Debugging **Bash scripts** is essential for finding errors and understanding execution flow. Here are the most common and effective methods, along with illustrative script examples.

---

1. Shell Execution Options (Trace & Verbose)

These are the most fundamental built-in debugging tools, activated either at the command line or within the script using the set command.

A. Trace Execution (-x)

The **Trace** option (-x) is the most widely used debugging tool.

- **What it does:** It prints each command after it has undergone **expansion** (variables replaced with values, command substitutions executed) but **before** it executes. The executed command is prefixed with a + symbol, which indicates the debug output.
- **Activation:**
  - **Command Line:** bash -x your_script.sh
  - **In Script:** set -x (to start tracing) and set +x (to stop tracing).

Bash Script Example (-x)

Bash

```
#!/bin/bash

# Define variables
USER_LIST="user1 user2"
LOG_DIR="/tmp/logs"

# Start debugging for the critical section
set -x
echo "Processing users..."
for user in $USER_LIST; do
    echo "Current user is $user"
    mkdir -p "$LOG_DIR/$user"
done
set +x # Stop debugging

echo "Debugging complete."
```

Trace Output (Execution):

Bash

```
+ echo 'Processing users...'
Processing users...
+ for user in user1 user2
+ echo 'Current user is user1'
Current user is user1
+ mkdir -p /tmp/logs/user1
+ for user in user1 user2
+ echo 'Current user is user2'
Current user is user2
+ mkdir -p /tmp/logs/user2
+ echo 'Debugging complete.'
Debugging complete.
```

B. Verbose Execution (-v)

- **What it does:** It prints each shell input line as it is read, **before** the expansion of variables or commands.
- **Activation:**
  - **Command Line:** bash -v your_script.sh
  - **In Script:** set -v

---

2. Using Debugging Flags for Error Checking

These flags help catch common scripting mistakes before they lead to unexpected behavior.

A. No Execution (Syntax Check) (-n)

- **What it does:** Reads commands but does not execute them. This is useful for quickly checking a script for **syntax errors** without running any potentially harmful code.
- **Activation:** bash -n your_script.sh

B. Exit on Error (-e)

- **What it does:** Exits the script immediately if a command exits with a non-zero status (indicating failure). This is crucial for **pinpointing the exact line** where the script failed, as Bash normally continues executing even after a failure.

- **Activation:** set -e or bash -e your_script.sh

C. Unset Variables as Error (-u)

- **What it does:** Treats the use of any **unset variable** as an error. This is a powerful technique for catching **typos** in variable names.
- **Activation:** set -u or bash -u your_script.sh

Bash Script Example (Combined Flags)

Bash

```
#!/bin/bash -eux # Combines -e, -u, and -x

MY_NAME="Alice" # MY_NAME is set
echo "Hello, $MY_NAME"

# The following line will fail and exit the script because MY_NANE is unset
echo "My name is $MY_NANE"
```

Combined Output (Execution):

Bash

```
+ MY_NAME=Alice
+ echo 'Hello, Alice'
Hello, Alice
+ echo 'My name is '
./script.sh: line 7: MY_NANE: unbound variable
```

The script immediately exits on line 7 due to the -u flag.

---

3. Redirecting Trace Output (BASH_XTRACEFD)

When using the trace option (-x), the output is sent to standard error (stderr), which can clutter the normal output of your script. You can redirect the trace to a separate file.

- **What it does:** The special variable **BASH_XTRACEFD** is set to the file descriptor (FD) number where the debug trace should be written.

Bash Script Example (BASH_XTRACEFD)

Bash

```
#!/bin/bash

# 1. Open File Descriptor 3 and redirect it to a log file
exec 3> debug.log

# 2. Assign the file descriptor (3) to the XTRACEFD variable
BASH_XTRACEFD="3"

# 3. Enable the trace option
set -x
echo "This output goes to stdout"
echo "This trace goes to debug.log"
set +x

# 4. Close the file descriptor
exec 3>&-

echo "Finished. Check debug.log for the trace."
```

---

4. Manual Debugging (ECHO Statements)

The simplest method is strategically placing echo statements to inspect the state of the script at specific points.

- **Purpose:** To print the values of variables or to confirm that execution reached a certain line.

Bash Script Example (Manual Checks)

Bash

```bash
#!/bin/bash

COUNTER=1
MAX_COUNT=3

while [ "$COUNTER" -le "$MAX_COUNT" ]; do

    # Check 1: Print variable value before action
    echo "DEBUG: Starting loop iteration $COUNTER" 1>&2

    # ... commands ...

    COUNTER=$((COUNTER + 1))
done

# Check 2: Confirm successful end point
echo "DEBUG: Loop finished successfully." 1>&2
```

> **Tip:** Adding 1>&2 redirects the echo output to **standard error (stderr)**, ensuring your debug messages don't interfere with the script's intended standard output (stdout).