PHY473/473A- Computational Physics

Lecture-2
Introduction to Unix and shell script

Gopal Hazra, IIT Kanpur

Source: Learning the bash Shell - Unix Shell Programming by Cameron Newham and web

Unix File System

Navigating the Shell :

You can access the shell by opening a terminal emulator

Paths and pwd:

Path of a directory can be either absolute paths or relative paths

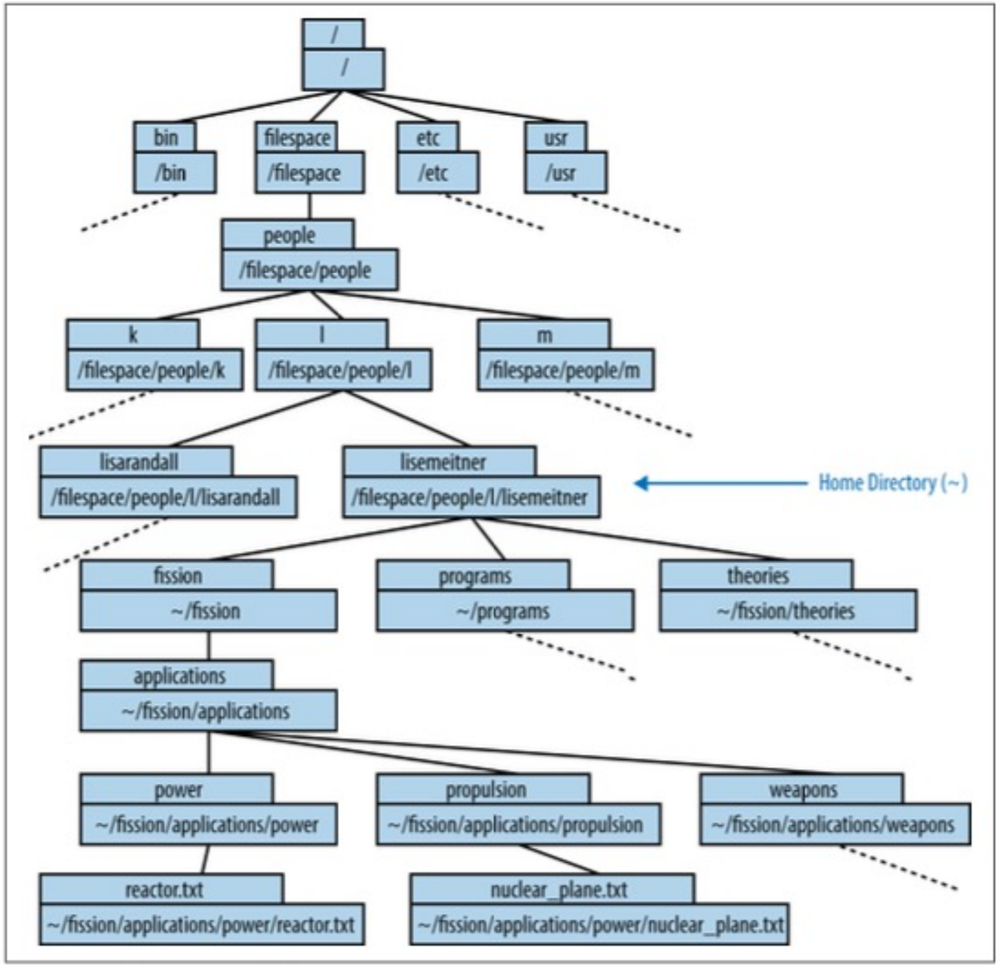


Figure 1-2. An example directory tree

A few important commands to remember

Listing the Contents (ls- command)

Creating Directories (mkdir)

Changing Directory (cd)

Removing files and directory (rm, rm –r)

File Inspection (head and tail) : sometimes we need to see what is the beginning and end of a file. This is usually useful for a very heavy file

Sometimes the command “more” also has been used.

Copy and Move, text editor

How to use Manual from terminals!

The ‘man’ command in Linux is used **to display the user manual of any command that we can run on the terminal**

```
$man [OPTION]... [COMMAND NAME]...
```

```
$ man printf
```

```
$ man 2 intro
```

```
$ man -f ls
```

-f option: One may not be able to remember the sections in which a command is present. So this option gives the section in which the given command is present.

Finding the Right Hammer (apropos)

The bash shell has so many built-in programs, practically no one has all of their names memorized.

```
~ $ apropos "text editor"
```

Example 1: Suppose you don’t know how to compress a file then you could type the following command in terminal and it will show all the related command and its short description or functionality.

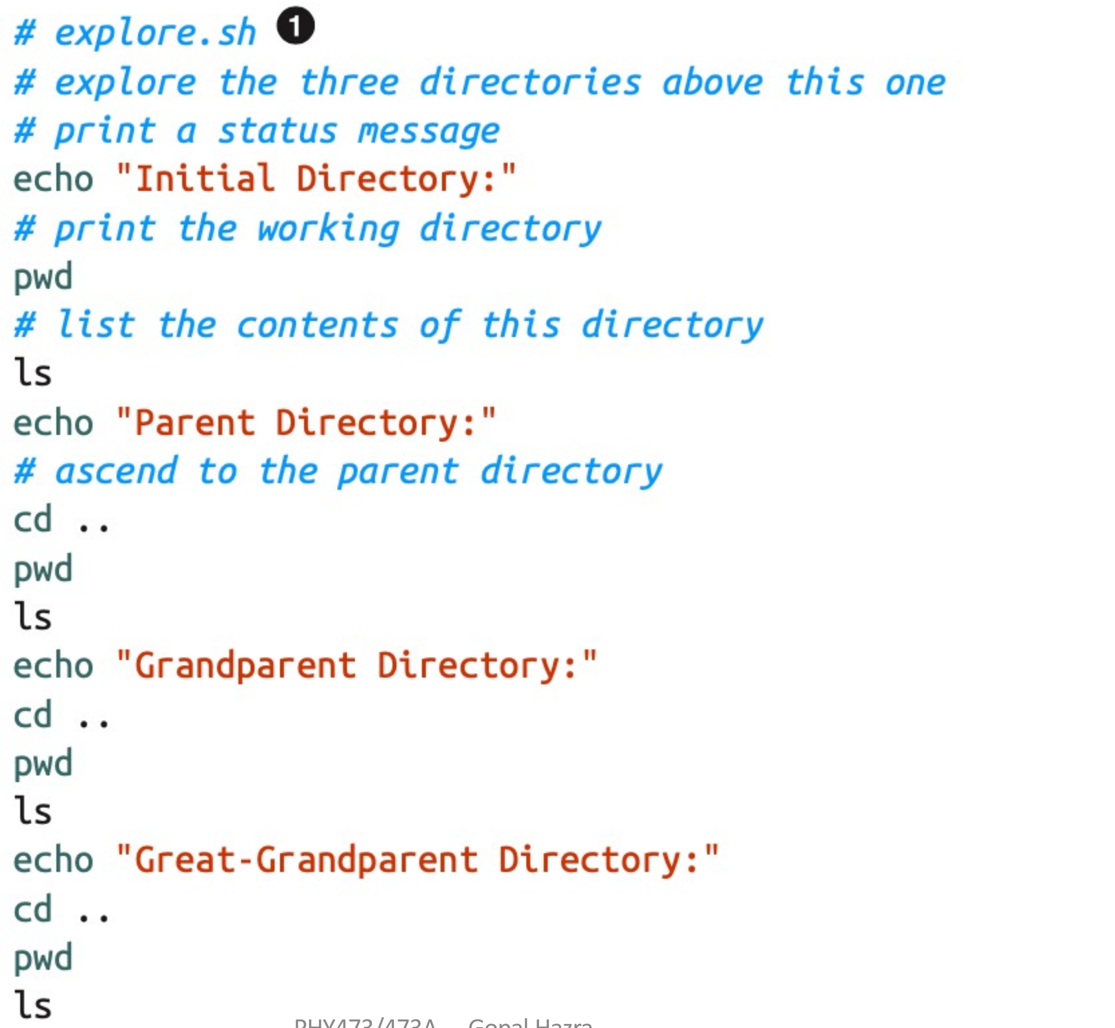
```
~ $ apropos ”compress"
```

- ✓ Now we have learned h mand line

- ✓ Repeating processes on .sh extension)

- ✓ This type of file, like a p are valid in the terminal are valid in a b

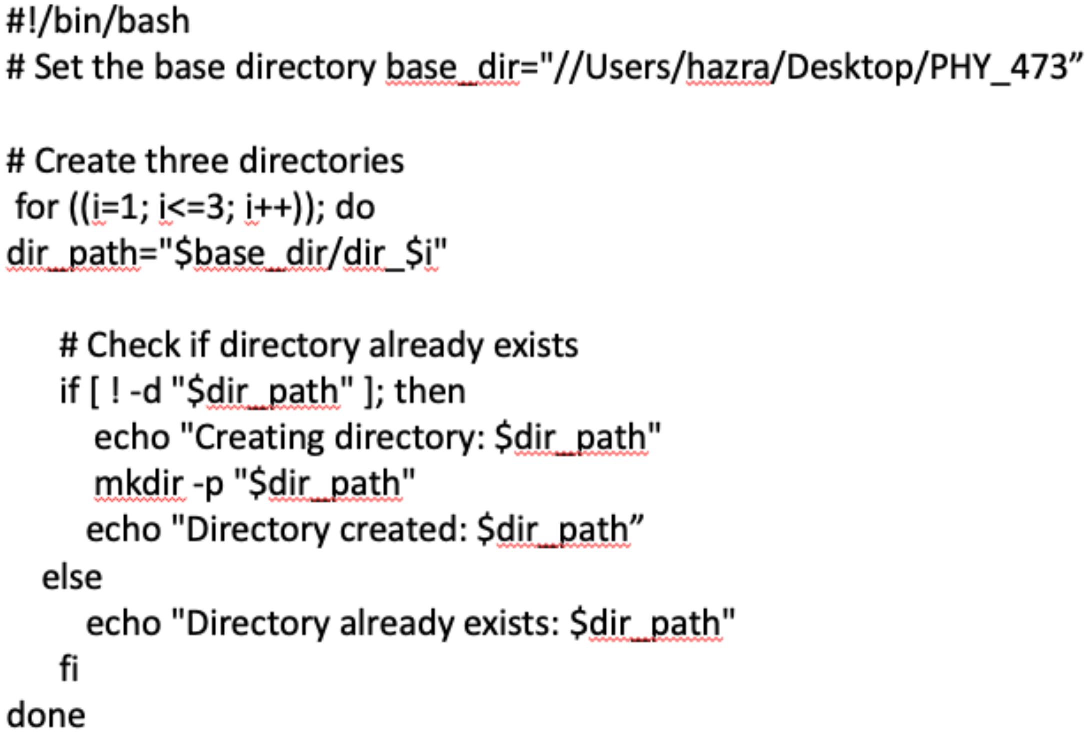
An example task: Enter in ory names!



```
# explore.sh ❶
# explore the three directories above this one
# print a status message
echo "Initial Directory:"
# print the working directory
pwd
# list the contents of this directory
ls
echo "Parent Directory:"
# ascend to the parent directory
cd ..
pwd
ls
echo "Grandparent Directory:"
cd ..
pwd
ls
echo "Great-Grandparent Directory:"
cd ..
pwd
ls
```

Creating Multiple directories?

Single directory → $ mkdir PHY473 → very simple But for 1000 directories you need to write a script



```
#!/bin/bash
# Set the base directory base_dir="//Users/hazra/Desktop/PHY_473”

# Create three directories
 for ((i=1; i<=3; i++)); do
dir_path="$base_dir/dir_$i"

    # Check if directory already exists
    if [ ! -d "$dir_path" ]; then
       echo "Creating directory: $dir_path"
       mkdir -p "$dir_path"
      echo "Directory created: $dir_path”
   else
      echo "Directory already exists: $dir_path"
    fi
done
```

Details about Shell Script – Bash Shell!

- ✓ The shell is a command interpreter.
- ✓ It is the layer between the operating system kernel and the user.

There are many shells one can use, Most widely used one is Called BASH shell (Bourne Again Shell)

The shell is a programming language that is run by the terminal. Like other programming languages, the shell:

-- Can collect many operations into single entities
-- Requires input
-- Produces output
-- Has variables and state
-- Uses irritating syntax
-- Uses special characters

Some Special characters used in shell scripts

#:Comments
~:home directory

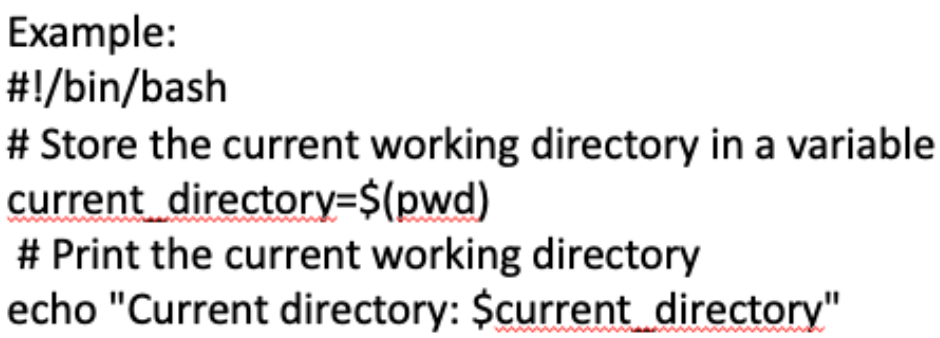
Find out which Shell you are using

Go to your terminal:

```
% Echo $0
```

How to Execute a shell script

- The first line must be “#!/bin/bash”.
  - setup the shell path
- **chmod u+x scriptname** (gives only the script owner execute permission)
- ./scripname



Example:

```
#!/bin/bash
# Store the current working directory in a variable
current_directory=$(pwd)
 # Print the current working directory
echo "Current directory: $current_directory"
```

Internal Built function and commands

- getopts:
  - parses command line arguments passed to the script.
- exit:
  - Unconditionally terminates a script
- set:
  - changes the value of internal script variables. e.g., set MY_VARIABLE="Hello"
- read:
  - **Reads" the value of a variable from stdin**
  - **also "read" its variable value from a file redirected to stdin**
- wait:
  - Stop script execution until all jobs running in background have terminated
- wc – word count, number counts

Internal Built function and commands …

- grep:
  - grep pattern file
    - search the files file, etc. for occurrences of *pattern*
- expr:
    - evaluates the arguments according to the operation given
      - **y=`expr $y + 1`** (same as **y=$(($y+1))**

```
# Multiplication
 result=$(expr 6 \* 2)
echo "6 * 2 = $result"
```

Example of the usage of ‘grep’

You can use grep to find out some word, some expression from large number of coding files

Search for a Pattern in a File: grep "pattern" filename

Search Recursively in Directories: grep -r "pattern" directory

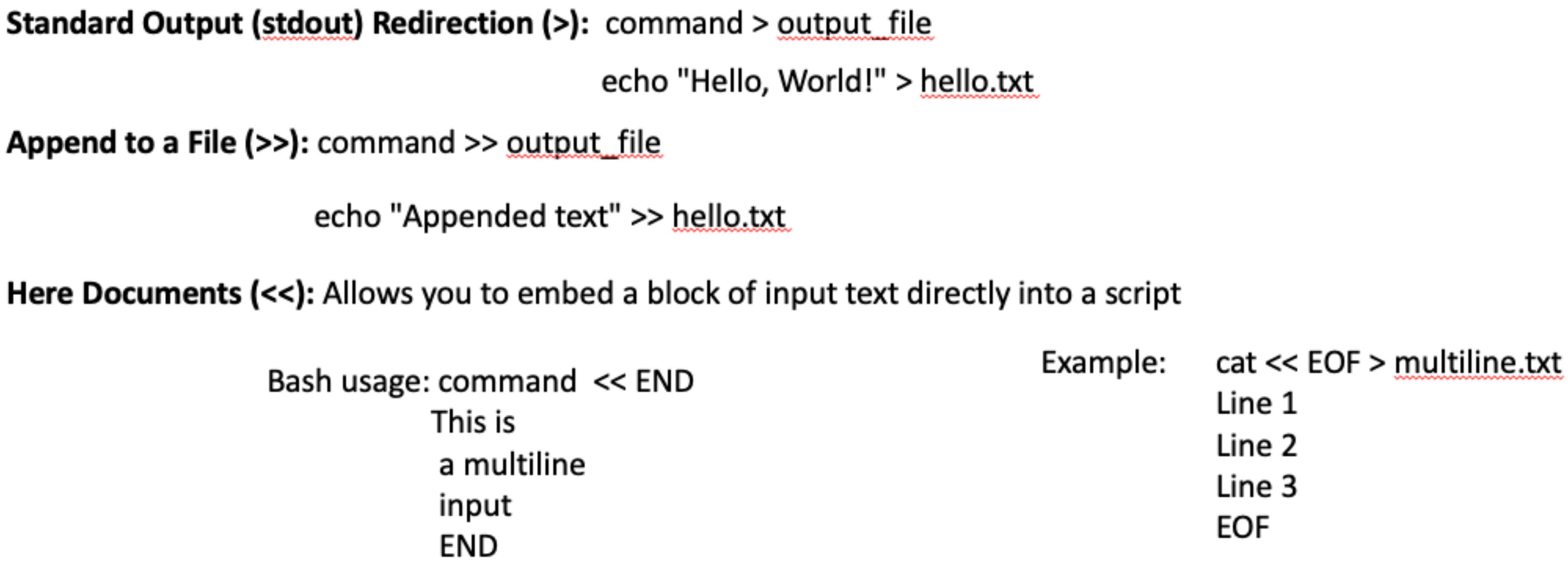
Display Line Numbers: grep -n "pattern" filename

Case-Insensitive Search: grep -i ”search_words" filename

Redirection of input output

- >: Redirect stdout to a file, Creates the file if not present, otherwise overwrites it
- < : Accept input from a file.
- >>: Creates the file if not present, otherwise appends to it.
- <<:
  - Forces the input to a command to be the shell's input, which until there is a line that contains only *label*.
  - cat >> mshfile << .
- |:pipe, similar to ">",

Redirection input output –Example



Standard Output (stdout) Redirection (>): command > output_file

```
echo "Hello, World!" > hello.txt
```

Append to a File (>>): command >> output_file

```
echo "Appended text" >> hello.txt
```

Here Documents (<<): Allows you to embed a block of input text directly into a script

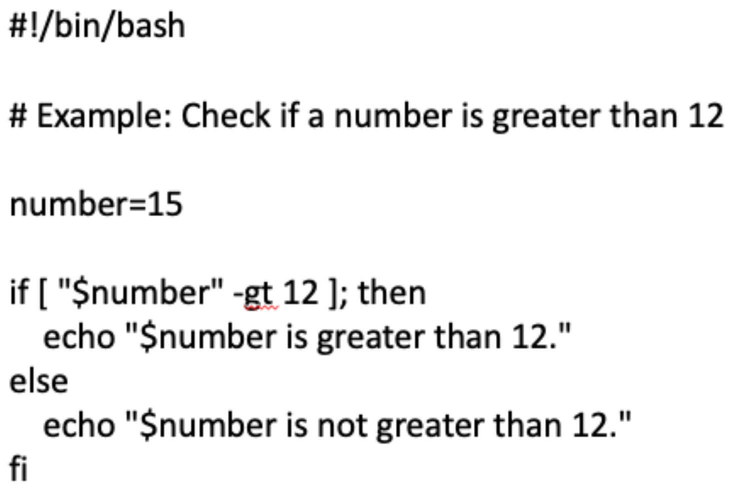
```
Bash usage: command  << END
             This is
              a multiline
              input
              END
```

```
Example:    cat << EOF > multiline.txt
            Line 1
            Line 2
            Line 3
            EOF
```

If condition in BASH

```
if [ condition ] then
  command1
elif   # Same as else if
 then
          command1
 else
  default-command
fi
```

Example:



```
#!/bin/bash

# Example: Check if a number is greater than 12

number=15

if [ "$number" -gt 12 ]; then
   echo "$number is greater than 12."
else
   echo "$number is not greater than 12."
fi
```

- -eq: equal to
- -ne: not equal to
- -lt: less than.

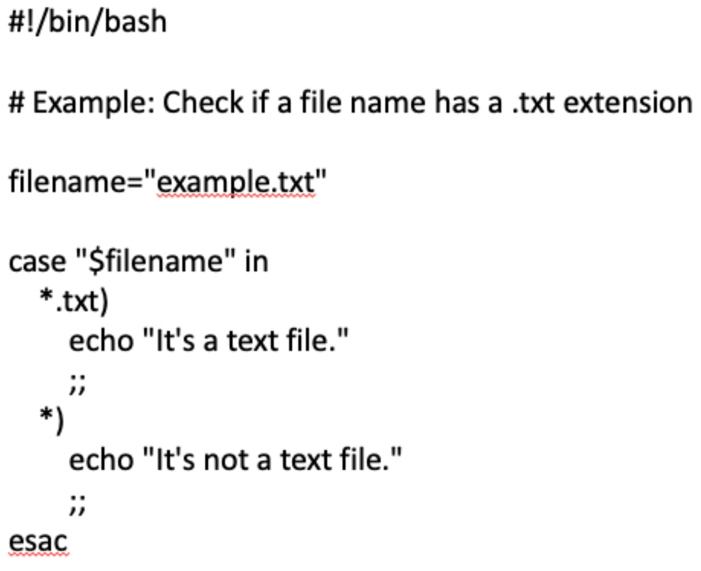
-

- -le: less than or equal to
- -gt: greater than
- -ge: greater than or equal to

case

```
x=5
 case $x in
  0) echo "Value of x is 0."
     ;
  5) echo "Value of x is 5."
     ;
  9) echo "Value of x is 9."
     ;
  *) echo "Unrecognized value."
esac
 done
```

Example:



```
#!/bin/bash

# Example: Check if a file name has a .txt extension

filename="example.txt"

case "$filename" in
   *.txt)
      echo "It's a text file."
      ;;
   *)
      echo "It's not a text file."
      ;;
esac
```

Loops in BASH shell (for, while and until)

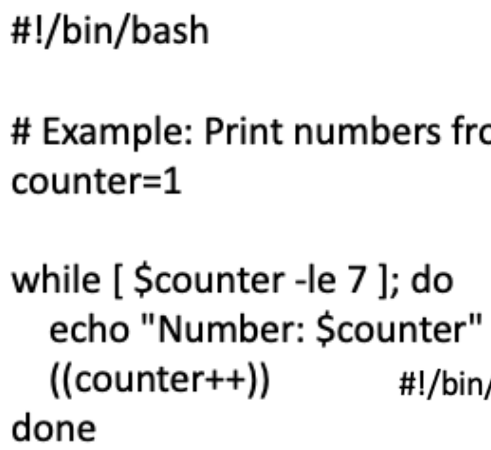
- for [arg] in [list];

  do

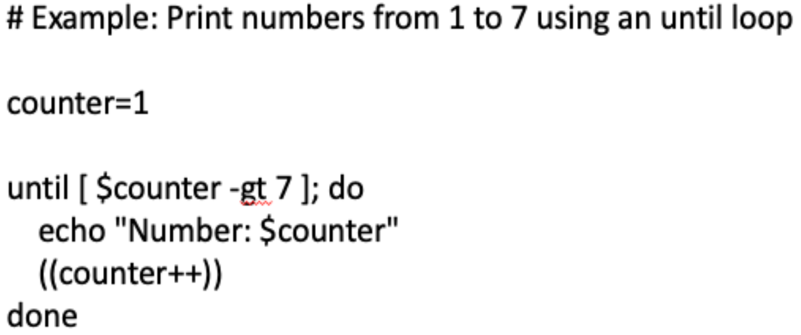
  command

  done
- while [condition];
  do
  □command...
  done

Example: For loop is easy, so we show only while and until



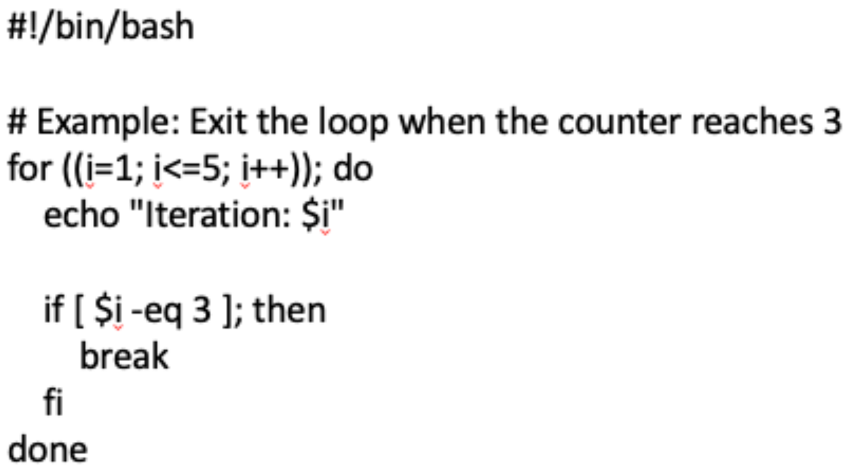
```
#!/bin/bash

# Example: Print numbers from 1 to 7 using a while loop
counter=1

while [ $counter -le 7 ]; do
   echo "Number: $counter"
   ((counter++))
done
```



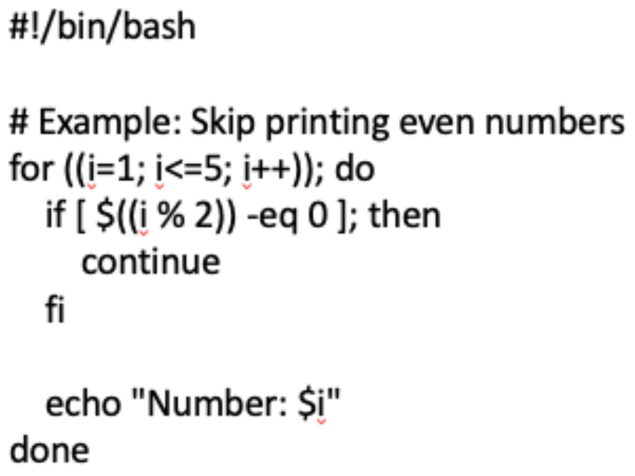
```
#!/bin/bash

# Example: Print numbers from 1 to 7 using an until loop

counter=1

until [ $counter -gt 7 ]; do
   echo "Number: $counter"
   ((counter++))
done
```

Loops in BASH shell (cont.)

- break, continue
  - **break** command terminates the loop
  - **continue** causes a jump to the next iteration of the loop



```
#!/bin/bash

# Example: Exit the loop when the counter reaches 3
for ((i=1; i<=5; i++)); do
   echo "Iteration: $i"

   if [ $i -eq 3 ]; then
      break
   fi
done
```



```
#!/bin/bash

# Example: Skip printing even numbers
for ((i=1; i<=5; i++)); do
   if [ $((i % 2)) -eq 0 ]; then
      continue
   fi

   echo "Number: $i"
done
```

Introduction to Variables in BASH

- $: variable substitution
  - If **variable1** is the name of a variable, then **$variable1** is a reference to its *value*.

Arithmetic Substitution:

```
#!/bin/bash

# Perform arithmetic operations
a=5
b=3

result=$((a + b))

echo "Result: $result"
```

Length of a String:

```
#!/bin/bash

# Get the length of a string
word="Bash"
length=${#word}

echo "Length of the word '$word': $length"
```

Pattern Matching –very important for file manipulations

- ${variable#pattern} -> removes the shortest match of pattern from the beginning (front) of the value stored in the variable

```
#!/bin/bash

# Example: Removing a pattern from the beginning of a string

original_string="prefix_hello_world"

# Remove the shortest match of "prefix_" from the beginning
of the string
result=${original_string#prefix_}

echo "Original String: $original_string"
echo "Result: $result"
```

Examples of Pattern Matching

${variable##pattern}
${variable%pattern}
${variable%%pattern}

x=/home/cam/book/long.file.name

echo ${x#/*/} → /cam/book/long.file.name

echo ${x##/*/}. → /book/long.file.name

echo ${x%.*}. → /home/cam/book/long.file

echo ${x%%.*} → /home/cam/book/long

Aliases are very important for scientific computing

- Avoiding typing a long command sequence
- Ex: alias lm="ls -l | more"

Array in BASH

- Declare:
  - declare -a array_name
- To dereference (find the contents of) an array variable, use *curly bracket* notation, that is, **${ array[xx]}**
- refers to *all* the elements of the array
  - *${array_name[@]}* or *${array_name[*]}*
- get a count of the number of elements in an array
  - *${#array_name[@]}* or *${#array_name[*]}*

Array Example in BASH

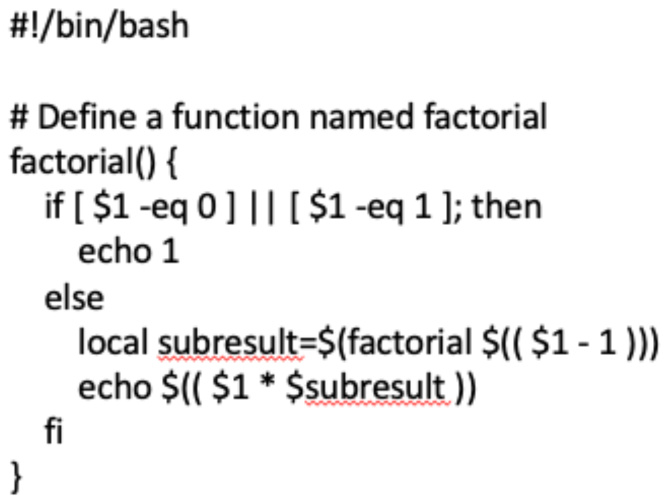
```
#!/bin/bash

# Example: Slicing an array
fruits=("Apple" "Banana" "Orange" "Grapes")

# Slice the array from index 1 to 2
sliced_array=("${fruits[@]:1:2}")

echo "Sliced array: ${sliced_array[@]}"
```

Defining Functions in BASH

- Type
  - *function function-name* {
    command...
    }
  - *function-name* () {
    command...
    }
- Local variables in function:
  - Declare: local var_name
- functions may have arguments
  - function-name $arg1 $arg2

```
#!/bin/bash
# Define a function named greet

greet() {
 {echo "Hello, $1!"
}

# Call the function with a parameter

 greet ”Jesus"

Result: Hello, Jesus
```

Defining Functions in BASH



```
#!/bin/bash

# Define a function named factorial
factorial() {
   if [ $1 -eq 0 ] || [ $1 -eq 1 ]; then
      echo 1
   else
      local subresult=$(factorial $(( $1 - 1 )))
      echo $(( $1 * $subresult ))
   fi
}
```

```
# Prompt the user for input
read -p "Enter a number to calculate its factorial: " num

# Call the factorial function with user input
result=$(factorial $num)

# Display the result
echo "The factorial of $num is: $result"
```

- The factorial function calculates the factorial of a number recursively. If the input is 0 or 1, it returns 1. Otherwise, it calculates the factorial by calling itself with a decremented argument.
- The script prompts the user to enter a number, calls the factorial function with the user's input, and then displays the result.

When you run this script, it will prompt you to enter a number, calculate its factorial, and then print the result.

Positional Parameters

- $1, $2, $3 …..
- $0 is the name of the script.
- The variable $# holds the number of positional parameter.

```
#!/bin/bash

# Example script: positional_parameters.sh

echo "The script name is: $0"
echo "The first argument is: $1"
echo "The second argument is: $2"
echo "All arguments are: \$@ -> $@"
echo "The number of arguments is: \$# -> $#"
```

If you run this:

```
bash positional_parameters.sh arg1 arg2 arg3
```

```
The script name is: positional_parameters.sh
The first argument is: arg1
The second argument is: arg2
All arguments are: $@ -> arg1 arg2 arg3
The number of arguments is: $# -> 3
```

Positional Parameters in Functions

Positional parameters in functions work similarly to how they work in scripts.

- $1, $2, $3….
- Not from $0

Files

- /etc/profile
  - systemwide defaults, mostly setting the environment
- /etc/bashrc
  - systemwide functions and and aliases for Bash
- $HOME/.bash_profile
  - user-specific Bash environmental default settings, found in each user's home directory
- $HOME/.bashrc
  - user-specific Bash init file, found in each user's home directory

Debugging

- The Bash shell contains no debugger, nor even any debugging-specific commands or constructs.
- The simplest debugging aid is the output statement, **echo**.
- Set option
    - -n: Don’t run command; check for syntax error only
    - -v: Echo commands before running them
    - -x: Echo commands after command-line processing