Category	Sub-Category	Concept/Operator	Description
I. Script Fundamentals	Shebang Line	#!/bin/bash or #!/usr/bin/env bash	The very first line of a script, specifying the interpreter to execute it. Ensures your script runs with Bash, not a different shell. env bash offers slightly better portability by letting env find bash in the PATH.
	Execution	chmod +x script.sh, ./script.sh	After creating a script, you must make it executable using chmod +x. Then, run it using its path (e.g., ./script.sh from the current directory).
	Comments	#	Used to add single-line comments within a script. Anything after # on a line is ignored by the interpreter. Essential for explaining code and improving readability.
II. Operators & Conditions	File Test Operators	-a file (exists - deprecated)	True if file exists.
		-b file (block special)	True if file exists and is a block special file (e.g., a disk device).
		-c file (character special)	True if file exists and is a character special file (e.g., a serial port).
		-d file (directory)	True if file exists and is a directory.
		-e file (exists - preferred)	True if file exists, regardless of its type. This is the most common way to check for file existence.
		-f file (regular file)	True if file exists and is a regular file (not a directory, symbolic link, or special file).
		-g file (SGID set)	True if file exists and has its set-group-id (SGID) bit set.
		-h file or -L file (symbolic link)	True if file exists and is a symbolic link.
		-k file (sticky bit set)	True if file exists and has its "sticky" bit set.
		-p file (named pipe/FIFO)	True if file exists and is a named pipe (FIFO).
		-r file (readable)	True if file exists and is readable by the effective user ID.
		-s file (not empty)	True if file exists and has a size greater than zero (i.e., it's not an empty file).
		-S file (socket)	True if file exists and is a socket.
		-t fd (terminal)	True if file descriptor fd is open and refers to a terminal (e.g., [-t 0] checks if standard input is a terminal).
		-u file (SUID set)	True if file exists and has its set-user-id (SUID) bit set.
		-w file (writable)	True if file exists and is writable by the effective user ID.
		-x file (executable)	True if file exists and is executable by the effective user ID.
		-O file (owned by effective user)	True if file exists and is owned by the effective user ID.
		-G file (owned by effective group)	True if file exists and is owned by the effective group ID.
		-N file (modified since last read)	True if file exists and has been modified since it was last read.
		file1 -nt file2 (newer than)	True if file1 is newer (according to modification date) than file2, or if file1 exists and file2 does not.
		file1 -ot file2 (older than)	True if file1 is older than file2, or if file2 exists and file1 does not.
		file1 -ef file2 (same inode/device)	True if file1 and file2 refer to the same device and inode numbers (i.e., they are hard links to the same file).

	String Comparison Operators	string1 = string2 or string1 == string2	True if string1 is equal to string2. == is Bash-specific, = is more portable.
		string1 != string2	True if string1 is not equal to string2.
		-z string (zero length)	True if the length of string is zero (i.e., it's an empty string).
		-n string (non-zero length)	True if the length of string is non-zero (i.e., it's not an empty string).
		string1 < string2 (lexicographical less than)	True if string1 sorts before string2 lexicographically. Requires [[ ]] or escaped \< in [ ].
		string1 > string2 (lexicographical greater than)	True if string1 sorts after string2 lexicographically. Requires [[ ]] or escaped \> in [ ].
	Integer Comparison Operators	num1 -eq num2 (equal)	True if num1 is equal to num2.
		num1 -ne num2 (not equal)	True if num1 is not equal to num2.
		num1 -gt num2 (greater than)	True if num1 is greater than num2.
		num1 -ge num2 (greater than or equal)	True if num1 is greater than or equal to num2.
		num1 -lt num2 (less than)	True if num1 is less than num2.
		num1 -le num2 (less than or equal)	True if num1 is less than or equal to num2.
	Logical Operators	&& (Logical AND)	Executes the command/condition on the right only if the command/condition on the left succeeds (returns exit status 0). Can be used directly in if statements or [[ ]].
		`	` (Logical OR)
		! (Logical NOT)	Reverses the exit status of a command or condition.
		-a (Logical AND - obsolescent in [ ])	Binary operator for logical AND within [ ]. Considered obsolescent; prefer && outside [ ] or within [[ ]].
		-o (Logical OR - obsolescent in [ ])	Binary operator for logical OR within [ ]. Considered obsolescent; prefer `
	Test Commands	[ ] (test command)	A POSIX-compliant command that evaluates conditions. Requires careful quoting of variables ("$VAR"). Spaces are mandatory around operators and variables. More portable.
		[[ ]] (extended test command)	Bash-specific (also Ksh, Zsh) extension. Offers more features like pattern matching, doesn't always require strict quoting of variables, and allows &&, `
		(( )) (arithmetic evaluation)	Bash-specific construct for integer arithmetic and comparison. Allows C-style comparison operators (==, !=, >, >=, <, <=). No need for variables to be quoted within this context.
III. Variables & Input	Variable Definition	VARIABLE_NAME="value"	Defines a variable. No spaces around the = sign. Variable names are typically uppercase by convention.
	Accessing Variables	$VARIABLE_NAME or ${VARIABLE_NAME}	Retrieves the value of a variable. Always quote variables ("$VARIABLE_NAME") when using them to prevent unexpected word splitting or globbing issues.
	Local Variables	local variable="value"	Declares a variable within a function to be local to that function's scope. Prevents unintended modification of global variables.

	User Input (read)	read -p "Prompt: " VAR	Prompts the user for input and stores it in the specified variable. Options like -p (prompt), -r (raw input, no backslash interpretation), -s (silent for passwords), -n N (read N chars), -t SECONDS (timeout).
	Positional Parameters	$0, $1, $2, ...	$0 is the script's name. $1, $2, etc., are the first, second, and subsequent arguments passed to the script.
	All Arguments	$@	Expands to all command-line arguments as separate words. Crucial for iterating over arguments correctly, especially when they contain spaces. Use for arg in "$@"; do ...
	Number of Arguments	$#	Returns the total number of command-line arguments passed to the script.
IV. Control Flow	If/Elif/Else	if condition; then ... elif condition; then ... else ... fi	Executes a block of code if a condition is true. elif (else if) allows for multiple conditions. else provides a fallback if no conditions are met.
	Case Statement	case "$VAR" in "pattern") ... ;; *) ... ;; esac	A cleaner way to handle multiple conditional branches based on the value of a variable. Patterns can include wildcards. ;; terminates each case. *) is the default case.
	For Loop	for item in list; do ... done	Iterates over a list of items (words, files, etc.). Each item is assigned to the loop variable in turn. Also supports C-style for ((i=0; i<N; i++)); do ... done.
	While Loop	while condition; do ... done	Executes a block of code repeatedly as long as a specified condition remains true. Useful for continuous monitoring or processing until a criterion is met.
	Until Loop	until condition; do ... done	Executes a block of code repeatedly until a specified condition becomes true. Essentially the inverse of while.
V. Functions	Function Definition	function_name () { ... } or function function_name { ... }	Encapsulates reusable blocks of code. Improves script organization and modularity.
	Function Arguments	$1, $2, ... within function	Inside a function, $1, $2, etc., refer to the arguments passed to that specific function, distinct from the script's command-line arguments.
	Return Status	return N	Sets the exit status of the function. 0 for success, non-zero for failure. Can be checked with $? after the function call.
VI. Error Handling & Debugging	Exit Status	$?	Stores the exit status of the last executed command. 0 indicates success, any non-zero value indicates failure.
	Strict Mode (set -euo pipefail)	set -e	Exits the script immediately if any command returns a non-zero exit status (fails). Prevents script from continuing with an error.
		set -u or set -o nounset	Exits the script if an attempt is made to use an unset variable. Helps catch typos and uninitialized variables.
		set -o pipefail	Ensures that the exit status of a pipeline (`command1
	Debugging Output	echo "Message"	Prints messages to standard output, useful for tracking script flow and variable values during debugging.
	Trace Mode (set -x)	set -x / set +x	Turns on (and off) "xtrace" mode, which prints each command and its arguments to standard error after parameter expansion and before execution. Invaluable for seeing exactly what your script is doing step-by-step.
	Syntax Check (bash -n)	bash -n your_script.sh	Performs a syntax check of the script without executing any commands. Useful for catching basic syntax errors before running.
	Static Analysis (shellcheck)	shellcheck your_script.sh	An external tool that provides static analysis for shell scripts, identifying common pitfalls, syntax errors, and style issues. Highly recommended for robust script development.
VII. Input/Output Redirection	Standard Output	> (overwrite), >> (append)	Redirects the standard output (stdout) of a command to a file. > overwrites the file if it exists, >> appends to it.

	Standard Input	< and << (redirect until EOF) <<< passes string as input	Redirects the content of a file to be used as standard input (stdin) for a command.
	Standard Error	2> (overwrite), 2>> (append)	Redirects the standard error (stderr) of a command to a file. 2> overwrites, 2>> appends. >&2 redirects stdout to stderr.
	Both Output/Error	&> (overwrite), &>> (append)	Bash-specific shorthand to redirect both standard output and standard error to a file.
	Pipes	|	Sends the output of a command to another
VIII. Quoting Mechanisms	Backslash	\	Escapes the next single character, telling the shell to treat it literally and disabling its special meaning (e.g., \$ for a literal dollar sign, \ for a literal space). Used to escape specific characters, often within double quotes.
	Single Quotes	'	Preserves the literal value of every character enclosed within them. No special interpretation (variable expansion, command substitution, globbing, etc.) occurs inside single quotes. Ideal for literal strings or regular expressions.
	Double Quotes	""	Preserves the literal value of most characters, but allows variable expansion ($VAR), command substitution ($(command)), and arithmetic expansion ($((...))). It disables word splitting and globbing, making it the recommended way to quote variables to prevent unexpected behavior with spaces or wildcards in variable values.
IX. General Best Practices	Quoting Variables	"$VARIABLE"	Always use double quotes around variable expansions ("$VAR") unless you have a very specific reason not to (and understand the implications). This prevents unexpected word splitting (splitting a single variable into multiple arguments) and globbing (interpreting wildcards like *).
	Spacing	[-f "$FILE"] (spaces are mandatory)	Spaces are critical around operators and arguments within [ ] and [[ ]] expressions. Incorrect spacing will lead to syntax errors.
	Read man pages	man command_name	The primary resource for detailed information on any command or utility in Linux/Unix.
	Incremental Development	Write small, test often.	Build scripts incrementally, testing each piece as you go, rather than writing a large script and debugging it all at once.
	Version Control	Use Git for tracking changes.	Especially for non-trivial scripts, using a version control system like Git helps manage changes, revert to previous versions, and collaborate.
X. Character Patterns & Regex	Anchors (Basic/Extended Regex)	^ (Start of line) , [!negates] $ (End of line)	^: Matches the beginning of a line., [^abc] negates the characters to not be included $: Matches the end of a line.
	Wildcards (Shell Globbing)	* (Zero or more characters) ? (Exactly one character) [] (Character range/set)	*: Matches zero or more of any character. ?: Matches exactly one of any character. []: Matches any single character within the brackets (e.g., [abc], [0-9], [a-zA-Z]). [^] negates the set (e.g., [^0-9] matches non-digits).
	Quantifiers (Extended Regex)	* (Zero or more of preceding) + (One or more of preceding) ? (Zero or one of preceding) {n} (Exactly n of preceding) {n,} (n or more of preceding) {n,m} (n to m of preceding)	*: Matches the preceding item zero or more times (e.g., a* matches "", "a", "aa", etc.). +: Matches the preceding item one or more times (e.g., a+ matches "a", "aa", etc.). ?: Matches the preceding item zero or one time (e.g., a? matches "" or "a"). {n}: Matches exactly n occurrences of the preceding item. {n,}: Matches n or more occurrences. {n,m}: Matches between n and m occurrences (inclusive).

	Character Classes (Extended Regex)	. (Any single character) \d (Digit) \w (Word character) \s (Whitespace character) \b (Word boundary) [:alnum:] (Alphanumeric) [:alpha:] (Alphabetic) [:digit:] (Digits) [:space:] (Whitespace)	.: Matches any single character (except newline). \d: Matches any digit (0-9). \w: Matches any word character (alphanumeric and underscore). \s: Matches any whitespace character. \b: Matches a word boundary (e.g., \bword\b matches "word" as a whole word). POSIX character classes like [:alnum:], [:alpha:], [:digit:], [:space:] are also available.
	Grouping & Alternation (Extended Regex)	() (Grouping) `	` (OR / Alternation)
	Escaping Regex Metacharacters	\ (Escape character)	Within a regular expression pattern, a backslash \ is used to escape a metacharacter (a character with special meaning in regex) to treat it as a literal character. For example, \. matches a literal dot, * matches a literal asterisk. This is distinct from shell escaping.