Bash Reference Sheet

Contents

Syntax

- *[word]* `[space]` *[word]*

  Spaces separate words. In bash, a *word* is a group of characters that belongs together. Examples are command names and arguments to commands. To put spaces inside an argument (or *word*), quote the argument (see next point) with single or double quotes.

- *[command]* `;` *[command]* `[newline]`

  Semi-colons and newlines separate synchronous commands from each other. Use a semi-colon *or* a new line to end a command and begin a new one. The first command will be executed synchronously, which means that Bash will wait for it to end before running the next command.

- *[command]* & *[command]*

  A single ampersand terminates an asynchronous command. An ampersand does the same thing as a semicolon or newline in that it indicates the end of a command, but it causes Bash to execute the command asynchronously. That means Bash will run it in the background and run the next command immediately after, without waiting for the former to end. Only the command before the `&` is executed asynchronously and you must not put a `;` after the `&`, the `&` replaces the `;`.

- *[command]* | *[command]*

  A vertical line or `pipe`-symbol connects the output of one command to the input of the next. Any characters streamed by the first command on `stdout` will be readable by the second command on `stdin`.

- *[command]* && *[command]*

  An *AND* conditional causes the second command to be executed only if the first command ends and exits successfully.

- *[command]* || *[command]*

  An *OR* conditional causes the second command to be executed only if the first command ends and exits with a failure exit code (any non-zero exit code).

- ' *[Single quoted string]* '

  Disables syntactical meaning of all characters inside the string. Whenever you want literal strings in your code, it's good practice to wrap them in single quotes so you don't run the risk of accidentally using a character that also has a syntactical meaning to Bash.

- " *[Double quoted string]* "

  Disables syntactical meaning of all characters except expansions inside the string. Use this form instead of single quotes if you need to expand a parameter or command substitution into your string.

  Remember: It's important to always wrap your expansions (`"$var"` or `"$(command)"`) in double quotes. This will, in turn, safely disable meaning of syntactical characters that may occur inside the expanded result.

Basic Structures

See BashSheet#Examples:_Basic_Structures for some examples of the syntax below.

Compound Commands

Compound commands are statements that can execute several commands but are considered as a sort of command group by Bash.

Command Lists

- { *[command list]*; }

  Execute the list of commands in the current shell as though they were one command.

  Command grouping on its own isn't very useful. However, it comes into play wherever Bash syntax accepts only one command while you need to execute multiple. For example, you may want to pass output of multiple commands via a pipe to another command's input:

  ```
  { cmd1; cmd2; } | cmd3
  ```

  Or you may want to execute multiple commands after a || operator:

  ```
  rm file || { echo "Removal failed, aborting."; exit 1; }
  ```

  It is also used for function bodies. Technically, this can also be used for loop bodies though this is **undocumented**, **not portable** and we normally prefer `do ...; done` for this):

  ```
  for digit in 1 9 7; { echo "$digit"; }       # non-portable, undocumented, unsupported
  for digit in 1 9 7; do echo "$digit"; done   # preferred
  ```

  Note: You **need** a `;` before the closing `}` (or it must be on a new line).

- (*[command list]*)

  Execute the list of commands in a subshell.

  This is exactly the same thing as the command grouping above, only, the commands are executed in a subshell. Any code that affects the environment such as variable assignments, `cd`, `export`, etc. do not affect the main script's environment but are scoped within the brackets.

  Note: You **do not** need a `;` before the closing `)`.

Expressions

- ((*[arithmetic expression]*))

Evaluates the given *expression* in an arithmetic context.

That means, strings are considered names of integer variables, all operators are considered arithmetic operators (such as `++`, `==`, `>`, `<=`, etc..) You should always use this for performing tests on numbers!

- `$((` *[arithmetic expression]* `))`

  Expands the result of the given *expression* in an arithmetic context.

  This syntax is similar to the previous, but expands into the result of the expansion. We use it inside other commands when we want the result of the arithmetic expression to become part of another command.

- `[[` *[test expression]* `]]`

  Evaluates the given *expression* as a `test`-compatible expression.

  All `test` operators are supported but you can also perform *Glob pattern matching* and several other more advanced tests. It is good to note that word splitting will **not** take place on unquoted parameter expansions here. You should always use this for performing tests on strings and filenames!

Loops

If you're new to loops or are looking for more details, explanation and/or examples of their usage, go read the BashGuide's section on Conditional Loops.

- `do` *[command list]*`; done`

  This constitutes the actual loop that is used by the next few commands.
  The list of commands between the `do` and `done` are the commands that will be executed in every iteration of the loop.

- `for` *[name]* `in` *[words]*

  The next loop will iterate over each *WORD* after the `in` keyword.
  The loop's commands will be executed with the value of the variable denoted by `name` set to the word.

- `for ((` *[arithmetic expression]*`;` *[arithmetic expression]*`;` *[arithmetic expression]* `))`

  The next loop will run as long as the second *arithmetic expression* remains *true*.
  The first *arithmetic expression* will be run before the loop starts. The third *arithmetic expression* will be run after the last command in each iteration has been executed.

- `while` *[command list]*

  The next loop will be repeated for as long as the last command ran in the *command list* exits successfully.

- `until` *[command list]*

  The next loop will be repeated for as long as the last command ran in the *command list* exits unsuccessfully ("fails").

- `select` *[name]* `in` *[words]*

  The next loop will repeat forever, letting the user choose between the given words.

  The iteration's commands are executed with the variable denoted by `name`'s value set to the word chosen by the user. Naturally, you can use `break` to end this loop.

Builtins

Builtins are commands that perform a certain function that has been compiled into Bash. Understandably, they are also the only types of commands (other than those above) that can modify the Bash shell's environment.

Dummies

- `true` (or `:`): **These commands do nothing at all**.

  They are *NOP*s that always return successfully.

- `false`: **The same as above, except that the command always "fails"**.

  It returns an exit code of `1` indicating failure.

Declarative

- `alias`: **Sets up a Bash alias**, or print the bash alias with the given name.

  Aliasses replace a word in the beginning of a command by something else. They only work in interactive shells (not scripts).

- `declare` (or `typeset`): **Assign a value to a variable**.

Each argument is a new variable assignment. Each argument's part before the equal sign is the name of the variable, and after comes the data of the variable. Options to declare can be used to toggle special variable flags (like read-only/export/integer/array).

- `export`: **Export the given variable to the environment** so that child processes inherit it.
  This is the same as `declare -x`. Remember that for the child process, the variable is not the same as the one you exported. It just holds the same data. Which means, you can't change the variable data and expect it to change in the parent process, too.
- `local`: **Declare a variable to have a scope limited to the current function**.
  As soon as the function exits, the variable disappears. Assigning to it in a function also doesn't change a global variable with the same name, should one exist. The same options as taken by `declare` can be passed to `local`.
- `type`: **Show the type of the command name specified as argument**.
  The type can be either: *alias*, *keyword*, *function*, *builtin*, or *file*.

Input

- `read`: **Read a line *(unless the `-d` option is used to change the delimiter from* newline *to something else)* and put it in the variables denoted by the arguments given to `read`**.
  If more than one variable name is given, split the line up using the characters in IFS as delimiters. If less variable names are given than there are split chunks in the line, the last variable gets all data left unsplit.

Output

- `echo`: **Output each argument given to `echo` on one line, separated by a single space**.
  The first arguments can be options that toggle special behaviour (like no newline at end/evaluate escape sequences).
- `printf`: **Use the first argument as a format specifier of how to output the other arguments**.
  See `help printf`.
- `pwd`: **Output the absolute pathname of the current working directory**.
  You can use the `-P` option to make `pwd` resolve any symlinks in the pathname.

Execution

- `cd`: **Changes the current directory to the given path**.
  If the path doesn't start with a slash, it is relative to the current directory.
- `command`: **Run the first argument as a command**.
  This tells Bash to skip looking for an alias, function or keyword by that name; and instead assume the command name is a builtin, or a program in `PATH`.
- `coproc`: **Run a command or compound command as a co-process**.
  Runs in bg, setting up pipes for communication. See http://wiki.bash-hackers.org/syntax/keywords/coproc for details.
- `.` *or* `source`: **Makes Bash read the filename given as first argument and execute its contents in the current shell**.
  This is kind of like `include` in other languages. If more arguments are given than just a filename to `source`, those arguments are set as the positional parameters during the execution of the sourced code. If the filename to source has no slash in it, `PATH` is searched for it.
- `exec`: **Run the command given as first argument and replace the current shell with it**.
  Other arguments are passed to the command as its arguments. If no arguments are given to exec but you do specify *Redirections* on the exec command, the redirections will be applied to the current shell.
- `exit`: **End the execution of the current script**.
  If an argument is given, it is the exit status of the current script (an integer between 0 and 255).
- `logout`: **End the execution of a *login* shell**.
- `return`: **End the execution of the current *function***.
  An exit status may be specified just like with the `exit` builtin.
- `ulimit`: **Modify resource limitations of the current shell's process**.
  These limits are inherited by child processes.

Jobs/Processes

- `jobs`: **List the current shell's active jobs**.
- `bg`: **Send the previous job *(or job denoted by the given argument)* to run in the background**.
  The shell continues to run while the job is running. The shell's input is handled by itself, not the job.
- `fg`: **Send the previous job *(or job denoted by the given argument)* to run in the foreground**.
  The shell waits for the job to end and the job can receive the input from the shell.
- `kill`: **Send a signal(3) to a process or job**.
  As argument, give the process ID of the process or the *jobspec* of the job you want to send the signal to.
- `trap`: **Handle a signal(3) sent to the current shell**.
  The code that is in the first argument is executed whenever a signal is received denoted by any of the other arguments to `trap`.
- `suspend`: **Stops the execution of the current shell until it receives a *SIGCONT* signal**.
  This is much like what happens when the shell receives a *SIGSTOP* signal.
- `wait`: **Stops the execution of the current shell until active jobs have finished**.
  In arguments, you can specify which jobs (by *jobspec*) or processes (by *PID*) to wait for.

Conditionals And Loops

- `break`: **Break out of the current loop**.
  When more than one loop is active, break out the last one declared. When a *number* is given as argument to `break`, break out of *number* loops, starting with the last one declared.
- `continue`: **Skip the code that is left in the current loop** and start a new iteration of that loop.
  Just like with `break`, a *number* may be given to skip out more loops.

Script Arguments

- `set`: **The `set` command normally sets various *Shell options*, but can also set *Positional parameters***.
  Shell options are options that can be passed to the shell, such as `bash -x` or `bash -e`. `set` toggles shell options like this: `set -x`, `set +x`, `set -e`, ... *Positional parameters* are parameters that hold arguments that were passed to the script or shell, such as `bash myscript -foo /bar`. `set` assigns positional parameters like this: `set -- -foo /bar`.
- `shift`: **Moves all positional parameters' values one parameter back**.
  This way, values that were in `$1` are discarted, values from `$2` go into `$1`, values from `$3` go into `$2`, and so on. You can specify an argument to shift which is an integer that specifies how many times to repeat this shift.
- `getopts`: **Puts an option specified in the arguments in a variable**.
  `getopts` Uses the first argument as a specification for which options to look for in the arguments. It then takes the first option in the arguments that is mentioned in this option specification (or next option, if getopts has been ran before), and puts this option in the variable denoted by the name in the second argument to `getopts`. This command is pretty much always used in a **loop**:

  ```
  while getopts abc opt
  do
     case $opt in
        a) ...;;
        b) ...;;
        c) ...;;
     esac
  done
  ```

  This way all options in the arguments are parsed and when they are either `-a`, `-b` or `-c`, the respective code in the `case` statement is executed. Following short style is also valid for specifying multiple options in the arguments that `getopts` parses: `-ac`.

Streams

If you're new to handling input and output in bash or are looking for more examples, details and/or explanations, go read BashGuide/InputAndOutput.

Bash is an excellent tool for managing streams of data between processes. Thanks to its excellent operators for connecting file descriptors, we take data from almost anywhere and send it to almost anywhere. Understanding streams and how you manipulate them in Bash is key to the vastness of Bash's power.

File Descriptors

A file descriptor is like a road between a file and a process. It's used by the process to send data to the file or read data from the file. A process can have a great many file descriptors, but by default, there are three that are used for standard tasks.

- `0`: **Standard Input**
  This is where processes normally read information from. Eg. the process may ask you for your name, after you type it in, the information is read over FD 0.
- `1`: **Standard Output**
  This is where processes normally write all their output to. Eg. the process may explain what it's doing or output the result of an operation.
- `2`: **Standard Error**
  This is where processes normally write their error messages to. Eg. the process may complain about invalid input or invalid arguments.

Redirection

- *[command]* `>` *[file]*, *[command]* `[n]>` *[file]*, *[command]* `2>` *[file]*
  File Redirection: The `>` operator redirects the command's *Standard Output* (or FD `n`) to a given *file*.
  This means all standard output generated by the command will be written to the file.
  You can optionally specify a number in front of the `>` operator. If not specified, the number defaults to `1`. The number indicates which file descriptor of the process to redirect output from.
  Note: The file will be truncated (emptied) before the command is started!
- *[command]* `>&`*[fd]*, *[command]* `[fd]>&`*[fd]*, *[command]* `2>&1`
  Duplicating File Descriptors: The `x>&y` operator copies FD `y`'s target to FD `x`.
  For the last example, FD `1` (the command's stdout)'s current target is copied to FD `2` (the command's stderr).
  As a result, when the command writes to its stderr, the bytes will end up in the same place as they would have if they had been written to the command's stdout.
- *[command]* `>>` *[file]*, *[command]* `[n]>>` *[file]*
  File Redirection: The `>>` operator redirects the command's *Standard Output* to a given *file*, appending to it.
  This means all standard output generated by the command will be added to the end of the file.
  Note: The file is not truncated. Output is just added to the end of it.
- *[command]* `<` *[file]*, *[command]* `[n]<` *[file]*
  File Redirection: The `<` operator redirects the given *file* to the command's *Standard Input*.
  You can optionally specify a number in front of the `<` operator. If not specified, the number defaults to `0`. The number indicates which file descriptor of the process to redirect input into.
- *[command]* `&>` *[file]*
  File Redirection: The `&>` operator redirects the command's *Standard Output* and *Standard Error* to a given *file*.
  This means all standard output and errors generated by the command will be written to the file.
- *[command]* `&>>` *[file] (Bash 4+)*
  File Redirection: The `&>>` operator redirects the command's *Standard Output* and *Standard Error* to a given *file*, appending to it.
  This means all standard output and errors generated by the command will be added to the end of the file.
- *[command]* `<<< "`*[line of data]*`"`
  Here-String: Redirects the single string of data to the command's *Standard Input*.
  This is a good way to send a single line of text to a command's input. Note that since the string is quoted, you can also put newlines in it safely, and turn it into multiple lines of data.
- *[command]* `<<[WORD]`
  [lines of data]
  `[WORD]`
  Here-Document: Redirects the lines of data to the command's *Standard Input*.
  This is a good way of sending multiple lines of text to a command's input.
  Note: The word after `<<` *must* be exactly the same as the word after the last line of data, and when you repeat that word after the last line of data, it *must* be in the beginning of the line, and there must be nothing else on that line.
  Note: You can `'quote'` the word after the `<<`. If you do so, anything in the lines of data that looks like expansions will not be expanded by bash.

Piping

- *[command]* `|` *[othercommand]*

  Pipe: The `|` operator connects the first command's *Standard Output* to the second command's *Standard Input*.

  As a result, the second command will read its data from the first command's output.

- *[command]* `|&` *[othercommand] (Bash 4+)*

  Pipe: The `|&` operator connects the first command's *Standard Output* and *Standard Error* to the second command's *Standard Input*.

  As a result, the second command will read its data from the first command's output and errors combined.

Expansions

- *[command]* `"$(` *[command list]* `)"`, *[command]* `"` `` *[command list]* `` `"`

  Command Substitution: captures the output of a command and expands it inline.

  We only use command substitution inside other commands when we want the output of one command to become part of another statement. An ancient and ill-advised alternative syntax for command substitution is the back-quote: `` `command` ``. This syntax has the same result, but it does not nest well and it's too easily confused with quotes (back-quotes have nothing to do with quoting!). Avoid this syntax and replace it with `$(command)` when you find it.

  It's like running the second command, taking its output, and pasting it in the first command where you would put `$(...)`.

- *[command]* `<(`*[command list]*`)`

  Process substitution: The `<(...)` operator expands into a new file created by bash that contains the other command's output.

  The file provides whomever reads from it with the output from the second command.

  It's like redirecting the output of the second command to a file called `foo`, and then running the first command and giving it `foo` as *argument*. Only, in a single statement, and `foo` gets created and cleaned up automatically afterwards.

  NOTE: DO NOT CONFUSE THIS WITH FILE REDIRECTION. The `<` here does **not** mean *File Redirection*. It is just a symbol that's part of the `<(...)` operator! This operator does **not** do *any* redirection. It *merely* expands into a *path* to a *file*.

- *[command]* `>(`*[command list]*`)`

  Process substitution: The `>(...)` operator expands into a new file created by bash that sends data you write to it to a second command's *Standard Input*.

  When the first command writes something to the file, that data is given to the second command as input.

  It's like redirecting a file called `foo` to the input of the second command, and then running the first command, giving it `foo` as *argument*. Only, in a single statement, and `foo` gets created and cleaned up automatically afterwards

Common Combinations

- *[command]* `<` `<(`*[command list]*`)`

  File Redirection and Process Substitution: The `<(...)` is replaced by a file created by bash, and the `<` operator takes that new file and redirects it to the command's *Standard Input*.

  This is almost the same thing as piping the second command to the first (`secondcommand | firstcommand`), but the first command is not sub-shelled like it is in a pipe. It is mostly used when we need the first command to modify the shell's environment (which is impossible if it is subshelled). For example, reading into a variable: `read var < <(grep foo file)`. This wouldn't work: `grep foo file | read var`, because the `var` will be assigned only in its tiny subshell, and will disappear as soon as the pipe is done.

  Note: Do not forget the *whitespace* between the `<` operator and the `<(...)` operator. If you forget that space and turn it into `<<(...)`, that will give errors!

  Note: This creates (and cleans up) a temporary implementation-specific file (usually, a FIFO) that channels output from the second command to the first.

- *[command]* `<<<` `"$(`*[command list]*`)"`

  Here-String and Command Substitution: The `$(...)` is replaced by the output of the second command, and the `<<<` operator sends that string to the first command's *Standard Input*.

  This is pretty much the same thing as the command above, with the small side-effect that `$()` strips all trailing newlines from the output and `<<<` adds one back to it.

  Note: This first reads all output from the second command, storing it in memory. When the second command is complete, the first is invoked with the output. Depending on the amount of output, this can be more memory-consuming.

Tests

If you're new to bash, don't fully understand what commands and exit codes are or want some details, explanation and/or examples on testing commands, strings or files, go read the BashGuide's section on Tests and Conditionals.

Exit Codes

An *Exit Code* or *Exit Status* is an unsigned 8-bit integer returned by a command that indicates how its execution went. It is agreed that an *Exit Code* of `0` indicates the command was successful at what it was supposed to do. Any other *Exit Code* indicates that something went wrong. Applications can choose for themselves what number indicates what went wrong; so refer to the manual of the application to find out what the application's *Exit Code* means.

Testing The Exit Code

- `if` *[command list]*`; then` *[command list]*`; elif` *[command list]*`; then` *[command list]*`; else` *[command list]*`; fi`
  The `if` command tests whether the last command in the first *command list* had an exit code of `0`.
  If so, it executes the *command list* that follows the `then`. If not, the next `elif` is tried in the same manner. If no `elif`s are present, the *command list* following `else` is executed, unless there is no `else` statement. To summarize, `if` executes a list of *command*s. It tests the exit code. On success, the `then` commands are executed. `elif` and `else` parts are optional. The `fi` part ends the entire `if` block (don't forget it!).
- `while` *[command list]*, and `until` *[command list]*
  Execute the next iteration depending on the exit code of the last command in the *command list*.
  We've discussed these before, but it's worth repeating them in this section, as they actually do the same thing as the `if` statement; except that they execute a loop for as long as the tested exit code is respectively `0` or `non-0`.

Patterns

Bash knows two types of patterns. *Glob Patterns* is the most important, most used and best readable one. Later versions of Bash also support the *"trendy" Regular Expressions*. However, it is ill-advised to use regular expressions in scripts unless you have absolutely no other choice or the advantages of using them are far greater than when using globs. Generally speaking, if you need a regular expression, you'll be using `awk(1)`, `sed(1)`, or `grep(1)` instead of Bash.

If you're new to bash or want some details, explanation and/or examples on pattern matching, go read the BashGuide's section on Patterns.

Glob Syntax

- `?`: **A question mark matches any character**.
  That is one single character.
- *: **A star matches any amount of any characters**.
  That is zero or more of whatever characters.
- `[...]`: **This matches *one of* any of the characters inside the braces**.
  That is one character that is mentioned inside the braces.
  - `[abc]`: **Matches either `a`, `b`, or `c` but not the string `abc`**.
  - `[a-c]`: **The dash tells Bash to use a range**.
    Matches any character between (inclusive) `a` and `c`. So this is the same thing as the example just above.
  - `[!a-c]` or `[^a-c]`: **The `!` or `^` in the beginning tells Bash to invert the match**.
    Matches any character that is *not* `a`, `b` or `c`. That means any other letter, but *also* a number, a period, a comma, or any other character you can think of.
  - `[[:digit:]]`: **The `[:`*class*`:]` syntax tells Bash to use a character class**.
    Character classes are groups of characters that are predefined and named for convenience. You can use the following classes:
    `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `word`, `xdigit`

Testing

- `case` *[string]* `in` *[glob pattern]*`)` *[command list]*`;;` *[glob pattern]*`)` *[command list]*`;; esac`:
  Using `case` is handy if you want to test a certain string that could match either of several different glob patterns.
  The *command list* that follows the *first* *glob pattern* that matched your *string* will be executed. You can specify as many *glob pattern* and *command lists* combos as you need.
- `[[` *[string]* = "*[string]*" `]]`, `[[` *[string]* = *[glob pattern]* `]]`, or `[[` *[string]* =~ *[regular expression]* `]]`:
  Test whether the left-hand *STRING* matches the right-hand *STRING* (if quoted), *GLOB* (if unquoted and using =) or *REGEX* (if unquoted and using =~).
  `[` and `test` are commands you often see in `sh` scripts to perform these tests. `[[` can do all these things (but better and safer) and it also provides you with *pattern* matching.

Do NOT use `[` or `test` in bash code. Always use `[[` instead. It has many benefits and no downsides.
Do NOT use `[[` for performing tests on *commands* or on *numeric operations*. For the first, use `if` and for the second use `((`.
`[[` can do a bunch of other tests, such as on files. See `help test` for all the types of tests it can do for you.

- `((` *[arithmetic expression]* `))`:
  This keyword is specialized in performing numeric tests and operations.
  See ArithmeticExpression

Parameters

Parameters are what Bash uses to store your script data in. There are *Special Parameters* and *Variables*.

Any parameters you create will be variables, since special parameters are read-only parameters managed by Bash. It is recommended you use lower-case names for your own parameters so as not to confuse them with the all-uppercase variable names used by Bash internal variables and environment variables. It is also recommended you use clear and transparent names for your variables. Avoid `x`, `i`, `t`, `tmp`, `foo`, etc. Instead, use the variable name to describe the kind of data the variable is supposed to hold.

It is also important that you understand the need for quoting. Generally speaking, whenever you use a parameter, you should quote it: `echo "The file is in: $filePath"`. If you don't, bash will tear the contents of your parameter to bits, delete all the whitespace from it, and feed the bits as arguments to the command. Yes, Bash mutilates your parameter expansions by default - it's called *Word Splitting* - so use quotes to prevent this.
The exception is *keywords* and *assignment*. After *`myvar=`* and inside *`[[`*, *`case`*, etc, you don't *need* the quotes, but they won't do any harm either - so if you're unsure: quote!

Last but not least: Remember that parameters are the *data structures* of bash. They hold your application data. They should **NOT** be used to hold your application logic. So while many ill-written scripts out there may use things like `GREP=/usr/bin/grep`, or `command='mplayer -vo x11 -ao alsa'`, you should **NOT** do this. The main reason is because you cannot possibly do it completely right *and* safe *and* readable/maintainable.
If you want to avoid retyping the same command multiple times, or make a single place to manage the command's command line, use a *function* instead. Not parameters.

Special Parameters

If you're new to bash or want some details, explanation and/or examples on parameters, go read the BashGuide's section on Special Parameters.

- `1`, `2`, ...: **Positional Parameters are the arguments that were passed to your script or your function.**
  When your script is started with `./script foo bar`, `"$1"` will become `"foo"` and `"$2"` will become `"bar"`. A script ran as `./script "foo bar" hubble` will expand `"$1"` as `"foo bar"` and `"$2"` as `"hubble"`.
- `*`: **When expanded, it equals the single string that concatenates all positional parameters using the first character of IFS to separate them (- by default, that's a space).**
  In short, `"$*"` is the same as `"$1x$2x$3x$4x..."` where x is the first character of `IFS`.
  With a default `IFS`, that will become a simple `"$1 $2 $3 $4 ..."`.
- `@`: **This will expand into multiple arguments: Each positional parameter that is set will be expanded as a single argument.**
  So basically, `"$@"` is the same as `"$1" "$2" "$3" ...`, all quoted separately.
  NOTE: You should always use `"$@"` before `"$*"`, because `"$@"` preserves the fact that each argument is its separate entity. With `"$*"`, you lose this data! `"$*"` is really only useful if you want to separate your arguments by something that's not a space; for instance, a comma: `(IFS=,; echo "You ran the script with the arguments: $*")` -- output all your arguments, separating them by commas.
- `#`: **This parameter expands into a number that represents how many positional parameters are set.**
  A script executed with 5 arguments, will have `"$#"` expand to `5`. This is mostly only useful to test whether any arguments were set:
  `if (( ! $# )); then echo "No arguments were passed." >&2; exit 1; fi`
- `?`: **Expands into the exit code of the previously completed foreground command.**
  We use `$?` mostly if we want to use the exit code of a command in multiple places; or to test it against many possible values in a `case` statement.
- `-`: **The dash parameter expands into the option flags that are currently set on the Bash process.**
  See *set* for an explanation of what option flags are, which exist, and what they mean.
- `$`: **The dollar parameter expands into the *Process ID* of the Bash process.**
  Handy mostly for creating a PID file for your bash process (`echo "$$" > /var/run/foo.pid`); so you can easily terminate it from another bash process, for example.
- `!`: **Expands into the *Process ID* of the most recently backgrounded command.**
  Use this for managing backgrounded commands from your Bash script: `foo ./bar & pid=$!; sleep 10; kill "$pid"; wait "$pid"`

- `_`: **Expanding the underscore argument gives you the last argument of the last command you executed.**
  This one's used mostly in interactive shells to shorten typing a little: `mkdir -p /foo/bar && mv myfile "$_"`.

Parameter Operations

If you're new to bash or want some details, explanation and/or examples on parameter operations, go read the BashGuide's section on Parameter Expansion and BashFAQ/073.

- `"$var"`, `"${var}"`
  Expand the value contained within the parameter `var`. The parameter expansion syntax is replaced by the contents of the variable.
- `"${var:-Default Expanded Value}"`
  Expand the value contained within the parameter `var` or the string `Default Expanded Value` if `var` is empty. Use this to expand a default value in case the value of the parameter is empty (unset or contains no characters).
- `"${var:=Default Expanded And Assigned Value}"`
  Expand the value contained within the parameter `var` but first assign `Default Expanded And Assigned Value` to the parameter if it is empty. This syntax is often used with the colon command (`:`): `: "${name:=$USER}"`, but a regular assignment with the above will do as well: `name="${name:-$USER}"`.
- `"${var:?Error Message If Unset}"`, `"${name:?Error: name is required.}"`
  Expand the value contained within the parameter `name` or show an error message if it's empty. The script (or function, if in an interactive shell) is aborted.
- `${name:+Replacement Value}`, `${name:+--name "$name"}`
  Expand the given string if the parameter `name` is not empty. This expansion is used mainly for expanding the parameter along with some context. The example expands **two arguments**: notice how, unlike all other examples, the main expansion is unquoted, allowing word splitting of the inside string. Remember to quote the parameter in the inside string, though!
- `"${line:5}"`, `"${line:5:10}"`, `"${line:offset:length}"`
  Expand a substring of the value contained within the parameter `line`. The substring begins at character number `5` (or the number contained within parameter `offset`, in the second example) and has a length of `10` characters (or the number contained within parameter `length`). The offset is `0`-based. If the length is omitted, the substring reaches til the end of the parameter's value.
- `"${@:5}"`, `"${@:2:4}"`, `"${array:start:count}"`
  Expand elements from an array starting from a `start` index and expanding all or a given `count` of elements. All elements are expanded as separate arguments because of the quotes. If you use `@` as the parameter name, the elements are taken from positional parameters (the arguments to your script - the second example becomes: `"$2" "$3" "$4" "$5"`).
- `"${!var}"`
  Expand the value of the parameter named by the value of the parameter `var`. This is **bad practice!** This expansion makes your code highly non-transparent and unpredictable in the future. You probably want an associative array instead.
- `"${#var}"`, `"${#myarray[@]}"`
  Expand into the length of the value of the parameter `var`. The second example expands into the number of elements contained in the array named `myarray`.
- `"${var#A Prefix}"`, `"${PWD#*/}"`, `"${PWD##*/}"`
  Expand the value contained within the parameter `var` after removing the string `A Prefix` from the beginning of it. If the value doesn't have the given prefix, it is expanded as is. The prefix can also be a glob pattern, in which case the string that matches the pattern is removed from the front. You can double the `#` mark to make the pattern match greedy.
- `"${var%A Suffix}"`, `"${PWD%/*}"`, `"${PWD%%/*}"`
  Expand the value contained within the parameter `var` after removing the string `A Suffix` from the end of it. Works just like the prefix trimming operation, only takes away from the end.
- `"${var/pattern/replacement}"`, `"${HOME/$USER/bob}"`, `"${PATH//:/ }"`
  Expand the value contained within the parameter `var` after replacing the given pattern with the given replacement string. The pattern is a glob used to search for the string to replace within `var`'s value. The first match is replaced with the replacement string. You can double the first `/` to replace all matches: The third example replaces all colons in `PATH`'s value by spaces.
- `"${var^}"`, `"${var^^}"`, `"${var^^[ac]}"`
  Expand the value contained within the parameter `var` after upper-casing all characters matching the pattern. The pattern *must* be match a single character and the pattern `?` (any character) is used if it is omitted. The first example upper-cases the first character from `var`'s value, the second upper-cases all characters. The third upper-cases all characters that are either `a` or `c`.
- `"${var,}"`, `"${var,,}"`, `"${var,,[AC]}"`
  Expand the value contained within the parameter `var` after lower-casing all characters matching the pattern. Works just like the upper-casing operation, only lower cases matching characters.

Arrays

Arrays are variables that contain multiple strings. Whenever you need to store multiple items in a variable, **use an array and NOT a string** variable. Arrays allow you to keep the elements nicely separated and allow you to cleanly expand the elements into separate arguments. This is **impossible** to do if you mash your items together in a string!

If you're new to bash or don't fully grasp what arrays are and why one would use them in favor of normal variables, or you're looking for more explanation and/or examples on arrays, go read the BashGuide's section on Arrays and BashFAQ/005

Creating Arrays

- `myarray=( foo bar quux )`

  Create an array `myarray` that contains three elements. Arrays are created using the `x=(y)` syntax and array elements are separated from each other by whitespace.
- `myarray=( "foo bar" quux )`

  Create an array `myarray` that contains *two* elements. To put elements in an array that contain whitespace, wrap quotes around them to indicate to bash that the quoted text belongs together in a single array element.
- `myfiles=( *.txt )`

  Create an array `myfiles` that contains all the filenames of the files in the current directory that end with `.txt`. We can use any type of expansion inside the array assignment syntax. The example use pathname expansion to replace a glob pattern by all the filenames it matches. Once replaced, array assignment happens like in the first two examples.
- `myfiles+=( *.html )`

  Add all HTML files from the current directory to the `myfiles` array. The `x+=(y)` syntax can be used the same way as the normal array assignment syntax, but append elements to the end of the array.
- `names[5]="Big John"`, `names[n + 1]="Long John"`

  Assign a string to a specific index in the array. Using this syntax, you explicitly tell Bash at what index in your array you want to store the string value. The index is actually interpreted as an *arithmetic expression*, so you can easily do math there.
- `read -ra myarray`

  Chop a line into fields and store the fields in an array `myarray`. The `read` commands reads a line from *stdin* and uses each character in the IFS variable as a delimiter to split that line into fields.
- `IFS=, read -ra names <<< "John,Lucas,Smith,Yolanda"`

  Chop a line into fields using `,` as the delimiter and store the fields in the array named `names`. We use the `<<<` syntax to feed a string to the `read` command's *stdin*. `IFS` is set to `,` for the duration of the `read` command, causing it to split the input line into fields separated by a comma. Each field is stored as an element in the `names` array.
- `IFS=$'\n' read -d '' -ra lines`

  Read all lines from *stdin* into elements of the array named `lines`. We use `read`'s `-d ''` switch to tell it not to stop reading after the first line, causing it to read in all of *stdin*. We then set `IFS` to a newline character, causing `read` to chop the input up into fields whenever a new line begins.
- `files=(); while IFS= read -d '' -r file; do files+=("$file"); done < <(find . -name '*.txt' -print0)`

  Safely read all TXT files contained recursively in the current directory into the array named `files`.
  We begin by creating an empty array named `files`. We then start a `while` loop which runs a `read` statement to read in a filename from *stdin*, and then appends that filename (contained in the variable `file`) to the `files` array. For the `read` statement we set `IFS` to empty, avoiding `read`'s behavior of trimming leading whitespace from the input and we set `-d ''` to tell `read` to continue reading until it sees a `NUL` byte (filenames **CAN** span multiple lines, so we don't want read to stop reading the filename after one line!). For the input, we attach the `find` command to `while`'s *stdin*. The `find` command uses `-print0` to output its filenames by separating them with `NUL` bytes (see the `-d ''` on `read`). **NOTE:** This is the **only** truly safe way of building an array of filenames from a command's output! You **must** delimit your filenames with `NUL` bytes, because it is the *only* byte that can't actually appear inside a filename! **NEVER** use `ls` to enumerate filenames! First try using the glob examples above, they are just as safe (no need to parse an external command), much simpler and faster.
- `declare -A homedirs=( ["Peter"]=~pete ["Johan"]=~jo ["Robert"]=~rob )`

  Create an *associative* array, mapping names to user home directories. Unlike normal arrays, associative arrays indices are strings (just like the values). Note: you *must* use `declare -A` when creating an associative array to indicate to bash that this array's indices are strings and not integers.
- `homedirs["John"]=~john`

  Add an element to an associative array, keyed at `"John"`, mapped to `john`'s home directory.

Using Arrays

- `echo "${names[5]}"`, `echo "${names[n + 1]}"`

  Expand a single element from an array, referenced by its index. This syntax allows you to retrieve an element's value given the index of the element. The index is actually interpreted as an *arithmetic expression*, so you can easily do math there.
- `echo "${names[@]}"`

  Expand each array element as a separate argument. This is the preferred way of expanding arrays. Each element in the array is expanded as if passed as a new argument, properly quoted.

- `cp "${myfiles[@]}" /destinationdir/`

  Copy all files referenced by the filenames within the `myfiles` array into `/destinationdir/`. Expanding an array happens using the syntax `"${array[@]}"`. It effectively replaces that expansion syntax by a list of all the elements contained within the array, properly quoted as separate arguments.
- `rm "./${myfiles[@]}"`

  Remove all files referenced by the filenames within the `myfiles` array. It's generally a *bad* idea to attach strings to an array expansion syntax. What happens is: the string is only prefixed to the *first* element expanded from the array (or suffixed to the last if you attached the string to the end of the array expansion syntax). If `myfiles` contained the elements `-foo.txt` and `bar-.html`, this command would expand into: `rm "./-foo.txt" "bar-.html"`. Notice only the first element is prefixed with `./`. In this particular instance, this is handy because `rm` fails if the first filename begins with a dash. Now it begins with a dot.
- `(IFS=,; echo "${names[*]}")`

  Expand the array `names` into a *single string* containing all elements in the array, merging them by separating them with a comma (`,`). The `"${array[*]}"` syntax is only very rarely useful. Generally, when you see it in scripts, it is a bug. The one use it has is to merge all elements of an array into a single string for displaying to the user. Notice we surrounded the statement with `(`brackets`)`, causing a subshell: This will scope the IFS assignment, resetting it after the subshell ends.
- `for file in "${myfiles[@]}"; do read -p "Delete $file? " && [[ $REPLY = y ]] && rm "$file"; done`

  Iterate over all elements of the `myfiles` array after expanding them into the `for` statement. Then, for each file, ask the user whether he wants to delete it.
- `for index in "${!myfiles[@]}"; do echo "File number $index is ${myfiles[index]}"; done`

  Iterate over all keys of the `myfiles` array after expanding them into the `for` statement. The syntax `"${!array[@]}"` (notice the `!`) gets expanded into a list of array *keys*, not values. Keys of normal arrays are numbers starting at `0`. The syntax for getting to a particular element within an array is `"${array[index]}"`, where `index` is the key of the element you want to get at.
- `names=(John Pete Robert); echo "${names[@]/#/Long }"`

  Perform a parameter expansion operation on every element of the `names` array. When adding a parameter expansion operation to an array expansion, the operation is applied to every single array element as it is expanded.
- `names=(John Pete Robert); echo "${names[@]:start:length}"; echo "${names[@]:1:2}"`

  Expand `length` array elements, starting at index `start`. Similar to the simple `"${names[@]}"` but expands a *sub*-section of the array. If `length` is omitted, the rest of the array elements are expanded.
- `printf '%s\n' "${names[@]}"`

  Output each array element on a new line. This `printf` statement is a very handy technique for outputting array elements in a common way (in this case, appending a newline to each). The format string given to `printf` is applied to each element (unless multiple `%s`'s appear in it, of course).
- `for name in "${!homedirs[@]}"; do echo "$name lives in ${homedirs[$name]}"; done`

  Iterate over all keys of the `homedirs` array after expanding them into the `for` statement. The syntax for getting to the keys of associative arrays is the same as that for normal arrays. Instead of numbers beginning at `0`, we now get the keys for which we mapped our associative array's values. We can later use these keys to look up values within the array, just like normal arrays.
- `printf '%s\n' "${#names[@]}"`

  Output the number of elements in the array. In this `printf` statement, the expansion expands to only one argument, regardless of the amount of elements in the array. The expanded argument is a number that indicates the amount of elements in the `names` array.

Examples: Basic Structures

Compound Commands

Command Lists

- `[[ $1 ]] || { echo "You need to specify an argument!" >&2; exit 1; }`

  We use a command group here because the `||` operator takes just one command.
  We want both the `echo` and `exit` commands to run if `$1` is empty.
- `(IFS=','; echo "The array contains these elements: ${array[*]}")`

  We use parenthesis to trigger a subshell here.
  When we set the IFS variable, it will only change in the subshell and not in our main script. That avoids us having to reset it to it's default after the expansion in the echo statement (which otherwise we would have to do in order to avoid unexpected behaviour later on).
- `(cd "$1" && tar -cvjpf archive.tbz2 .)`

Here we use the subshell to temporarily change the current directory to what's in `$1`.
After the `tar` operation (when the subshell ends), we're back to where we were before the `cd` command because the current directory of the main script never changed.

Expressions

- `((completion = current * 100 / total))`
  Note that arithmetic context follows completely different parsing rules than normal bash statements.
- `[[ $foo = /* ]] && echo "foo contains an absolute pathname."`
  We can use the `[[` command to perform all tests that `test(1)` can do.
  But as shown in the example it can do far more than `test(1)`; such as glob pattern matching, regular expression matching, test grouping, etc.

Loops

- `for file in *.mp3; do openssl md5 "$file"; done > mysongs.md5`
  For loops iterate over all arguments after the `in` keyword.
  One by one, each argument is put in the variable name `file` and the loop's body is executed.

  DO NOT PASS A COMMAND'S OUTPUT TO `for` BLINDLY!
  `for` will iterate over the WORDS in the command's output; which is almost NEVER what you really want!
- `for file; do cp "$file" /backup/; done`
  This concise version of the `for` loop iterates the positional parameters.
  It's basically the equivalent of `for file in "$@"`.
- `for (( i = 0; i < 50; i++ )); do printf "%02d," "$i"; done`
  Generates a comma-separated list of numbers zero-padded to two digits.
  (The last character will be a comma, yes, if you really want to get rid of it; you can - but it defeats the simplicity of this example)
- `while read _ line; do echo "$line"; done < file`
  This `while` loop continues so long as the `read` command is successful.
  (Meaning, so long as lines can be read from the file). The example basically just throws out the first column of data from a file and prints the rest.
- `until myserver; do echo "My Server crashed with exit code: $?; restarting it in 2 seconds .."; sleep 2; done`
  This loop restarts `myserver` each time it exits with a non-successful exit code.
  It assumes that when `myserver` exits with a non-successful exit code; it crashed and needs to restart; and if it exist with a successful exit code; you ordered it to shut down and it needn't be restarted.
- `select fruit in Apple Pear Grape Banana Strawberry; do (( credit -= 2, health += 5 )); echo "You purchased some $fruit.  Enjoy!"; done`
  A simple program which converts credits into health.
  Amazing.

Builtins

Dummies

- `while true; do ssh lhunath@lyndir.com; done`
  Reconnect on failure.

Declarative

- `alias l='ls -al'`
  Make an alias called `l` which is replaced by `ls -al`.
  Handy for quickly viewing a directory's detailed contents.
- `declare -i myNumber=5`
  Declare an integer called `myNumber` initialized to the value `5`.
- `export AUTOSSH_PORT=0`

Export a variable on the bash process environment called `AUTOSSH_PORT` which will be inherited by any process this bash process invokes.

- `foo() { local bar=fooBar; echo "Inside foo(), bar is $bar"; }; echo "Setting bar to 'normalBar'"; bar=normalBar; foo; echo "Outside foo(), bar is $bar"`
  An exercise in variable scopes.
- `if ! type -P ssh >/dev/null; then echo "Please install OpenSSH." >&2; exit 1; fi`
  Check to see if `ssh` is available.
  Suggest the user install *OpenSSH* if it is not, and exit.

Input

- `read firstName lastName phoneNumber address`
  Read data from a single line with four fields into the four named variables.

Output

- `echo "I really don't like $nick.  He can be such a prick."`
  Output a simple string on standard output.
- `printf "I really don't like %s.  He can be such a prick." "$nick"`
  Same thing using `printf` instead of `echo`, nicely separating the text from the data.

Execution

- `cd ~lhunath`
  Change the current directory to `lhunath`'s home directory.
- `cd() { builtin cd "$@" && echo "$PWD"; }`
  Inside the function, execute the builtin `cd` command, not the function (which would cause infinite recursion) and if it succeeds, `echo` out the new current working directory.
- `source bashlib; source ./.foorc`
  Run all the bash code in a file called `bashlib` which exists somewhere in `PATH`; then do the same for the file `.foorc` in the current directory.
- `exec 2>/var/log/foo.log`
  Send all output to standard error from now on to a log file.
- `echo "Fatal error occurred!  Terminating!"; exit 1`
  Show an error message and exit the script.

CategoryShell

BashSheet (last edited 2016-06-23 02:36:41 by tor2r)