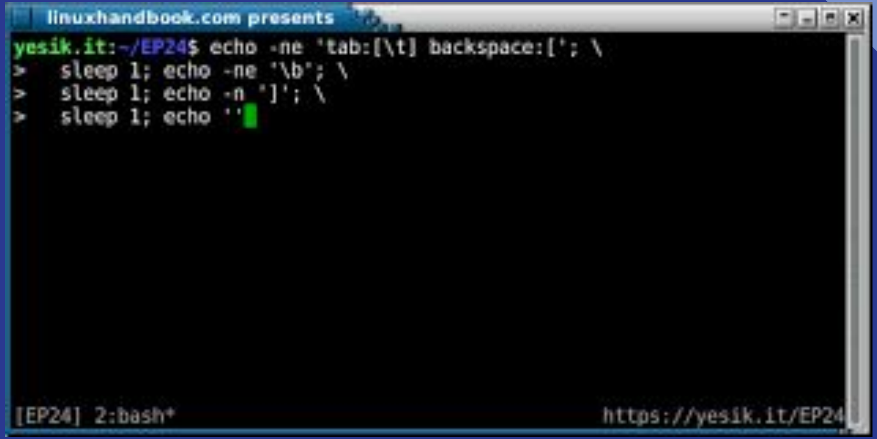
Bash Scripting!
Class 3



```
yesik.it:~/EP24$ echo -ne 'tab:[\t] backspace:['; \
>   sleep 1; echo -ne '\b'; \
>   sleep 1; echo -n ']'; \
>   sleep 1; echo ''
```

```
[EP24] 2:bash*                                          https://yesik.it/EP24
```

Class 1 Overview

- Core concepts, fundamentals, advantages
- Terminal basics, commands, CLI
- How to create and use variables
- Types of variables, special variables
- Operators - file, string, arithmetic, comparison

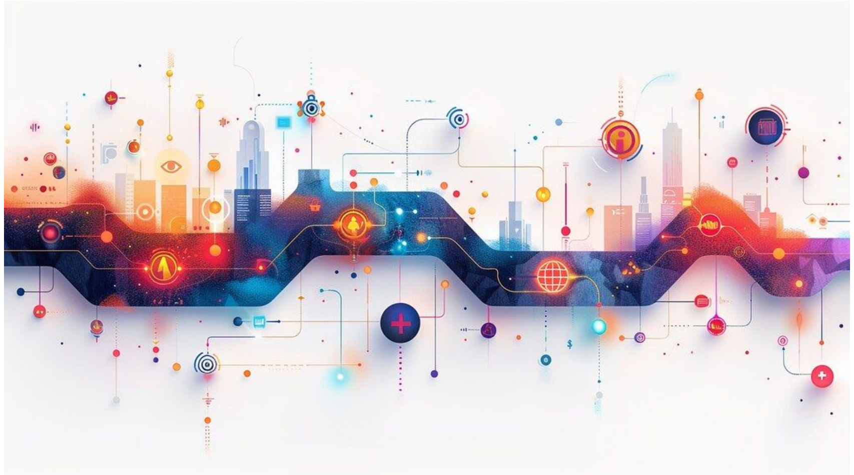


Image credit:
https://flickr.com

Class 2 Overview

- If statements, exit statuses, exit command
- Loops (for, while, until)
- Case statements
- For loops
- Many more
- Practice: Hands-on, code examples

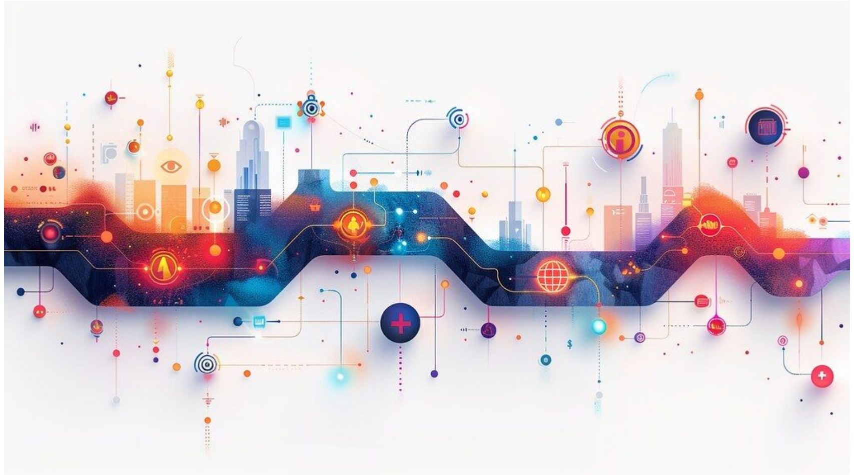


Image credit:
https://flickr.com

Class 3 Overview

- Functions
- Wildcards
- Redirection Operators
- Error handling
- Logging and debugging
- Practice: Hands-on exercises, code examples

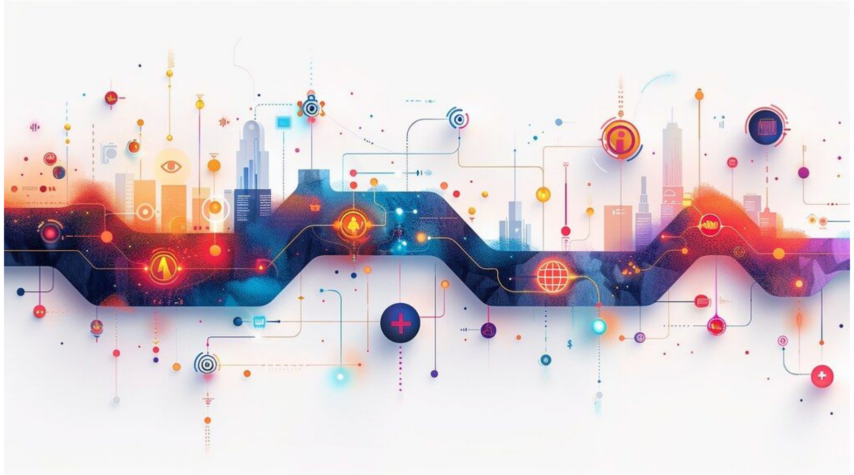


Image credit: https://flickr.com

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**

```bash
#!/bin/bash

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

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

hi
```

```bash
#!/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]*
    1. cat
    2. can
    3. catch
    4. candy
- **[!] - Matches any chars that are not included in the [ ]**
  - [!aeiou]*
    1. sky
    2. fly
    3. computer
    4. 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
```

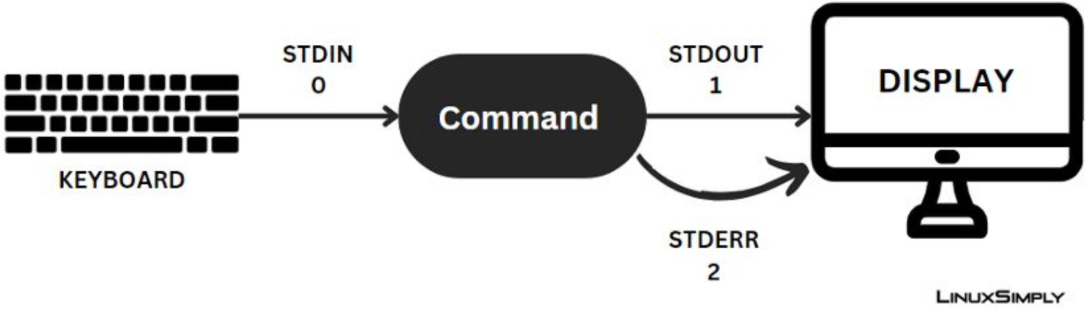
Case Statements - Examples (Real Case)

- 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.

Redirection Operator

In Linux, everything, including input and output, is treated as a file. The operating system assigns identifiers called file descriptors to represent these files. Each process can have a maximum of nine open file descriptors. In the Bash shell, the first three descriptors are reserved with specific IDs for these standard streams:

- **0 (represents STDIN)**
- **1 (represents STDOUT)**
- **2 (represents STDERR)**



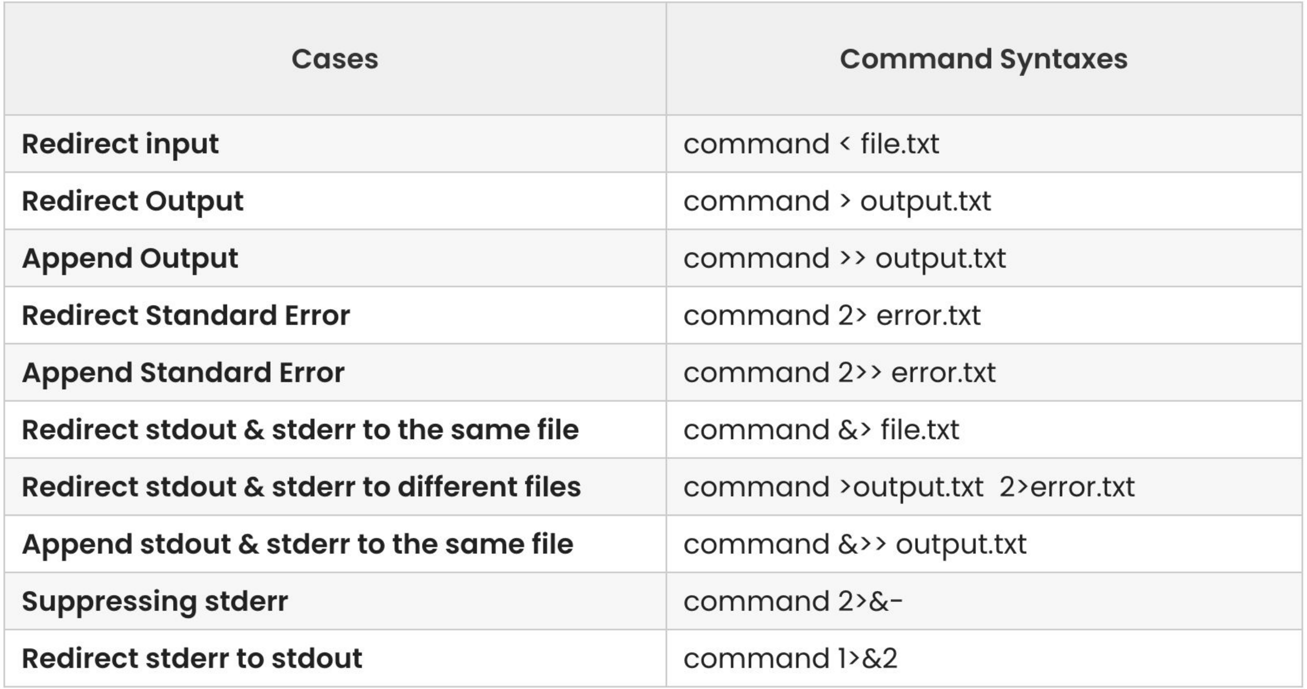
Credit: linuxsimply

Redirection is a process which allows us to change the default input source or output destination of a command.

Redirection Operator - Usages

- **Input Redirection**
  - cat < file.txt
- **Output Redirection**
  - whoami > file.txt
- **Appending redirected output**
  - whoami >> file.txt
  - cat file.txt
- **Std Error Redirection**
  - whoami -l 2> file.txt
  - Bash echo to Std Error - echo “The error message here!” >&2
  - Std Error to Stdout - command > file 2>&1
- **Redirecting Standard Output and Standard Error**
  - ls /nonexistent_directory &> log.txt
- **Appending Standard Output and Standard Error**
  - ls /nonexistent_directory >> log.txt 2>&1

Redirection Operator - Summary



Cases	Command Syntaxes
Redirect input	command < file.txt
Redirect Output	command > output.txt
Append Output	command >> output.txt
Redirect Standard Error	command 2> error.txt
Append Standard Error	command 2>> error.txt
Redirect stdout & stderr to the same file	command &> file.txt
Redirect stdout & stderr to different files	command >output.txt 2>error.txt
Append stdout & stderr to the same file	command &>> output.txt
Suppressing stderr	command 2>&-
Redirect stderr to stdout	command 1>&2

Source: https://linuxsimply.com/bash-scripting-tutorial/redirection-and-piping/redirection/

Logging

- **What and why?**
  - The process of recording events, activities, or messages generated by a system
  - That may scroll off the screen
  - Who, what, when, where, why something
  - Script may run via cron and others
- **Syslog Standard**
  - Syslog is a standard logging mechanism used in Unix-like operating systems to collect, process, and store log messages generated by various components of the system
  - Centralized Logging
  - Severity Levels - ranging from "emergency" to "debug", elert, warning,
  - Facilities - "auth" for authentication-related messages, "mail" for mail server messages, and "kernel" for messages from the operating system kernel
  - Configurability - Log file location are configurable - var/log/messages, var/log/syslog
- **Generating log messages**
- **Custom logging functions**

Logging - Examples

```
#!/bin/bash

logger "Message"
logger -p local0.info "Message"
logger -s -p local0.info "Message"
logger -t myscript -p local0.info "Message"
logger -i -t myscript "Message"
```

Output:

```
Mar  3 19:06:12  root[55028] <Info>: Message
```

- This sends a log message with the content **"Message"** to the syslog daemon. By default, the message is logged with the facility "user" and severity level "notice".
- **-p local0.info** specifies the facility as "local0" and severity level as "info".
- **-s** logs the message to the system console.
- **-t myscript** tags the message with the program name "myscript".
- **-i** Log the message with the process ID included.

Logging - Examples

```
VERBOSE=false
HOST="amazon.com"
PID="$$"
PROGRAM_NAME="$0"
MY_HOST=$(hostname)

logit () {
  local LOG_LEVEL=$1
  shift
  MSG=$@
  TIMESTAMP=$(date +"%Y-%m-%d %T")
  if [ $LOG_LEVEL = 'ERROR' ] || $VERBOSE
  then
    echo "${TIMESTAMP} ${MY_HOST} ${PROGRAM_NAME}[${PID}]: ${LOG_LEVEL} ${MSG}"
  fi
}

logit INFO "Processing data."

fetch-data $HOST || logit ERROR "Could not fetch data from $HOST"
```

Output:

```
2024-03-03 21:31:36 BS847s-MacBook-Pro.local
/bin/zsh[45508]: ERROR Could not fetch data from
amazon.com
```

Logging - Examples

- VERBOSE=false: A variable indicating whether verbose logging is enabled or not. By default, it is set to false.
- HOST="google.com": A variable storing the hostname.
- PID="$$": A variable storing the process ID of the script.
- PROGRAM_NAME="$0": A variable storing the name of the script.
- THIS_HOST=$(hostname): A variable storing the hostname of the current machine.
- logit () { ... }: Definition of the logit function, which takes two arguments: LOG_LEVEL and MSG. Inside the function:
  - TIMESTAMP=$(date +"%Y-%m-%d %T"): Variable storing the current timestamp in the format "YYYY-MM-DD HH:MM:SS".
  - The function checks whether the LOG_LEVEL is 'ERROR' or VERBOSE is true. If it is an error message or VERBOSE is true, the log message is printed.
  - The log message includes the timestamp, hostname, script name, process ID, log level, and message.
- logit INFO "Processing data.": Calls the logit function with log level INFO and the message "Processing data.".
- fetch-data $HOST || logit ERROR "Could not fetch data from $HOST": Calls the fetch-data function with the hostname stored in HOST. If the fetch-data function fails (returns a non-zero exit status), it logs an error message.

Debugging

Several reasons why debugging is essential:

- **Identifying Errors**
- **Examine the inner working of the script**
- **Find out the root cause of unexpected behaviour**
- **Fixing bugs**
- **Testing and Validation**

Debugging

```
#!/bin/bash

# Enable debugging mode
bash -x script.sh

# Or, within the script itself
set -x

# Your script commands here

# Disable debugging mode
set +x
```

- The -x option (or -v) is used to enable debugging mode. When this option is set, Bash displays each command before executing it, allowing you to trace the execution flow and identify any issues in the script.
- Called as **x-trace, tracing,** or **print debugging**

Debugging - Example

```
# Enable debugging mode
set -x

factorial() {
    local n=$1
    local result=1

    # Iterate from 1 to n and calculate the factorial
    for ((i = 1; i <= n; i++)); do
        result=$((result * i))
    done

    echo "Factorial of $n is $result"
}

factorial 5

# Disable debugging mode
set +x
```

- set -x: Enables debugging mode, which causes Bash to display each command before executing
- factorial() function: Defines a function to calculate the factorial of a number.
- Within the function:
    - local variables are used to ensure variable scope.
    - A for loop calculates the factorial of the input number.
    - An echo statement displays the result.
- factorial 5: Calls the factorial function with the argument 5 to calculate the factorial of 5.
- set +x: Disables debugging mode

Debugging - (Built-in)

- **-e**: This option, when set, ensures that the script exits immediately if any command returns a non-zero exit status. It's often used to enforce strict error handling in scripts.

- **-x**: This option, when set, enables debugging mode, where Bash displays each command before executing it. It's useful for tracing the execution flow of the script and identifying any issues.

- **-v**: This option, when set, displays shell input lines as they are read, useful for debugging interactive sessions or scripts.

- **-ex or -xe**: These combinations of options enable both -e (exit on error) and -x (debugging mode) simultaneously. It's commonly used for debugging scripts where immediate exit on error and command tracing are desired.

Thank you!