Scripting:

* The line starts with a “shebang”, the pair of characters #!  
  Example:  
  #!/bin/bash
* read – reads one line from the standard input  
  read MESSAGE  
  waits for a line from input; one types:  
  hello  
  and value hello is stored in a shell variable (“MESSAGE”)
* read –d : MESSAGE (reads up to : instead of newline)
* read –n 100 MESSAGE (reads for no more than 100 characters)
* read –t 10 MESSAGE (waits for 10 sec for input to be entered)
* Example: create, read and remove a shell variable:  
  MY\_VAR=massimo  
  echo $MY\_VAR  
  massimo  
  unset MY\_VAR
* echo -e "Your username is: **`whoami`** \n" will display the result of whoami command
* To execute any executable in terminal use: ./filename
* A pipeline is the sequence of one or more commands separated by the character | (pipe, or vertical bar)  
  Example:  
  ls –l | wc –l  
  tells you how many files and directories are in the current directory (NB: plus, one, since ls –l produces an extra line)
* Redirection: Standard input, output and error can also be redirected to files  
  Examples:  
  – ls –l > l.txt  
  saves the file listing in long format to file l.txt; if file exists, overwrites it  
  – cat < l.txt  
  redirects the input of cat to come from file l.txt  
  – ls –l >> l.txt  
  appends the file listing in long format to existing file l.txt; if file doesn’t exits creates it.
* The shell has several special parameters. These parameters are automatically assigned and can only be read. To read their value, use the $ sign in front of their name:  
  \* expands to the positional parameters with which the script (or a  
  new shell) are called (i.e. the script’s arguments) as a single  
  string. The separator (default: a space) can be chosen  
  1 - 9 each expands to one positional parameter (more than nine is  
  possible)  
  @ expands to the positional parameters, as separate strings  
  # expands to the number of positional parameters  
  ? expands to the exit status of the most recently executed command  
  $ expands to the process ID of the script (or shell)  
  0 expands to the name of the script (or shell)
* Operator $(( )) evaluates an arithmetic expression and replaces it with its value
* Quoting is a mechanism to remove the special meaning of certain characters or words to the shell
* Characters | & ; ( ) < > space tab ! and others must be quoted to remove their special meaning and use them literally.
* Three quoting mechanisms exist: the escape character, single quotes, and double quotes.
* The escape character (\ aka backslash) preserves the literal value of the next character  
  Examples: \|, \&, \\
* Single quotes preserve the literal value of each character within the quotes. Cannot be nested. Example: 'this is the backslash \'
* Double quotes are similar, but some characters ($, `  
  and \) retain their special value. Example: "The value of variable A is: $A"

|  |  |  |  |
| --- | --- | --- | --- |
| **Command** | **One-line Explanation** | **Example** | **Options** |
| awk | awk is a programming language designed to search for, match patterns, and perform actions on files. awk programs are generally quite small, and are interpreted – great for tabular data | awk '/pattern/ { print $1 }' file.txt | Selected features: ● Reads input file one record at a time and splits into fields ● $n field values of the current record ● $NR number of records ● $NF number of fields in current record ● Some support for formatted output ● Supports + & ? in RE's (same as Perl) ● Much more |
| grep | Command-line utility for searching plain-text data sets for lines that match a regular expression. | grep 'keyword' file.txt | -c print only matching lines -i ignore uppercase/lowercase differences -h don't show file names -l only give file names when match is found -n display line numbers -v print lines except those that match |
| sed | Edits the input based on regular expressions – great for text replacement | sed 's/search/replace/g' file.txt | -n no print on output -e editing command (not required) -f sed script filename |
| sort | Command-line utility for sorting lines of text files. | sort file.txt | -r, -n, -k |
| uniq | Tool for filtering adjacent matching lines in a file and suppressing duplicates. | uniq file.txt | display only the lines that are not repeated -d display only repeated lines and one copy -c print only a count of the each line occurrence -fn ignore first n fields +N same as -fN |
| join | Command for joining two files based on a common field. | join file1.txt file2.txt | -an display unmatched pairs from n-th file -o n,m m-th field of n-th file -j n m m-th field of n-th file will be the join field -t field separator |
| paste | Utility to merge lines of files. | paste file1.txt file2.txt | -d field delimiter (separator) -s merge lines instead of paste row by row - use standard input instead of file(s) |
| split | Command to split a file into pieces. | split -l 100 file.txt | -b, -l, -d |
| cut | Command for extracting specific columns or fields from files. | cut -d',' -f1,3 file.csv | -k specify start & end fields for sort & sort type -c Test if is already sorted -m Merge two sorted files -u Delete all duplicate lines (unique sort) -o Output file name |
| tr | Translates by substitution or deletion and use to strip control characters and change case | tr -c -d -s "fromthis" "tothis” | -c replace characters in from this with a complement -d delete all input characters in from string -s removes all but first (squeezes) repeating chars |

Conditional statements in Bash scripting provide the ability to control the flow of execution based on certain conditions. There are five primary types of conditional statements used in Bash:

* if statement:This statement executes a block of code if a specified condition is true.

```bash

if [ expression ]

then

statement

fi

```

* if-else statement: Executes one block of code if the condition is true and another block if it is false.

```bash

if [ expression ]

then

statement1

else

statement2

fi

```

* if..elif..else..fi statement (Else If ladder):Allows multiple conditions to be evaluated sequentially. If one condition is true, it executes its block of code and skips the rest.

```bash

if [ expression1 ]

then

statement1

elif [ expression2 ]

then

statement3

else

statement4

fi

```

* if..then..else..if..then..fi..fi.. (Nested if):Used for complex conditions where one set of conditions leads to another set of conditions.

```bash

if [ expression1 ]

then

statement1

else

if [ expression2 ]

then

statement3

fi

fi

```

* switch statement: Works similarly to switch-case in other programming languages, allowing multiple conditions to be evaluated against a single variable.

```bash

case $variable in

Pattern1) statement1 ;;

Pattern2) statement2 ;;

\*) default\_statement ;;

esac

```

In conclusion, understanding and using these conditional statements effectively in Bash scripting allows for creating scripts that can make decisions based on various conditions, enhancing script functionality and automation capabilities.

Functions in scripting languages, such as Bash or Python, are essential for organizing code into reusable blocks. They are defined using a syntax that includes the function name, parameters (optional), and the body enclosed within curly braces `{}`.

\*\*Example (Bash)\*\*:

```bash

# Function definition

my\_function() {

echo "Hello, world!"

}

```

Function with parameters

greet() {

local name="$1" # First parameter

echo "Hello, $name!"

}

# Calling the function

greet "Alice"

```

Function returning a value

add() {

local num1="$1"

local num2="$2"

local sum=$((num1 + num2))

echo "$sum"

}

# Calling the function and capturing its output

result=$(add 5 3)

echo "Sum: $result"

```

Scope:

# Global variable

global\_var="I'm global"

# Function with local variable

local\_var\_func() {

local local\_var="I'm local"

echo "$local\_var"

}

# Accessing variables

echo "$global\_var"

local\_var\_func

```

Factorial function using recursion

factorial() {

local n="$1"

if [ "$n" -eq 0 ]; then

echo 1

else

local previous=$(factorial $((n - 1)))

echo $((n \* previous))

fi

}

# Calling the factorial function

result=$(factorial 5)

echo "Factorial of 5 is: $result"

```

Functions in scripting languages enhance code modularity, readability, and reusability, making them crucial for writing efficient and maintainable scripts.

# Shell Scripting – Standard Input, Output and Error:

A diagram of a software application

Description automatically generated

In shell scripting, managing standard input (stdin), standard output (stdout), and standard error (stderr) is crucial for handling data streams and error messages effectively. Here’s a breakdown of each:

* Standard Input (stdin): Standard input (`stdin`) is the default input stream where a program reads data from by default. Used to accept user input or data piped from another command.

File Descriptor: `0`

Reading input from the user or a file.

```bash

# Read from user input

echo "Enter your name:"

read name

echo "Hello, $name!"

# Read from a file

while read line; do

echo "Line: $line"

done < input.txt

```

* Standard Output (stdout): Standard output (`stdout`) is the default output stream where a program writes normal output. Used to display results, messages, or data to the user or redirect to a file.

File Descriptor: `1`

Example: Printing output to the console or redirecting to a file.

```bash

# Print to stdout

echo "This is a message"

# Redirect stdout to a file

echo "Output to file" > output.txt

# Append to a file

echo "More output" >> output.txt

```

* Standard Error (stderr): Standard error (`stderr`) is the output stream where a program writes error messages and diagnostics. Used to report errors separately from normal output to avoid mixing error messages with regular data.

File Descriptor: `2`

Example: Writing error messages to stderr.

```bash

# Print error message to stderr

echo "Error: File not found" >&2

# Redirect stderr to a file

command\_not\_exist 2> error.log

```

Link: For A-Z Linux commands

<https://ss64.com/bash/>