PW SKILLS | DevOps and Cloud Computing

Advanced Bash Scripting - 2

Objective

- Utilize regular expressions for advanced pattern matching in Bash scripts.
- Understand and use arrays and associative arrays in Bash.
- Implement control structures such as if-else and loops for decision-making and iteration.
- Work with Bash variables and parameters effectively.
- Apply best practices for writing clean, efficient, and maintainable Bash scripts.

PW SKILLS

Regular expressions and their importance in pattern matching

Introducing Regular expressions

Regular expressions, often shortened to "regex," are sequences of characters that define a search pattern in text. They are a powerful tool used by programmers to locate, validate, or manipulate specific strings or patterns within text.

Regular expressions are essential for text processing, input validation, and searching algorithms.

Importance:

- Pattern matching
- Special characters
- Flexibility
- Use cases
- Equivalents

PW SKILLS

Demonstrating using grep, sed, and awk with regular expressions for text processing

Let’s do it

Here are examples of how to use grep, sed, and awk with regular expressions for text processing:

Grep: This command searches for patterns in a file and prints the lines that match.

- For example, grep "pattern" file.txt searches file.txt for lines containing "pattern“

Sed: This command replaces text using regular expressions.

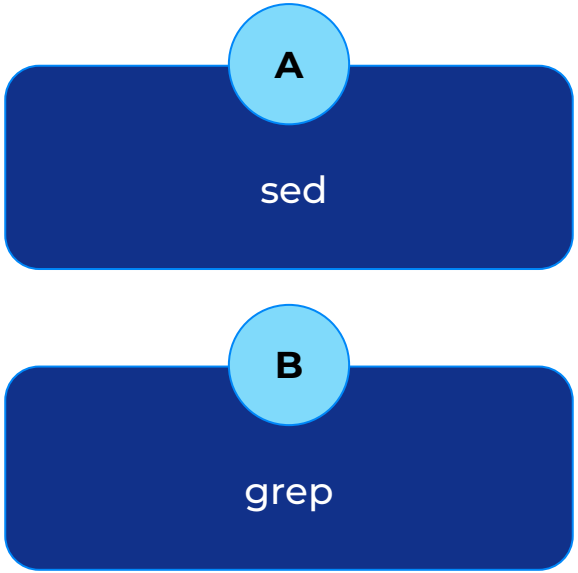
- For example, sed 's/oldstring/newstring/g' log.txt replaces all occurrences of "oldstring" with "newstring" in log.txt.

Awk: This command processes and manipulates text based on patterns.

- For example, awk '{print $1, $3}' log.txt prints the first and third fields of each line in log.txt.

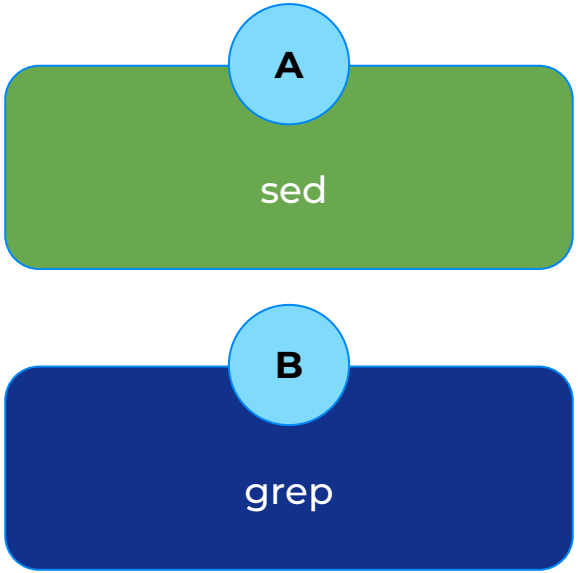
Pop Quiz

Q. For pattern-matching, awk uses regular expressions in which style?



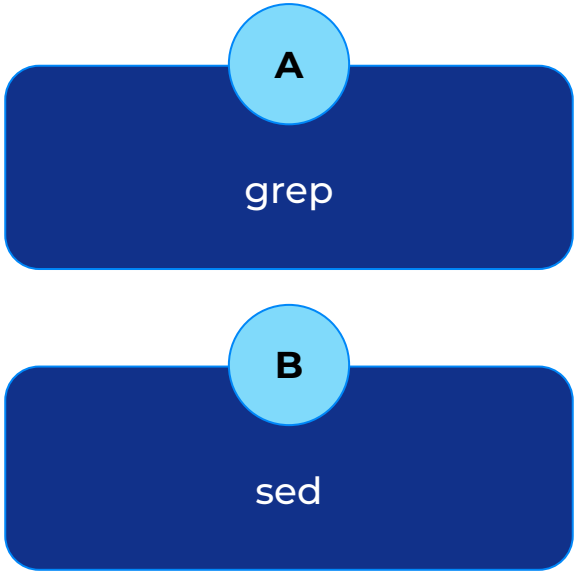
Pop Quiz

Q. For pattern-matching, awk uses regular expressions in which style?



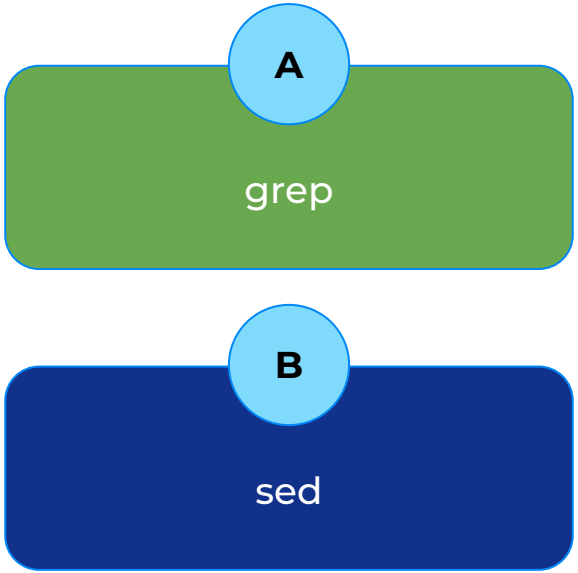
Pop Quiz

Q. Which command is used to search for a pattern in a file and print the lines that match the pattern?



Pop Quiz

Q. Which command is used to search for a pattern in a file and print the lines that match the pattern?



PW SKILLS

Providing examples of matching specific patterns

Examples

Here are examples of regular expressions that match specific patterns:

- Matching an email address
- Matching IP addresses
- Matching 8 digits in a row
- Matching positive integer literals
- Matching full integer literals
- Matching identifiers (or names)
- Matching a single "a" followed by zero or more "b"s followed by "c"

PW SKILLS

Explain the difference between standard arrays and associative arrays.

Standard & Associated arrays

Standard arrays and associative arrays differ primarily in how elements are accessed:

- **Standard Arrays:** Use sequential numeric indices to access elements. The index has to be an integer.
- **Associative Arrays:** Use keys (any scalar value) to access elements. Keys are unique within the array. Associative arrays are also known as maps, dictionaries, or symbol tables.

PW SKILLS

Demonstrating creating, accessing, and manipulating arrays

Let’s do it

Here's how to create, access, and manipulate arrays in Bash:

Standard Arrays (Indexed Arrays):

- Creation: ‘array=(val1 val2 val3)’
- Declaration (optional): ‘declare -a array name’
- Accessing Elements: ‘${array[index num]}’

  ${array} will retrieve val1.
- Looping through elements:

```
for item in ${array[@]}; do
  echo $item
done
```

Let’s do it

- Append elements: ‘array+=(new element)’
- Insert elements: ‘array name[position]=element to be inserted’
- Update elements: ‘example array[index]=updated element’

Associative Arrays:

- Declaration: ‘declare -A assoc array’
- Creation/Assignment: ‘assoc array[key]=value’
- Accessing Elements: ‘${assoc array[key]}’

For example:

```
assoc_array[element1]="Hello World"
echo ${assoc_array[element1]}
```

Take A 5-Minute Break!

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

PW SKILLS

Explaining if-else statements for conditional logic

Let’s do it

‘if’, ‘if...else’, and ‘if...elif...else’ statements in Bash allow you to execute code based on conditions.

Basic syntax:

```
if TEST-COMMAND
then
  STATEMENTS
fi
```

```
if TEST-COMMAND
then
  STATEMENTS1
else
  STATEMENTS2
fi
```

```
if TEST-COMMAND1
then
  STATEMENTS1
elif TEST-COMMAND2
then
  STATEMENTS2
else
  STATEMENTS3
fi
```

Let’s do it

Demonstrations:

- **Checking File Existence:**

```
FILE="example.txt"
if [ -f "$FILE" ]; then
  echo "File exists."
else
  echo "File does not exist."
fi
```

Let’s do it

- **Comparing Numbers:**

```
NUM1=10
NUM2=20
if [[ $NUM1 -gt $NUM2 ]]; then
  echo "$NUM1 is greater than $NUM2."
else
  echo "$NUM1 is not greater than $NUM2."
fi
```

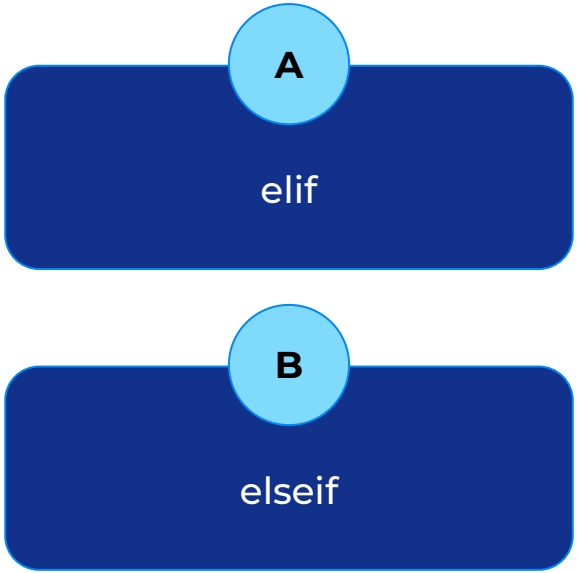
Let’s do it

- **Comparing Strings:**

```
STR1="hello"
STR2="world"
if [ "$STR1" = "$STR2" ]; then
  echo "Strings are equal."
else
  echo "Strings are not equal."
fi
```

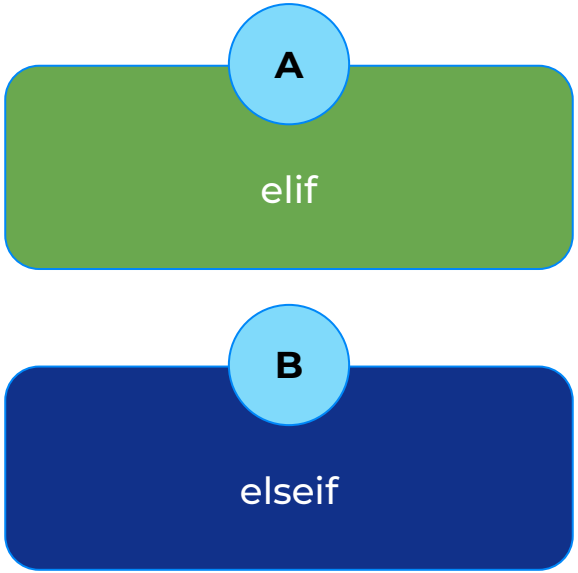
Pop Quiz

Q. Which keyword is used to check multiple conditions in an if statement?



Pop Quiz

Q. Which keyword is used to check multiple conditions in an if statement?



PW SKILLS

Introducing loops (for, while, until)

Let’s do it

Here's a short explanation of ‘for’, ‘while’, and ‘until’ loops in Bash, with examples of iteration:

1. **‘for’ Loop:**

Used to iterate a known number of times, often through a list.

- **Iterating through an array:**

```
array=(item1 item2 item3)
for item in "${array[@]}"; do
  echo "$item"
done
```

Let’s do it

- **Iterating through a file:**

```
for line in $(cat file.txt); do
  echo "$line"
done
```

- **Iterating through command output:**

```
for file in $(ls *.txt); do
  echo "$file"
done
```

Let’s do it

2. while Loop:

- Executes as long as a condition is true.

```
count=0
while [ $count -lt 5 ]; do
  echo "Count: $count"
  ((count++))
done
```

Let’s do it

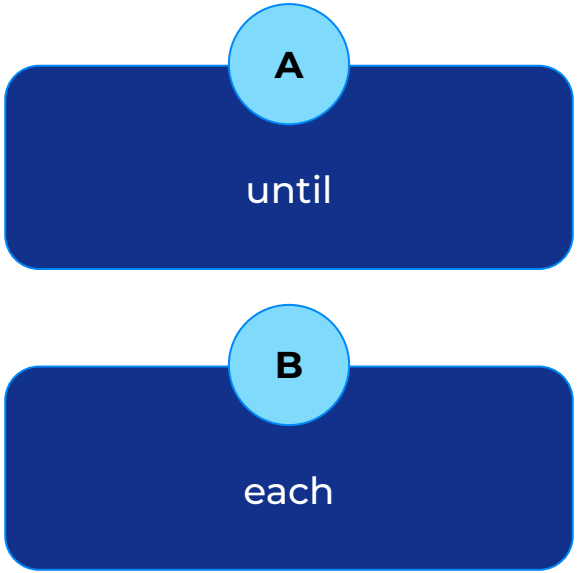
3. **‘until’ Loop:**

- Executes as long as a condition is false.
- It is the opposite of the ‘while’ loop

```
count=0
until [ $count -gt 5 ]; do
  echo "Counter: $count"
  ((count++))
done
```

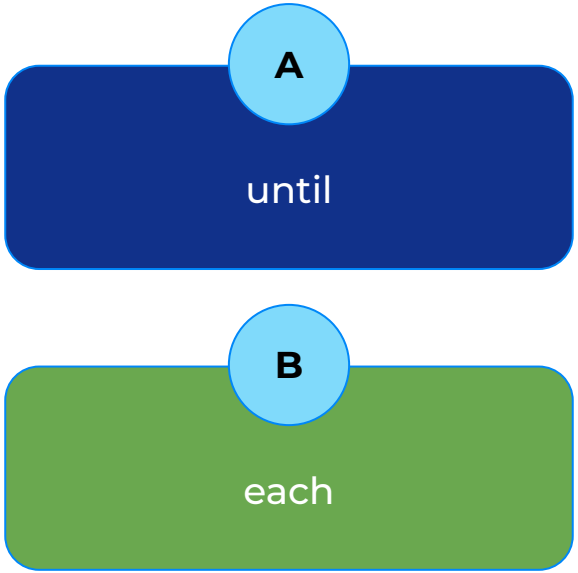
Pop Quiz

Q. What is NOT a keyword for looping in Bash?



Pop Quiz

Q. What is NOT a keyword for looping in Bash?



PW SKILLS

Explaining types of variables (local, global, special)

Types of variables

In Bash, variables can be categorized into three main types based on their scope and behavior:

- **Local Variables:** These variables are accessible only within the function or code block where they are defined.
- **Global Variables:** These variables are accessible throughout the script. Any variable defined outside a function is considered a global variable.
- **Special Variables (Environment Variables):** These variables are set by the system and contain information about the environment, such as the user's home directory or the system path. To make a variable an environment variable, use the ‘export’ command. To list all environment variables, use the ‘env’ command.

PW SKILLS

Demonstrating parameter substitution

Let’s do it

Here's a demonstration of parameter substitution in Bash:

1. Default Values: ${var:-default}

- If var is unset or null, the expansion results in default. Otherwise, the value of var is substituted

```
unset myvar
echo ${myvar:-"Hello"}  # Output: Hello
myvar="World"
echo ${myvar:-"Hello"}  # Output: World
```

Let’s do it

2. String Manipulation:

- ${var%pattern}: Removes the shortest matching pattern from the suffix (end) of $var.
- ${var#pattern}: Removes the shortest matching pattern from the prefix (beginning) of $var.

```
var="abc1234zip1234abc"
echo ${var#abc}   # Output: 1234zip1234abc (removes shortest "abc" from the beginning)
echo ${var%abc}   # Output: abc1234zip1234 (removes shortest "abc" from the end)

var="filename.txt"
echo ${var%.txt}  # Output: filename (removes ".txt" from the end)
echo ${var#*.}    # Output: txt (removes everything before the last "." from the beginning)
```

PW SKILLS

Examples of passing arguments to scripts and accessing them

Examples

- Passing Arguments: Arguments are passed to a Bash script by typing them on the command line after the script's name. For example: bash script.sh arg1 arg2.
- Accessing Arguments: Inside the script, arguments are accessed using positional parameters:

Example:

```
#!/bin/bash
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "Total number of arguments: $#"
echo "All arguments: $@"
```

Examples

- If you run above script with ‘bash script.sh hello world’, the output will be:

```
Script name: script.sh
First argument: hello
Second argument: world
Total number of arguments: 2
All arguments: hello world
```

- **Handling Multiple Arguments:**

```
#!/bin/bash
echo "All arguments: $@"
echo "Looping through the arguments:"
while (( "$#" )); do
  echo "Argument: $1"
  shift
done
```

Examples

- Running ‘bash script.sh one two three four’ will output:

```
All arguments: one two three four
Looping through the arguments:
Argument: one
Argument: two
Argument: three
Argument: four
```

- **Passing output of one script to another:** You can also pass the output of one shell script as an argument to another shell script.

Examples

```
#!/bin/bash
while getopts ":f:o:" opt; do
case $opt in
f) echo "File option passed with value: $OPTARG" ;;
o) echo "Output option passed with value: $OPTARG" ;;
\?) echo "Invalid option: -$OPTARG" ;;
esac
done
```

- Running ‘bash script.sh -f input.txt -o output.txt’ will output:

```
File option passed with value: input.txt
Output option passed with value: output.txt
```

PW SKILLS

Emphasizing writing readable and maintainable scripts

Let’s learn

To write readable and maintainable Bash scripts, emphasize the following:

- Meaningful Variable Names
- Comments
- Input Validation
- Error Handling
- Consistency
- Double Quote Variables
- Use Local Variables
- Use Descriptive Function Names
- Use ‘[[‘ instead of ‘[‘ or test
- Use ‘$(...)’ instead of backticks
- Avoid Option Flags

PW SKILLS

Demonstrating the use of set -e, set -u, and set -o pipefail for robust scripts.

Let’s do it

To create robust Bash scripts, use the following ‘set’ commands:

- ‘set –e’ (or ‘set -o errexit’): This option causes the script to exit immediately if a command exits with a non-zero status.
- ‘set –u’ (or ‘set -o nounset’): This option treats unset variables as an error, causing the script to exit if an unset variable is used.
- ‘set -o pipefail’: This option causes a pipeline to return the exit status of the last command to exit with a non-zero status, even if other commands in the pipeline succeed.

Let’s do it

- Example: In this example, if ‘grep’ fails or ‘var2’ is unset, the script will exit.

```
#!/usr/bin/env bash
set -euo pipefail  # Exit immediately if a command exits with a non-zero status or an unset variable is used, or a command in a pipeline fails

var1="Hello"
echo "$var1" | grep "Hello" #this command works and returns 0
echo "$var2" #the script exits due to the -u option
```

PW SKILLS

Providing examples of modular scripting with functions

Examples

Here are examples of modular scripting with functions in Bash:

1. Greeting Function:

```
#!/bin/bash

# Define the greet function
greet() {
  local name=$1
  echo "Hello, $name! Welcome!"
}

# Call the greet function with a name
greet "Kalevi"
```

Examples

2. Arithmetic Functions:

```
#!/bin/bash

# Function for addition
add() {
  local num1=$1
  local num2=$2
  local sum=$((num1 + num2))
  echo "The sum of $num1 and $num2 is: $sum"
}

# Function for subtraction
subtract() {
  local num1=$1
  local num2=$2
  local difference=$((num1 - num2))
  echo "The difference between $num1 and $num2 is: $difference"
}

# Call the functions
add 5 3
subtract 5 3
```

Examples

3. String Manipulation Functions:

```
#!/bin/bash

# Function to get the length of a string
string_length() {
  local str=$1
  echo "Length of '$str' is: ${#str}"
}

# Function to extract a substring from a string
substring_extraction() {
  local str=$1
  local start=$2
  local length=$3
  local substring=${str:start:length}
  echo "Substring from position $start with length $length in '$str' is: $substring"
}

# Function to concatenate two strings
string_concatenation() {
  local str1=$1
  local str2=$2
  local concatenated="$str1$str2"
  echo "Concatenated string of '$str1' and '$str2' is: $concatenated"
}

# Test the string manipulation functions
string_length "Hello, world!"
substring_extraction "Hello, world!" 6 4
string_concatenation "Bash, " "Scripting!"
```

PW SKILLS

Time for case study!

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this session
- Review your lecture notes
- Note down questions and queries regarding this session and consult the teaching assistants

Thanks !