *)
        echo "unknown"
        ;;
esac
```



You can use logical operators “&&” and “||”

```
whovian@phy504:~/host$ bash caseloop.sh
Enter the name of a country: Brazil
The official language of Brazil is unknown
```

while + case loops (option processing)

```
#!/bin/bash
# set flag vars to empty
file= verbose= quiet=

while [ $# -gt 0 ]           # Loop until no args left
do
    case $1 in               # Check first arg
        -f) file=$2          # Shift off "-f" so that shift at end gets value in $2
            shift
            ;;
        -v) verbose=true
            quiet=
            ;;
        --) shift             # By convention, -- ends options
            break
            ;;
        -*) echo "$0: $1: unrecognized option" >&2
            ;;
        *) break             # Nonoption argument, break while loop
            ;;
    esac
    shift                   # Set up for next iteration
done
```

Number of positional parameters

one limitation: no implementation of compound options like **-fg**.

The preferred method for options is with **getopts**

while + case + getopts (option processing)

getopts allows multiple options in the same dash, e.g., **-fv**

```
#!/bin/bash
file=
while getopts ":fvg:" optname
do
```

Start with :, then for each option that needs an argument, write an :

```
    case ${optname} in
    f)    echo "-f option!"
          ;;
    v)    echo "-v option!"
          ;;
    g)    file=${OPTARG}
          echo "-g option! File=${file}"
          ;;
    :)    echo -e "Error: -${OPTARG} requires an argument\r"
          exit 1
          ;;
```

Variable **OPTARG** provides the arguments

:) = whenever an argument is not provided

```
    *)    echo "Option does not exist"
          break
          ;;
    esac
done
```

no need to use **shift**

You don't need the dash

while + case + getops (option processing)

```
whovian@phy504:~/host$ bash whileloop5.sh -f
-f option!
whovian@phy504:~/host$ bash whileloop5.sh -v
-v option!
whovian@phy504:~/host$ bash whileloop5.sh -fv
-f option!
-v option!
whovian@phy504:~/host$ bash whileloop5.sh -g vivian.txt
-g option! File=vivian.txt
whovian@phy504:~/host$ bash whileloop5.sh -g
Error: -g requires an argument
```

mixing **getops** and positional parameters is quite advanced

(I won't ask that – see StackOverflow [answer](#))

getops is to deal with options using dashes

Positional parameters do not need to come with a dash

Functions

Bash Functions

Functions can be considered a box

Input comes in, and output goes out.

How many inputs? Depends on the function.

```
whovian@phy504:~$ say_hello() {  
    echo "Hello, $1!"  
}
```

```
whovian@phy504:~$ calculate_square() {  
    local number=$1  
    local result=$((number * number))  
    echo $result  
}
```

```
whovian@phy504:~$ greet() {  
    local name=${1:-"World"} # Default value if no argument provided  
    echo "Hello, $name!"  
}
```

```
whovian@phy504:~$ say_hello  
Hello, !
```

```
whovian@phy504:~$ calculate_square 3  
9
```

```
whovian@phy504:~$ greet Vivian  
Hello, Vivian!
```

local (required
keyword) variable:
defined only inside
the function

Input positional parameters

Bash Functions

To write >1 commands in a single line, use semicolons

```
whovian@phy504:~$ say_hello() { echo "Hello, $1!"; }
whovian@phy504:~$ calculate_square() { local number=$1; local result=$((number * number)); echo $result; }
whovian@phy504:~$ say_hello
Hello, !
whovian@phy504:~$ calculate_square 3
9
whovian@phy504:~$
```



In bash, arguments are separated by space

```
whovian@phy504:~$ unset -f say_hello
whovian@phy504:~$ unset -f calculate_square
whovian@phy504:~$ say_hello
bash: say_hello: command not found
whovian@phy504:~$ calculate_square 3
bash: calculate_square: command not found
whovian@phy504:~$
```

To unset function definitions, use

unset -f

Homework I

- Write a script that reads two strings, checks if they are empty, and then outputs the longer string
- Write a script that takes user input and checks whether it is a positive or negative number. If the number is positive, the script should output **“Positive number”**. If the number is negative, the script should output "Negative number"
- Write a script that takes two user inputs and checks whether the first input is divisible by the second input. If the first input is divisible by the second input, the script should output **“First input is divisible by the second input”**. If the first input is not divisible by the second input, the script should output **“First input is not divisible by the second input”**

Homework II

- Write a script that checks whether a file exists in the current directory. If the file exists, the script should output "**File exists**". If the file does not exist, the script should output "**File does not exist**"
- Write a script that ask the user to enter a number between 20 and 30. If the user enters an invalid number or a non-number, ask again. Repeat until a satisfactory number is entered. Hint: solution uses **while true** and **break** commands
- Write a script that prompts the user to enter the name of a file. Repeat until the user enters a file that exists
- Write a script that loops through the command line arguments and print them (Hint: use **@** and **shift**)

Homework III

- Write a script that print all odd numbers up to 50
- Write a program that asks the user in the terminal to create a password. Then, the program should check if the password length is less than four characters. If it is smaller than four chars, the program should complain with the sentence “**Password is too short: x characters\n**” where **x** is the number of chars.
- Write a program with two options. **-f** option makes the program write a positive message (e.g., “**The World is Wonderful**”). **-g** option makes the program write a negative message (e.g., “**The World is Terrible**”). All other letters (e.g., **-g** or **-z**), makes the program write a neutral message (e.g., “**The world is average**”)

Homework IV

- Explain in words what this function does. Code it yourself and run an example. What is the most surprising or challenging bash feature of this code?
- I get this output when I run the extract function without an argument. Modify the script to test if there is an argument and provide a personalized, more explanatory error message if it does not.

```
extract() {  
    if [ -f "$1" ]; then  
        case "$1" in  
            *.tar.bz2)    tar xjf "$1"      ;;  
            *.tar.gz)    tar xzf "$1"      ;;  
            *.bz2)       bunzip2 "$1"      ;;  
            *.rar)       unrar e "$1"      ;;  
            *.gz)        gunzip "$1"       ;;  
            *.tar)       tar xf "$1"       ;;  
            *.tbz2)      tar xjf "$1"      ;;  
            *.tgz)       tar xzf "$1"      ;;  
            *.zip)       unzip "$1"        ;;  
            *.Z)         uncompress "$1"    ;;  
            *)           echo "'$1' cannot be extracted" ;;  
        esac  
    else  
        echo "'$1' is not a valid file"  
    fi  
}
```

```
whovian@phy504:~$ extract  
' ' is not a valid file
```