Lecture 10

408/508 *Computational Techniques for Linguists*

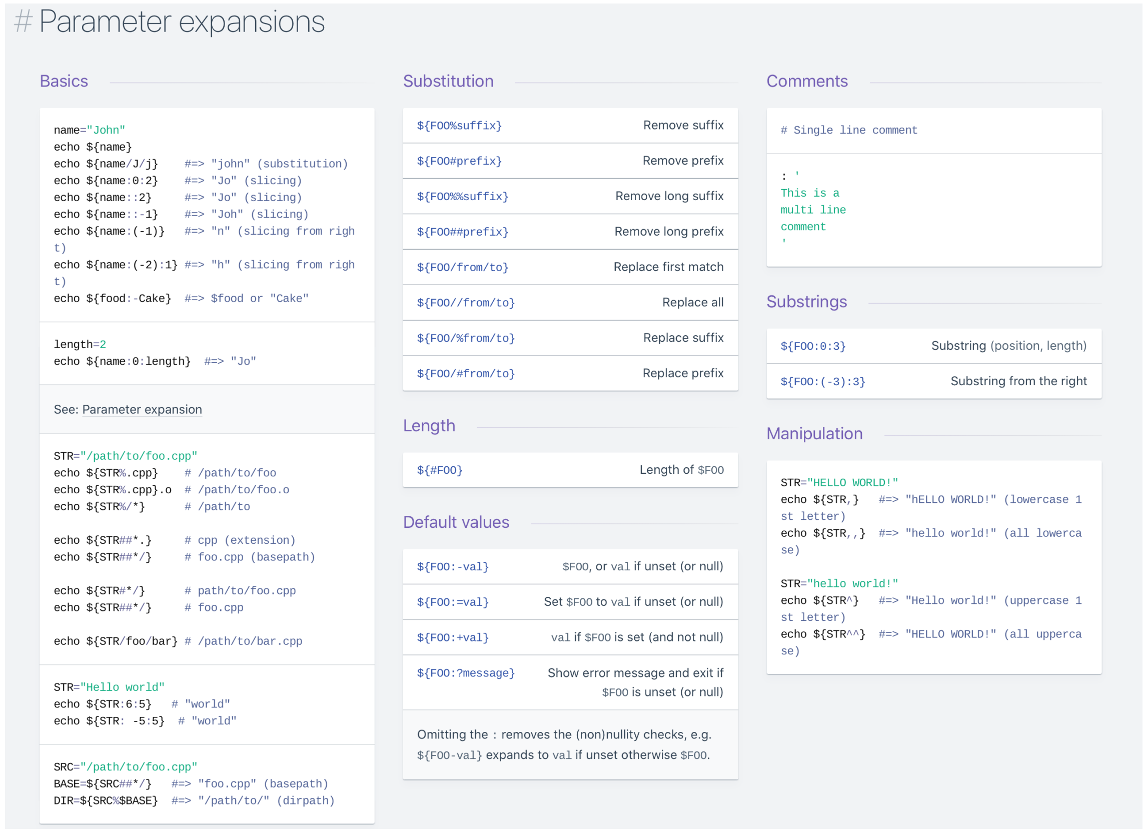
Last Time

- Usefulness of (*filename*) **expansion** on the command line
  - to rename files:
    - `mv "$filename" "${filename%suffix}newsuffix"`
    - e.g. `JPG` to `jpg`
  - to backup files:
    - `for file in f{1..3}.jpg; do cp $file $file.bak; done`

% = shortest suffix

https://devhints.io/bash

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 righ
t)
echo ${name:(-2):1} #=> "h" (slicing from righ
t)
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

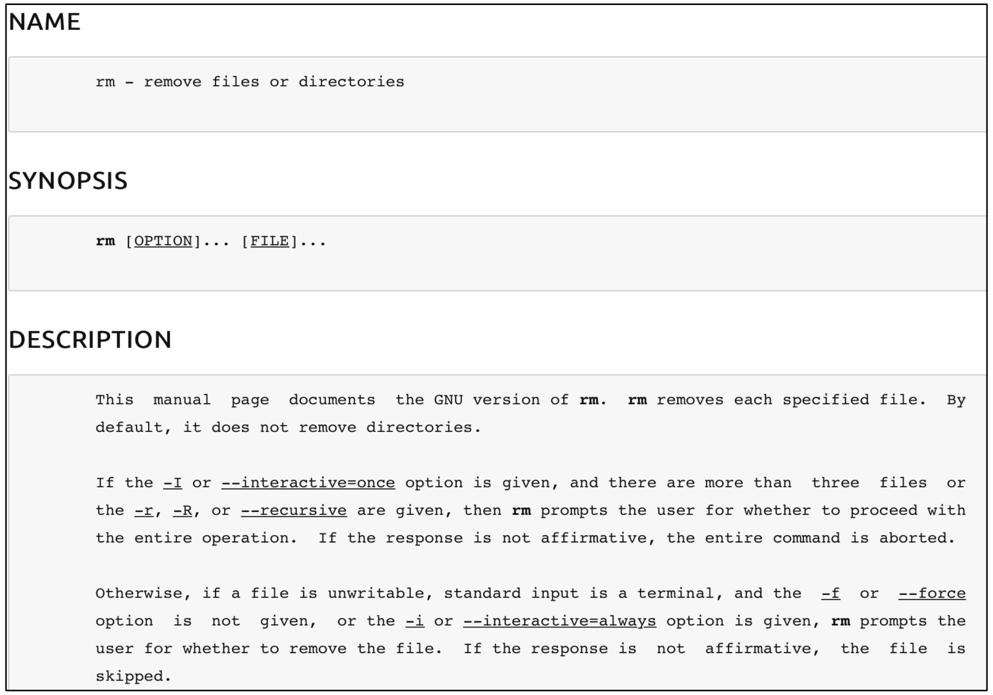
```
STR="HELLO WORLD!"
echo ${STR,}   #=> "hELLO WORLD!" (lowercase 1
st letter)
echo ${STR,,}  #=> "hello world!" (all lowerca
se)

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

Today's Topics

- Final lecture on bash
  - *we start again with something more friendly next week*
  - Four exercises today:
    1. deleting files
    2. double-spacing a file
    3. removing blank lines from a file
    4. find with sed example

Exercise 1: deleting files - `rm`

- `man rm`



NAME

```
       rm - remove files or directories
```

SYNOPSIS

```
       rm [OPTION]... [FILE]...
```

DESCRIPTION

```
       This  manual  page  documents  the GNU version of rm.  rm removes each specified file.  By
       default, it does not remove directories.

       If the -I or --interactive=once option is given, and there are more than  three  files  or
       the -r, -R, or --recursive are given, then rm prompts the user for whether to proceed with
       the entire operation.  If the response is not affirmative, the entire command is aborted.

       Otherwise, if a file is unwritable, standard input is a terminal, and the  -f  or  --force
       option  is  not  given,  or the -i or --interactive=always option is given, rm prompts the
       user for whether to remove the file.  If the response is  not  affirmative,  the  file  is
       skipped.
```

Exercise 1: deleting files

Remove File/Directory

• `rm` *`FILEPATTERN`*	removes a file or files, e.g. * (*dangerous*!), any expansion pattern (*we've seen*)
• `rm –d` *`DIR`*	removes a directory (assuming directory is *empty*)
• `rm –r` *`FILEPATTERN`*	recursive remove (*extreme danger*!)
• `rm –rf` *`FILEPATTERN`*	forced recursive remove (!!!)

- Examples:
  - **`touch file.txt`**
  - `rm file.txt` (*you have default write permission*)
  - `touch protected.txt`
  - `chmod u-w protected.txt` (*u = user, -w = remove write permission*)
  - `rm protected.txt`

```
override r––r––r––  sandiway/staff for protected.txt?
```

  - `rm –f protected.txt` (*no interaction*: *forced removal*)
  - `rm –i file.txt` (*ask it to ask you for confirmation*)

```
remove file.txt?
```

Exercise 1: deleting files

best used in interactive shell

- *can put alias shortcut in Terminal startup ~/.bash_profile (MacOS) or ~/.bashrc*

At least two reasons:
1. another computer
2. shell scripts

- `alias rm="rm -i"` not recursively expanded (*considered dangerous*: *why*?)
- `alias` (*list defined aliases*)
- `unalias rm` (*remove alias*)
- Aliases don't work in shell scripts (`rm.sh` *on course website*):

```
#!/bin/bash
if [ $# -ne 1 ]; then
    echo "usage: filename"
    exit 1
fi
touch $1
rm $1
```

rm -i won't be called!

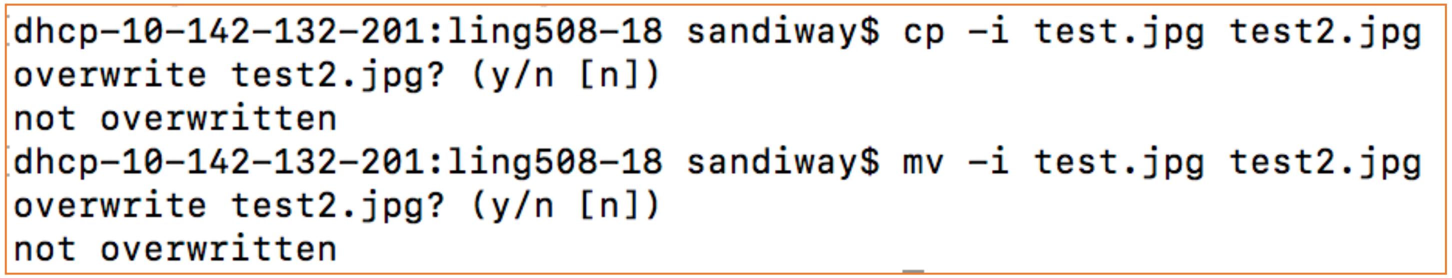
define a function in ~/.bash_profile
(absolute path: otherwise recursively defined)

```
rm () {
    /bin/rm -i "$@"
}
export -f rm
```

Other commands with `-i`

- `-i` (interactive confirm option)

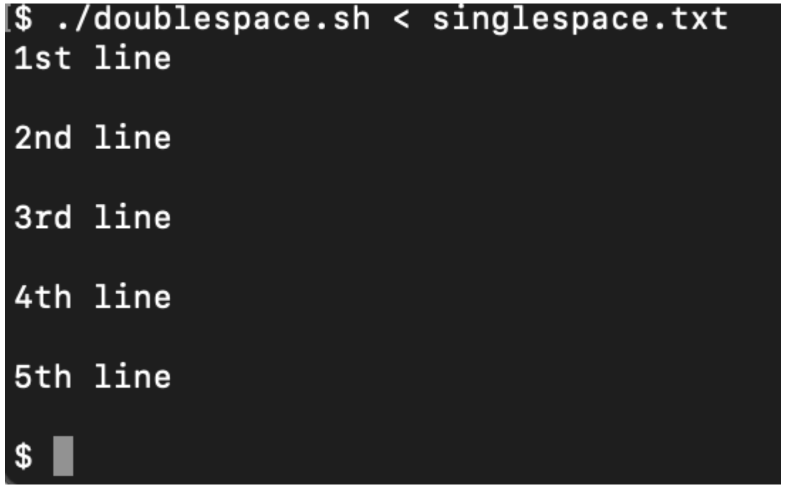
  before overwriting a file
  - `mv -i` *rename file*
  - `cp -i` *copy file*



```
dhcp-10-142-132-201:ling508-18 sandiway$ cp -i test.jpg test2.jpg
overwrite test2.jpg? (y/n [n])
not overwritten
dhcp-10-142-132-201:ling508-18 sandiway$ mv -i test.jpg test2.jpg
overwrite test2.jpg? (y/n [n])
not overwritten
```

Exercise 2: double-spacing a text file

- Write a script that reads each line of a file, then writes the line back out, but with an extra blank line following. This has the effect of *double-spacing* the file.



```
$ ./doublespace.sh < singlespace.txt
1st line

2nd line

3rd line

4th line

5th line

$ 
```

Note:

- `< filename` means take input from `filename`

What you need to know to solve this:

1. `read`
2. `test [[ … ]]`
3. while loop

Exercise 2: double-spacing a text file

- *double-spacing* the file (`doublespace.sh`):

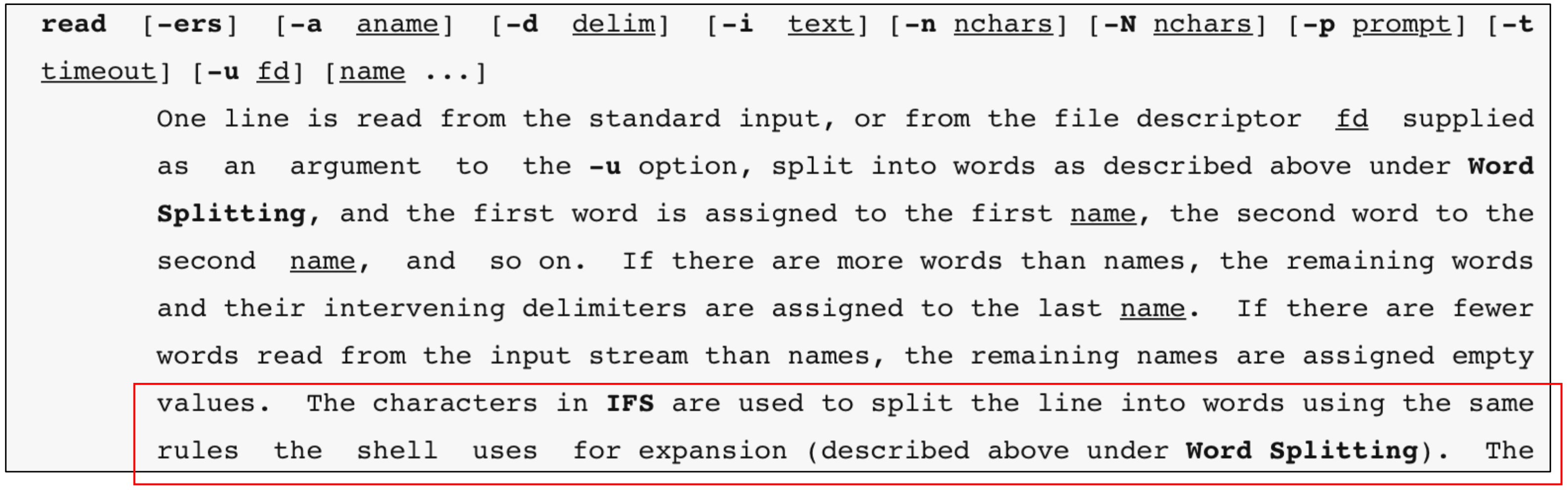
```
#!/bin/bash
read ln
while [[ -n $ln  ]]; do
     echo $ln
     echo
     read ln
done
```

```
while [ -n "$ln" ]; do
```
also works

−n = non-zero

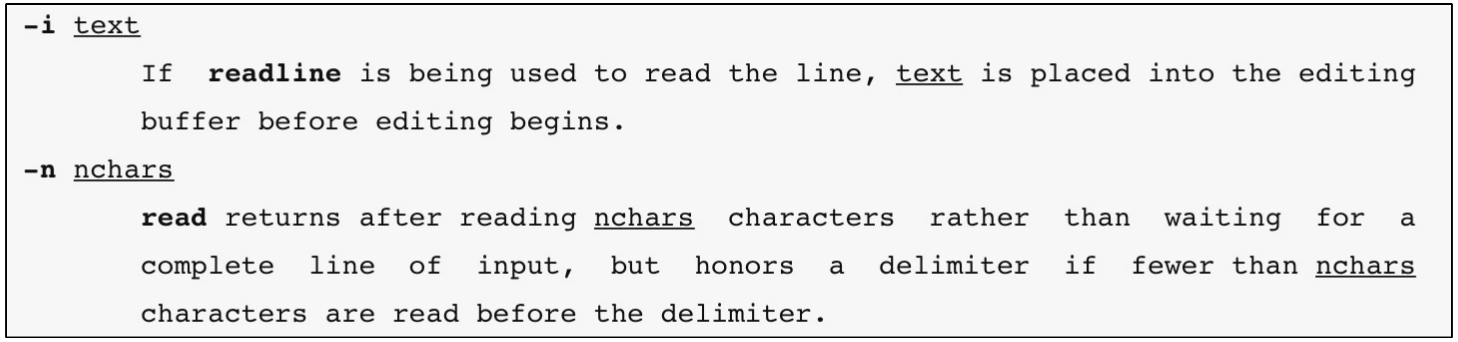
`read –r`
If this option is given, backslash does not act as an escape character.

Exercise 2: read



```
read  [-ers]  [-a  aname]  [-d  delim]  [-i  text] [-n nchars] [-N nchars] [-p prompt] [-t
timeout] [-u fd] [name ...]
          One line is read from the standard input, or from the file descriptor  fd  supplied
          as  an  argument  to  the -u option, split into words as described above under Word
          Splitting, and the first word is assigned to the first name, the second word to the
          second  name,  and  so on.  If there are more words than names, the remaining words
          and their intervening delimiters are assigned to the last name.  If there are fewer
          words read from the input stream than names, the remaining names are assigned empty
          values.  The characters in IFS are used to split the line into words using the same
          rules  the  shell  uses  for expansion (described above under Word Splitting).  The
```

Exercise 2: read



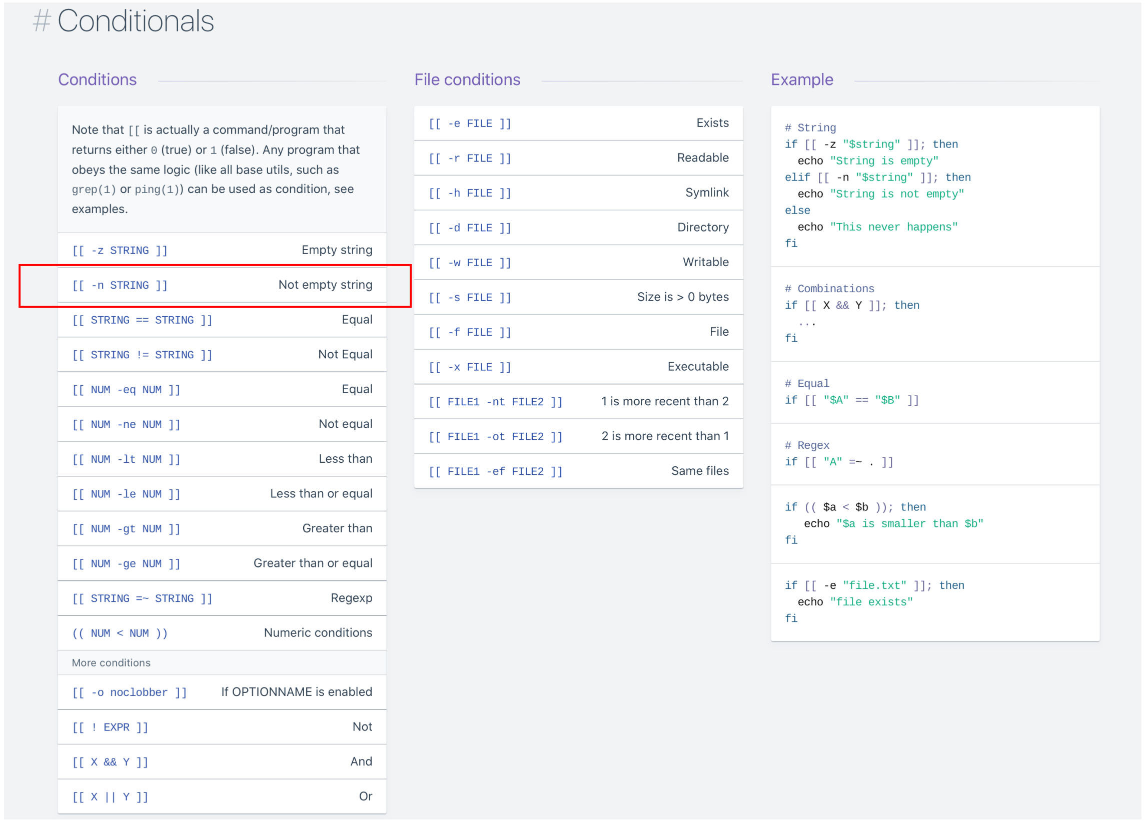
```
-i text
       If  readline is being used to read the line, text is placed into the editing
       buffer before editing begins.
-n nchars
       read returns after reading nchars  characters  rather  than  waiting  for  a
       complete  line  of  input,  but  honors  a  delimiter  if  fewer than nchars
       characters are read before the delimiter.
```

Exercise 2: read

```
-p prompt
       Display prompt  on  standard  error,  without  a  trailing  newline,  before
       attempting  to  read  any  input.   The prompt is displayed only if input is
       coming from a terminal.
-r     Backslash does not act as an escape character.  The backslash is  considered
       to  be part of the line.  In particular, a backslash-newline pair may not be
       used as a line continuation.
-s     Silent mode.  If input is coming from a terminal, characters are not echoed.
-t timeout
       Cause read to time out and return failure if a complete line of input (or  a
       specified number of characters) is not read within timeout seconds.  timeout
       may be a decimal number with a  fractional  portion  following  the  decimal
       point.   This  option  is  only  effective  if  read is reading input from a
       terminal, pipe, or other special file; it has no effect  when  reading  from
       regular  files.   If  read times out, read saves any partial input read into
       the specified variable name.  If timeout is  0,  read  returns  immediately,
```

https://devhints.io/bash

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

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

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

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

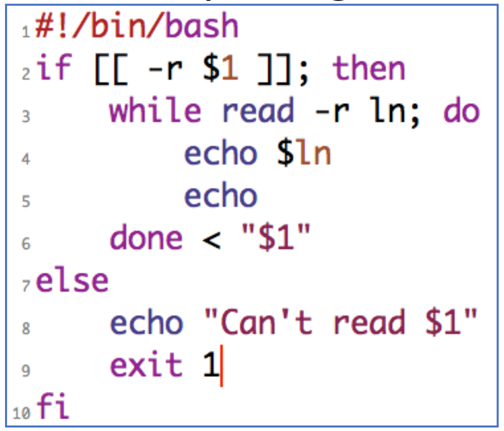
```bash
# Regex
if [[ "A" =~ . ]]
```

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

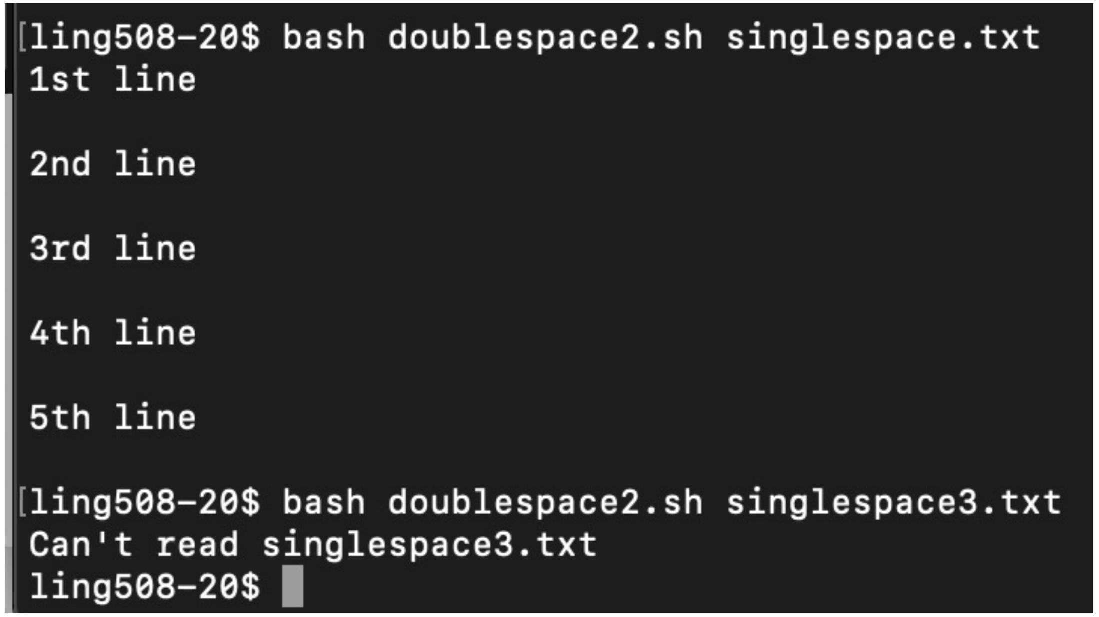
```bash
if [[ -e "file.txt" ]]; then
  echo "file exists"
fi
```

Exercise 2b: double-spacing from filename

- *double-spacing* the file (`doublespace2.sh`):



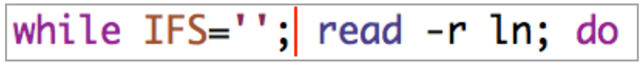
```
#!/bin/bash
if [[ -r $1 ]]; then
    while read -r ln; do
        echo $ln
        echo
    done < "$1"
else
    echo "Can't read $1"
    exit 1
fi
```



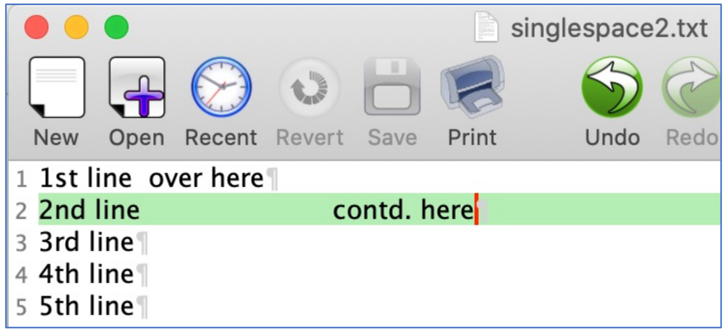
```
ling508-20$ bash doublespace2.sh singlespace.txt
1st line

2nd line

3rd line

4th line

5th line

ling508-20$ bash doublespace2.sh singlespace3.txt
Can't read singlespace3.txt
ling508-20$
```

Exercise 2b: double-spacing from filename

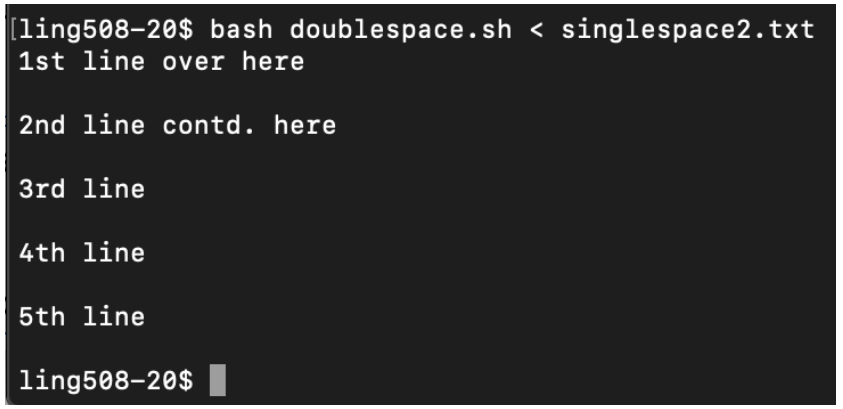
- **whitespace trim problem workaround:**



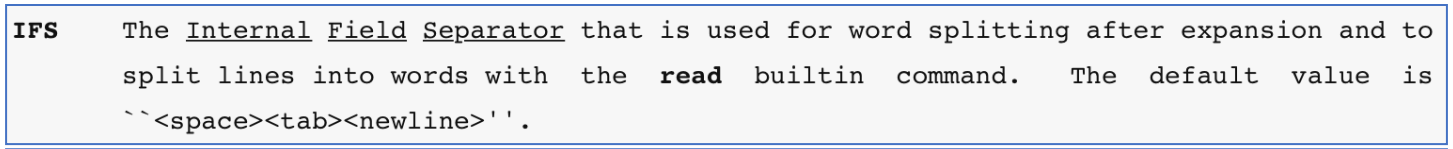
```
while IFS='';read -r ln; do
```



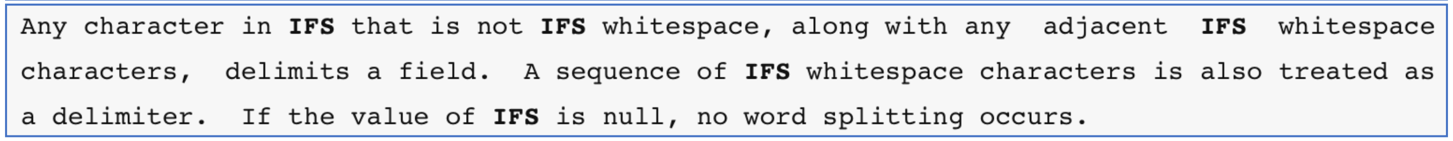
```
1st line  over here
2nd line                    contd. here
3rd line
4th line
5th line
```



```
ling508-20$ bash doublespace.sh < singlespace2.txt
1st line over here

2nd line contd. here

3rd line

4th line

5th line

ling508-20$ 
```



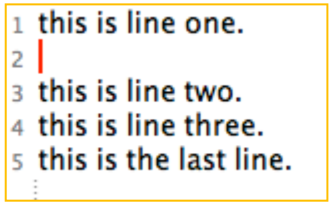
IFS The Internal Field Separator that is used for word splitting after expansion and to split lines into words with the **read** builtin command. The default value is ``<space><tab><newline>''.



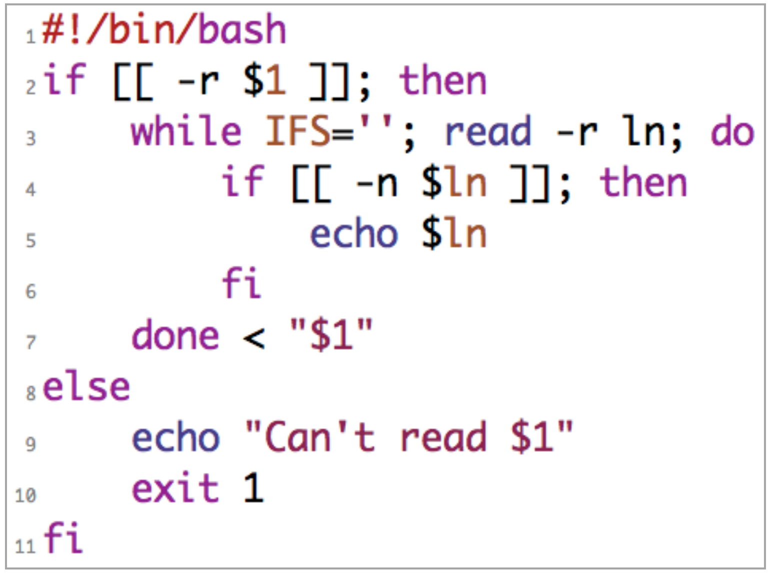
Any character in **IFS** that is not **IFS** whitespace, along with any adjacent **IFS** whitespace characters, delimits a field. A sequence of **IFS** whitespace characters is also treated as a delimiter. If the value of **IFS** is null, no word splitting occurs.

Exercise 3: all except blank lines

- **Changing the line spacing of a text file:**
- write a script to echo all lines of a file except for blank lines (`nonblank.sh`).



```
this is line one.

this is line two.
this is line three.
this is the last line.
```



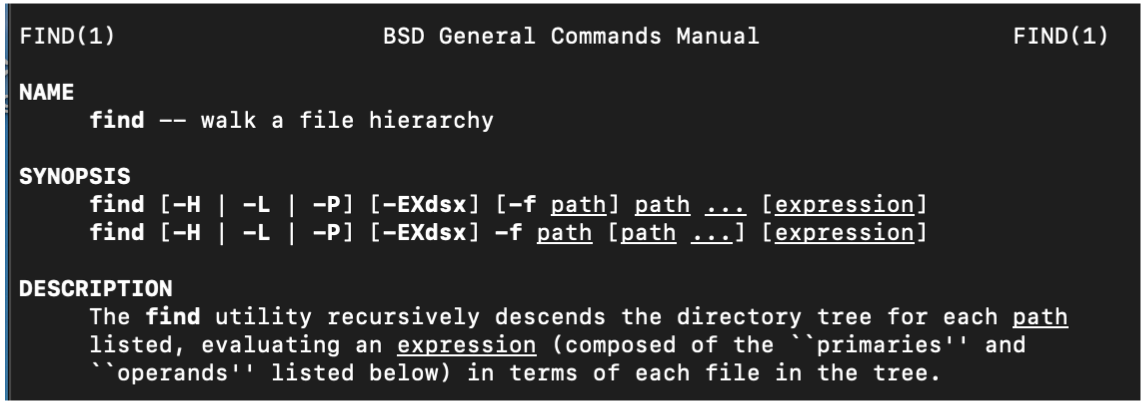
```bash
#!/bin/bash
if [[ -r $1 ]]; then
    while IFS=''; read -r ln; do
        if [[ -n $ln ]]; then
            echo $ln
        fi
    done < "$1"
else
    echo "Can't read $1"
    exit 1
fi
```

Exercise 4: `find` with `sed`

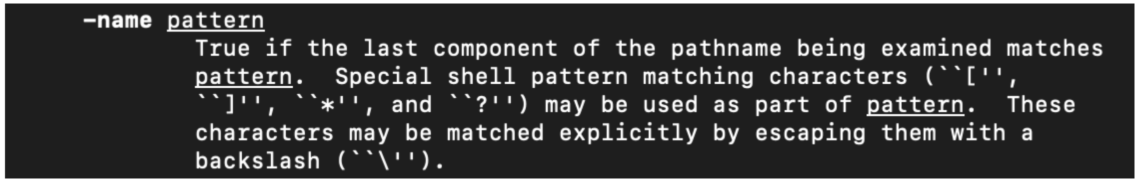
- Using `sed` to edit all .html files in a directory
  - combine with `find -exec … {} \;`
  - `{}` is the placeholder for each filename found by find
  - `\;` ensures `;` is passed to find, lets find know the end of the -exec command
    - `;` is escaped because it is also the shell command separator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

```
1. grep 'see footnote 3' *.html
2. find . -name '*.html' -print
3. find . -name '*.html' -print -exec sed -i.bak 's/see
   footnote 3/see footnote 4/' {} \;
4. grep 'see footnote 3' *.html
```

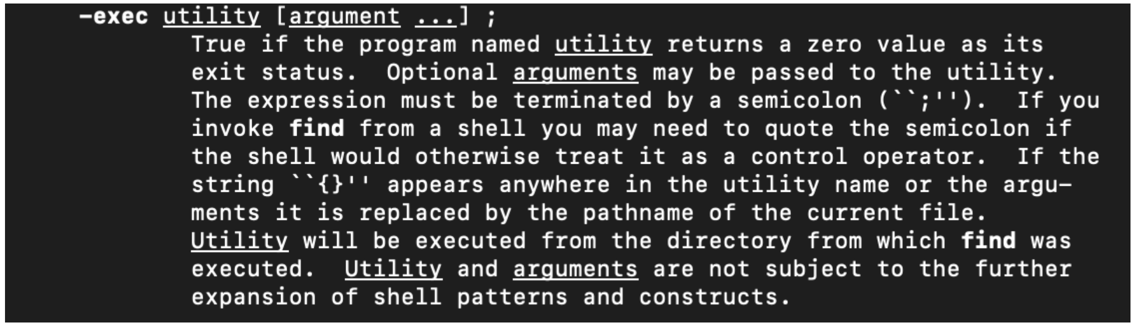
Exercise 4: `find`



```
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.
```

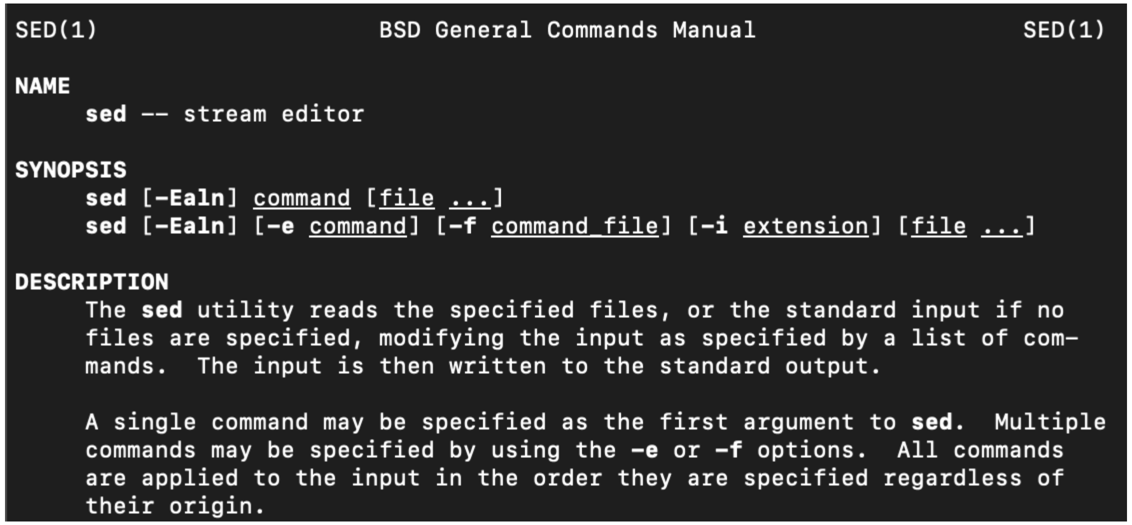


```
     -name pattern
             True if the last component of the pathname being examined matches
             pattern.  Special shell pattern matching characters (``['',
             ``]'', ``*'', and ``?'') may be used as part of pattern.  These
             characters may be matched explicitly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

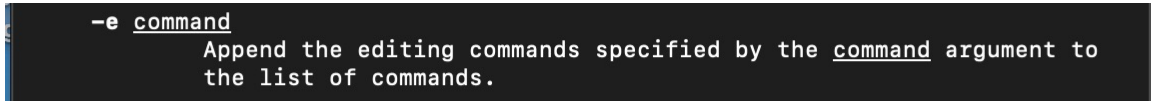


```
     -exec utility [argument ...] ;
             True if the program named utility returns a zero value as its
             exit status.  Optional arguments may be passed to the utility.
             The expression must be terminated by a semicolon (``;'').  If you
             invoke find from a 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.
```

Exercise 4: `sed`



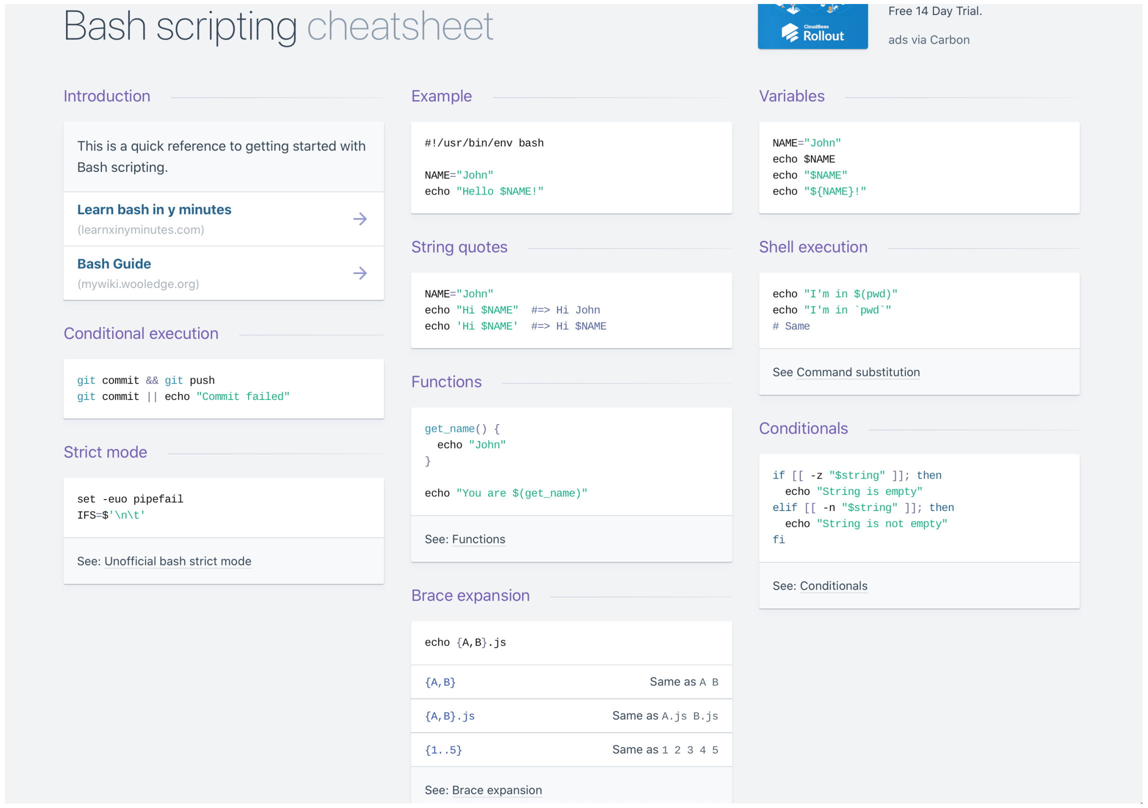
```
SED(1)                    BSD General Commands Manual                   SED(1)

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.
```

https://devhints.io/bash

Good Resource: Bash cheat sheet



Free 14 Day Trial.

ads via Carbon

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

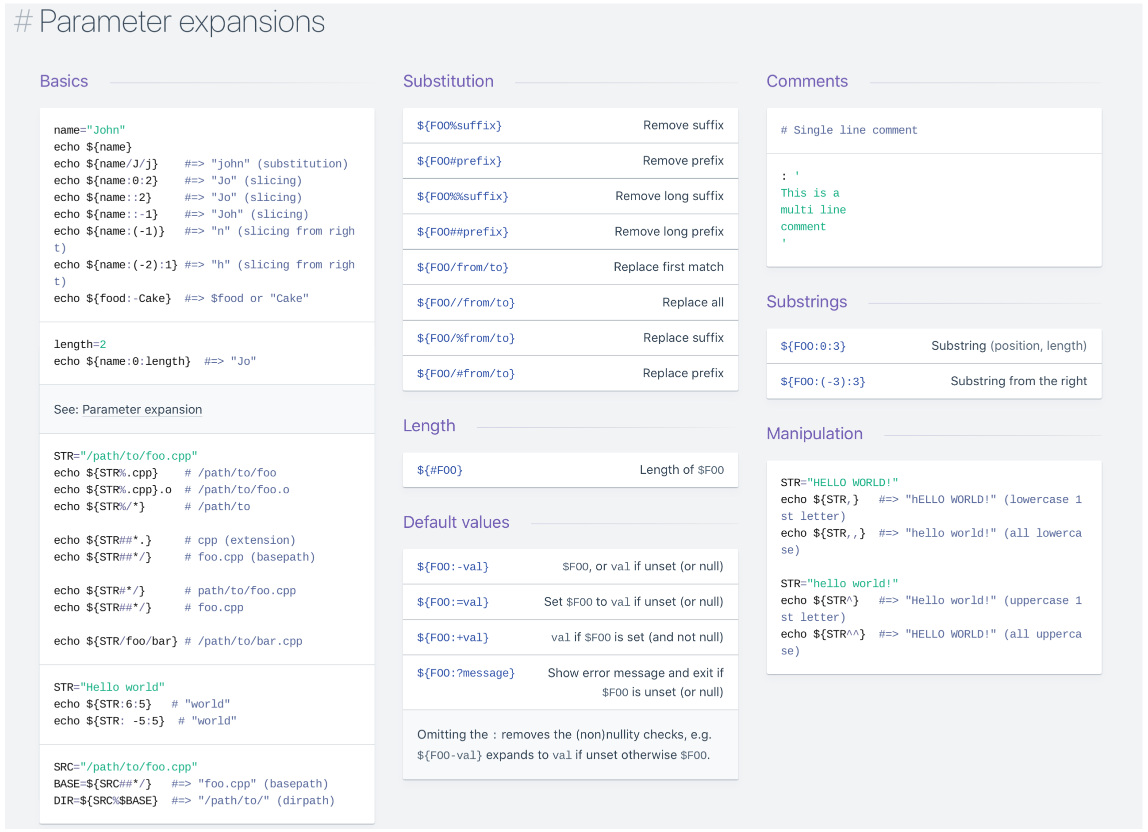
Conditionals

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

See: Conditionals

https://devhints.io/bash

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 righ
t)
echo ${name:(-2):1} #=> "h" (slicing from righ
t)
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

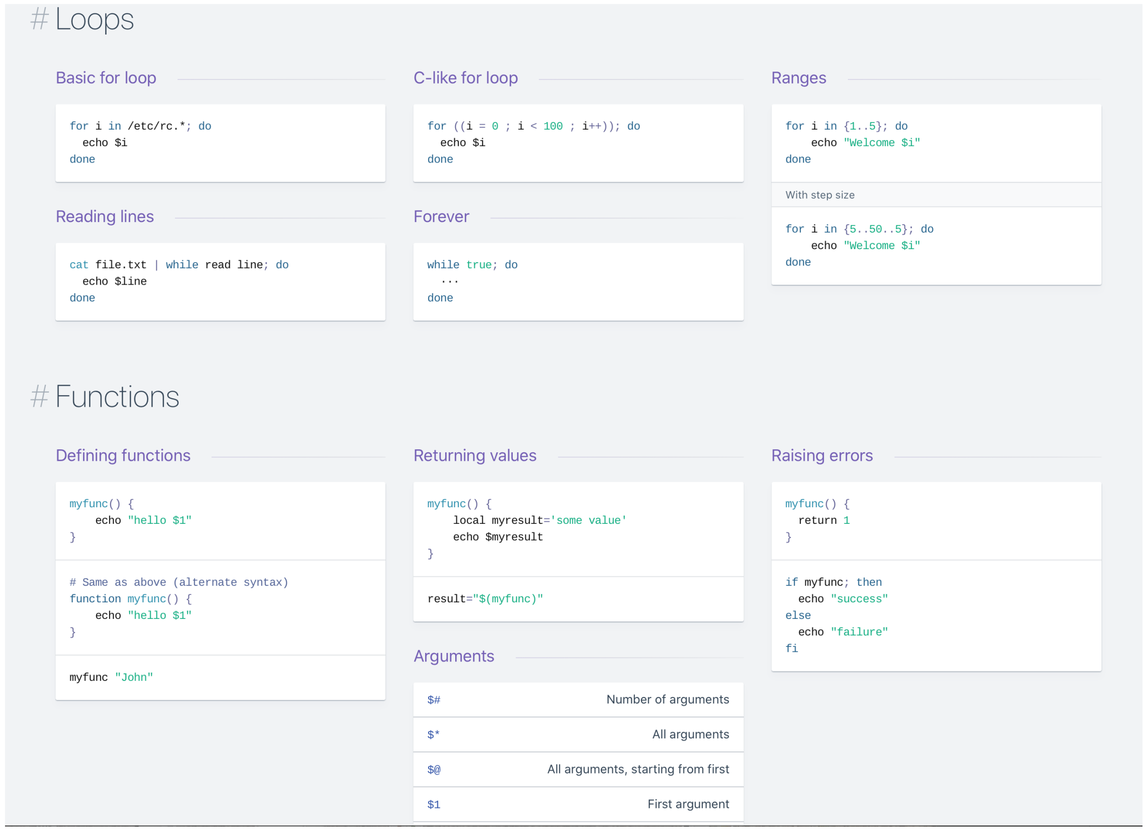
`${FOO:0:3}`	Substring (position, length)
`${FOO:(-3):3}`	Substring from the right

Manipulation

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

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

https://devhints.io/bash

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)"
```

Arguments

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

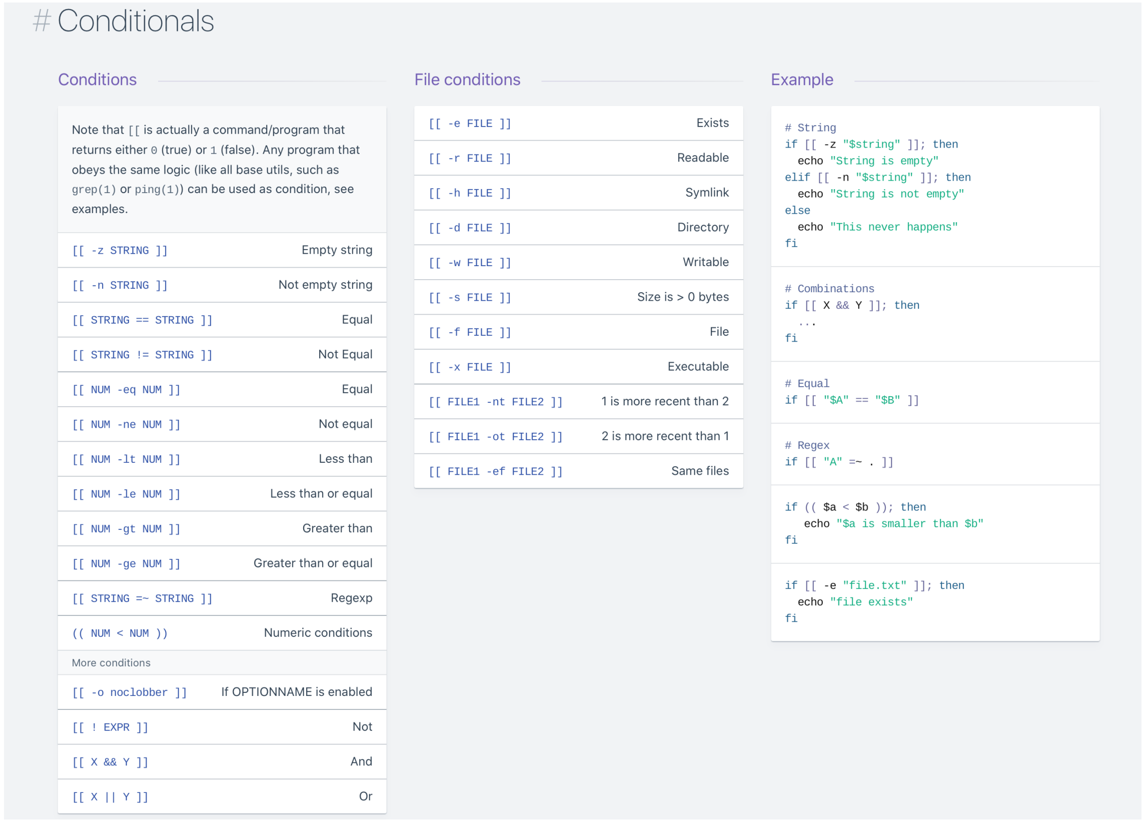
Raising errors

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

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

https://devhints.io/bash

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

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

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

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

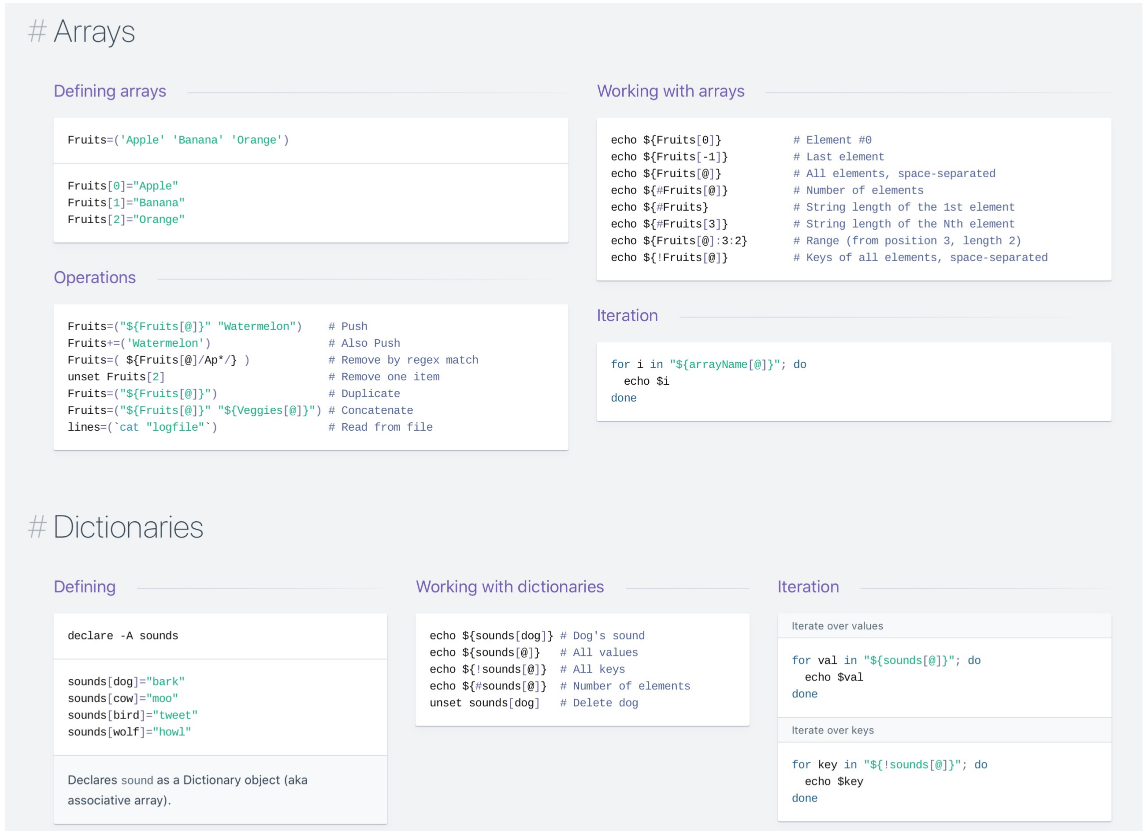
```bash
# Regex
if [[ "A" =~ . ]]
```

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

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

https://devhints.io/bash

Good Resource: Bash cheat sheet



Arrays

Defining arrays

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

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

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
```

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
```

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

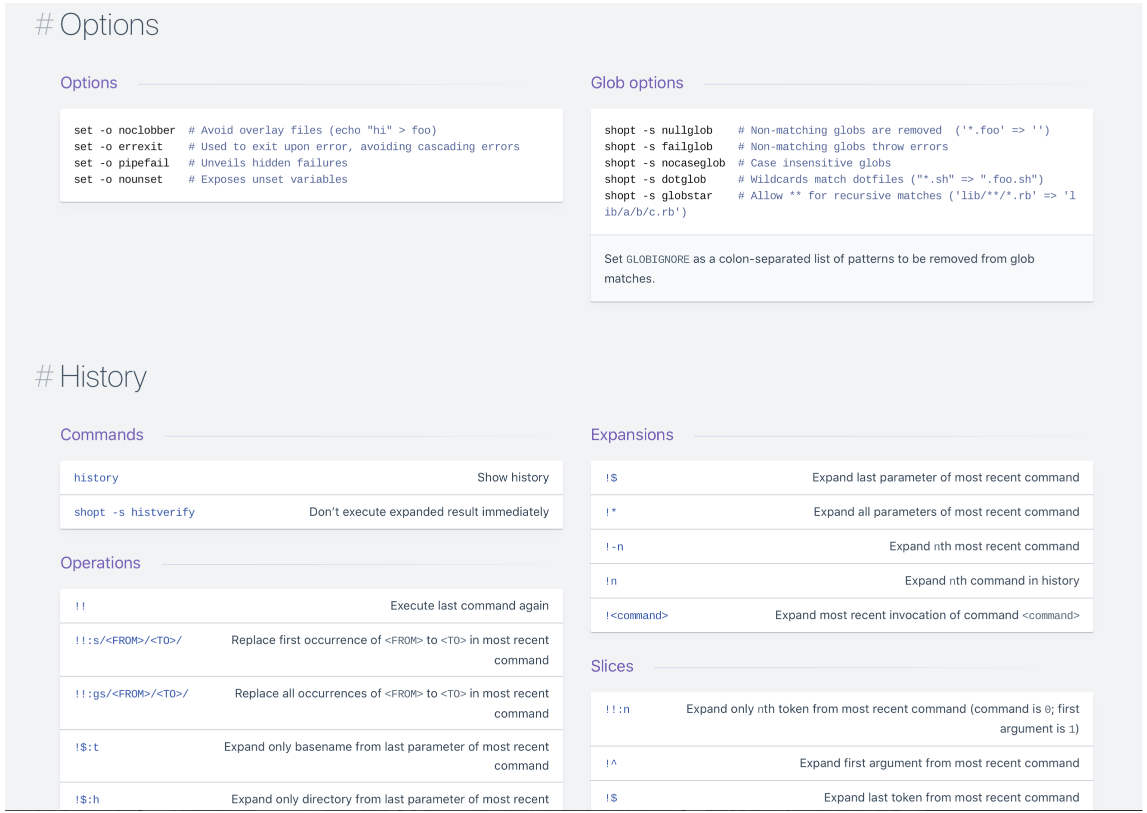
```
for val in "${sounds[@]}"; do
  echo $val
done
```

Iterate over keys

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

https://devhints.io/bash

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' => 'l
ib/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

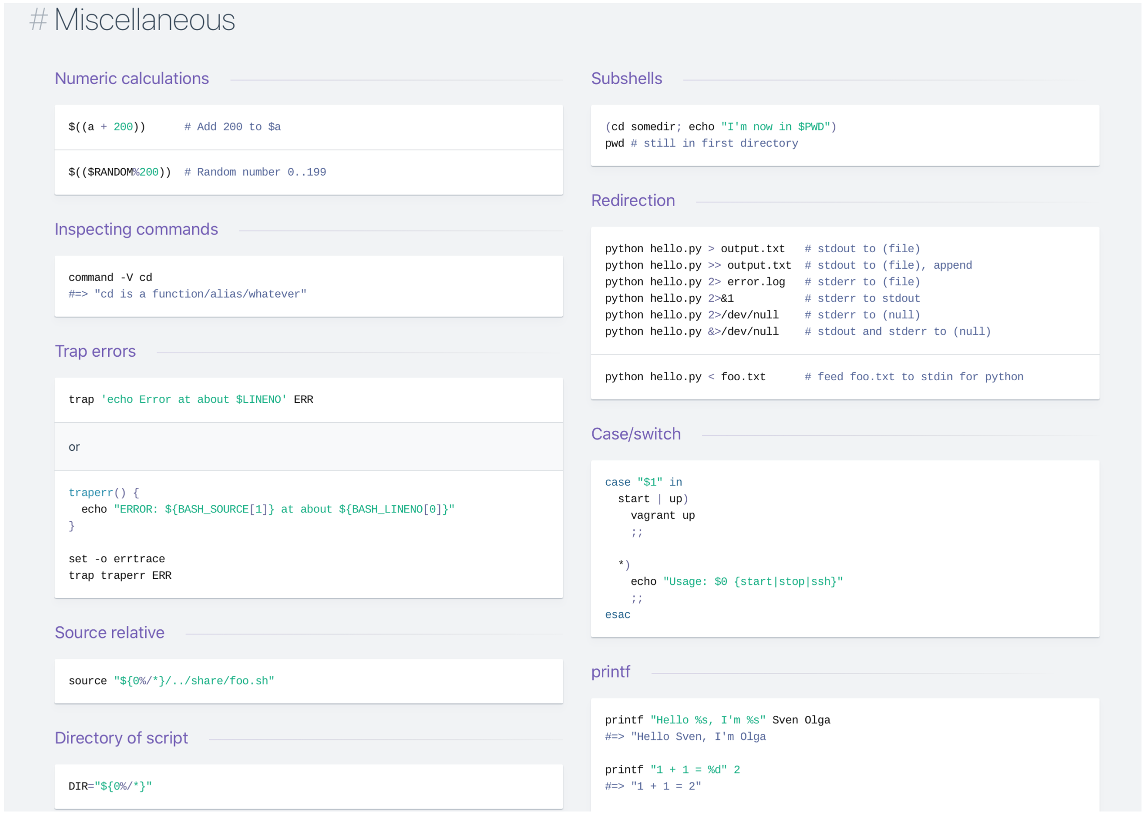
`!$`	Expand last parameter of most recent command
`!*`	Expand all parameters of most recent command
`!-n`	Expand `n`th most recent command
`!n`	Expand `n`th command in history
`!<command>`	Expand most recent invocation of command `<command>`

Slices

`!!:n`	Expand only `n`th 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

https://devhints.io/bash

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

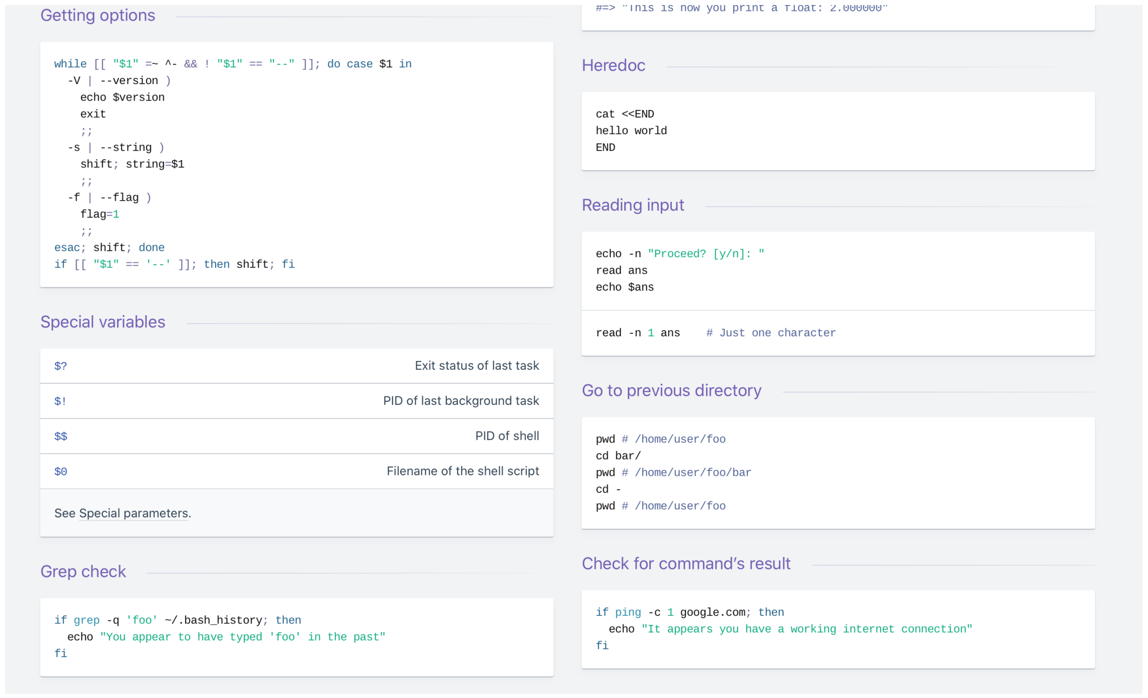
```
case "$1" in
  start | up)
    vagrant up
    ;;

  *)
    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"
```

https://devhints.io/bash

Good Resource: Bash cheat sheet



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

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
```

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
```