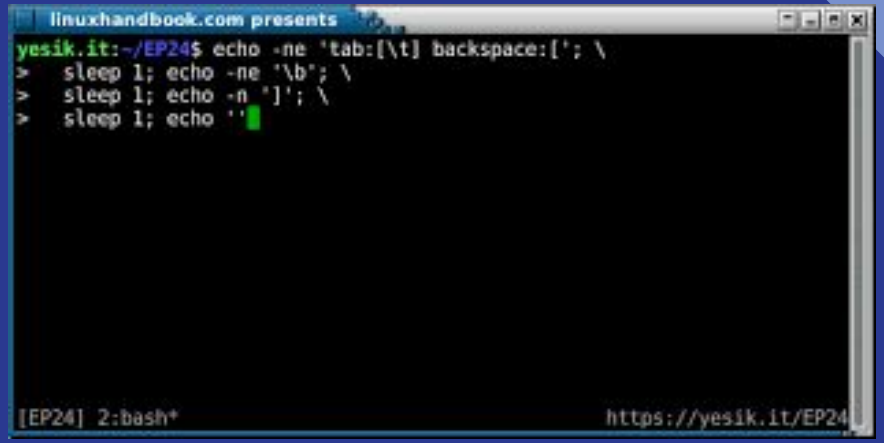
Shell Scripting!
Class 2



Special Variables - examples

```
# $@ - Stores arguments as an array
echo "Arguments stored in \$@:"
for arg in "$@"; do
    echo "$arg"
done

# $# - Show the number of arguments supplied in a
given script
echo "Number of arguments: $#"

# $$ - Displays the process ID of the current shell
echo "Process ID of the current shell: $$"
```

```
# $* - Groups all given arguments by
connecting them together
echo "Arguments grouped together using \$*:"
echo "$*"

# $! - Shows the ID of the last background job
echo "ID of the last background job: $!"

# $? - Displays the exit status code for the last
executed command
echo "Exit status code of the previous
command: $?"
echo
```

File Operators

File operators are used to check various attributes and properties of files.

- **-e** FILE: True if FILE exists.
- **-f** FILE: True if FILE exists and is a regular file.
- -d FILE: True if FILE exists and is a directory.
- -r FILE: True if FILE exists and is readable.
- -w FILE: True if FILE exists and is writable.
- -x FILE: True if FILE exists and is executable.
- -s FILE: True if FILE exists and has a size greater than zero.
- -L FILE: True if FILE exists and is a symbolic link.
- -G FILE: True if FILE exists and is owned by the effective group ID.
- -O FILE: True if FILE exists and is owned by the effective user ID.

File Operators - Examples

```
if [ -e file.txt ]; then
    echo "File exists"
fi

if [ -f file.txt ]; then
    echo "File is a regular file"
fi

if [ -d directory ]; then
    echo "Directory exists"
fi

if [ -r file.txt ]; then
    echo "File is readable"
fi

if [ -s file.txt ]; then
    echo "File is not empty"
fi
```

```
if [ -w file.txt ]; then
    echo "File is writable"
fi

if [ -x script.sh ]; then
    echo "Script is executable"
fi

if [ -x script.sh ]; then
    echo "Script is executable"
fi

if [ -x script.sh ]; then
    echo "Script is executable"
fi
```

String Operators

String operators are used to manipulate and compare strings

- **-z:** This operator returns true if the length of the string is zero (i.e., the string is empty).
- **-n:** This operator returns true if the length of the string is non-zero (i.e., the string is not empty).
- **Length Operator - ${#string}:** Returns the length of the string.

- **Substring Removal (Prefix):**
  - **${string#substring}:** Removes the shortest match of substring from the beginning of the string.
  - **${string##substring}:** Removes the longest match of substring from the beginning of the string.
- **Substring Removal (Suffix):**
  - **${string%substring}:** Removes the shortest match of substring from the end of the string.
  - **${string%%substring}:** Removes the longest match of substring from the end of the string.

String Operators

- **Substring Extraction - ${string:start:length}:** Extracts a substring starting at the specified position with the specified length.

- **Substring Replacement:**
  - **${string/substring/replacement}:** Replaces the first occurrence of substring with replacement.
  - **${string//substring/replacement}:** Replaces all occurrences of substring with replacement.

- **Substring Test:${string:substring}:** Tests if substring is present in string.
  If present, returns true (0); otherwise, returns false (1).

String Operators - Examples

```
string="Hello, World!"
echo "Length of the string: ${#string}"
# Output: 13

echo "Substring: ${string:7:5}"
# Output: World

echo "Prefix Removal: ${string#Hello, }"
# Output: World!

echo "Suffix Removal: ${string%World!}"
# Output: Hello,

echo "Substring Replacement: ${string/Hello/Hi}"
# Output: Hi, World!
```

```
if [[ $string == *"Hello"* ]]; then
    echo "Substring 'Hello' is present."
else
    echo "Substring 'Hello' is not present."
fi

# Output: Substring 'Hello' is present.
```

String Operators - Example (-z)

```
#!/bin/bash

string1="Hello"
string2=""

if [ -z "$string1" ]; then
    echo "string1 is empty."
else
    echo "string1 is not empty."
fi

if [ -z "$string2" ]; then
    echo "string2 is empty."
else
    echo "string2 is not empty."
fi
```

Output:

string1 is not empty.

string2 is empty.

String Operators - Example (-n)

```
#!/bin/bash

string1="Hello"
string2=""

if [ -n "$string1" ]; then
    echo "string1 is not empty."
else
    echo "string1 is empty."
fi

if [ -n "$string2" ]; then
    echo "string2 is not empty."
else
    echo "string2 is empty."
fi
```

Output:

string1 is not empty.

string2 is empty.

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on numeric values.

Addition (+): Adds two numbers.

```
sum=$((5 + 3))
echo "Sum: $sum" # Output: 8
```

Subtraction (-): Subtracts the second number from the first.

```
difference=$((10 - 3))
echo "Difference: $difference" # Output: 7
```

Multiplication (*): Multiplies two numbers.

```
product=$((5 * 4))
echo "Product: $product" # Output: 20
```

Division (/): Divides the first number by the second. Note: If the divisor is zero, Bash will throw an error.

```
quotient=$((20 / 5))
echo "Quotient: $quotient" # Output: 4
```

Arithmetic Operators

Modulus (%): Returns the remainder of the division operation.

```
remainder=$((10 % 3))
echo "Remainder: $remainder" # Output: 1
```

Increment (++): Increases the value of a variable by 1.

```
count=5
((count++))
echo "Incremented value: $count" # Output: 6
```

Decrement (--): Decreases the value of a variable by 1.

```
count=5
((count--))
echo "Decremented value: $count" # Output: 4
```

Comparison Operators with Arithmetic Value

Equal (-eq): Checks if two values are equal.

```
if [ "$num1" -eq "$num2" ]; then
    echo "num1 is equal to num2"
fi
```

Not Equal (-ne): Checks if two values are not equal.

```
if [ "$num1" -ne "$num2" ]; then
    echo "num1 is not equal to num2"
fi
```

Greater Than (-gt): Checks if the first value is greater than the second.

```
if [ "$num1" -gt "$num2" ]; then
    echo "num1 is greater than num2"
fi
```

Greater Than or Equal To (-ge): Checks if the first value is greater than or equal to the second.

```
if [ "$num1" -ge "$num2" ]; then
    echo "num1 is greater than or
equal to num2"
fi
```

num1=5
num2=10

Comparison Operators with Arithmetic Value

Less Than (-lt): Checks if the first value is less than the second.

```
if [ "$num1" -lt "$num2" ]; then
    echo "num1 is less than num2"
fi
```

Less Than or Equal To (-le): Checks if the first value is less than or equal to the second.

```
if [ "$num1" -le "$num2" ]; then
    echo "num1 is less than or equal to num2"
fi
```

```
#!/bin/bash

num1=5
num2=10

if [ "$num1" -eq "$num2" ]; then
    echo "num1 is equal to num2"
elif [ "$num1" -lt "$num2" ]; then
    echo "num1 is less than num2"
else
    echo "num1 is greater than
num2"
fi
```

Exit Status and Return Codes

- Every command returns an exit status
- Exit status: Numerical code indicating script execution outcome
- Range from 0-255
- 0: Script executed successfully
- Non-zero: Script encountered errors or issues
- Specific codes may indicate specific errors (e.g., 1: permission denied, 2: no such file or directory, etc.)
- Use echo $? after script execution to check status
- Use for error checking
- Use man or info to find meaning of exit status

Checking the Exit Status

ls /not/there/

echo “$?”

Output: 2

Echo “Hello World”

echo “$?”

Output: 0

```
HOST="google.com"

# Check if the host is reachable
ping -c 1 $HOST

# Check the exit status of the ping command
if [ “$?” -eq “0” ];
then
    echo "$HOST is reachable."
else
    echo "Unable to reach $HOST."
fi
```

Checking the Exit Status

```
HOST="google.com"

# Check if the host is reachable
ping -c 1 $HOST

# Check the exit status of the ping
command
if [ “$?” -ne “0” ];
then
    echo "$HOST is unreachable."
fi
```

```
HOST="google.com"

# Check if the host is reachable
ping -c 1 $HOST
RETURN_CODE=$?

# Check the exit status of the ping
command
if [ “RETURN_CODE” -eq “0” ];
then
    echo "$HOST is unreachable."
fi
```

Checking the Exit Status

&& = AND
mkdir /tmp/bak && cp test.txt /tmp/bak/

```
#!/bin/bash
HOST="google.com"

# Check if the host is reachable
ping -c 1 $HOST && echo "$HOST is reachable."
```

|| = OR
cp test.text /tmp/bak/ || cp test.txt /tmp

```
#!/bin/bash
HOST="google.com"

# Check if the host is reachable
ping -c 1 $HOST || echo "$HOST is unreachable."
```

The Semicolon

To get all executed in a single line use a semicolon separating commands

```
# Declare and assign values to variables
NAME="John"; AGE=30; CITY="New York"

# Print the values of variables
echo "Name: $NAME; Age: $AGE; City: $CITY"
```

Same as:

```
NAME="John"
AGE=30
CITY="New York"

echo "Name: $NAME"
echo "Age: $AGE"
echo "City: $CITY"
```

The Semicolon

To get all executed in a single line use a semicolon separating commands

cp text.txt /temp/test; cp text.txt /temp

Same as:

cp text.txt /temp/test

cp text.txt /temp

ls /not/there; hostname

Output:
ls: /not/there: No such file or directory
username

date; uptime

Output:
Wed Feb 21 15:13:10 +06 2024
15:13 up 13 days, 20:48, 4 users, load averages: 3.07 3.89 4.22

The Exit Command

- **Explicitly define the return code**
  - exit 0
  - exit 1
  - exit 2
  - exit 3
  - exit 4
  - ….
  - exit 255
- **The last executed command is the default value**

The Exit Command

```
#!/bin/bash

# Check if a file exists
if [[ ! -f "important_data.txt" ]];
then
  echo "Error: File not found!"
  exit 2
fi

# Process the file successfully
# ...

# All good, signal success!
exit 0
```

```
#!/bin/bash
HOST="google.com"

# Check if the host is reachable
ping -c 1 $HOST

# Check the exit status of the ping
command
if [ “$?” -ne “0” ];
then
    echo "$HOST is unreachable."
    exit 1
fi
exit 0
```

The Exit Command - Summary

- All command return an exit status
- 0 - 255 status codes
- 0 - Success status
- Other than 0 - Error status
- 1 - General errors or failure
- 2 - Incorrect usage (e.g., missing arguments)
- (3-254) - Custom errors (based on script requirements)
- 255 - Fatal errors (non-recoverable issues)
- $? Contains the last exit status
- exit command
- Decision making with if, &&, ||

Loops

- **Reading files line-by-line**
- **for loop**
  - Iterates over a list of items
- **while loop**
  - Continues executing a block of code as long as a condition is true
  - Infinite loops
- **until loop**
  - Continues executing a block of code until a condition becomes true
- **`break`** and **`continue`**
- ***If the command fails, it returns non-zero exit status and then continue the loop until it is closed by force***

```
while [ is_condition_true ];
do
    command 1
    command 2
    …command N
done
```

Loops

While loop

```
while [ is_condition_true ];
do
    command 1
    command 2
    …command N
done
```

Infinite loop

```
while true;
do
    command 1
    command 2
    …command N
done
```

Commands change the condition

Commands do not change the condition

Loops (Examples - while, infinite)

 # Example of a while loop

```
count=1

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

The while loop iterates as long as the variable count is less than or equal to 5.

 # Example of an infinite loop

```
while true; do
    echo "This loop runs forever!"
done
```

The infinite loop, if uncommented, will continue to execute indefinitely, printing "This loop runs forever!" repeatedly until manually stopped.

Loops (Examples - while, for)

 # Print numbers from 1 to 5

```
i=1
while ((i<=5)); do
    echo $i
    ((i++))
done
```

 # Read user input until 'quit' is entered
input=""

```
while [[ "$input" != "quit" ]]; do
    read -p "Enter something (or 'quit' to exit):
" input
    echo "You entered: $input"
done
```

 # Print numbers from 1 to 5

```
for ((i=1; i<=5; i++)); do
    echo $i
done
```

 # Iterate over elements in an array

```
fruits=("apple" "banana" "orange")
for fruit in "${fruits[@]}"; do
    echo "I like $fruit"
done
```

Loops (Examples - until vs while)

 # Print numbers from 1 to 5

```
i=1
until ((i>5)); do
    echo $i
    ((i++))
done
```

 # Keep prompting until 'yes' is entered

```
input=""
until [[ "$input" == "yes" ]]; do
    read -p "Do you want to continue? (yes/no): "
input
    echo "You entered: $input"
done
```

 # Example of while loop

```
counter=0
while (( counter < 5 )); do
    echo "Counter: $counter"
    (( counter++ ))
done
```

 # The loop continues executing as long as the condition evaluates to false.

 # Once the condition becomes true, the loop terminates.

Loops (Examples - until vs while)

 # Example: Waiting for a service to become available

```
echo "Waiting for database service to become available..."

until nc -z localhost 5432; do
    echo "Database service is not yet available. Retrying in 5 seconds..."
    sleep 5
done

echo "Database service is now available. Proceeding with the script."
```

 # The until loop continues executing as long as the condition evaluates to true.

 # Once the until loop condition becomes false, the loop terminates.

In the provided example, `nc` is used to check if a TCP connection to a specific port is successful. The `-z` option tells nc to scan for open ports, and if successful, it returns zero indicating the port is open, otherwise, it returns a non-zero exit status indicating the port is closed.

Loops (Examples - until vs while)

 # Example: Polling a remote server until it responds

```
echo "Polling remote server..."

# Using 'while' loop to repeatedly send ping requests until a response is received

while ! ping -c 1 -W 1 example.com &> /dev/null; do
    echo "Remote server is not reachable. Retrying in 5 seconds..."
    sleep 5
done

echo "Remote server is reachable. Proceeding with operations."
```

 # Example: Waiting for a file to be created

```
echo "Waiting for log file to be created..."

# Using 'until' loop to wait until the file exists

until [[ -f /var/log/application.log ]]; do
    echo "Log file does not exist yet. Retrying in 3 seconds..."
    sleep 3
done

echo "Log file detected. Proceeding with processing."
```

Loops (Examples - until vs while)

- The **until** loop waits until the log file /var/log/application.log is created. It checks for the existence of the file using the -f test condition. If the file does not exist, it prints a message and retries after 3 seconds until the file is created.

- The **while** loop continuously sends ping requests to the remote server example.com until it receives a response. It uses the ping command to send a single ICMP echo request with a timeout of 1 second (-c 1 -W 1). If the server is unreachable, it prints a message and retries after 5 seconds until a response is received.

- **-c** for count and **-w** for deadline in bash ping command
- **!** for negation of the result of the ping command
- **&>** for the combination of both stdout and stdin operators `>` and `2>&1`
- **>** Redirects standard output (stdout) of a command to a file.
- **2>&1** Redirects standard error (stderr) to the same location as stdout.

Loops (Examples - infinite)

```
while true
do
  read -p "Select your choice: 1: uptime.  2: disk usage. " MY_CHOICE
  case "$MY_CHOICE" in
    1)
      uptime
      ;;
    2)
      df -h
      ;;
    *)
      break
      ;;
  esac
done
```

Infinite loop: It continues until it matches `*` and `*break*` statement

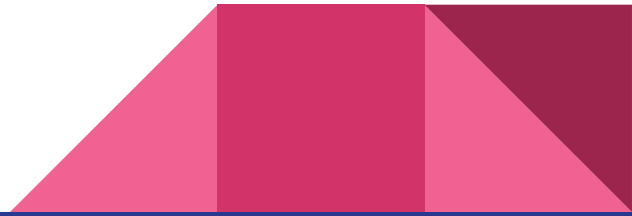
Functions

- **Why to use**
  - Keep it DRY
  - Make reusable
  - Reduce script length
  - Easier to maintain and troubleshoot
- **How to create**
  - Must be defined before using it
  - Has parameter support
- **How to use**
  - Use after defining function
  - Best practice to define at starting the script

```
function function-name() {
   # code goes here
}
```

```
function-name() {
   # code goes here
}
```

```
function-name
```



Functions

- **Function can call other functions**

```
#!/bin/bash

function hi() {
    echo "Hi!"
    now
}

function now() {
    echo "It's $(date +%r)"
}

hi
```

```
#!/bin/bash

function hi() {
   echo "Hi!"
   now
}

hi

function now() {
   echo "It's $(date +%r)"
}
```

 # This will cause an error as the "now()" function is not yet defined.

Functions

- **Functional parameters**
  - variables that are declared as part of a function definition
- **Positional parameters**
  - Functions can accept parameters
  - 1st parameter is stored in $1
  - 2nd parameter is stored in $2
  - And so on
  - $@ contains all the parameters
- **Just like shell scripts**
  - $0 is the script itself, not function name

```
function hello() {
    local param1 = $1
    echo "Hello $param1"
}

hello Mostafa

function hello() {
    for NAME in $@
    do
        echo "Hello $NAME"
    done
}

hello Mostafa Mahmud
```

Functions - with Global Variable

- **Variable scope**
  - Variables are global by default
  - Must be defined before using

```
#!/bin/bash

my_function() {
    echo "$GLOBAL_VAR"
}

GLOBAL_VAR=1
# The value of GLOBAL_VAR is
available to my_function

my_function
```

```
#!/bin/bash

my_function() {
   echo "$GLOBAL_VAR"
}

# The value of GLOBAL_VAR is NOT
available to my_function since GLOBAL_VAR
was defined after my_function was called.

my_function

GLOBAL_VAR=1
```

Functions - with Local Variable

- **Local Variable**
  - Only access within function
  - Using **local** keyword
  - Only functions have **local** var
  - Best practices to use var in functions as **local**

```
my_function() {
    local LOCAL_VAR=1
    echo "LOCAL_VAR can be accessed inside of the function: $LOCAL_VAR"
}

my_function

# LOCAL_VAR is not available outside of the function.
echo "LOCAL_VAR can NOT be accessed outside of the function: $LOCAL_VAR"
```

Functions - exit status/return code

- **Every functions have exit status**
  - Valid exit codes range from 0-255
  - 0 = success
  - Non-zero = error
  - $? = exit status
- **Explicitly usage**
  - return <return code>
- **Implicitly usage**
  - The exit status of the last command executed in the function

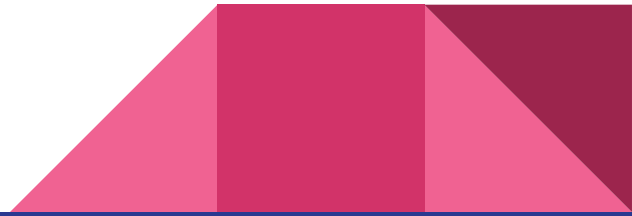
```
my_function() {
        #code goes here
}

my_function

echo “$?”
```

Functions - Example (exit status/return code, local)

```
function backup_file () {
  if [ -f "$1" ]
  then
    local BACKUP_FILE="/tmp/$(basename
${1}).$(date +%F).$$"

    echo "Backing up $1 to ${BACKUP_FILE}"

    # The exit status of the function will be the exit
status of the cp command.
    cp $1 $BACKUP_FILE
  else
    # The file does not exist, so return an non-zero
exit status.
    return 1
  fi
}

backup_file /etc/hosts

# Make a decision based on the
exit status of the function.
if [ $? -eq "0" ]
then
  echo "Backup succeeded!"
else
  echo "Backup failed!"
  # Abort the script and return a
non-zero exit status.
  exit 1
fi
```

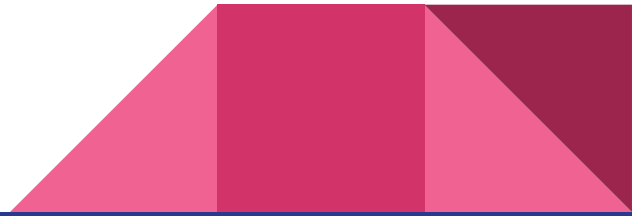
Functions - Summary

- DRY
- Global and local variables
- Parameters
- Exit statuses/return codes



Shell Script Order and Checklist

- **Shebang (#!)**
  Ensure the script begins with a shebang line specifying the interpreter (e.g., #!/bin/bash)
- **Comment/File Header**
  Include comments at the beginning of the script to describe its purpose, author, creation date, and any other relevant information.
- **Global Variables**
  Use descriptive variable names and initialize them properly.
- **Functions**
  Use local variable and organize functions logically based on their purpose or functionality
- **Main body/script content**
  Write the main logic of the script.
- **Error Handling, Logging, Testing etc.**
- **Exit with status code**

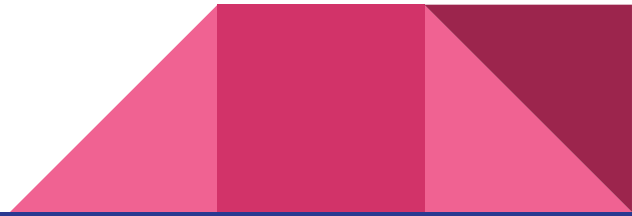


Wildcards

- **What is**
  - A character or string used for a pattern matching
  - Globbing that refers to the process of expanding wildcard patterns into a list of filenames or directories or strings that match the specified pattern
- **Different types of**
  - *** –>** *.txt, a*.txt, a* (matches zero or more characters)
  - **? –>** ?.txt, a?.txt, a? (matches exactly one character)
- **When and where can be used**
  - Shell command
  - Shell script
  - File operations
  - Regular expressions
- **How to use with various commands**
  - ls, rm, cp etc.

Wildcards - Character Class

- **[ ] - A Character Class**
  - It allows you to match any one character from a set of characters.
  - Matched exactly one character
  - [aeiou]
  - ca[nt]*
    i. cat
    ii. can
    iii. catch
    iv. candy
- **[!] - Matches any chars that are not included in the [ ]**
  - [!aeiou]*
    i. sky
    ii. fly
    iii. computer
    iv. desk



Wildcards - Ranges

- **Ranges separated by hyphen (-) allow you to specify a range of characters to match**
  - [a-z]: Matches any lowercase letter from 'a' to 'z'.
  - [A-Z]: Matches any uppercase letter from 'A' to 'Z'.
  - [0-9]: Matches any digit from '0' to '9'.

  - [a-z]: Matches any lowercase letter from 'a' to 'z'.
  - [A-Z]: Matches any uppercase letter from 'A' to 'Z'.
  - [0-9]: Matches any digit from '0' to '9'.
  - [a-zA-Z]: Matches any uppercase or lowercase letter.
  - [0-9a-f]: Matches any hexadecimal digit (0-9, a-f).

  - [a-d]*: Matches all files that start with a, b, c, or d
  - [4-7]*: Matches all files that start with 4, 5, 6, or 7

Wildcards - Named Character Classes

- **Named character classes such as alpha, alnum, and digit are shorthand representations for common character groups**
  - **[[:alpha:]]**
  - **[[:alnum:]]**
  - **[[:digit:]]**
  - **[[:lower:]]**
  - **[[:upper:]]**
  - **[[:space:]]**

```
string="Hello World 123!"

# Match alphabetic characters
if [[ $string =~ [[:alpha:]] ]]; then
   echo "Alphabetic character found: ${BASH_REMATCH[0]}"
fi

# Match digit characters
if [[ $string =~ [[:digit:]] ]]; then
   echo "Digit character found: ${BASH_REMATCH[0]}"
fi
```

Wildcards - \ - Escape Character

- **Double Quotes**

text1="a $(echo b) c"
text2="a \$(echo b) c"

echo "${text1}" **Output: a b c**

echo "${text2}" **Output: a $(echo b) c**

text="!event"
bash: !event: event not found

text="\a \$ \` \!event \\"

echo ${text} **Output: \a $ ` \!event **

- **No Quotes**

Namely, any sequence without quotes wouldn't be unified without escaping all characters, which are not alphanumeric or part of the following group: <comma>, <period>, <underscore>, <plus-sign>, <colon>, <commercial-at>, <percent-sign>, <slash>, <hyphen>:

text=a\ \&\ b\ \&\ c

echo "${text}" **Output: a & b & c**

Character Classes vs Character Patterns

- **Character patterns** represent the overall structure or format of the string being matched.
- Include literal characters, wildcard characters **(*, ?)**, and metacharacters **(., +, (), {})**.
- **Usage:** Used for more complex pattern matching, including repetition, alternation, and grouping.
- Applications: Employed in commands like **grep, sed, awk,** and Bash built-in constructs like **case statements**.
- **Purpose:** Provides a flexible and powerful means of matching strings and sequences of characters in Bash scripting.

- **Character classes** represent sets of characters used for pattern matching.
- Defined within square brackets **[ ]**.
- **Usage:** Allows matching any single character from a specified set of characters.
- Examples: **[aeiou]** matches any vowel, **[0-9]** matches any digit, **[^0-9]** matches any non-digit character.
- **Purpose:** Useful for specifying specific sets of characters to match against in patterns.

Different Patterns - Examples

- **Repetition:**
  - .*: Matches zero or more occurrences of any character.
  - .+: Matches one or more occurrences of any character.
  - [0-9]*: Matches zero or more occurrences of digits.

- **Alternation:**
  - pattern1|pattern2: Matches either pattern1 or pattern2.
  - (pattern1|pattern2): Matches either pattern1 or pattern2 within a group.
  - cat|dog: Matches either "cat" or "dog".

- **Grouping:**
  - (pattern): Groups patterns together for applying quantifiers or alternation.
  - ([0-9]{3}): Matches exactly three digits within a group.
  - (word1|word2|word3): Matches either "word1", "word2", or "word3".

Different Patterns - Examples

- **Anchors:**
  - ^pattern: Matches pattern at the beginning of a line.
  - pattern$: Matches pattern at the end of a line.
  - ^pattern$: Matches pattern as the entire line.

- **Quantifiers:**
  - pattern{m}: Matches exactly m occurrences of pattern.
  - pattern{m,n}: Matches at least m and at most n occurrences of pattern.
  - [0-9]{3,5}: Matches 3 to 5 occurrences of digits.

- **Character Classes and Ranges:**
  - [aeiou]: Matches any vowel character.
  - [A-Za-z]: Matches any uppercase or lowercase letter.
  - [0-9A-Fa-f]: Matches any hexadecimal digit.

- **Negation:**
  - [^0-9]: Matches any non-digit character.
  - [^aeiou]: Matches any non-vowel character.

Different Patterns - Examples

```
#!/bin/bash

# Example patterns for complex pattern matching
input="abc123def456ghi"

# Matching digits in the input string using repetition
if [[ $input =~ [0-9]+ ]]; then
    echo "Digits found: ${BASH_REMATCH[0]}"
fi

# Matching alternating patterns in the input string
if [[ $input =~ (abc|def|ghi) ]]; then
    echo "Alternating pattern found: ${BASH_REMATCH[0]}"
fi

# Matching groups of characters in the input string
if [[ $input =~ (abc[0-9]+def[0-9]+ghi) ]]; then
    echo "Grouped pattern found: ${BASH_REMATCH[0]}"
fi
```

- The first pattern matches one or more digits using **[0-9]+**, demonstrating repetition.
- The second pattern matches alternating substrings **(abc, def, ghi)**, showing alternation.
- The third pattern matches a group of characters (abc, followed by one or more digits, def, followed by one or more digits, ghi), illustrating grouping.
- We use the **=~** operator to perform pattern matching against the input string.

Different Patterns - Examples

```
#!/bin/bash
input="apple123 banana789 orange456 pineapple"

# Anchors:
# Match words that start with "apple"
if [[ $input =~ ^apple ]]; then
    echo "Anchor - Start of line: ${BASH_REMATCH[0]}"
fi

# Match words that end with "pineapple"
if [[ $input =~ pineapple$ ]]; then
    echo "Anchor - End of line: ${BASH_REMATCH[0]}"
fi

# Quantifiers:
# Match numbers consisting of three digits
if [[ $input =~ [0-9]{3} ]]; then
    echo "Quantifier - Three digits: ${BASH_REMATCH[0]}"
fi
```

```
# Negation:
# Match words that do not contain digits
if [[ $input =~ ^[^0-9]*$ ]]; then
    echo "Negation - No digits:
${BASH_REMATCH[0]}"
fi

# Match words that contain "na" followed
by any single character
if [[ $input =~ na. ]]; then
    echo "Character Range - 'na' followed
by any character:
${BASH_REMATCH[0]}"
fi
```

Wildcards - Usage in for loop

```
for FILE in /var/www/*.html
do
    echo "Copying $FILE"
    cp $FILE /var/www-just-html
done
```

```
# This will loop through all the
"html" files in the current directory.

for FILE in *.html
do
   echo "Copying $FILE"
   cp $FILE /var/www-just-html
done
```

Case Statements

- **Alternative to if statement**
  - If [“$VAR” == “one”]
  - elif [“$VAR” == “one”]
  - elif [“$VAR” == “one”]
  - elif [“$VAR” == “one”]
- **Might be easier to read and understand than complex if statements**
- **Patterns can include wildcards**
- **Multiple pattern matching using a pipe**

```
#!/bin/bash

case "$VAR" in
   pattern_1)
      # commands go here
      ;;
   pattern_N)
      # commands go here
      ;;
esac
```

Case Statements - Examples

```
#!/bin/bash

case "$1" in
   start)
      /usr/sbin/sshd
      ;;
   stop)
      kill $(cat /var/run/sshd.pid)
      ;;
esac
```

```
#!/bin/bash

case "$1" in
   start)
      /usr/sbin/sshd
      ;;
   stop)
      kill $(cat /var/run/sshd.pid)
      ;;
    *)
      echo "Usage: $0 start|stop" ; exit 1
      ;;
esac
```

Case Statements - Examples

```
#!/bin/bash

case "$1" in
    start|START)
        /usr/sbin/sshd
        ;;
    stop|STOP)
        kill $(cat /var/run/sshd.pid)
        ;;
    *)
        echo "Usage: $0 start|stop" ; exit 1
        ;;
esac
```

```
#!/bin/bash

read -p "Enter y or n:" ANSWER

case "$ANSWER" in
    [yY]|[yY][eE][sS])
        echo "You answered yes."
        ;;
    [nN]|[nN][oO])
        echo "You answered no."
        ;;
   *)
        echo "Invalid answer."
        ;;
esac
```

Case Statements - Examples

```
echo "Enter a fruit name: "
read fruit

# Match the input against multiple patterns
case $fruit in
    apple|orange)
        echo "It's a common fruit."
        ;;
    banana|pineapple)
        echo "It's a tropical fruit."
        ;;
    grape|kiwi)
        echo "It's a small fruit."
        ;;
    *)
        echo "Unknown fruit."
        ;;
esac
```

```
#!/bin/bash

read -p "Enter y or n: " ANSWER

case "$ANSWER" in
    [yY]*)
        echo "You answered yes."
        ;;
    *)
        echo "You answered something else."
        ;;
esac
```

Case Statements - Examples (pipe, wildcard)

```
process_file() {
    case $1 in
        *.txt|*.md)
            echo "Text file: $1"
            # Process text files
            ;;
        *.jpg|*.png|*.gif)
            echo "Image file: $1"
            # Process image files
            ;;
        *.sh)
            echo "Shell script: $1"
            # Process shell scripts
            ;;
        *)
            echo "Unknown file type: $1"
            # Handle other file types
            ;;
    esac
}
```

```
# Loop through all files in the current
directory
for file in *; do
    # Check if the file exists and is a regular
file
    if [ -f "$file" ]; then
        # Process the file based on its type
        process_file "$file"
    fi
done
```

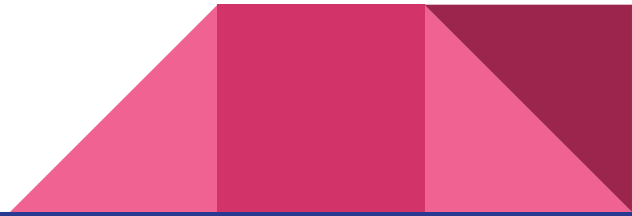
Case Statements - Examples (Real Case)

- We define a process_file function that takes a filename as an argument.
- The case statement inside the function matches different file extensions using pipes (|) and performs corresponding actions based on the matched patterns.
- We loop through all files in the current directory and call the process_file function for each file.
- Inside the loop, we check if the file exists and is a regular file before processing it.

Assignment 1

Write a script that renames all files in the current directory that end in ".txt" to begin with today's date in the following format: YYYY-MM-DD. For example, if a text file named "notes.txt" was in the current directory and today was January 15, 2023, it would change the name to "2023-01-15-notes.txt".

Extra Credit: Ensure graceful handling for instances where there are no ".txt" files in the current directory.

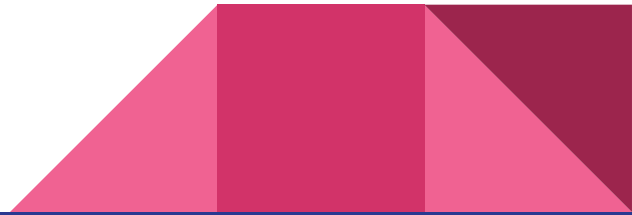


Assignment 2

Write a script that renames files based on the file extension. The script should prompt the user for a file extension. Next, it should ask the user what prefix to prepend to the file name(s). By default, the prefix should be the current date in YYYY-MM-DD format. So, if the user simply presses Enter, the date will be used. Otherwise, whatever the user enters will be used as the prefix. Next, it should display the original file name and the new name of the file. Finally, it should rename the file.

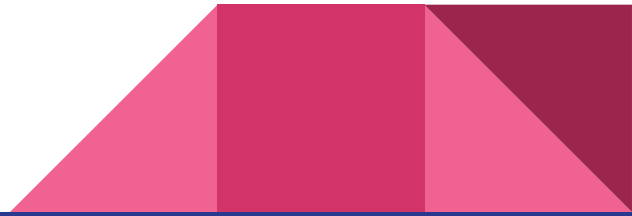
Example output 1:
Please enter a file extension: txt
Please enter a file prefix: (Press ENTER for 2023-01-15). notes
Renaming document.txt to 2023-01-15-document.txt.

Example output 2:
Please enter a file extension: txt
Please enter a file prefix: (Press ENTER for 2023-01-15).
Renaming notes.txt to 2023-01-15-notes.txt.



Assignment 3

Write a shell script that accepts a file or directory name as an argument. Have the script report if it is a regular file, a directory, or other type of file. If it is a regular file, exit with a 0 exit status. If it is a directory, exit with a 1 exit status. If it is some other type of file, exit with a 2 exit status.



Thank you!