Bash scripting

Learning outcome

- Combining linux commands
- Overview of the features
- Command exit status
- Variables
- Arithmetic expressions
- Job control
- Constructs

How to print lines having only numbers

```
cat minima.dat
```

```
97.11	70.15
-82.63	79.14
-177.00		106.10
97.11 106.10
-78.14sand110.59
-177.00 178.00
97.11 178.00
97.11 178.00ssa
1010	101000
```

How to print lines having only numbers

```
grep -E "^[0-9.-]+\s+[0-9.-]+$" minima.dat  OR
egrep "^[0-9.-]+\s+[0-9.-]+$" minima.dat OR
```

OR

```
while IFS= read -r line; do
    if [[ $line =~ ^[0-9.-]+\s+[0-9.-]+$ ]]; then echo "$line" ; fi
done < "minima.dat"
```

OR

```
for line in $(cat "minima.dat"); do
    if [[ $line =~ ^[0-9.-]+\s+[0-9.-]+$ ]]; then
        echo "$line"
    fi
done
```

OR

```
grep -E "^[0-9.-]+\s+[0-9.-]+$" <(cat "minima.dat")
```

History

- 1979: Bourne shell - /bin/sh
  - Only a few additional features are added over the time to the shell
  - 1980: built-in `test` command
  - 1984: hashing and shell functions
  - 1989: job control features
- Competitors to Bourne shell are csh and ksh
- 1989: GNU developed Bourne-Again SHell (bash)
  - written from scratch by incorporating the features from other existing shells.

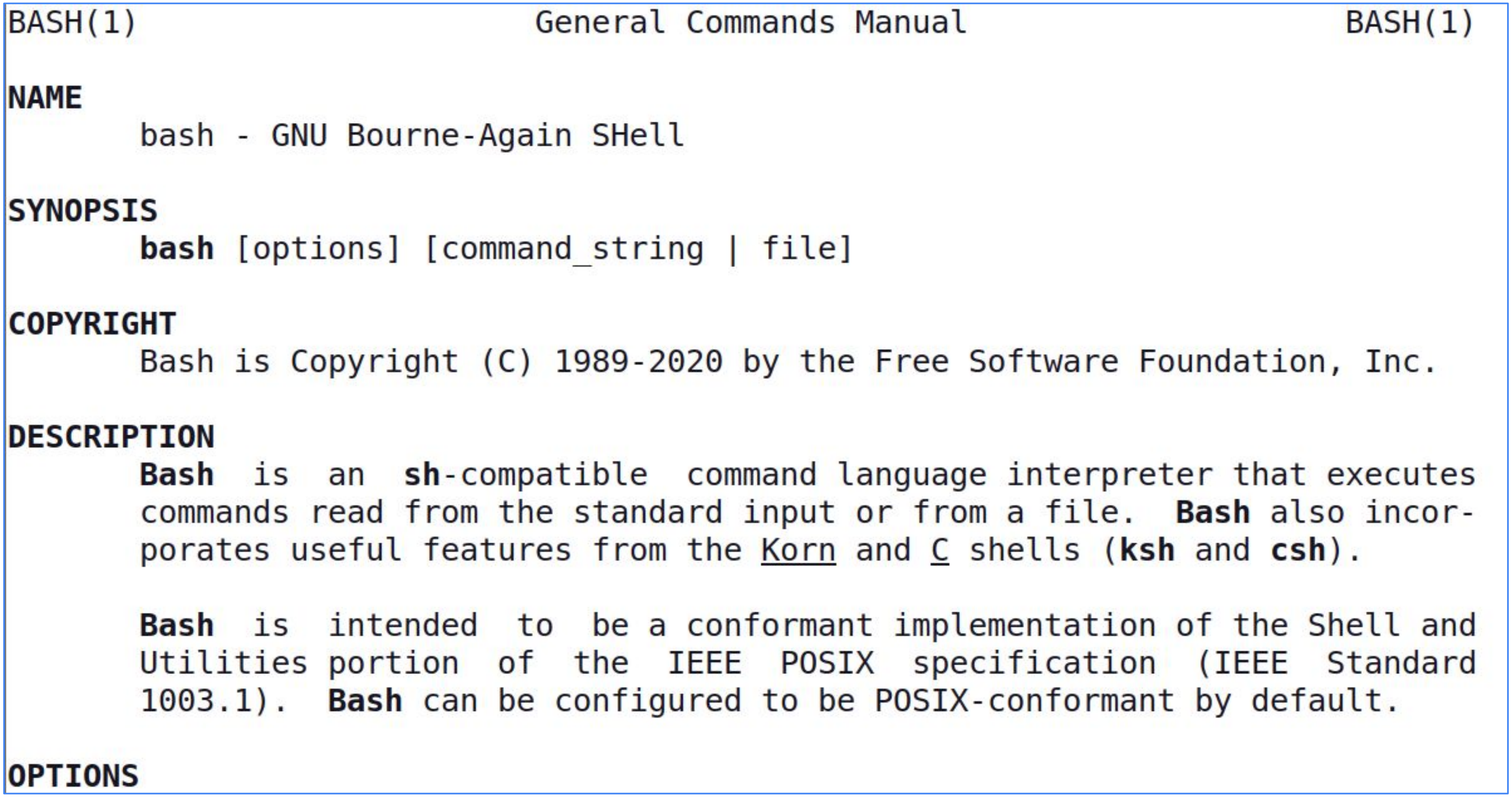
bash overview

- bash is a powerful scripting language. It is open source
- Bourne shell subset of Bash, with additional features
- Learning Bash and shell scripting is learning Unix/Linux
- Vital role in automating tasks and system administration
- A sequence of commands becomes a script file, with added command-line options
- Define variable, functions, loops, etc enable efficient execution of more complex tasks
- Command-line options are used to enable different options to be passed to the commands

bash features

- Input/output redirection
- Wildcard characters
- variables
- Built-in command set for writing programs
- Job control
- Command-line editing
- History
- Integer arithmetic
- Arrays and arithmetic expressions
- for loop

man bash



```
BASH(1)                     General Commands Manual                    BASH(1)

NAME
       bash - GNU Bourne-Again SHell

SYNOPSIS
       bash [options] [command_string | file]

COPYRIGHT
       Bash is Copyright (C) 1989-2020 by the Free Software Foundation, Inc.

DESCRIPTION
       Bash  is  an  sh-compatible  command language interpreter that executes
       commands read from the standard input or from a file.  Bash also incor-
       porates useful features from the Korn and C shells (ksh and csh).

       Bash  is  intended  to  be a conformant implementation of the Shell and
       Utilities portion  of  the  IEEE  POSIX  specification  (IEEE  Standard
       1003.1).  Bash can be configured to be POSIX-conformant by default.

OPTIONS
```

First command

```
bash [options] [arguments]
```

- `bash hpc.sh`
- `bash hpc.sh arg1 arg2 arg3`

- First line: `#!/bin/bash`
- Comments： These lines start with a `#` symbol

Script arguments

- Arguments are values passed to the script on the command line
- Arguments are accessed using special variables: `$0, $1, $2, ...`
- Examples:

`$0`: Script Name

`$1, $2, ...`: Positional Arguments

`$#`: Number of Arguments

`$*`: All Arguments as a Single String

`$@`: All Arguments as Separate Strings

Example

```
#!/bin/bash
x=$1
y=$2
echo "No. of arguments: "$#
echo "All arguments as a separate string"
for i in "$@"; do
  echo $i
done
echo "----- "
echo "All arguments as a single string"
for i in "$*"; do
  echo $i
done
```

```
./1.bash "1 "  "2 4"
No. of arguments: 2
All arguments as a separate string
1
2 4
-----
All arguments as a single string
1 2 4
```

Example

```
#!/bin/bash

x=$1
y=$2

if [ "$#" -ne 2 ]; then
  echo ./$0 var1 var2
  echo "exiting ... "
  exit
fi
```

Essential elements of the bash script

- Input/output: read input from user or command line or a file and display output on terminal
- `read`, `echo` or `printf`
- Control structures:
  - `if` statements
  - `case` statements
  - `for` loops
  - `while` loops
- Functions: Set of commands can be grouped to define as a function, and reuse it at multiple times in the script

Essential elements of the bash script

- Command substitution: Output of one command can be used as input for another command
  - \`...\` or $(...)
- Exit status and error handling:
- Redirection: Input and output can be redirected
- Arithmetic operations and conditional expressions

Arithmetic Evaluation and Expansion with **integers**

- Syntax:

```
    $((expression))
```

where expression is a valid arithmetic expression

```
#!/bin/bash
x=5; y=10; z=2
addition=$((x + y))
multiplication=$((x *y))
division=$((y / x))
result=$(( (x + y) * z ))
square=$((z*z))
```

Arithmetic operators

- + / - / * / / : addition/subtraction/multiplication/integer division
- ** : exponentiation
- % : modulo (remainder)
- -= : subtraction assignment
- += : addition assignment
- *= : multiplication assignment
- /= : integer division assignment
- parameter++ : post-increment
- parameter-- : post-decrement

Examples

```
echo "5*2 = "$(( 5*2 ))
echo "5/2 = "$(( 5/2 ))
echo "5%2 = "$(( 5%2 ))
echo "5**2 = "$(( 5**2 ))
num=10
(( num += 10 ))
echo $num
echo $((num++))
echo $num
```

exit status

- Every command exits with a numeric status
  - 0 - true or success
  - non zero value denotes a particular type of error
- For eg: when you type the command `ls`, it returns an exit status (not displayed on standard output). You can display the exit status using the command echo $?

Common Exit Status Codes

- 0 - Success: Command executed successfully without errors.
- 1 - General Errors: Often used to indicate that something went wrong, but not specific.
- 2 - Misuse of Shell Builtins: Incorrect usage of shell built-in commands.
- 126 - Command Not Executable: Permission issues or command not found.
- 127 - Command Not Found: The command couldn't be found or isn't executable.
- 128+x - Fatal Errors: Signals and process interruptions (x represents the signal number).
- 130 - Script Terminated: User interrupted the script using Ctrl+C.
- 255 - Exit Status Out of Range: Used when the exit status exceeds the valid range (0-255).

Using Exit Status

- Use the special variable $? to access the exit status of the last command.
- $? contains the exit status of the most recently executed command.
- You can use conditional statements and control flow based on exit status.

Examples

```
#!/bin/bash

ls file.dat > /dev/null
status=$?
echo $status
```

```
#!/bin/bash

ls file.dat > /dev/null
if [ $? -ne 0 ]; then
    echo "file not found"
fi
```

File Redirection

- `>`: Redirects standard output to a file, overwriting if exists.

```
echo "NewFile" > output.txt
```

- `>>`: Appends standard output to a file.

```
echo "Appended" >> output.txt
```

- `2>`: Redirects standard error to a file.

```
ls non_existent 2> error.txt
```

- `&>`: Redirects both output and error to the same file.

```
ls non_existent &> output_err.txt
```

- `>/dev/null`: Redirects output to null device, discarding it.

```
ls non_existent > /dev/null
```

File Redirection

- `<`: Redirects input from a file.
- Example: `while read line; do ... done < input.txt`
- `<<`: Takes input from script
- Example:

```
cat << EOF > file.txt
Multiline text
EOF
```

Constructs

For Loop:

```
for variable in value1 value2 ...; do
    # code to execute for each value
done
```

While Loop:

```
while [ condition ]; do
    # code to execute while condition is true
done
```

Constructs

`For` Loop:

```
for num in 1 2 3 4 5; do
 echo "Number: $num"
done
```

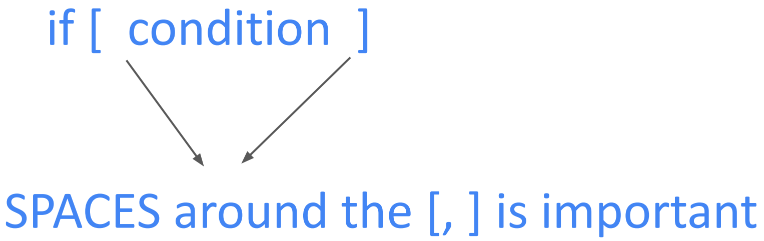
`While` Loop:

```
while [[ $count -le 5 ]]; do
 echo "Count: $count"
 ((count++))
done
```

Constructs

If Statement:

```
if [ condition ]; then
    # code to execute if condition is true
elif [ another_condition ]; then
    # code to execute if another_condition is true
else
    # code to execute if none of the conditions are true
fi
```



if [condition]

SPACES around the [,] is important

Constructs

If Statement:

```
if [ condition ]; then
    # code to execute if condition is true
elif [ another_condition ]; then
    # code to execute if another_condition is true
else
    # code to execute if none of the conditions are true
fi
```

Optional

Constructs

If Statement:

- [[...]] Construct - construct is an extended conditional expression in Bash.
- offers enhanced features compared to the [...] command.
- Supports logical operators: && (AND), || (OR)
- Supports pattern matching using * and ?
- Regular expressions: =~
- Example:
  - `if [[ $x -lt $y ]]; then echo "$x is less than $y"; fi`
  - `if [[ $x -lt $y && $string == "Hello, World!" ]]; then echo "$x is less than $y AND string matches" ; fi`
  - `if [[ $string =~ o{2} ]]; then echo "String contains two consecutive o’s"; fi`

Constructs

If Statement:

- -eq: Equal to
- -ne: Not equal to
- -lt: Less than
- -le: Less than or equal to
- -gt: Greater than
- -ge: Greater than or equal to
- =: Equal to
- !=: Not equal to
- =~: Regular expression match operator
- &&: Logical AND
- ||: Logical OR
- !: Logical NOT
- -e: File exists
- -f: File exists and is a regular file
- -d: File exists and is a directory
- -s: File is not empty
- -r: File is readable
- -w: File is writable
- -x: File is executable
- -z: String is empty
- -n: String is not empty

Example

```bash
#!/bin/bash
read -p "Enter the path to an existing file: " file
if [[ -e "$file" ]]; then
    if [[ -f "$file" ]]; then
        echo "$file is a regular file."
    fi
    if [[ -d "$file" ]]; then
        echo "$file is a directory."
    fi
    if [[ -s "$file" ]]; then
        echo "$file is empty."
    fi
else
    echo "$file does not exist."
    exit 1
fi
```

Constructs

`case` Statement:

```
case "$variable" in
    pattern1)
        # code to execute for pattern1
        ;;
    pattern2)
        # code to execute for pattern2
        ;;
    *)
        # code to execute for other patterns
        ;;
esac
```

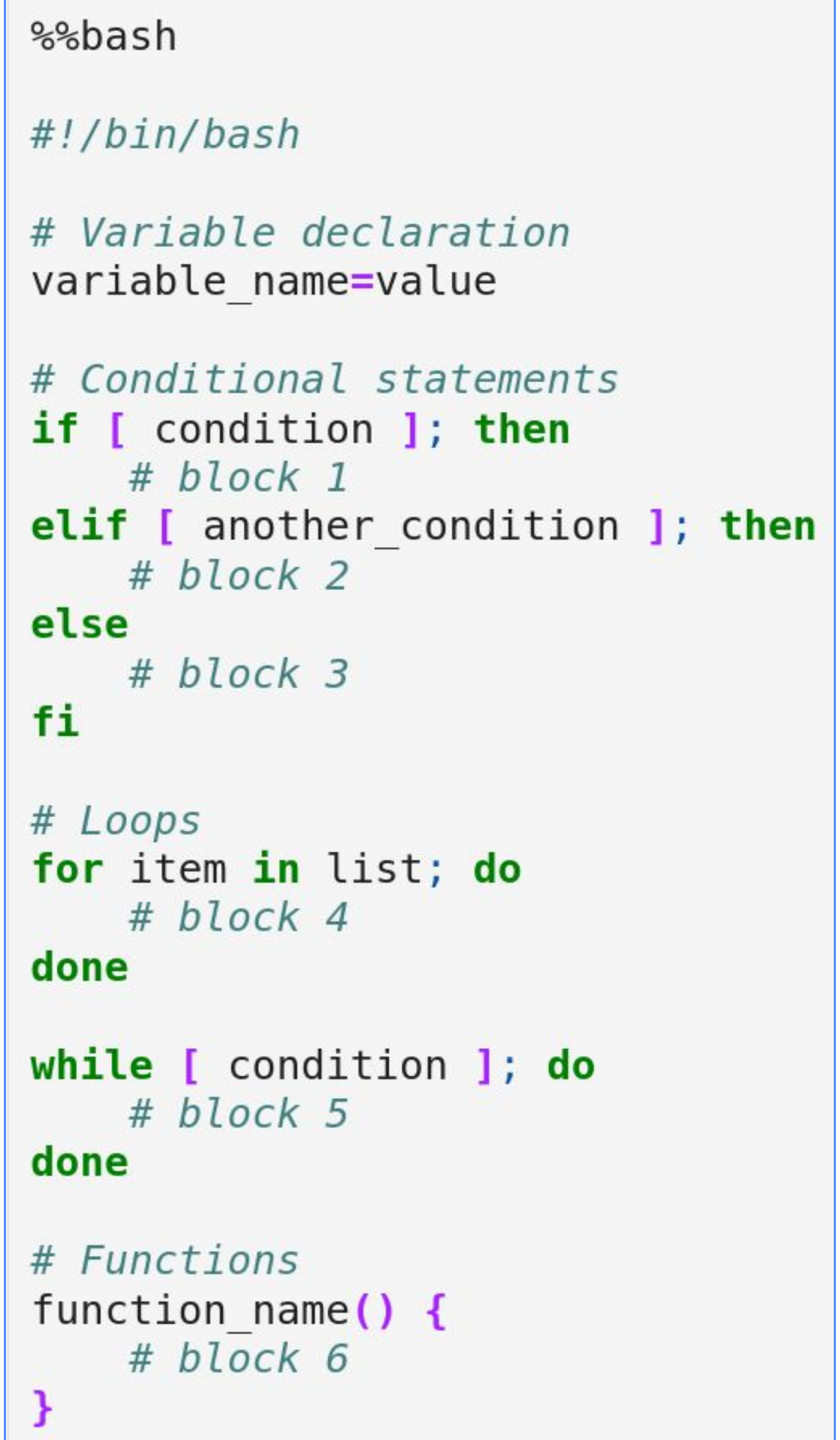
Constructs - example

```
#!/bin/bash


    read -p "Enter choice: " choice
    case $choice in
        1) ls -l ;;
        2) ps -f ;;
        3|4) date ;;
        5) who ;;
        *) break ;;
    esac
```

Constructs - example

```
#!/bin/bash

while true; do
    read -p "Enter choice: " choice
    case $choice in
        1) ls -l ;;
        2) ps -f ;;
        3|4) date ;;
        5) who ;;
        *) break ;;
    esac
done
```

Bash Script Basic Syntax



```
%%bash

#!/bin/bash

# Variable declaration
variable_name=value

# Conditional statements
if [ condition ]; then
    # block 1
elif [ another_condition ]; then
    # block 2
else
    # block 3
fi

# Loops
for item in list; do
    # block 4
done

while [ condition ]; do
    # block 5
done

# Functions
function_name() {
    # block 6
}
```

Examples

```
#!/bin/bash

for i in apple banana grape; do
    echo 'Fruit: $i'
done

for j in red white brown; do
    ecno "Color: $j"
done

echo "Loop finished"
```

OUTPUT:

Total: ??

Examples

```
#!/bin/bash
count=3
for ((i = 1; i <= count; i++))
do
    echo "Iteration: $i"
done

for i in $(seq 1 5); do
    echo "Number: $i"
    echo "End of loop"
done
echo "Loops finished"
```

OUTPUT:

Total: ??

Examples

```
#!/bin/bash


for num in {1..5
do
    echo "Number: $num"
done
```

OUTPUT:

Total: ??

Examples

```
#!/bin/bash

while read -r username rest; do
    echo $username
done </etc/passwd
```

OUTPUT:

Total: ??

Examples

```
#!/bin/bash


IFS=’:’
while read -r username rest; do
    echo $username
done </etc/passwd
```

OUTPUT:

Total: ??

Examples

```
#!/bin/bash
read -p "Enter your choice: " choice
case $choice in
    [[:upper:]])
        echo "Uppercase letter."
    ;;
    [[:lower:]])
        echo "Lowercase letter."
    ;;
    *)
        echo "Something else."
    ;;
esac
```

OUTPUT:

Total: ??

Examples

```
#!/bin/bash
x=10
y=5

if [$x==$y]; then echo "x is equal to y"; fi

if [ $x -gt $y && $x -lt 20 ]; then
    echo "x is between 5 and 20"
fi
```

OUTPUT:

Total: ??

Examples

```
#!/bin/bash


do
    read -p "Enter a number (0 to exit): " num
    echo "You entered: $num"
while [[ $num -ne 0 ]]
```

OUTPUT:

Total: ??

Examples

```
#!/bin/bash

while [[ $num -ne 0 ]]; do
    read -p "Enter a number (0 to exit): " num
    echo "You entered: $num"

done
```

OUTPUT:

Total: ??

Examples

```
#!/bin/bash


num=-1
while [[ $num -ne 0 ]]; do
    read -p "Enter a number (0 to exit): " num
    echo "You entered: $num"


done
```

OUTPUT:

Total: ??

Simple calculator in bash

Algorithm

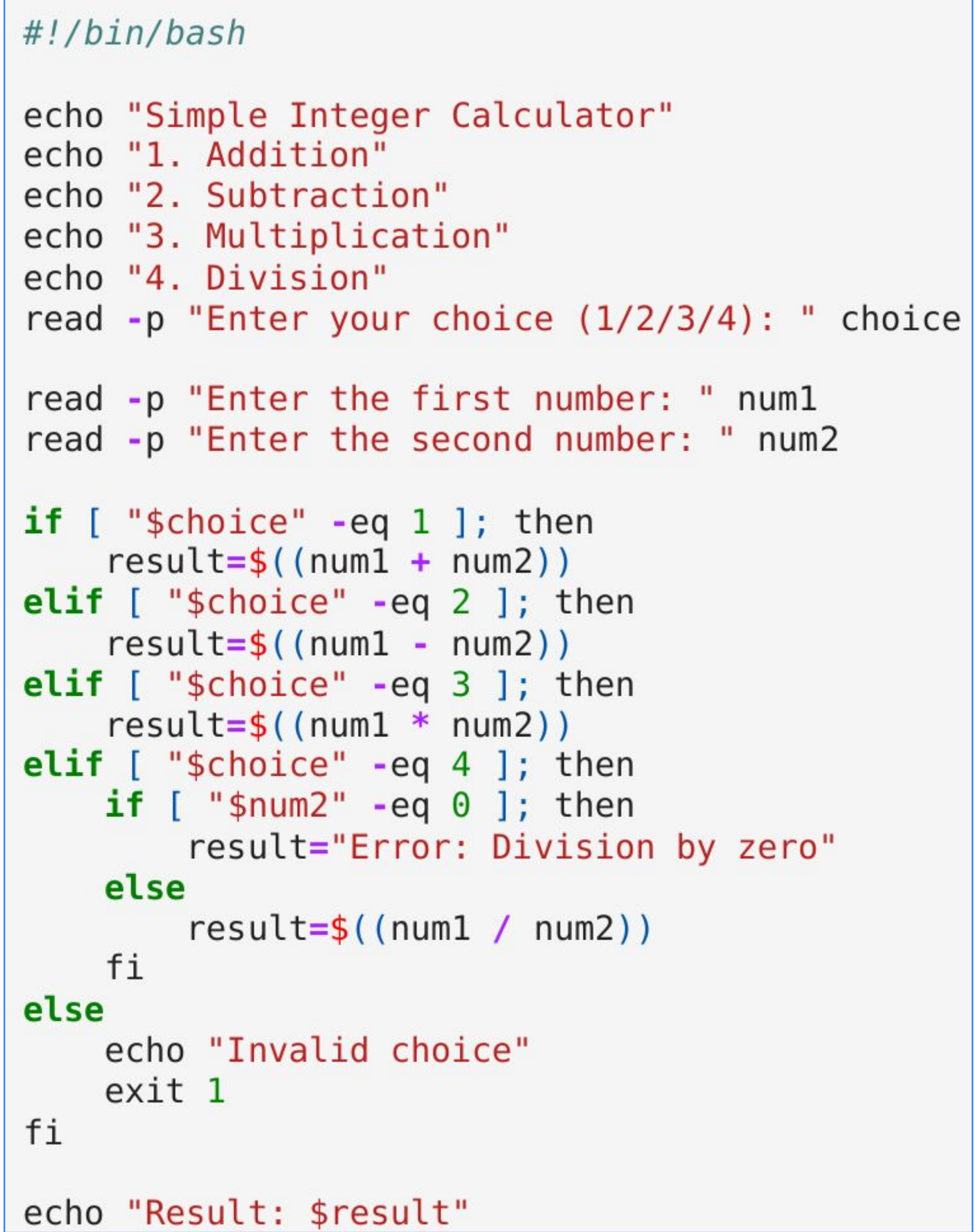
```
Start
|
|__  Read user's choice  (multiplication, division, etc)
|__  Read first number
|__  Read second number
|    |
|    |__ If choice is X (addition / subtraction / multiplication / division)
|    |__ Else (invalid choice)
|
|__ 7. Print result
|__ End
```

Example 1

Integers calculator

Important points to be noted:

- Usage of If statement
- Works for integer numbers



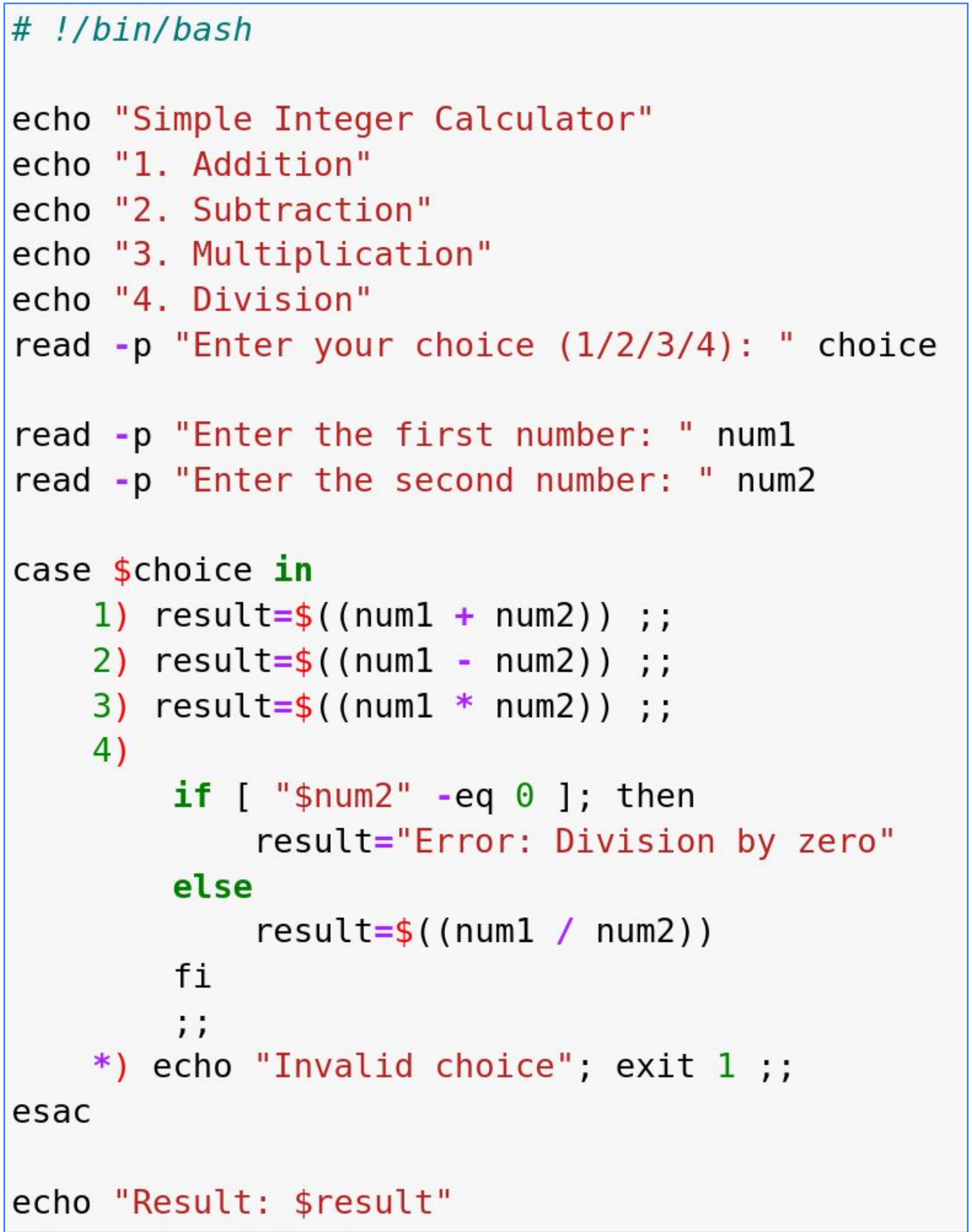
```bash
#!/bin/bash

echo "Simple Integer Calculator"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
read -p "Enter your choice (1/2/3/4): " choice

read -p "Enter the first number: " num1
read -p "Enter the second number: " num2

if [ "$choice" -eq 1 ]; then
    result=$((num1 + num2))
elif [ "$choice" -eq 2 ]; then
    result=$((num1 - num2))
elif [ "$choice" -eq 3 ]; then
    result=$((num1 * num2))
elif [ "$choice" -eq 4 ]; then
    if [ "$num2" -eq 0 ]; then
        result="Error: Division by zero"
    else
        result=$((num1 / num2))
    fi
else
    echo "Invalid choice"
    exit 1
fi

echo "Result: $result"
```

Example 2

Integers calculator

Important points to be noted:

- Usage of case statement
- Works for integer numbers



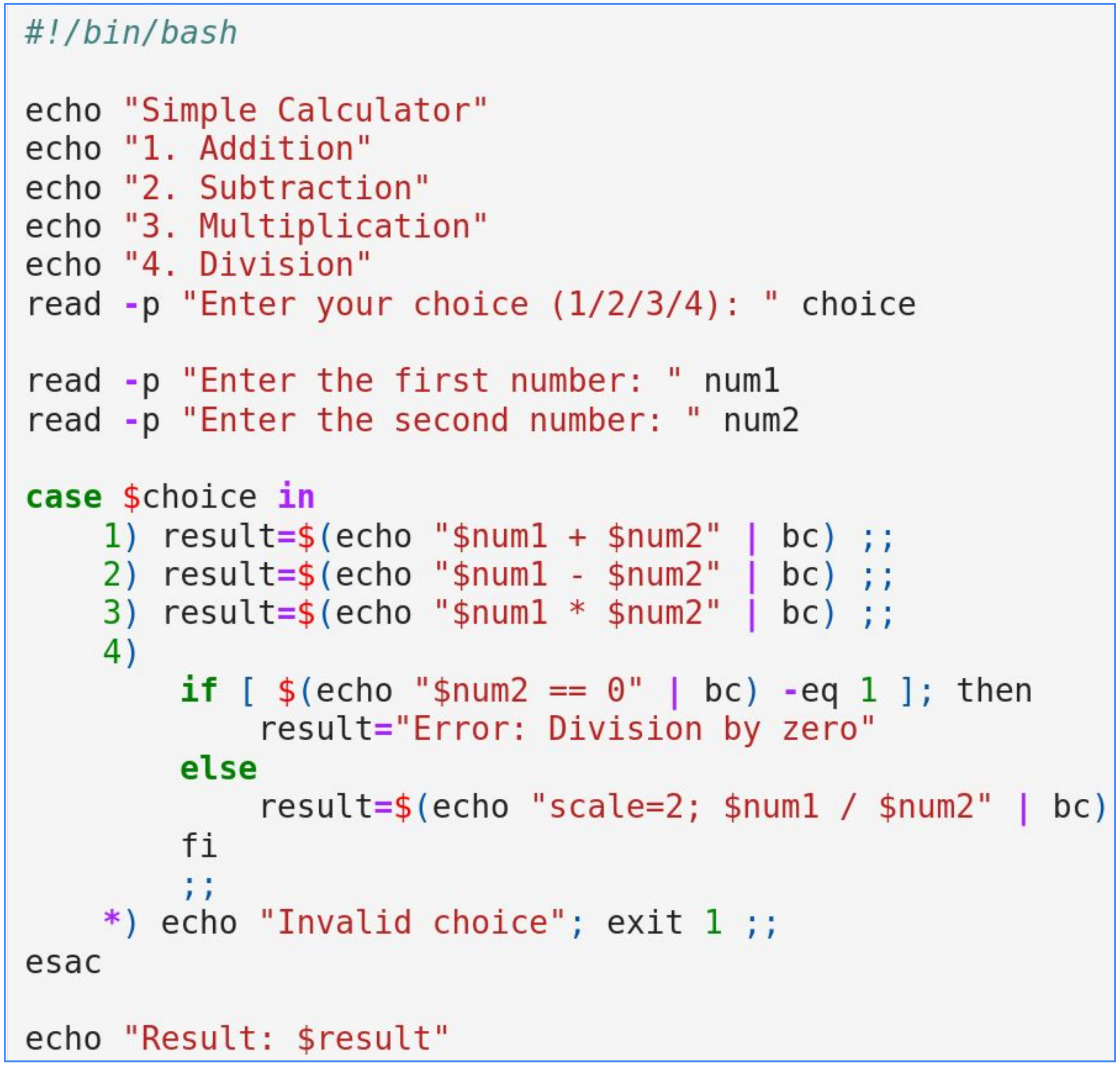
```
# !/bin/bash

echo "Simple Integer Calculator"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
read -p "Enter your choice (1/2/3/4): " choice

read -p "Enter the first number: " num1
read -p "Enter the second number: " num2

case $choice in
    1) result=$((num1 + num2)) ;;
    2) result=$((num1 - num2)) ;;
    3) result=$((num1 * num2)) ;;
    4)
        if [ "$num2" -eq 0 ]; then
            result="Error: Division by zero"
        else
            result=$((num1 / num2))
        fi
        ;;
    *) echo "Invalid choice"; exit 1 ;;
esac

echo "Result: $result"
```

Example 3

Real numbers calculator

Important points to be noted:

- Usage of case statement
- Works for both integers and real numbers



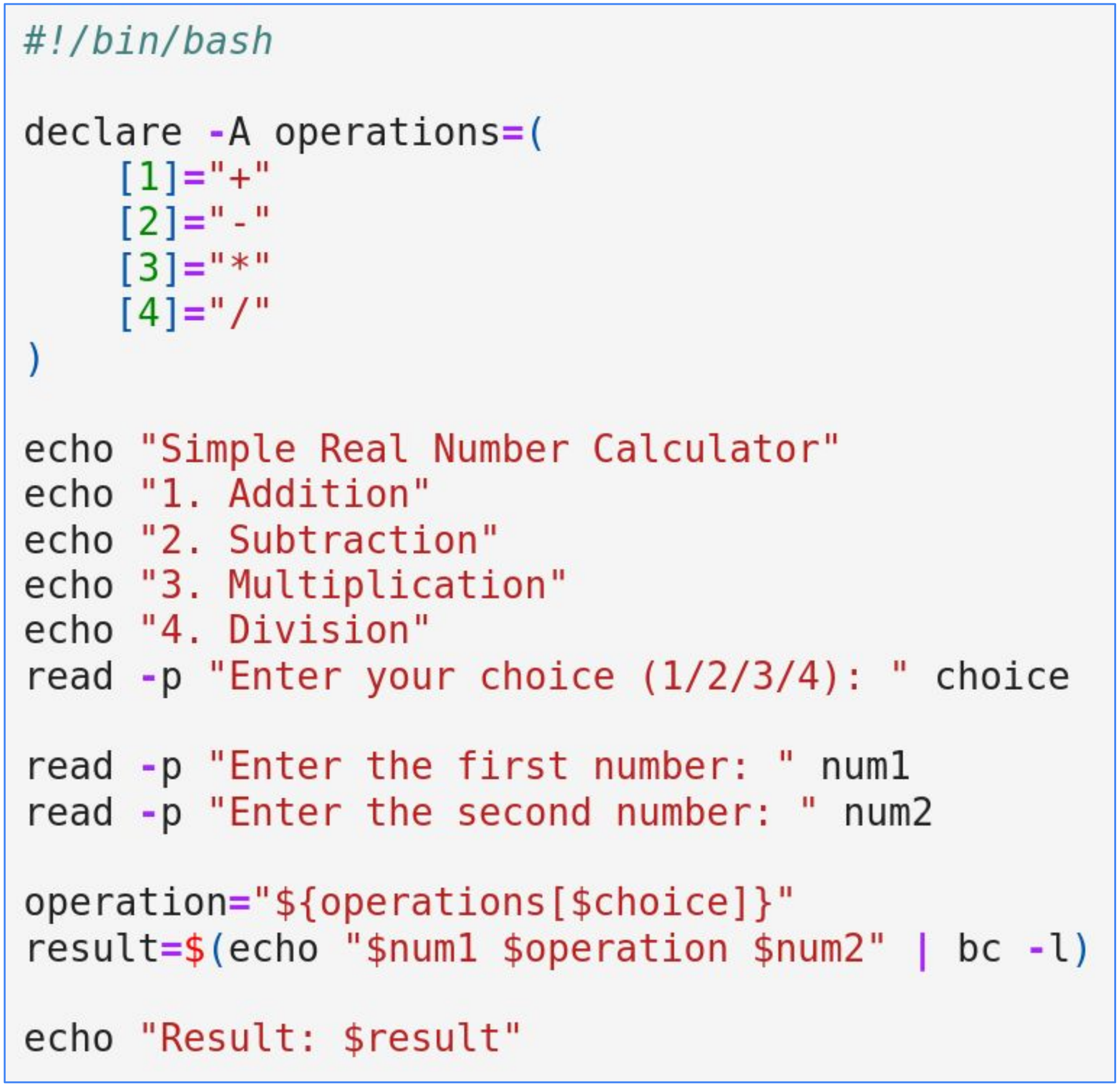
```bash
#!/bin/bash

echo "Simple Calculator"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
read -p "Enter your choice (1/2/3/4): " choice

read -p "Enter the first number: " num1
read -p "Enter the second number: " num2

case $choice in
    1) result=$(echo "$num1 + $num2" | bc) ;;
    2) result=$(echo "$num1 - $num2" | bc) ;;
    3) result=$(echo "$num1 * $num2" | bc) ;;
    4)
        if [ $(echo "$num2 == 0" | bc) -eq 1 ]; then
            result="Error: Division by zero"
        else
            result=$(echo "scale=2; $num1 / $num2" | bc)
        fi
        ;;
    *) echo "Invalid choice"; exit 1 ;;
esac

echo "Result: $result"
```

Example 4

Real numbers calculator

Important points to be noted:

- Simple code



```bash
#!/bin/bash

declare -A operations=(
    [1]="+"
    [2]="-"
    [3]="*"
    [4]="/"
)

echo "Simple Real Number Calculator"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
read -p "Enter your choice (1/2/3/4): " choice

read -p "Enter the first number: " num1
read -p "Enter the second number: " num2

operation="${operations[$choice]}"
result=$(echo "$num1 $operation $num2" | bc -l)

echo "Result: $result"
```

arrays

declare command

- -i: Declare a variable as an integer.
- -a: Declare a variable as an indexed array.
- -A: Declare a variable as an associative array.
- -p: Display attributes and options of variables.
- -x: Export a variable for child processes.

declare command - examples

```
#!/bin/bash

declare -i age=25

declare -i quantity=10

result=$((age + quantity))

echo "Total: $result"
```

OUTPUT:

```
Total: 35
```

declare command - examples

```bash
#!/bin/bash
declare -a colors=("red" "green" "blue")
echo "First color: ${colors[0]}"
echo "Second color: ${colors[1]}"
```

OUTPUT:

```
First color: red
Second color: green
```

```bash
#!/bin/bash
declare -a numbers=("2.5" "3.14" "1.618")
sum=$(echo "${numbers[0]} + ${numbers[1]} + ${numbers[2]}" | bc -l)
echo "Sum of numbers: $sum"
echo "Display all variables: ${numbers[@]}"
declare -p numbers
```

OUTPUT:

```
Sum of numbers: 7.258
Display all variables: 2.5 3.14 1.618
declare -a numbers=([0]="2.5" [1]="3.14" [2]="1.618")
```

declare command - examples

```
#!/bin/bash
declare -A fruits
fruits["apple"]="red"
fruits["banana"]="yellow"
fruits["grape"]="purple"

echo "Color of apple: ${fruits["apple"]}"
echo "Color of banana: ${fruits["banana"]}"
declare -p fruits
```

OUTPUT:

```
First color: red
Second color: green
```

OUTPUT:

```
Sum of numbers: 7.258
Display all variables: 2.5 3.14 1.618
declare -a numbers=([0]="2.5" [1]="3.14" [2]="1.618")
```

declare command - examples

```
#!/bin/bash
declare -x fruits

fruits=”apple”

./another_script.sh
```

```
#!/bin/bash
echo $fruits
```

OUTPUT:

```
apple
```

declare command - examples

```
#!/bin/bash
set -a

fruits="apple"
colors="red"
./another_script.sh
```

```
#!/bin/bash
echo $fruits $red
```

OUTPUT:

```
apple red
```

declare command - examples

```
#!/bin/bash
declare -i age=25
declare -i quantity=2.5
result=$((age + quantity))
echo "Total: $res1.1ult"
```

OUTPUT:

Total: ??

```
#!/bin/bash

declare -i age="25abc"
echo "Age: $age"
```

declare command - examples

```
#!/bin/bash

declare -a colors=("red", "green", "blue")

echo "First color: ${colors[0]}"
```

OUTPUT:

Total: ??

declare command - examples

```
#!/bin/bash

declare -A fruits
fruits["apple"]="red"
fruits["banana"]="yellow"
fruits["grape"]="purple"

echo "Color of apple: ${fruits['apple']}"
```

OUTPUT:

Total: ??

declare command - examples

```
#!/bin/bash


temperatures_celsius=(20 25 30 15 10 35 22 18 28 32)


echo "Celsius    Fahrenheit"
echo "---------------------"
for celsius in "${temperatures_celsius[@]}"; do
    fahrenheit=$(echo "scale=2; ($celsius * 9/5) + 32" | bc)
    echo "$celsius°C" "$fahrenheit°F"
done
```

OUTPUT:

Total: ??

declare command - examples

```
#!/bin/bash

temperatures_celsius=(20 25 30 15 10 35 22 18 28 32)

echo "Celsius    Fahrenheit"
echo "---------------------"
for celsius in "${temperatures_celsius[@]}"; do
    fahrenheit=$(echo "($celsius * 9/5) + 32" | bc)
    echo "$celsius°C" "$fahrenheit°F"
done
```

OUTPUT:

Total: ??

Working with practical examples

Retrieving a value from specified file and printing

You are provided with a directory structure where the last subdirectory contains a file named 'out'. Write a bash script to extract the final energy for each case. The output should be printed in the following format:

Output should be printed in the following format:

```
# Header
Dir1
Sub-dir1    ener_val1
Sub-dir2    ener_val2
...
Dir2
...
```

Download the input files from here:

https://www.dropbox.com/sh/d18w4jmye9gmayo/AAA74Aiz_fC9sXj9sshWZDT6a?dl=0

script

```bash
#!/bin/bash
# Directories structure with 'out' file in the last subdirectory
dir=$PWD/base
# Pattern to match
x='Geometry converged'
# Loop through each directory
for i in `ls -d $dir/*/`; do
  pushd $i > /dev/null
  dir_name=$(basename $i)
  ener_found=0
  # Loop through subdirectories within each directory
  echo "$dir_name"
  for j in `ls -d [0-9]*/`; do
    grep "$x" $j/out > /dev/null
```

Script contd. on next page

script

```
    if [ $? -eq 0 ]; then
      ener=$(grep 'Total Energy' $j/out | tail -n 1 | tr -s ' ' | cut -d ' ' -f 3)
      echo "$j    $ener"
      ener_found=1
    fi
  done
  # If no energy was found, print a message
  if [ $ener_found -eq 0 ]; then
    echo "$dir_name"
    echo "No energy found"
  fi
  popd > /dev/null
done
```

Automated daily file backup script

Create a script that performs file backups daily after 3:00 am. The script must exclusively back up files that are either new or have been modified. The backup location should be set to /home1/user/backup, while the source files are located at /home/user/work.

script

```
#!/bin/bash
source_dir="/home/user/work"
backup_dir="/home1/user/backup"
log_file="/var/log/backup.log"
current_time=$(date +"%Y-%m-%d %H:%M:%S")
echo "Backup started at $current_time" >> "$log_file"

# Check if the backup directory exists, if not, create it
if [ ! -d "$backup_dir" ]; then
    mkdir -p "$backup_dir"
fi

# Sync the source directory to the backup directory
rsync -av --update --delete "$source_dir/" "$backup_dir/" >> "$log_file" 2>&1

echo "Backup completed at $(date +'%Y-%m-%d %H:%M:%S')" >> "$log_file"
```

Hard disk space monitoring and threshold alert script

Write a script to keep track of the available space on your hard drives. The script should issue a warning when the available space falls below a specified threshold value.

code

```
#!/bin/bash
# Set the threshold value in percentage
threshold_percentage=10

# Loop through mounted filesystems and check disk space
while read -r fs size used avail percentage mount; do
    if [[ "$fs" =~ ^/dev/ ]]; then
        available_percentage=$(echo "$percentage" | tr -d '%')
        disk=$(echo "$fs" | tr -d '/')

        if [ "$available_percentage" -lt "$threshold_percentage" ]; then
            echo "Warning: Available space on $disk is less than  threshold_percentage%."
        fi
    fi
done < <(df -h)
```

Recursive counting of files and directories

Write a script that reports the total number of files and directories. The counting should be performed recursively.

code

```
#!/bin/bash

if [ $# -ne 1 ]; then echo "Usage: $0 <directory_path>"; exit 1; fi
directory="$1"
file_count=0; dir_count=0

# Use 'find' to loop through items in the directory and subdirectories
while IFS= read -r item; do
    if [ -f "$item" ]; then
        ((file_count++))
    elif [ -d "$item" ]; then
        ((dir_count++))
    fi
done < <(find "$directory")

echo "Total number of files: $file_count"
echo "Total number of directories: $dir_count"
```

Start, stop or monitor processes on remote machines

Write a script that allows you to manage processes on remote machines using SSH, enabling you to start, stop, or monitor processes. It should do the following

1. Function to start a process remotely
2. Function to stop a process remotely
3. Function to monitor a process remotely

script

```
#!/bin/bash
u="your_remote_user"
h="remote_machine_address"

s(){ p=$1;ssh $u@$h "nohup $p &";echo "Started $p on $h";}
t(){ p=$1;ssh $u@$h "pkill -f $p";echo "Stopped $p on $h";}
m(){ p=$1;ssh $u@$h "pgrep -fl $p";}

while true; do
  clear;echo "Remote Process Management";echo "1. Start a process";echo "2. Stop a process";echo "3. Monitor a process";echo "4. Exit";read -p "Enter your choice: " c

  case $c in
    1) read -p "Enter the name of the process to start: " p; s "$p";;
    2) read -p "Enter the name of the process to stop: " p; t "$p";;
    3) read -p "Enter the name of the process to monitor: " p; m "$p";;
    4) echo "Exiting...";exit 0;;
    *) echo "Invalid choice. Please select a valid option.";;
  esac;read -p "Press Enter to continue...";done
```

Job queue

Write a bash script that handles a job queue, allowing the execution of up to 4 jobs concurrently. Upon the completion of any job, the script should initiate the next job in the queue.

code

```
#!/bin/bash

ncores=4

jobs=("Job1" "Job2" "Job3" "Job4" "Job5" "Job6" "Job7" "Job8" "Job9" "Job10")

run_job() {
    sleep_time=$((1 + $RANDOM % 5))
    echo "Running $1"; sleep "$sleep_time"; echo "Completed $1"
}

total_jobs="${#jobs[@]}"
submitted_jobs=0
running_jobs=0
index=0
```

Script contd. on next page

code

```
while [ "$submitted_jobs" -lt "$total_jobs" ]; do
    job="${jobs[$index]}"
    run_job "$job" &
    ((running_jobs++))
    ((index++))

    while [ "$running_jobs" -ge "$ncores" ]; do
      wait -n
      running_jobs=$((running_jobs - 1))
    done
    submitted_jobs=$((submitted_jobs + 1))
done

wait
echo "All jobs submitted"
```