LING 408/508: Programming for Linguists

Lecture 10 (= 9)

Today's Topics

Sincere apologies for canceling Lecture 9 at short notice.

- [Lecture 10 = Lecture 9]
- Homework 5 Review
- Bash shell:
  - Last time: globbing (contd.)
  - string manipulation
    - an example, renaming files using a loop
  - find with sed example
  - Cheat Sheet

Homework 5 review

- Modify the BMI calculator to:
  1. accept either command line arguments or read from the terminal if they're missing
  2. print the weight status message according to the following table:
  3. modify the calculator to accept input in both metric and traditional units
- make sure you supply examples of your program working!

BMI	Weight Status
Below 18.5	Underweight
18.5 – 24.9	Normal
25.0 – 29.9	Overweight
30.0 and Above	Obese

```
[ling508-20$ bash hw5.sh
weight in kg (lbs): 68
height in cm (in): 172
units kg/lbs: k
22.98
normal
[ling508-20$ bash hw5.sh 160 68 lbs
24.32
normal
ling508-20$
```

Homework 5 review

```
1#!/bin/bash¶
2# usage: weight height kg/lbs¶
3if [ $# -ne 3 ]; then¶
4    read -p "weight in kg (lbs): " weight¶
5    read -p "height in cm (in): " height¶
6    read -n 1 -p "units kg/lbs: " units¶
7    echo¶
8else¶
9    weight=$1¶
10   height=$2¶
11   units=${3:0:1}¶
12fi¶
```

Notice variables `weight`, `height` and `units`
are set no matter whether the inputs come from command line arguments or the Terminal

Homework 5 review

```
13((height2= $height * $height))¶
14if [ $units = "k" ]; then¶
15    ((bmi = $weight * 1000000 / $height2))¶
16else¶
17    ((bmi = $weight * 70300 / $height2))¶
18fi¶
19((bmi2 = $bmi / 100))¶
20echo $bmi2.${bmi#$bmi2}¶
```

- `string1 = string2` and `string1 == string2` - The equality operator returns true if the operands are equal.
  - Use the `=` operator with the `test` `[` command.
  - Use the `==` operator with the `[[` command for pattern matching.

Homework 5 review

```
13((height2= $height * $height))¶
14if [ $units = "k" ]; then¶
15    ((bmi = $weight * 1000000 / $height2))¶
16else¶
17    ((bmi = $weight * 70300 / $height2))¶
18fi¶
19((bmi2 = $bmi / 100))¶
20echo $bmi2.${bmi#$bmi2}¶
```

```
13((height2= $height * $height))¶
14if [ $units = "k" ]; then¶
15    ((bmi = $weight * 1000000 / $height2))¶
16else¶
17    ((bmi = $weight * 70300 / $height2))¶
18fi¶
19echo "scale=2;$bmi/100" | bc -q¶
```

Homework 5 review

```
20 if [ $bmi -lt 1850 ]; then
21     echo "underweight"
22 elif [ $bmi -lt 2500 ]; then
23     echo "normal"
24 elif [ $bmi -lt 3000 ]; then
25     echo "overweight"
26 else
27     echo "obese"
28 fi
```

```
[ling508-20$ bash hw5b.sh 68 172 kg
 22.98
 normal
[ling508-20$ bash hw5b.sh 70 172 kg
 23.66
 normal
[ling508-20$ bash hw5b.sh 74 172 kg
 25.01
 overweight
```

Expansion

- Pathname expansion (aka ***globbing***):
  - similar (but not the same) as regular expressions
    - `*` (*wild card string*)
    - `?` (*wild card character*)
    - `[ab]` (*a or b*)
    - `[^ab]` (*not* (*a or b*))
  - Examples:
    - `ls f[23]*.jpg`
    - `ls f[^4]*.jpg`

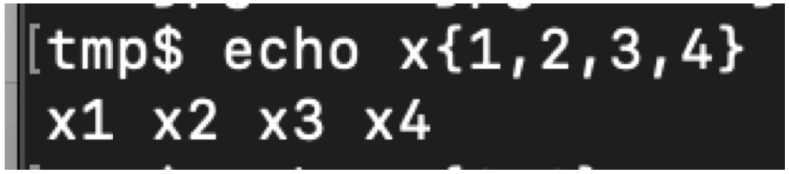
```
tmp$ ls
f1.jpg          f2.jpg.bak      f3.jpg.bak      f5.jpg          sum.sh
f1.jpg.bak      f22.jpg         f4.jpg          f5.jpg.bak
f2.jpg          f3.jpg          f4.jpg.bak      line.sh
tmp$
```

```
tmp$ ls f[23]*.jpg
f2.jpg  f22.jpg f3.jpg
tmp$ ls f[^4]*.jpg
f1.jpg  f2.jpg  f22.jpg f3.jpg  f5.jpg
tmp$
```

Expansion

- (curly) Brace expansion:
  - `mkdir ~/class/examples/{ex1,ex2}`
  - shorthand for:
  - `mkdir ~/class/examples/ex1 ~/class/examples/ex2`

  - `echo x{1,2,3,4}`
  - shorthand for:
  - `echo x1 x2 x3 x4`



String operations: useful for dealing with filenames

- String length:
  - ${#var}
- Substring:
  - ${string:position} — starting at position (0,1,2…), (-N) from the end
  - ${string:position:length}

```
units=${3:0:1}
```

- Delete prefix:
  - ${string#substring} — shortest match
  - ${string##substring} — longest match
- Delete suffix:
  - ${string%substring} — shortest match
  - ${string%%substring} — longest match

```
cp $file ${file%.*}.bak
```

- Substring substitution:
  - ${string/substring/replacement} — replace first match
  - ${string//substring/replacement} — replace all matches
- Prefix substitution: ${string/#substring/replacement}
- Suffix substitution: ${string/%substring/replacement}

File extension renaming

Shell script (`rmext.sh`):

```
#!/bin/bash
if [ $# -ne 2 ]; then
echo “usage: ext1 ext2”
exit 1
fi
for filename in *.$1
# Traverse list of files ending with 1st argument.
do
  mv "$filename" "${filename%$1}$2"
done
exit 0
```

Exercise:

- *create a subdirectory with some .JPG files*

```
sh ../rmext.sh JPG jpg
```

- ⇑ delete suffix: `${string%substring}`
- "… " just in case there are spaces in the filenames

File extension renaming

```
[tmp$ ls
 f1.jpg          f2.jpg.bak      f3.jpg.bak      f5.jpg          sum.sh
 f1.jpg.bak      f22.jpg         f4.jpg          f5.jpg.bak
 f2.jpg          f3.jpg          f4.jpg.bak      line.sh
[tmp$ bash ../rmext.sh jpg JPG
[tmp$ ls
 f1.JPG          f2.jpg.bak      f3.jpg.bak      f5.JPG          sum.sh
 f1.jpg.bak      f22.JPG         f4.JPG          f5.jpg.bak
 f2.JPG          f3.JPG          f4.jpg.bak      line.sh
 tmp$
```

```
[tmp$ bash ../rmext.sh jpg JPG
 mv: rename *.jpg to *.JPG: No such file or directory
 tmp$
```

File renaming

- Example:
  - append a suffix –1 to all jpg files
  - `for f in *.jpg; do mv $f ${f/./–1.}; done`
- Levels of quoting:

Substring substitution:
`${string/substring/replacement}`

```
$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 sandiway
$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt {a,b} foo 4 sandiway
$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

http://linuxcommand.org

sed

```
SED(1)                    BSD General Commands Manual                    SED(1)

NAME
     sed -- stream editor

SYNOPSIS
     sed [-Ealn] command [file ...]
     sed [-Ealn] [-e command] [-f command_file] [-i extension] [file ...]

DESCRIPTION
     The sed utility reads the specified files, or the standard input if no
     files are specified, modifying the input as specified by a list of com-
     mands.  The input is then written to the standard output.

     A single command may be specified as the first argument to sed.  Multiple
     commands may be specified by using the -e or -f options.  All commands
     are applied to the input in the order they are specified regardless of
     their origin.
```

```
     -e command
             Append the editing commands specified by the command argument to
             the list of commands.
```

find

```
FIND(1)                    BSD General Commands Manual                    FIND(1)

NAME
     find -- walk a file hierarchy

SYNOPSIS
     find [-H | -L | -P] [-EXdsx] [-f path] path ... [expression]
     find [-H | -L | -P] [-EXdsx] -f path [path ...] [expression]

DESCRIPTION
     The find utility recursively descends the directory tree for each path
     listed, evaluating an expression (composed of the ``primaries'' and
     ``operands'' listed below) in terms of each file in the tree.
```

```
     -name pattern
             True if the last component of the pathname being examined matches
             pattern.  Special shell pattern matching characters (``['',
             ``]'', ``*'', and ``?'') may be used as part of pattern.  These
             characters may be matched explicitly by escaping them with a
             backslash (``\'').
```

```
     -exec utility [argument ...] ;
             True if the program named utility returns a zero value as its
             exit status.  Optional arguments may be passed to the utility.
             The expression must be terminated by a semicolon (``;'').  If you
             invoke find from a shell you may need to quote the semicolon if
             the shell would otherwise treat it as a control operator.  If the
             string ``{}'' appears anywhere in the utility name or the argu-
             ments it is replaced by the pathname of the current file.
             Utility will be executed from the directory from which find was
             executed.  Utility and arguments are not subject to the further
             expansion of shell patterns and constructs.
```

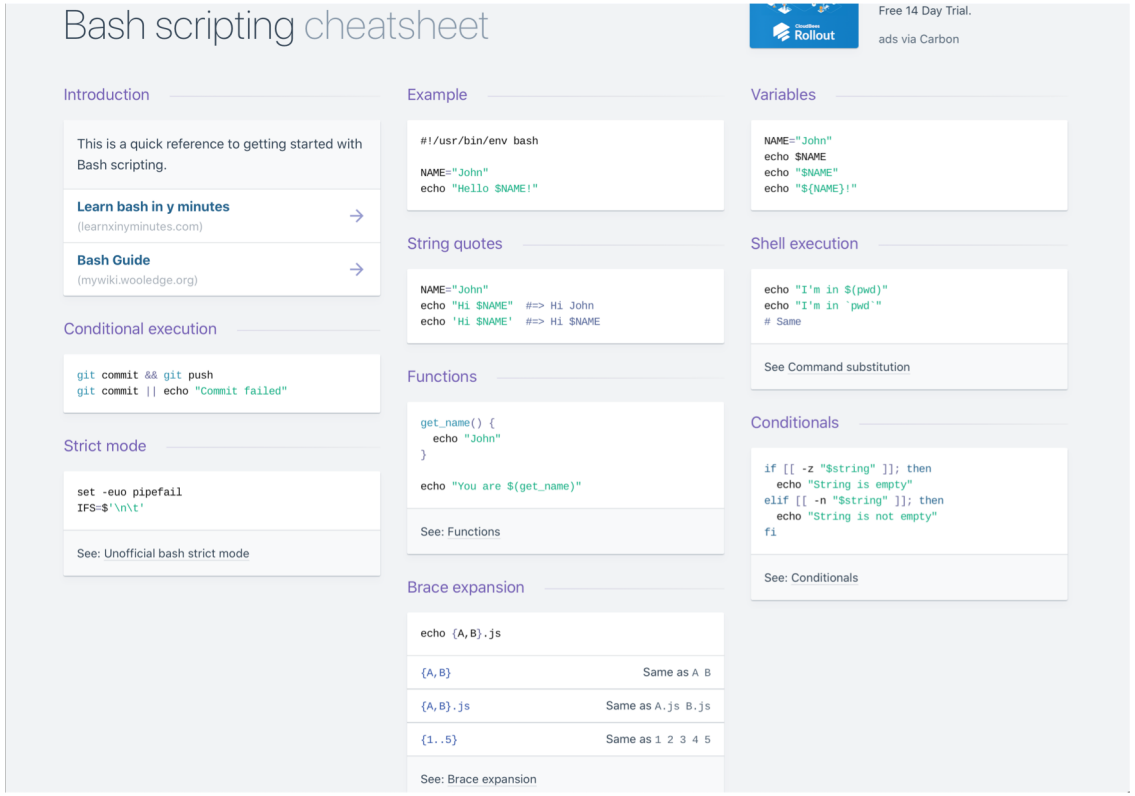
`find` with `sed`

- Using `sed` to edit all .html files in a directory
  - combine with `find –exec … {} \;`
  - `{}` is a placeholder for each filename found by find
  - `\;` ensures `;` is passed to find, lets find know the end of the -exec command
    - `;` is escaped because it is also the shell command separator
  - **-i[SUFFIX]**, edit files in place (makes backup if extension supplied).
- **Example**:

```
1. grep 'begin at step 30' *.html
2. find . –name '*.html' –print
3. find . –name '*.html' –print –exec sed –i.bak 's/begin
   at step 30/begin at step 51/' {} \;
4. grep 'begin at step 30' *.html
```

Good Resource: Bash cheat sheet

- https://devhints.io/bash



Bash scripting cheatsheet

Introduction

This is a quick reference to getting started with Bash scripting.

Learn bash in y minutes
(learnxinyminutes.com)

Bash Guide
(mywiki.wooledge.org)

Conditional execution

```
git commit && git push
git commit || echo "Commit failed"
```

Strict mode

```
set -euo pipefail
IFS=$'\n\t'
```

See: Unofficial bash strict mode

Example

```
#!/usr/bin/env bash

NAME="John"
echo "Hello $NAME!"
```

String quotes

```
NAME="John"
echo "Hi $NAME"  #=> Hi John
echo 'Hi $NAME'  #=> Hi $NAME
```

Functions

```
get_name() {
  echo "John"
}

echo "You are $(get_name)"
```

See: Functions

Brace expansion

```
echo {A,B}.js
```

`{A,B}`	Same as `A B`
`{A,B}.js`	Same as `A.js B.js`
`{1..5}`	Same as `1 2 3 4 5`

See: Brace expansion

Variables

```
NAME="John"
echo $NAME
echo "$NAME"
echo "${NAME}!"
```

Shell execution

```
echo "I'm in $(pwd)"
echo "I'm in `pwd`"
# Same
```

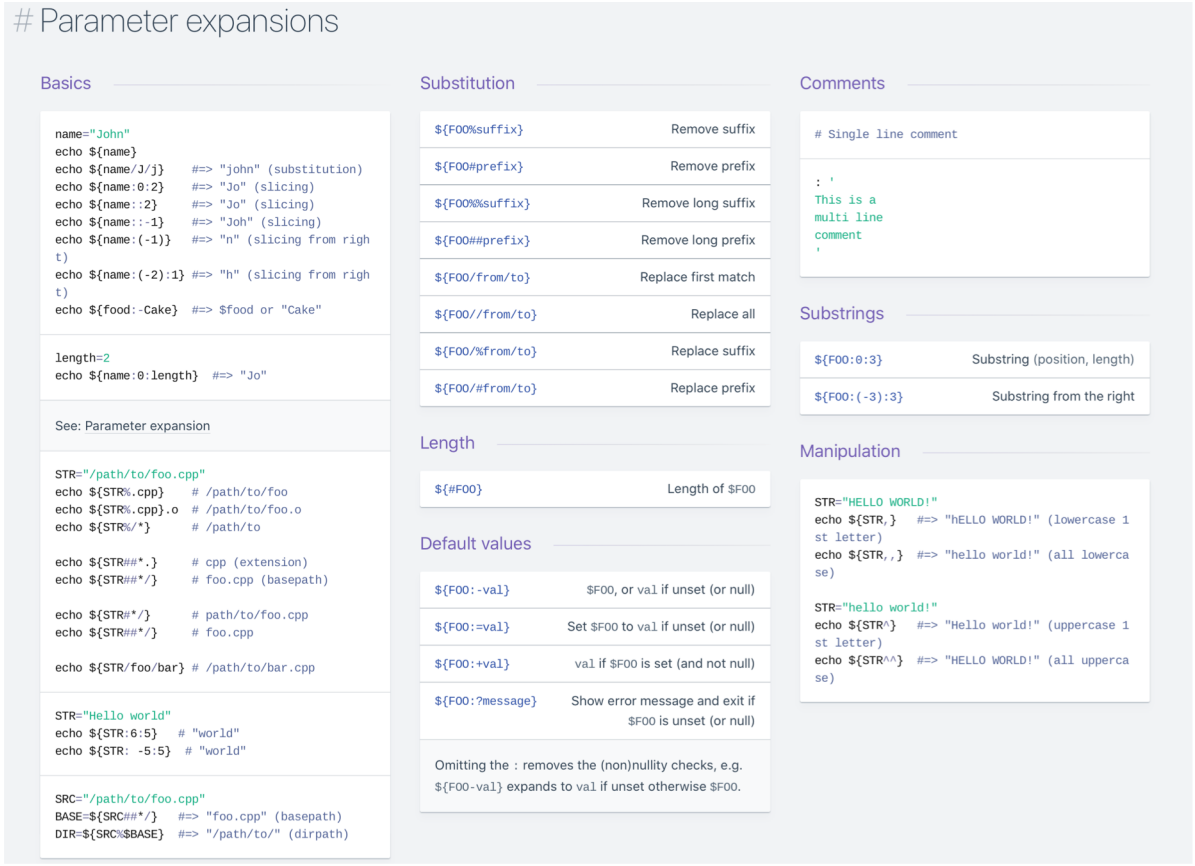
See Command substitution

Conditionals

```
if [[ -z "$string" ]]; then
  echo "String is empty"
elif [[ -n "$string" ]]; then
  echo "String is not empty"
fi
```

See: Conditionals

Good Resource: Bash cheat sheet



Parameter expansions

Basics

```
name="John"
echo ${name}
echo ${name/J/j}    #=> "john" (substitution)
echo ${name:0:2}    #=> "Jo" (slicing)
echo ${name::2}     #=> "Jo" (slicing)
echo ${name::-1}    #=> "Joh" (slicing)
echo ${name:(-1)}   #=> "n" (slicing from right)
echo ${name:(-2):1} #=> "h" (slicing from right)
echo ${food:-Cake}  #=> $food or "Cake"

length=2
echo ${name:0:length}  #=> "Jo"
```

See: Parameter expansion

```
STR="/path/to/foo.cpp"
echo ${STR%.cpp}    # /path/to/foo
echo ${STR%.cpp}.o  # /path/to/foo.o
echo ${STR%/*}      # /path/to

echo ${STR##*.}     # cpp (extension)
echo ${STR##*/}     # foo.cpp (basepath)

echo ${STR#*/}      # path/to/foo.cpp
echo ${STR##*/}     # foo.cpp

echo ${STR/foo/bar} # /path/to/bar.cpp

STR="Hello world"
echo ${STR:6:5}   # "world"
echo ${STR: -5:5}  # "world"

SRC="/path/to/foo.cpp"
BASE=${SRC##*/}   #=> "foo.cpp" (basepath)
DIR=${SRC%$BASE}  #=> "/path/to/" (dirpath)
```

Substitution

`${FOO%suffix}`	Remove suffix
`${FOO#prefix}`	Remove prefix
`${FOO%%suffix}`	Remove long suffix
`${FOO##prefix}`	Remove long prefix
`${FOO/from/to}`	Replace first match
`${FOO//from/to}`	Replace all
`${FOO/%from/to}`	Replace suffix
`${FOO/#from/to}`	Replace prefix

Length

`${#FOO}`	Length of `$FOO`

Default values

`${FOO:-val}`	`$FOO`, or `val` if unset (or null)
`${FOO:=val}`	Set `$FOO` to `val` if unset (or null)
`${FOO:+val}`	`val` if `$FOO` is set (and not null)
`${FOO:?message}`	Show error message and exit if `$FOO` is unset (or null)

Omitting the `:` removes the (non)nullity checks, e.g. `${FOO-val}` expands to `val` if unset otherwise `$FOO`.

Comments

```
# Single line comment

: '
This is a
multi line
comment
'
```

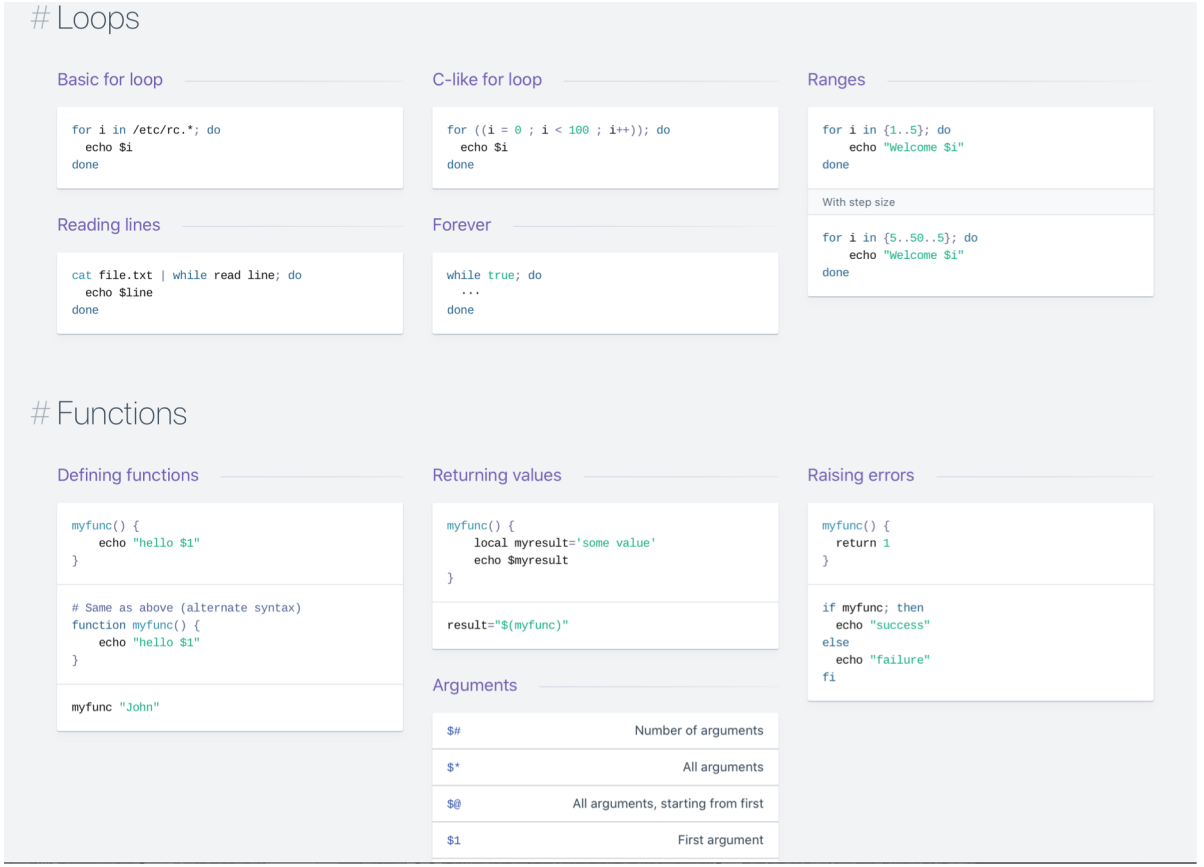
Substrings

`${FOO:0:3}`	Substring (position, length)
`${FOO:(-3):3}`	Substring from the right

Manipulation

```
STR="HELLO WORLD!"
echo ${STR,}   #=> "hELLO WORLD!" (lowercase 1st letter)
echo ${STR,,}  #=> "hello world!" (all lowercase)

STR="hello world!"
echo ${STR^}   #=> "Hello world!" (uppercase 1st letter)
echo ${STR^^}  #=> "HELLO WORLD!" (all uppercase)
```

Good Resource: Bash cheat sheet



Loops

Basic for loop

```
for i in /etc/rc.*; do
  echo $i
done
```

C-like for loop

```
for ((i = 0 ; i < 100 ; i++)); do
  echo $i
done
```

Ranges

```
for i in {1..5}; do
    echo "Welcome $i"
done
```

With step size

```
for i in {5..50..5}; do
    echo "Welcome $i"
done
```

Reading lines

```
cat file.txt | while read line; do
  echo $line
done
```

Forever

```
while true; do
  ...
done
```

Functions

Defining functions

```
myfunc() {
    echo "hello $1"
}
```

```
# Same as above (alternate syntax)
function myfunc() {
    echo "hello $1"
}
```

```
myfunc "John"
```

Returning values

```
myfunc() {
    local myresult='some value'
    echo $myresult
}
```

```
result="$(myfunc)"
```

Raising errors

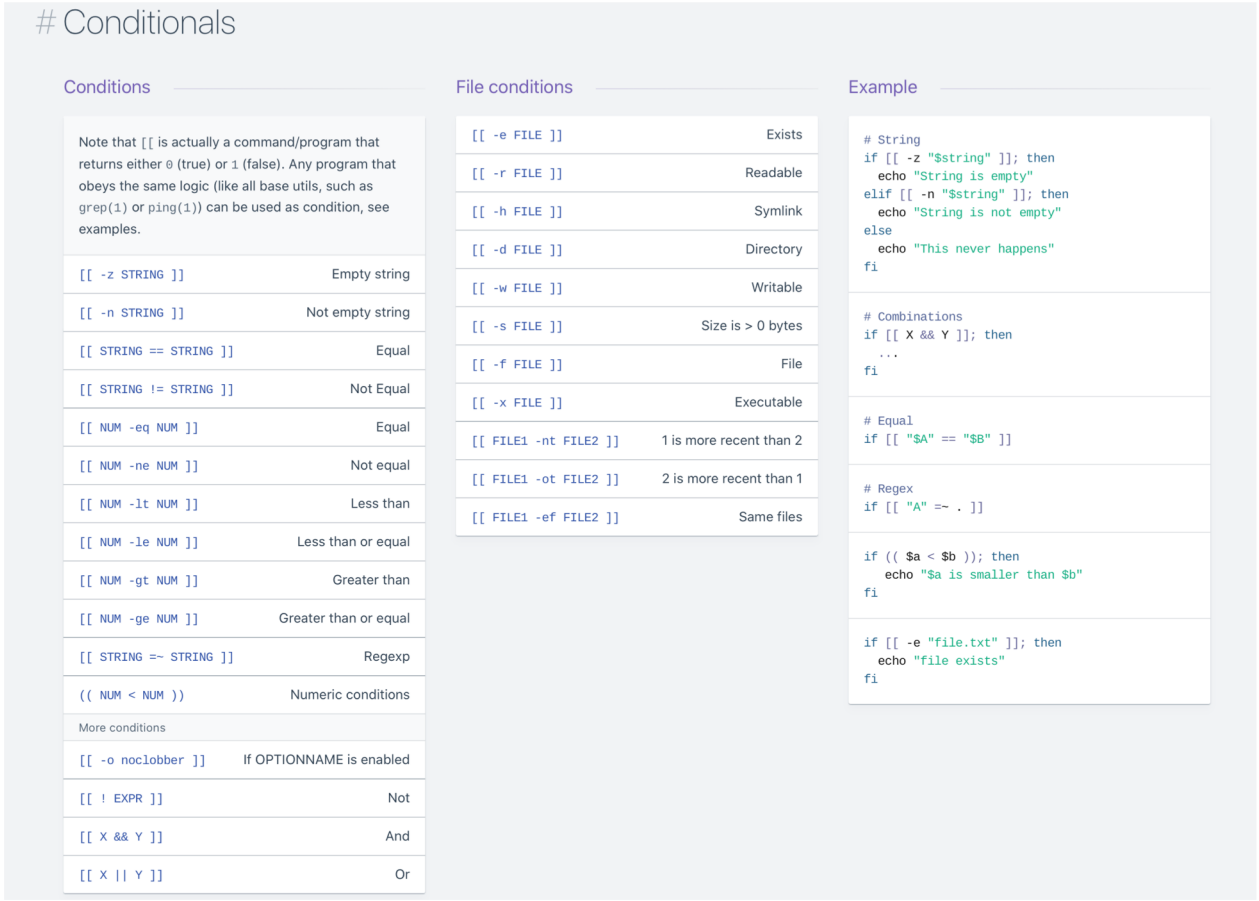
```
myfunc() {
  return 1
}
```

```
if myfunc; then
  echo "success"
else
  echo "failure"
fi
```

Arguments

$#	Number of arguments
$*	All arguments
$@	All arguments, starting from first
$1	First argument

Good Resource: Bash cheat sheet



Conditionals

Conditions

Note that `[[` is actually a command/program that returns either `0` (true) or `1` (false). Any program that obeys the same logic (like all base utils, such as `grep(1)` or `ping(1)`) can be used as condition, see examples.

`[[ -z STRING ]]`	Empty string
`[[ -n STRING ]]`	Not empty string
`[[ STRING == STRING ]]`	Equal
`[[ STRING != STRING ]]`	Not Equal
`[[ NUM -eq NUM ]]`	Equal
`[[ NUM -ne NUM ]]`	Not equal
`[[ NUM -lt NUM ]]`	Less than
`[[ NUM -le NUM ]]`	Less than or equal
`[[ NUM -gt NUM ]]`	Greater than
`[[ NUM -ge NUM ]]`	Greater than or equal
`[[ STRING =~ STRING ]]`	Regexp
`(( NUM < NUM ))`	Numeric conditions

More conditions

`[[ -o noclobber ]]`	If OPTIONNAME is enabled
`[[ ! EXPR ]]`	Not
`[[ X && Y ]]`	And
`[[ X || Y ]]`	Or

File conditions

`[[ -e FILE ]]`	Exists
`[[ -r FILE ]]`	Readable
`[[ -h FILE ]]`	Symlink
`[[ -d FILE ]]`	Directory
`[[ -w FILE ]]`	Writable
`[[ -s FILE ]]`	Size is > 0 bytes
`[[ -f FILE ]]`	File
`[[ -x FILE ]]`	Executable
`[[ FILE1 -nt FILE2 ]]`	1 is more recent than 2
`[[ FILE1 -ot FILE2 ]]`	2 is more recent than 1
`[[ FILE1 -ef FILE2 ]]`	Same files

Example

```
# String
if [[ -z "$string" ]]; then
  echo "String is empty"
elif [[ -n "$string" ]]; then
  echo "String is not empty"
else
  echo "This never happens"
fi
```

```
# Combinations
if [[ X && Y ]]; then
  ...
fi
```

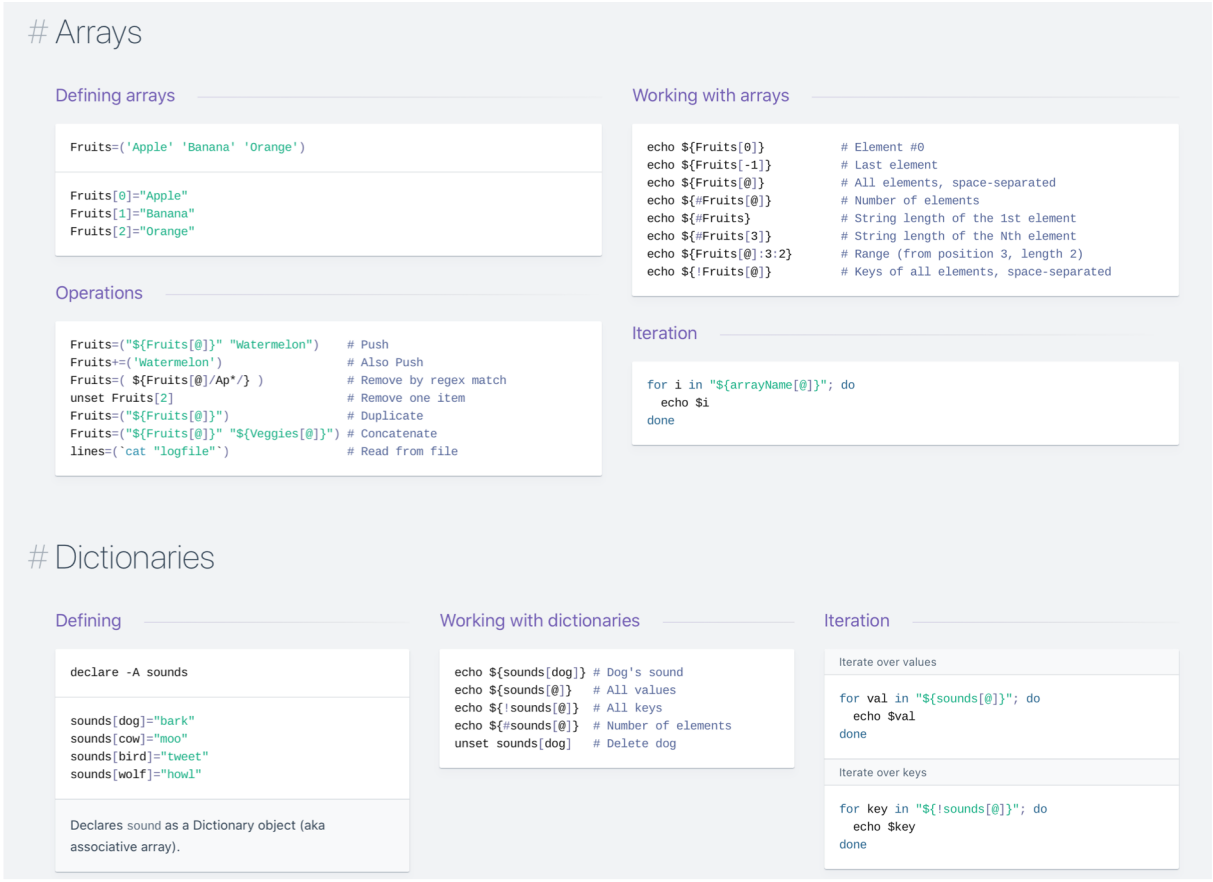
```
# Equal
if [[ "$A" == "$B" ]]
```

```
# Regex
if [[ "A" =~ . ]]
```

```
if (( $a < $b )); then
   echo "$a is smaller than $b"
fi
```

```
if [[ -e "file.txt" ]]; then
  echo "file exists"
fi
```

Good Resource: Bash cheat sheet



Arrays

Defining arrays

```
Fruits=('Apple' 'Banana' 'Orange')
```

```
Fruits[0]="Apple"
Fruits[1]="Banana"
Fruits[2]="Orange"
```

Working with arrays

```
echo ${Fruits[0]}           # Element #0
echo ${Fruits[-1]}          # Last element
echo ${Fruits[@]}           # All elements, space-separated
echo ${#Fruits[@]}          # Number of elements
echo ${#Fruits}             # String length of the 1st element
echo ${#Fruits[3]}          # String length of the Nth element
echo ${Fruits[@]:3:2}       # Range (from position 3, length 2)
echo ${!Fruits[@]}          # Keys of all elements, space-separated
```

Operations

```
Fruits=("${Fruits[@]}" "Watermelon")    # Push
Fruits+=('Watermelon')                  # Also Push
Fruits=( ${Fruits[@]/Ap*/} )            # Remove by regex match
unset Fruits[2]                         # Remove one item
Fruits=("${Fruits[@]}")                 # Duplicate
Fruits=("${Fruits[@]}" "${Veggies[@]}") # Concatenate
lines=(`cat "logfile"`)                 # Read from file
```

Iteration

```
for i in "${arrayName[@]}"; do
  echo $i
done
```

Dictionaries

Defining

```
declare -A sounds
```

```
sounds[dog]="bark"
sounds[cow]="moo"
sounds[bird]="tweet"
sounds[wolf]="howl"
```

Declares sound as a Dictionary object (aka associative array).

Working with dictionaries

```
echo ${sounds[dog]} # Dog's sound
echo ${sounds[@]}   # All values
echo ${!sounds[@]}  # All keys
echo ${#sounds[@]}  # Number of elements
unset sounds[dog]   # Delete dog
```

Iteration

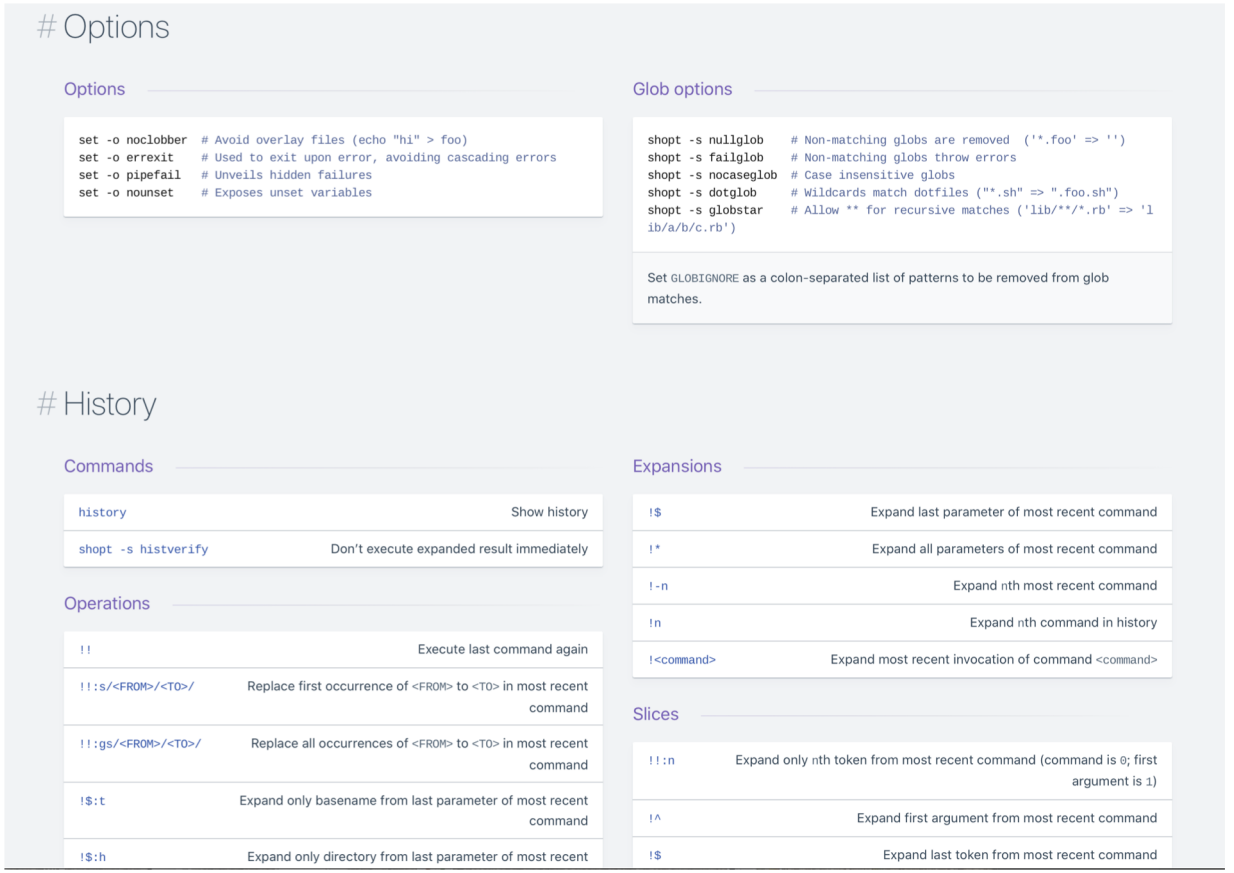
Iterate over values

```
for val in "${sounds[@]}"; do
  echo $val
done
```

Iterate over keys

```
for key in "${!sounds[@]}"; do
  echo $key
done
```

Good Resource: Bash cheat sheet



Options

Options

```
set -o noclobber  # Avoid overlay files (echo "hi" > foo)
set -o errexit    # Used to exit upon error, avoiding cascading errors
set -o pipefail   # Unveils hidden failures
set -o nounset    # Exposes unset variables
```

Glob options

```
shopt -s nullglob    # Non-matching globs are removed  ('*.foo' => '')
shopt -s failglob    # Non-matching globs throw errors
shopt -s nocaseglob  # Case insensitive globs
shopt -s dotglob     # Wildcards match dotfiles ("*.sh" => ".foo.sh")
shopt -s globstar    # Allow ** for recursive matches ('lib/**/*.rb' => 'lib/a/b/c.rb')
```

Set `GLOBIGNORE` as a colon-separated list of patterns to be removed from glob matches.

History

Commands

`history`	Show history
`shopt -s histverify`	Don't execute expanded result immediately

Operations

`!!`	Execute last command again
`!!:s/<FROM>/<TO>/`	Replace first occurrence of `<FROM>` to `<TO>` in most recent command
`!!:gs/<FROM>/<TO>/`	Replace all occurrences of `<FROM>` to `<TO>` in most recent command
`!$:t`	Expand only basename from last parameter of most recent command
`!$:h`	Expand only directory from last parameter of most recent

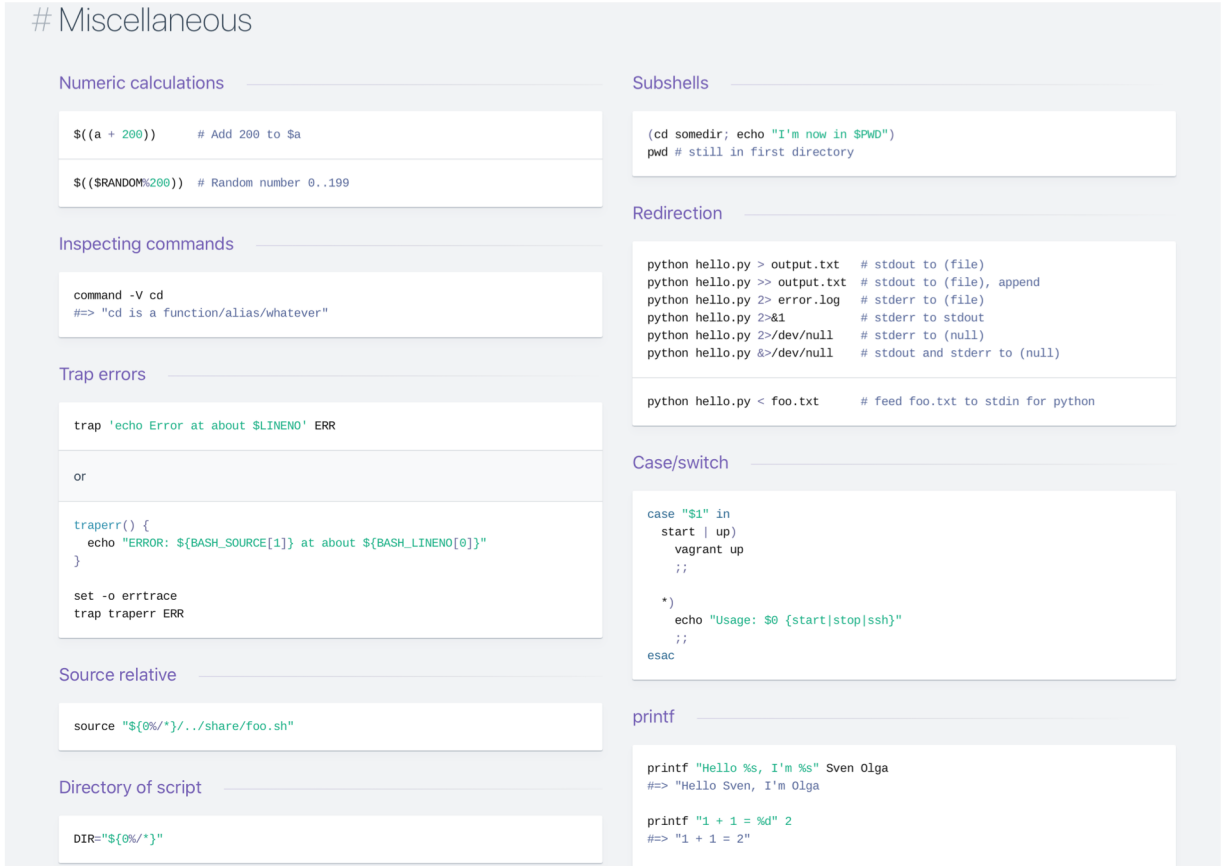
Expansions

`!$`	Expand last parameter of most recent command
`!*`	Expand all parameters of most recent command
`!-n`	Expand nth most recent command
`!n`	Expand nth command in history
`!<command>`	Expand most recent invocation of command `<command>`

Slices

`!!:n`	Expand only nth token from most recent command (command is `0`; first argument is `1`)
`!^`	Expand first argument from most recent command
`!$`	Expand last token from most recent command

Good Resource: Bash cheat sheet



Miscellaneous

Numeric calculations

```
$((a + 200))      # Add 200 to $a
```

```
$(($RANDOM%200))  # Random number 0..199
```

Inspecting commands

```
command -V cd
#=> "cd is a function/alias/whatever"
```

Trap errors

```
trap 'echo Error at about $LINENO' ERR
```

or

```
traperr() {
  echo "ERROR: ${BASH_SOURCE[1]} at about ${BASH_LINENO[0]}"
}

set -o errtrace
trap traperr ERR
```

Source relative

```
source "${0%/*}/../share/foo.sh"
```

Directory of script

```
DIR="${0%/*}"
```

Subshells

```
(cd somedir; echo "I'm now in $PWD")
pwd # still in first directory
```

Redirection

```
python hello.py > output.txt   # stdout to (file)
python hello.py >> output.txt  # stdout to (file), append
python hello.py 2> error.log   # stderr to (file)
python hello.py 2>&1           # stderr to stdout
python hello.py 2>/dev/null    # stderr to (null)
python hello.py &>/dev/null    # stdout and stderr to (null)
```

```
python hello.py < foo.txt      # feed foo.txt to stdin for python
```

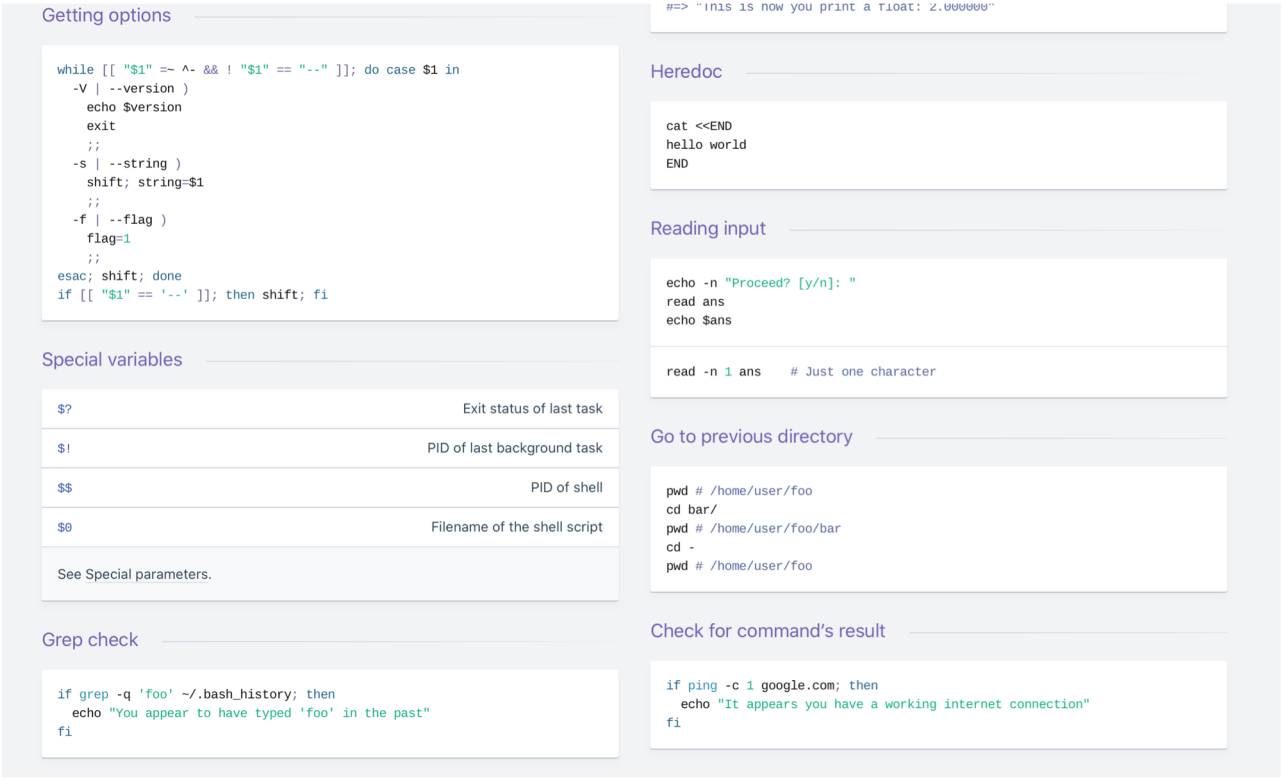
Case/switch

```
case "$1" in
  start | up)
    vagrant up
    ;;

  *)
    echo "Usage: $0 {start|stop|ssh}"
    ;;
esac
```

printf

```
printf "Hello %s, I'm %s" Sven Olga
#=> "Hello Sven, I'm Olga

printf "1 + 1 = %d" 2
#=> "1 + 1 = 2"
```

Good Resource: Bash cheat sheet



Getting options

```
while [[ "$1" =~ ^- && ! "$1" == "--" ]]; do case $1 in
  -V | --version )
    echo $version
    exit
    ;;
  -s | --string )
    shift; string=$1
    ;;
  -f | --flag )
    flag=1
    ;;
esac; shift; done
if [[ "$1" == '--' ]]; then shift; fi
```

Special variables

$?	Exit status of last task
$!	PID of last background task
$$	PID of shell
$0	Filename of the shell script

See Special parameters.

Grep check

```
if grep -q 'foo' ~/.bash_history; then
  echo "You appear to have typed 'foo' in the past"
fi
```

```
#=> "This is how you print a float: 2.000000"
```

Heredoc

```
cat <<END
hello world
END
```

Reading input

```
echo -n "Proceed? [y/n]: "
read ans
echo $ans
```

```
read -n 1 ans    # Just one character
```

Go to previous directory

```
pwd # /home/user/foo
cd bar/
pwd # /home/user/foo/bar
cd -
pwd # /home/user/foo
```

Check for command's result

```
if ping -c 1 google.com; then
  echo "It appears you have a working internet connection"
fi
```