Linux Shell/Bash Scripting Fundamentals (Concise Guide)

This guide provides a brief overview of fundamental concepts in bash scripting.

---

1. Variables

Variables store data in a named memory location. Assign values using `NAME="Value"` (no spaces around `=`). Access their content by prepending a dollar sign: `$`NAME`or` `` ``${NAME} ``. Using curly braces `${}` is good practice for clarity, especially when concatenating with other text.

- **User-Defined**: Variables you create, e.g., `MY_VAR="hello world"`.
- **Environment**: Pre-set variables available to all processes, like `$`HOME`,` `` ``$USER ``, `$PATH`.
- **Special**: Variables providing script information:
  - `$0`: Script name.
  - `$`1`` ,  ``$2 ``, ...: Positional arguments passed to the script.
  - `$#`: Number of arguments.
  - `$?`: Exit status of the last executed command (0 for success, non-zero for failure).

```
# Example: Define and access variables
NAME="World"
AGE=30
CURRENT_YEAR=$(date +%Y)

echo "Hello, $`NAME! You are `$AGE years old in $CURRENT_YEAR."
```

---

2. Comments

Comments are non-executable lines in a script used to explain the code. They are ignored by the shell but are crucial for human readability and understanding, especially in complex scripts or when collaborating.

- **Single-line comments**: Start with a `#` symbol. Anything after `#` on the same line is a comment.

```
# This is a single-line comment
echo "Hello"

MY_VAR="value" # This is an inline comment
```

---

3. User Input and Validation

Interact with users using the `read` command to get input. Options like `-p` display a prompt, and `-s` hides input for sensitive data (e.g., passwords).

- **Basic Input**: `read USER_NAME`
- **Prompted Input**: `read -p "Enter your name: " NAME`
- **Silent Input**: `read -s -p "Enter password: " PASSWORD`
- **Validation**: Essential for robust scripts. Check if input is empty (`-z`), numeric (`[[ "`$NUM” = {}^{[}0-9]+$` ]]`), or within a range.

```
# Example: Get and validate numeric input
read -p "Enter a number between 1 and 10: " USER_NUM

if [[ "$`USER_NUM" =~ ^[0-9]+`$ ]] && (( USER_NUM >= 1 && USER_NUM <= 10 ));
then
  echo "Valid number entered: $USER_NUM."
else
  echo "Invalid input. Please enter a number between 1 and 10."
fi
```

---

4. Arguments

Command-line arguments allow passing data to a script at execution time. They are accessed via positional parameters.

- `$0` : The name of the script itself.
- `$`1 , ``$2 , ... `$ : The first, second, and Nth arguments respectively. Use curly braces for arguments beyond ``$9` (e.g., `${10}`).
- `$#` : The total number of arguments passed.
- `"$@"` : Expands to all arguments as separate, quoted strings. Ideal for iterating through arguments, especially those with spaces.
- `"$*"` : Expands to all arguments as a single string.

```
 # Usage: ./script.sh apple banana "cherry pie"

echo "Script name: $0"
echo "First argument: $1"
echo "All arguments: $@"
echo "Number of arguments: $#"

# Iterate over arguments
for arg in "$@"; do
  echo "Processing: $arg"
done
```

---

5. Arrays

Arrays store ordered collections of values. Each value is an element, accessed by its index (starting from 0).

- **Indexed Arrays**: `MY_ARRAY=("value1" "value2" "value3")`
  - Access element: `$ (e.g., ``${MY_ARRAY[0]}`)
  - All elements: `"${MY_ARRAY[@]}"` (preferred for iteration)
  - Number of elements: `${#MY_ARRAY[@]}`
- **Associative Arrays (Bash 4.0+)**: Store key-value pairs. Declare with `declare -A MY_MAP` .
  - Assign: `MY_MAP[key]="value"`

    - Access: `${MY_MAP[key]}`
    - All keys: `"${!MY_MAP[@]}"`

```
 # Example: Indexed array
FRUITS=("Apple" "Banana" "Cherry")
echo "First fruit: ${FRUITS[0]}"

# Example: Associative array
declare -A CAPITAL_CITIES
CAPITAL_CITIES["USA"]="Washington D.C."
CAPITAL_CITIES["France"]="Paris"
echo "Capital of France: ${CAPITAL_CITIES["France"]}"
```

---

6. Conditional Expressions

Used to evaluate conditions, returning true or false. The `[[ ... ]]` construct is a modern and powerful bash-specific feature, generally preferred over `[ ... ]`.

- **File Tests**: Check file properties.
    - `[[ -f "FILE" ]]`: True if FILE is a regular file.
    - `[[ -d "DIR" ]]`: True if DIR is a directory.
    - `[[ -e "PATH" ]]`: True if PATH exists.
- **String Tests**: Compare strings.
    - `[[ "STR1" == "STR2" ]]`: True if STR1 equals STR2.
    - `[[ -z "STR" ]]`: True if STR is empty.
    - `[[ "STR" =~ REGEX ]]`: True if STR matches REGEX (regular expression).
- **Arithmetic Tests**: Compare numbers. Use `(( ... ))` for arithmetic expressions.
    - `(( NUM1 > NUM2 ))`: True if NUM1 is greater than NUM2.
    - `(( NUM1 == NUM2 ))`: True if NUM1 equals NUM2.

```
 # Example: Conditional expressions
FILE="/etc/passwd"
if [[ -f "$`FILE" && "`$(whoami)" == "root" ]]; then
  echo "$FILE exists and you are root."
else
  echo "Condition not met."
fi
```

---

7. Conditional Statements

Control script execution flow based on conditions. The primary statements are `if` and `case`.

- **`if-elif-else`**: Executes different blocks of code based on sequential conditions. `bash # if-elif-else example SCORE=85 if (( SCORE >= 90 )); then echo "Grade: A" elif (( SCORE >= 80 )); then echo "Grade: B" else echo "Grade: C or lower" fi`
- **`case`**: Selects a block of code to execute based on pattern matching a variable's value. Useful for multiple choices. `bash # case example ACTION="start" case "$ACTION" in start|run) echo "Starting service...";; stop) echo "Stopping service...";; *) echo "Invalid action.";; esac`

Example: Checking a file's existence and permissions

```bash
#!/bin/bash

FILE_TO_CHECK="/tmp/my_test_file.txt"

if [ -f "$FILE_TO_CHECK" ]; then
  echo "File '$FILE_TO_CHECK' exists."
  if [ -r "$FILE_TO_CHECK" ]; then
    echo "File is readable."
  else
    echo "File is not readable."
  fi
  if [ -w "$FILE_TO_CHECK" ]; then
    echo "File is writable."
  else
    echo "File is not writable."
  fi
else
  echo "File '$FILE_TO_CHECK' does not exist."
fi
```

8. Loops

Loops execute a block of code repeatedly. Bash offers `for`, `while`, and `until` loops.

- **`for` loop**: Iterates over a list of items or a numeric range.
  - **List iteration**: `for ITEM in item1 item2; do ... done`

    - **C-style numeric**: `for (( i=1; i<=5; i++ )); do ... done`
- `while` **loop**: Continues as long as a condition is true.
    - `while [ CONDITION ]; do ... done`
    - Often used for reading files line by line: `while IFS= read -r LINE; do ... done < file.txt`
- `until` **loop**: Continues as long as a condition is false (i.e., until it becomes true).
    - `until [ CONDITION ]; do ... done`
- **Loop Control**: `break` (exits the loop), `continue` (skips to the next iteration).

```
# Example: while loop with counter
COUNT=0
while (( COUNT < 3 )); do
  echo "Loop iteration: $COUNT"
  ((COUNT++))
done
```

---

9. Functions

Functions are reusable blocks of code. They improve script organization and maintainability.

- **Definition**: `my_function() { commands; }` or `function my_function { commands; }`.
- **Calling**: Simply use the function name: `my_function`.
- **Arguments**: Accessed inside the function using `$`1```, ``$2```, etc., similar to script arguments.
- **Return Status**: Functions return an exit status (0 for success, non-zero for failure) using the `return` command. Check with `$?`.
- **Return Value**: To return data, `echo` the data from the function and capture it using command substitution: `RESULT=$(my_function)`.
- **Local Variables**: Use `local VAR_NAME="value"` inside a function to prevent conflicts with global variables.

```bash
# Example: Function with arguments and local variable
greet_user() {
  local name="$1"
  if [ -z "$name" ]; then
    echo "Error: Name is required." >&2
    return 1
  fi
  echo "Hello, $name!"
  return 0
}

greet_user "Alice" # Calls the function
greet_user         # Calls with no argument, triggers error
```

---

10. Debugging

Essential for finding and fixing errors. Bash provides built-in tools and common techniques.

- `echo` **Statements**: Print variable values and execution flow to trace issues.
- `set` **Options**: Modify shell behavior for debugging:
  - `set -x`: Prints commands and their arguments as they are executed (execution trace).
  - `set -v`: Prints shell input lines as they are read (verbose).
  - `set -e`: Exits immediately if a command exits with a non-zero status (exit on error).
  - `set -u`: Treats unset variables as an error (unset variable check).
- **Exit Status (** `$?` **)**: Always check the exit status of commands to understand success or failure.

```bash
#!/bin/bash -ex # Enable execution trace and exit on error

MY_VAR="Debugging in progress"
echo "Value: $MY_VAR"

# This command will fail if /nonexistent does not exist, and script will exit
due to -e
ls /nonexistent_directory

echo "This line will not be reached if previous command failed."
```

Thank You!

Thank you for reading this concise guide on Linux Shell/Bash Scripting Fundamentals.

I hope you found it helpful.

Connect with me:

- **LinkedIn**: linkedin.com/in/alamgirweb11