Shell Scripting for Absolute Beginners: Detailed Guide with Pokémon API Tasks

Generated by Grok 3

June 25, 2025

Contents

1 Introduction

This guide is for people who've never used a computer before and want to learn shell scripting in Bash, a way to give your computer commands to do tasks automatically. We'll start from zero, explaining every example from your notes in detail, like you're a kid learning for the first time. You'll understand what each command does, why it's useful, and how it works, with extra examples to make it stick. Then, we'll do the tasks (0–5) step-by-step, explaining every single line. Everything is written in plain English, with analogies to make it fun and clear. By the end, you'll be scripting like a Pokémon trainer catching Pokémon!

2 Setting Up Your Computer

2.1 What You Need to Know

- **Linux**: A free operating system, like Windows, but awesome for scripting. It's what runs websites like Google.
- **Terminal**: A window where you type commands, like texting your computer what to do.
- **Bash**: The program that reads your commands, like a waiter taking your order to the kitchen.
- **Script**: A text file with commands, like a to-do list your computer follows.

2.2 Step-by-Step Setup

1. **Open a Terminal**:
   - **On Linux (e.g., Ubuntu)**: Search "Terminal" in your apps or press `Ctrl+Alt+T`.
   - **On Windows**: Install Ubuntu via WSL:
     - Open PowerShell (search "PowerShell" in Start menu).
     - Run: `wsl --install`.
     - Follow prompts to set up Ubuntu (pick a username/password).
     - Open Ubuntu from Start menu.
   - **Online Option**: Go to https://replit.com, sign up, create a "Bash" project.
2. **Install Tools**: You need `curl` (fetches web data), `jq` (reads JSON), `nano` (text editor), and `texlive` (for PDF). Run:

```
sudo apt update
sudo apt install curl jq nano texlive-full
```

   - `sudo`: Unlocks admin mode (type your password).
   - `apt update`: Refreshes software list.
   - `apt install`: Downloads tools.

- Press `Y` if asked.

3. **Create a Folder**: Organize your work:

```
mkdir shell_learning
cd shell_learning
```

- `mkdir`: Makes a folder called `shell_learning`.`cd` : $Goesintothatfolder, likeopeningadrawer.$

4. **Learn Nano**: To write scripts:
   - Run: `nano myfile.sh`.
   - Type code.
   - Save: `Ctrl+O`, `Enter`.
   - Exit: `Ctrl+X`.
5. **Check Tools**: Verify installation:

```
curl --version
jq --version
nano --version
```

   Each should show a version number. If not, repeat step 2.

3 Understanding the Examples from Your Notes

We'll go through every example in your notes, from easiest to hardest in each section, explaining what it does, why it's useful, and how it works, like you're learning for the first time. Each gets an analogy and an extra example.

3.1 Kernel, Shell, and Terminal

Example:

```
whoami
# Output: username
```

What: Shows your username, like checking who's logged into your computer.

Why: Helps you know who you are on a shared computer (e.g., at school).

How:

- `whoami`: A command asking Linux, "Who's using me?"
- Output: Your username (e.g., `john`).

Analogy: Like asking, "Who's sitting at this desk?" The computer says your name.

Extra Example:

```
whoami
# Output: ash
```

Run it to see your username.

3.2 Basic Commands

Here are the examples, from simple to complex:

1. **Example: echo**

```
echo "Hello, World!"
# Output: Hello, World!
```

What: Prints text, like writing on a chalkboard.

Why: Shows messages in scripts or checks values (e.g., debugging).

How:

- `echo`: Command to print text.
- `"Hello, World!"`: Text to print, in quotes to keep it together.
- Output: `Hello, World!`

Analogy: Like yelling, "Say this out loud!" to your computer.

Extra Example:

```
echo "I love Pokémon!"
# Output: I love Pokémon!
```

2. **Example: ls**

```
ls
# Output: file1.txt  file2.txt
```

What: Lists files/folders, like looking in a drawer.

Why: Shows what files you can work with.

How:

- `ls`: Stands for "list". Shows visible files.
- Output: File names (e.g., `file1.txt file2.txt`).

Analogy: Like asking, "What's in this folder?"

Extra Example:

```
touch note.txt
ls
# Output: note.txt
```

3. **Example: pwd**

```
pwd
# Output: /home/username/shell_learning
```

What: Shows your current folder's path, like your GPS location.

Why: Keeps you from getting lost in the file system.

How:

- `pwd`: "Print working directory".

- Output: Path like `/home/username/shell_learning`.

Analogy: Like asking, "Where am I in the house?"

Extra Example:

```
pwd
# Output: /home/ash/shell_learning
```

4. **Example: touch**

```
touch test.txt
# Output: (creates test.txt)
```

What: Creates an empty file, like making a blank note.

Why: Prepares files for scripts or data.

How:

- `touch`: Creates `test.txt` if it doesn't exist.
- No output, but `ls` shows `test.txt`.

Analogy: Like grabbing a blank paper and labeling it.

Extra Example:

```
touch todo.txt
ls
# Output: todo.txt
```

5. **Example: cat**

```
cat test.txt
# Output: (displays contents of test.txt)
```

What: Shows a file's contents, like reading a note aloud.

Why: Lets you see what's in a file without editing it.

How:

- `cat`: "Concatenate", but here it displays `test.txt`.
- Output: File contents (empty if new).

Analogy: Like opening a letter and reading it.

Extra Example:

```
echo "Hi" > greet.txt
cat greet.txt
# Output: Hi
```

3.3 Writing Your First Script

Example:

```
#!/bin/bash
echo "My first script!"
ls
```

What: A script that prints a message and lists files, like a to-do list.

Why: Automates multiple commands, saving time.

How:

- `#!/bin/bash`: Tells Linux to use Bash, like picking a program to open a file.
- `echo "My first script!"`: Prints the message.
- `ls`: Lists files.
- Save as `first.sh`, run:

  ```
  chmod u+x first.sh
  ./first.sh
  ```

- Output:

  ```
  My first script!
  test.txt
  ```

Analogy: Like a note saying, "Say this, then list the drawer's contents."

Extra Example:

```
#!/bin/bash
echo "Starting work!"
pwd
```

Save as `start.sh`, run:

```
chmod u+x start.sh
./start.sh
# Output:
# Starting work!
# /home/ash/shell_learning
```

3.4 Variables

1. **Example: Simple Variable**

```
name="Pikachu"
echo "Hello, $name"
# Output: Hello, Pikachu
```

What: Stores "Pikachu" in `name` and prints it, like labeling a box.

Why: Reuses data (e.g., names) in scripts.

How:

- `name="Pikachu"`: Stores "Pikachu" in `name`. No spaces around `=`.
- `echo "Hello, $name"`: Prints "Hello," plus the value of `name`.

Analogy: Like writing "Pikachu" on a sticky note and reading it.

Extra Example:

```
animal="Dog"
echo "I have a $animal"
# Output: I have a Dog
```

2. **Example: Script with Variable**

```
#!/bin/bash
name="Ash"
echo "Hello, $name! Welcome to Pokémon training."
```

What: A script that greets "Ash".

Why: Shows variables in scripts for personalized messages.

How:

- `name="Ash"`: Stores "Ash".
- `echo "Hello, $name! ..."`: Prints the greeting.
- Save as `greet.sh`, run:

```
chmod u+x greet.sh
./greet.sh
```

Analogy: Like a play script with the actor's name swapped in.

Extra Example:

```
#!/bin/bash
game="Pokémon"
echo "'Im playing $game!"
# Output: 'Im playing Pokémon!
```

3.5 Control Structures

1. **Example: If Statement**

```
#!/bin/bash
age=18
if [ $age -ge 18 ]; then
    echo "You can vote!"
else
    echo "You're too young to vote."
fi
```

What: Checks if `age` is 18+ and prints a message, like a club entry check.

Why: Makes decisions in scripts (e.g., checking user input).

How:

- `age=18`: Sets `age` to 18.
- `if [ $age -ge 18 ]; then`: Tests if `age` is greater/equal to 18.
- `echo "You can vote!"`: Runs if true.
- `else`: Runs if false.
- `fi`: Ends the if.

Analogy: Like a bouncer checking your ID.

Extra Example:

```
#!/bin/bash
grade=90
if [ $grade -ge 80 ]; then
    echo "A grade!"
else
    echo "Try harder."
fi
```

2. **Example: For Loop**

```
#!/bin/bash
for fruit in apple banana orange; do
    echo "I like $fruit"
done
```

What: Prints a message for each fruit, like reading a list.

Why: Automates repetitive tasks.

How:

- `for fruit in apple banana orange; do`: Loops through each item.
- `echo "I like $fruit"`: Prints for current `fruit`.
- `done`: Ends the loop.

Analogy: Like saying, "For each fruit, say you like it."

Extra Example:

```
#!/bin/bash
for color in red blue green; do
    echo "Color: $color"
done
```

3. **Example: While Loop**

```
#!/bin/bash
count=1
while [ $count -le 3 ]; do
    echo "Count: $count"
    count=$((count + 1))
done
```

What: Counts from 1 to 3, like a timer.

Why: Repeats tasks until a condition changes.

How:

- `count=1`: Starts at 1.
- `while [ $count -le 3 ]; do`: Loops while `count` is 3 or less.
- `count=$((count + 1))`: Adds 1 to `count`.

Analogy: Like counting until you reach a goal.

Extra Example:

```
#!/bin/bash
num=5
while [ $num -gt 0 ]; do
    echo "Countdown: $num"
    num=$((num - 1))
done
```

3.6 Process Management

1. **Example: Foreground Process**

```
ping google.com
```

What: Checks if `google.com` is reachable, locking the terminal.

Why: Tests internet connection.

How:

- `ping`: Sends packets to `google.com`.
- Stop with `Ctrl+C`.

Analogy: Like calling someone and waiting for their answer.

Extra Example:

```
ping localhost
```

2. **Example: Background Process**

```
ping google.com &
# Output: [1] 1234
```

What: Runs `ping` in the background, freeing the terminal.

Why: Runs tasks without locking the terminal.

How:

- `&`: Runs in background.
- Output: Job number (`[1]`) and process ID (`1234`).

Analogy: Like asking someone else to make the call.

Extra Example:

```
sleep 10 &
```

3. **Example: Jobs**

```
jobs
# Output: [1]+  Running  ping google.com &
```

What: Lists background jobs.

Why: Manages multiple tasks.

How:

- `jobs`: Shows running jobs.

Analogy: Like checking who's working in the background.

Extra Example:

```
sleep 5 &
jobs
```

4. **Example: Foreground Job**

```
fg %1
```

What: Brings job 1 to the foreground.

Why: Interacts with a background job.

How:

- `fg %1`: Moves job 1 to foreground.

Analogy: Like taking back a phone call.

Extra Example:

```
sleep 15 &
fg %1
```

5. **Example: Suspend and Resume**

```
ping google.com
# (Ctrl+Z)
# Output: [1]+  Stopped  ping google.com
bg %1
```

What: Pauses a job and resumes it in the background.

Why: Temporarily stops a task without ending it.

How:

- `Ctrl+Z`: Pauses job.
- `bg %1`: Resumes in background.

Analogy: Like putting a call on hold, then handing it off.

Extra Example:

```
sleep 20
# Ctrl+Z
bg %1
```

3.7 Text Processing Tools

1. **Example: jq**

```
echo '{"name": "Pikachu"}' | jq '.name'
# Output: "Pikachu"
```

What: Gets the `name` from JSON, like picking a label from a box.

Why: Processes API data (e.g., Pokémon API).

How:

- `echo '"name": "Pikachu"'`: Prints JSON.
- `|`: Sends to `jq`.
- `jq '.name'`: Extracts `name`.

Analogy: Like opening a package and reading the label.

Extra Example:

```
echo '{"type": "Electric"}' | jq '.type'
# Output: "Electric"
```

2. **Example: sed**

```
echo "hello world" | sed 's/hello/hi/'
# Output: hi world
```

What: Replaces "hello" with "hi".

Why: Edits text (e.g., configs).

How:

- `echo "hello world"`: Prints text.
- `sed 's/hello/hi/'`: Substitutes `hello` with `hi`.

Analogy: Like find-and-replace in a document.

Extra Example:

```
echo "I like tea" | sed 's/tea/coffee/'
# Output: I like coffee
```

3. **Example: awk**

```
echo -e "1\n2\n3" > numbers.txt
awk '{sum += $1} END {print sum}' numbers.txt
# Output: 6
```

What: Sums numbers in a file.

Why: Processes data (e.g., logs).

How:

- `echo -e "123"`: Writes three lines to `numbers.txt`.
- `awk 'sum += $1 END print sum'`: Adds first column, prints sum.

Analogy: Like a cashier totaling a receipt.

Extra Example:

```
echo -e "5\n10" > scores.txt
awk '{sum += $1} END {print sum}' scores.txt
# Output: 15
```

3.8 Error Handling

1. **Example: Exit Status**

```
ls fake_file
echo $?
# Output: 2
```

What: Checks if `ls` worked (non-zero means error).

Why: Detects command failures in scripts.

How:

- `ls fake`$_{file}$: $Tries to list a non-existent file.$`echo $?` : $Shows exit code (2 = error).$

  Analogy: Like checking if a delivery arrived.

  Extra Example:

```
ls test.txt
echo $?
# Output: 0 (success if test.txt exists)
```

2. **Example: Trap**

```
#!/bin/bash
trap 'echo "Error occurred!"; exit 1' ERR
ls fake_file
echo "This 'wont run"
```

What: Catches errors and stops the script.

Why: Prevents scripts from continuing after failures.

How:

- `trap 'echo "Error occurred!"; exit 1' ERR`: Runs on error.

- `ls fake`$_{file}$: $Fails, trigger strap.$`echo "This won't run"` : $Skipped.$

  Analogy: Like a safety switch stopping a machine.

  Extra Example:

```
#!/bin/bash
trap 'echo "Oops!"; exit 1' ERR
cat missing.txt
echo "Skipped"
```

3.9 Automation with Cron

Example:

```
#!/bin/bash
echo "Backup at $(date)" >> backup.log
```

And crontab:

```
0 0 * * * /home/username/shell_learning/backup.sh
```

What: Runs a script daily at midnight to log a backup.

Why: Automates tasks (e.g., backups).

How:

- Script:

- `echo "Backup at $(date)"`: Logs timestamp.
- `>> backup.log`: Appends to file.

- Crontab:
  - `crontab -e`: Opens cron editor.
  - `0 0 * * *`: Means "midnight every day".
  - Path: Full path to script.

Analogy: Like setting an alarm to do a task daily.

Extra Example:

```
#!/bin/bash
echo "Check at $(date)" >> check.log
```

Crontab:

```
0 12 * * * /home/ash/shell_learning/check.sh
```

Runs at noon.

4 Task Implementations

Now, we'll do the tasks (0–5), explaining every step and line like you're new.

4.1 Task 0: API Request Automation

Goal: Fetch Pikachu's data from https://pokeapi.co/api/v2/pokemon/pikachu, save to `data.json`, log errors to `errors.txt`.

Steps:

1. Open `nano apiAutomation-0x00`.
2. Write the script below.
3. Save and exit.
4. Make executable: `chmod u+x apiAutomation-0x00`.
5. Run: `./apiAutomation-0x00`.

Script:

```
#!/bin/bash
# Fetch 'Pikachus data
curl -s "https://pokeapi.co/api/v2/pokemon/pikachu" > data.json
# Check if curl worked
if [ $? -ne 0 ]; then
    echo "$(date): Failed to fetch Pikachu data" >> errors.txt
    exit 1
fi
# Check if data.json is empty
```

```
if [ ! -s data.json ]; then
    echo "$(date): Empty response from API" >> errors.txt
    exit 1
fi
echo "Data saved to data.json"
```

Explanation:

- `#!/bin/bash`: Uses Bash to run the script.
- `curl -s "https://..."`: Fetches data silently (`-s` hides progress). `>` saves to `data.json`.
- `if [ $? -ne 0 ]; then`: Checks if `curl` failed (`$?` is exit code, `-ne 0` means "not zero").
- `echo "$(date): Failed..."`: Logs error with timestamp.
- `exit 1`: Stops script on error.
- `if [ ! -s data.json ]; then`: Checks if `data.json` is empty (`! -s` means "not non-empty").
- `echo "Data saved..."`: Confirms success.

Verify:

```
cat data.json | jq . | head -n 50
```

4.2 Task 1: Extract Pokémon Data

Goal: Extract Pikachu's name, height, weight, type from `data.json`, print: "Pikachu is of type Electric, weighs 6kg, and is 0.4m tall."

Steps:

1. Open `nano` $data_extraction_automation - 0x01.Writescript$.
2. Save, exit.
3. Run: `chmod u+x` $data_extraction_automation - 0x01$; $./data_extraction_automation - 0x01$.

Script:

```
#!/bin/bash
# Extract data
name=$(jq -r '.name' data.json)
height=$(jq -r '.height' data.json)
weight=$(jq -r '.weight' data.json)
type=$(jq -r '.types[0].type.name' data.json)
# Convert units
height_m=$(echo "$height / 10" | bc -l)
weight_kg=$(echo "$weight / 10" | bc -l)
# Print
echo "$name is of type $type, weighs ${weight_kg}kg, and is ${
   height_m}m tall."
```

Explanation:

- `name=$(jq -r '.name' data.json)`: Gets `name` from JSON (`-r` removes quotes).
- `type=$(jq -r '.types[0].type.name')`: Gets first type's name.

- `height`$_m = \$(echo"\$height/10"|bc-l)$: $Divides height by 10 (decimeters to meters)$.`bc -l` $does math$.`echo` $Prints formatted sentence$.

4.3 Task 2: Batch Pokémon Data Retrieval

Goal: Fetch data for Bulbasaur, Ivysaur, Venusaur, Charmander, Charmeleon, save to `pokemon`$_data/ <pokemon> .json, with 2 - second delay$.

Steps:

1. Open nano batchProcessing-0x02.
2. Write script.
3. Save, exit.
4. Run: chmod u+x batchProcessing-0x02; ./batchProcessing-0x02.

Script:

```
#!/bin/bash
# Create folder
mkdir -p pokemon_data
# Pokémon list
pokemon_list=("bulbasaur" "ivysaur" "venusaur" "charmander" "
    charmeleon")
# Loop
for pokemon in "${pokemon_list[@]}"; do
    echo "Fetching data for $pokemon..."
    curl -s "https://pokeapi.co/api/v2/pokemon/$pokemon" > "
       pokemon_data/$pokemon.json"
    if [ $? -eq 0 ] && [ -s "pokemon_data/$pokemon.json" ]; then
        echo "Saved data to pokemon_data/$pokemon.json "
    else
        echo "$(date): Failed to fetch $pokemon data" >> errors.txt
    fi
    sleep 2
done
```

Explanation:

- `mkdir -p pokemon`$_data$: $Creates folder$(`-p` $avoids errors if exists$).`pokemon`$_list$ = (...) : $Stores Pokmon$

4.4 Task 3: Summarize Pokémon Data

Goal: Create `pokemon`$_report.csv with name, height, weight, and calculate averages$.

Steps:

1. Open nano summaryData-0x03.

2. Write script.

3. Save, exit.

4. Run: chmod u+x summaryData-0x03; ./summaryData-0x03.

Script:

```
#!/bin/bash
# CSV header
echo "Name,Height (m),Weight (kg)" > pokemon_report.csv
# Process files
for file in pokemon_data/*.json; do
    name=$(jq -r '.name' "$file")
    height=$(jq -r '.height' "$file")
    weight=$(jq -r '.weight' "$file")
    height_m=$(echo "$height / 10" | bc -l)
    weight_kg=$(echo "$weight / 10" | bc -l)
    echo "$name,$height_m,$weight_kg" >> pokemon_report.csv
done
# Averages
awk -F',' 'NR>1 {sum_h+=$2; sum_w+=$3; count++} END {print "Average
   Height: " sum_h/count " m"; print "Average Weight: " sum_w/count
   " kg"}' pokemon_report.csv
echo "CSV Report generated at: pokemon_report.csv"
cat pokemon_report.csv
```

Explanation:

- echo "Name,...": Writes CSV header.

- for file in pokemon$_{d}ata/*.json$: $LoopsthroughJSONfiles$.awk -F',' 'NR>1 ...' : $Skipsheader, sumsheight/weight, calculatesaverages.$

4.5 Task 4: Error Handling and Retry Logic

Goal: Modify Task 2 to retry failed requests 3 times.

Steps:

1. Open nano batchProcessing-0x02.

2. Write script.

3. Save, exit.

4. Run: chmod u+x batchProcessing-0x02; ./batchProcessing-0x02.

Script:

```
#!/bin/bash
# Retry function
fetch_pokemon() {
    local pokemon=$1
    local retries=3
    local count=0
```

```
    local success=0
    while [ $count -lt $retries ]; do
        curl -s "https://pokeapi.co/api/v2/pokemon/$pokemon" > "
           pokemon_data/$pokemon.json"
        if [ $? -eq 0 ] && [ -s "pokemon_data/$pokemon.json" ]; then
            success=1
            break
        fi
        count=$((count + 1))
        sleep 2
    done
    if [ $success -eq 1 ]; then
        echo "Saved data to pokemon_data/$pokemon.json  "
    else
        echo "$(date): Failed to fetch $pokemon data after $retries
           attempts" >> errors.txt
    fi
}
# Setup
mkdir -p pokemon_data
pokemon_list=("bulbasaur" "ivysaur" "venusaur" "charmander" "
   charmeleon")
# Loop
for pokemon in "${pokemon_list[@]}"; do
    echo "Fetching data for $pokemon..."
    fetch_pokemon "$pokemon"
    sleep 2
done
```

Explanation:

- fetch$_{p}okemon() : Functiontohandleretries.$local pokemon=$1 : TakesPokmonnameasargume
- while [$count -lt $retries]: Tries 3 times.
- break: Stops loop on success.

4.6 Task 5: Parallel Data Fetching

Goal: Fetch data in parallel using background processes.

Steps:

1. Open nano batchProcessing-0x04.
2. Write script.
3. Save, exit.
4. Run: chmod u+x batchProcessing-0x04; ./batchProcessing-0x04.

Script:

```
#!/bin/bash
# Retry function
```

```
fetch_pokemon() {
    local pokemon=$1
    local retries=3
    local count=0
    local success=0
    while [ $count -lt $retries ]; do
        curl -s "https://pokeapi.co/api/v2/pokemon/$pokemon" > "
           pokemon_data/$pokemon.json"
        if [ $? -eq 0 ] && [ -s "pokemon_data/$pokemon.json" ]; then
            success=1
            break
        fi
        count=$((count + 1))
        sleep 2
    done
    if [ $success -eq 1 ]; then
        echo "Saved data to pokemon_data/$pokemon.json  "
    else
        echo "$(date): Failed to fetch $pokemon data after $retries
           attempts" >> errors.txt
    fi
}
# Setup
mkdir -p pokemon_data
pokemon_list=("bulbasaur" "ivysaur" "venusaur" "charmander" "
   charmeleon")
# Parallel fetch
for pokemon in "${pokemon_list[@]}"; do
    echo "Fetching data for $pokemon..."
    fetch_pokemon "$pokemon" &
done
# Wait
wait
```

Explanation:

- fetch$_{p}okemon$"$pokemon"& : $Runsinbackground.$wait : $Waitsforalljobstofinish.$

5 Resources

Watch these videos:

- Bash Scripting for Beginners (freeCodeCamp)
- Linux Shell Scripting (TechWorld with Nana)

6 How to Create the PDF

1. Save this file as shell$_{s}cripting_{b}eginner_{t}utorial.tex.Compile$:

2. # 7 Conclusion

You've learned shell scripting from scratch, with every example explained like you're new. Save scripts to ALXprodev-Devops/Advanced$_shellonGitHub.Practiceandwatchthevideostomasters$