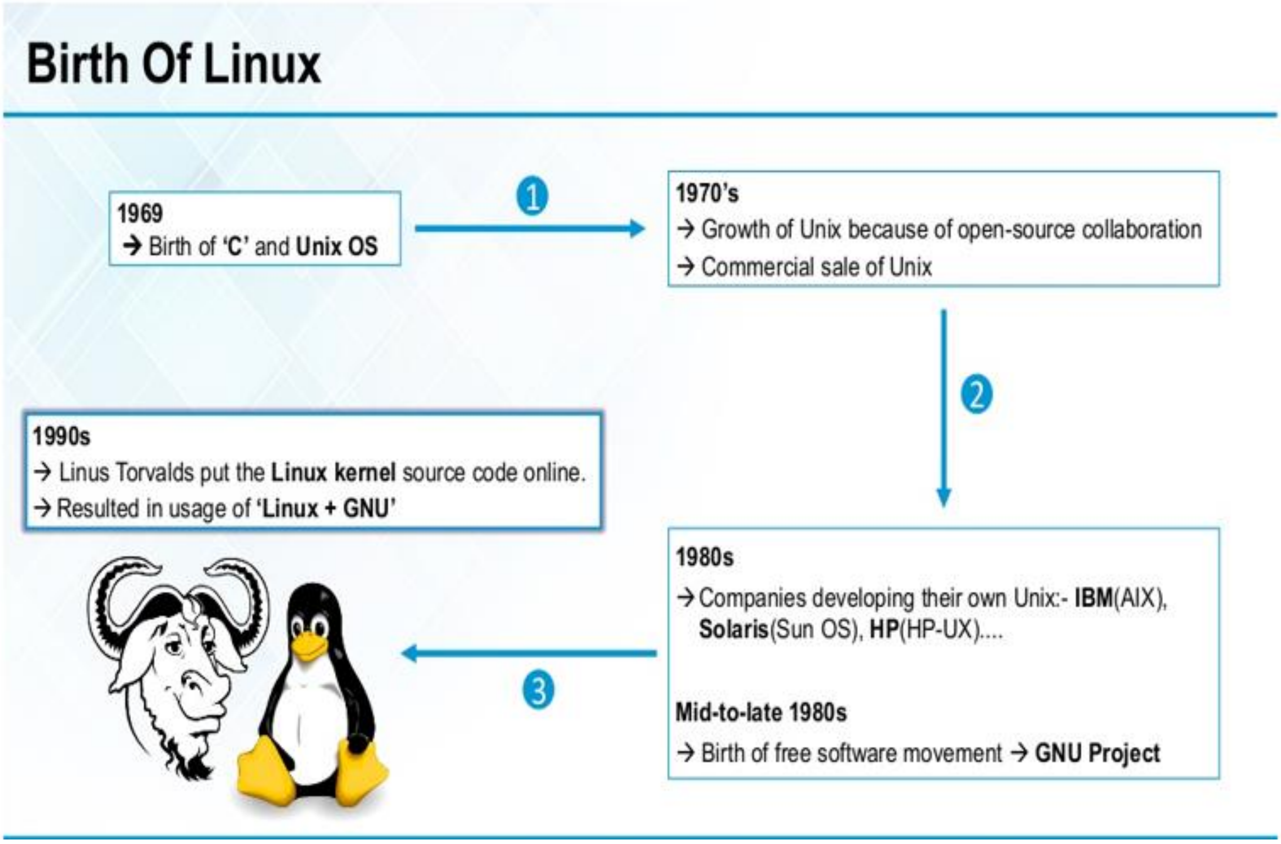
**Shell Scripting for beginners**
https://www.linkedin.com/in/vijaykumar-biradar-29b710161/
**https://github.com/vijaybiradar/shell_scripting**

Check below post for more info
**https://www.linkedin.com/posts/vijaykumar-biradar-29b710161_linux-fundamentals-activity-6834000725916434432-MCV3/**

**https://www.linkedin.com/posts/vijaykumar-biradar-29b710161_linuxsimplenotes-activity-6834383915680182273-ElfY/**



## Birth Of Linux

1969
→ Birth of **'C'** and **Unix OS**

1970's
→ Growth of Unix because of open-source collaboration
→ Commercial sale of Unix

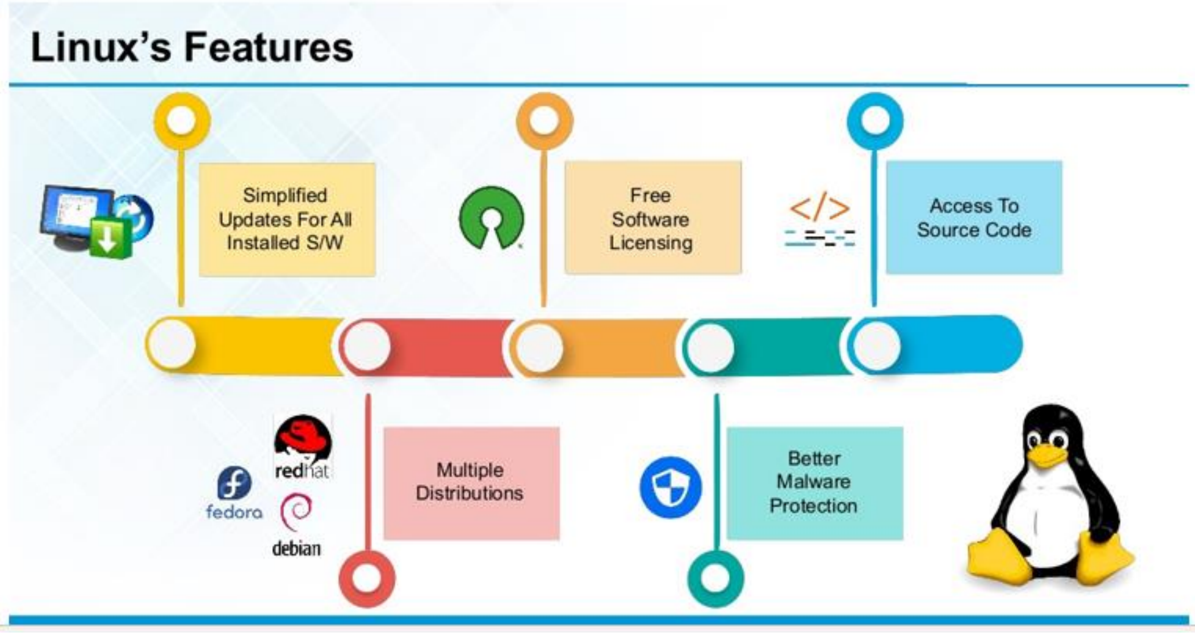
1980s
→ Companies developing their own Unix:- **IBM**(AIX), **Solaris**(Sun OS), **HP**(HP-UX)....

**Mid-to-late 1980s**
→ Birth of free software movement → **GNU Project**

1990s
→ Linus Torvalds put the **Linux kernel** source code online.
→ Resulted in usage of **'Linux + GNU'**

**What is LINUX?**

Linux is a Unix-like, open source and community-developed operating system for which is capable of handling activities from multiple users at the same time.



Linux's Features

Simplified Updates For All Installed S/W

Free Software Licensing

Access To Source Code

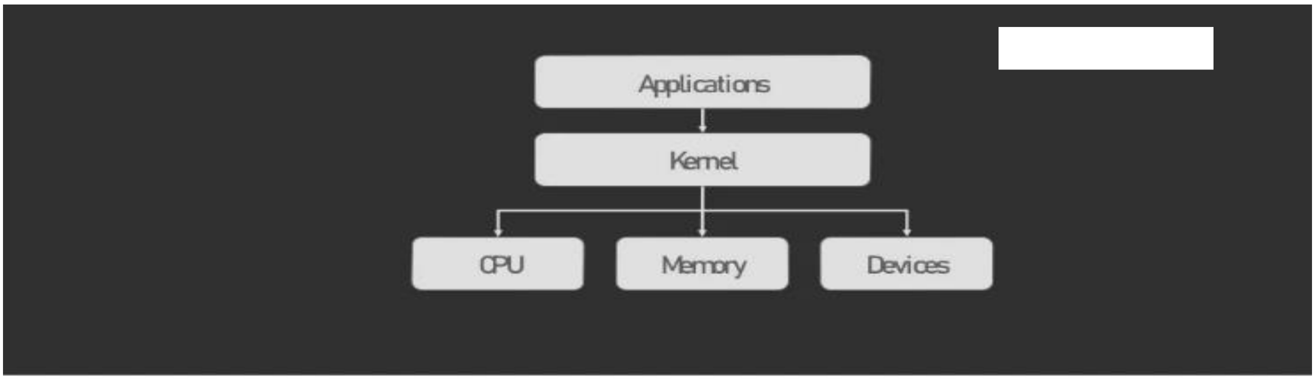
Multiple Distributions

Better Malware Protection

# Why use Linux?

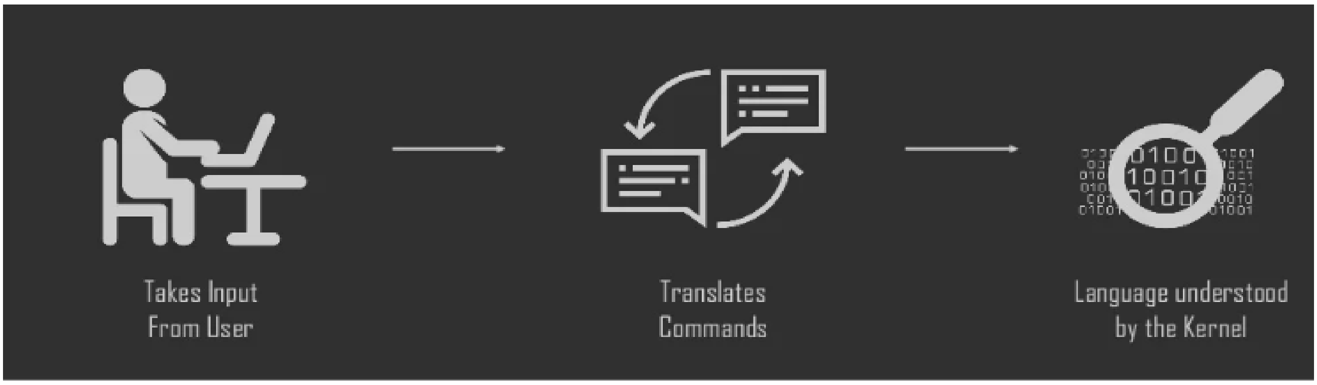
Some of the reasons to use Linux are:

- Low cost and very stable (some Linux servers are not rebooted for over a year, try that with Windows server!)
- Best computing power and inbuilt network support.
- Fastest developing OS, with the most number of developers.
- Most secure OS.
- Configurability
- Convenience
- freedom



**What is a Kernel?**

The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the operating system or the Kernel. Users communicate with the OS through a program called the Shell.

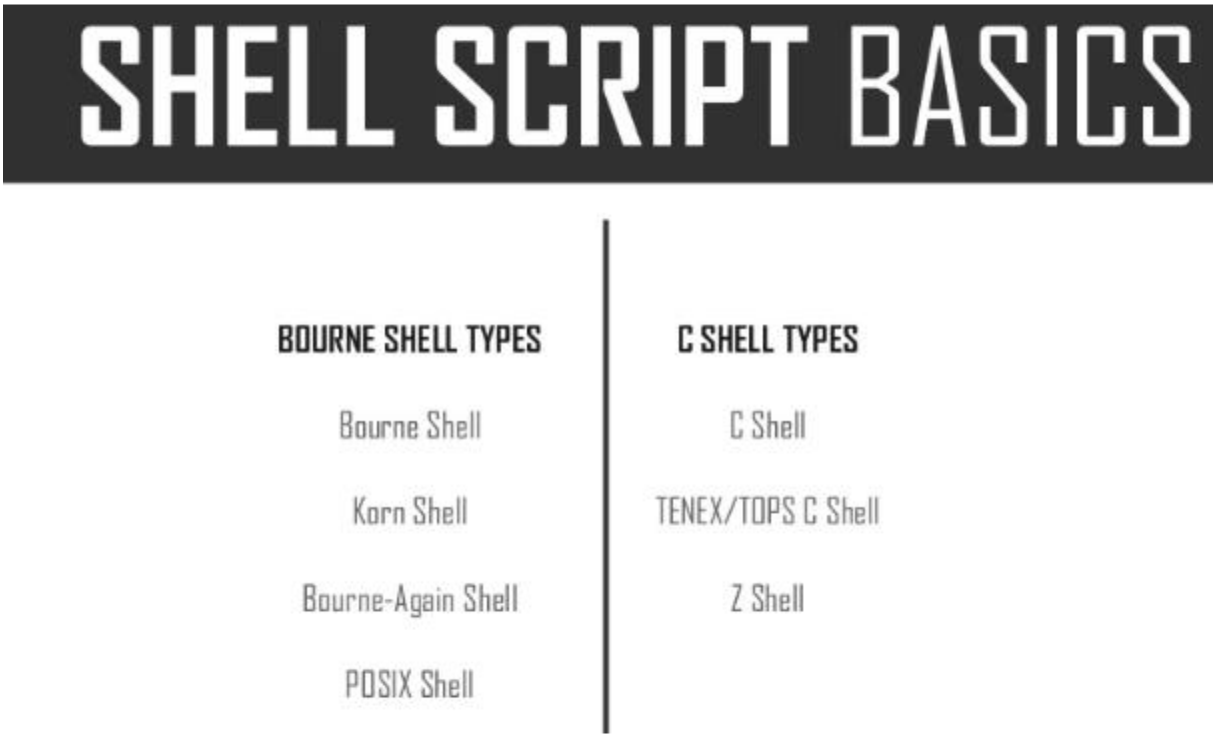


**What is a Shell?**

The Shell is a Command Line Interpreter. It translates commands entered by the user and converts them into a language that is understood by the Kernel.

## What is a Shell Script?

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by #sign, describing the steps



# SHELL SCRIPT BASICS

| BOURNE SHELL TYPES | C SHELL TYPES |
|---|---|
| Bourne Shell | C Shell |
| Korn Shell | TENEX/TOPS C Shell |
| Bourne-Again Shell | Z Shell |
| POSIX Shell | |

First you need to find out where is your Bash interpreter located. Enter the following into your command line:
This command reveals that the Bash shell is stored in **/bin/bash**

```
$ which bash
/bin/bash
```

## simple hello_world script

Just like in every programming course, we start with a simple **hello_world** script. The following script will output **Hello World**.

```
echo Hello World
```

After creating this simple script in **vi** or with **echo**, you'll have to **chmod +x hello_world** to make it executable. And unless you add the scripts directory to your path, you'll have to type the path to the script for the shell to be able to find it.

```
$ echo Hello World > hello_world
$ chmod +x hello_world
$ ./hello_world
Hello World
```

## she-bang

Let's expand our example a little further by putting **#!/bin/bash** on the first line of the script. The **#!** is called a **she-bang** (sometimes called **sha-bang**), where the **she-bang** is the first two characters of the script.

```
#!/bin/bash
echo Hello World
```

You can never be sure which shell a user is running. A script that works flawlessly in **bash** might not work in **ksh**, **csh**, or **dash**. To instruct a shell to run your script in a certain shell, you can start your script with a **she-bang** followed by the shell it is supposed to run in. This script will run in a bash shell.

```
#!/bin/bash
echo -n hello

echo A bash subshell `echo -n hello`
```

```
#!/bin/ksh
echo -n hello

echo a Korn subshell `echo -n hello`
```

This script will run in a Korn shell (unless **/bin/ksh** is a hard link to **/bin/bash**). The **/etc/shells** file contains a list of shells on your system.

# comment

Let's expand our example a little further by adding comment lines.

```
#!/bin/bash
#

# Hello World Script
#

echo Hello World
```

# variables

Here is a simple example of a variable inside a script.

```
#!/bin/bash #

# simple variable in script #

var1=4

echo var1 = $var1
```

Scripts can contain variables, but since scripts are run in their own shell, the variables do not survive the end of the script.

```
$ echo $var1


$ ./vars
var1 = 4

$ echo $var1
```

# sourcing a script

Luckily, you can force a script to run in the same shell; this is called **sourcing** a script.

```
$ source ./vars
var1 = 4

$ echo $var1
 4

$
```

The above is identical to the below

```
$ . ./vars

var1 = 4

$ echo $var1

4
```

# troubleshooting a script

Another way to run a script in a separate shell is by typing **bash** with the name of the script as a parameter.

```
$ cat runme

# the runme script

var4=42
echo $var4


$ bash runme

42
```

Expanding this to **bash -x** allows you to see the commands that the shell is executing (after shell expansion).

```
$ bash -x runme

+ var4=42

+ echo 42

42
```

Notice the absence of the commented (#) line, and the replacement of the variable before execution of **echo**.

# Scripting loops

# test [ ]

The **test** command can test whether something is true or false. Let's start by testing whether 10 is greater than 55.

```
$ test 10 -gt 55 ; echo $?
1
$
```

The test command returns 1 if the test fails. And as you see in the next screenshot, test returns 0 when a test succeeds.

```
$ test 56 -gt 55 ; echo $?
0
$
```

If you prefer true and false, then write the test like this.

```
$test 56 -gt 55 && echo true || echo false
true
$test 6 -gt 55 && echo true || echo false
false
```

The test command can also be written as square brackets, the screenshot below is identical to the one above.

```
$ [ 56 -gt 55 ] && echo true || echo false
true
$ [ 6 -gt 55 ] && echo true || echo false
false
```

# if then else

The **if then else** construction is about choice. If a certain condition is met, then execute something, else execute something else. The example below tests whether a file exists, and if the file exists then a proper message is echoed.

```
#!/bin/bash


if [ -f isit.txt ]

then echo isit.txt exists!
else echo isit.txt not found!
fi
```

If we name the above script 'choice', then it executes like this.

```
[vijay@biradar scripts]$ ./choice
isit.txt not found!

[vijay@biradar scripts]$ touch isit.txt
[vijay@biradar scripts]$ ./choice
isit.txt exists!

[vijay@biradar scripts]$
```

# if then elif

You can nest a new **if** inside an **else** with **elif**. This is a simple example.

```
#!/bin/bash
count=42

if [ $count -eq 42 ]
then

  echo "42 is correct."
elif [ $count -gt 42 ]
then

  echo "Too much."
else

  echo "Not enough."

fi
```

# for loop

The example below shows the syntax of a classical **for loop** in bash.

```
for i in 1 2 4
do

   echo $i
done
```

An example of a **for loop** combined with an embedded shell.

```
#!/bin/ksh

for counter in `seq 1 20`
do

   echo counting from 1 to 20, now at $counter
   sleep 1

done
```

The same example as above can be written without the embedded shell using the bash **{from..to}** shorthand.

```
#!/bin/bash

for counter in {1..20}
do

   echo counting from 1 to 20, now at $counter
   sleep 1

done
```

This **for loop** uses file globbing (from the shell expansion). Putting the instruction on the command line has identical functionality.

```
kahlan@solexp11$ ls
count.ksh  go.ksh

kahlan@solexp11$ for file in *.ksh ; do cp $file $file.backup ; done
kahlan@solexp11$ ls

count.ksh  count.ksh.backup  go.ksh  go.ksh.backup
```

# while loop

Below a simple example of a **while loop**.

```
i=100;

while [ $i -ge 0 ] ;
do

   echo Counting down, from 100 to 0, now at $i;
   let i--;

done
```

Endless loops can be made with **while true** or **while :** , where the **colon** is the equivalent of **no operation** in the **Korn** and **bash** shells.

```
#!/bin/ksh

# endless loop
while :

do

 echo hello
 sleep 1

done
```

# until loop

Below a simple example of an **until loop**.

```
let i=100;

until [ $i -le 0 ] ;
do

   echo Counting down, from 100 to 1, now at $i;
   let i--;

done
```

# Input Output Redirection in Linux

**Types of Redirections**

1. Overwrite
   - “>” standard output
   - “<” standard input
2. Appends
   - “>>” standard output
   - “<<” standard input
3. Merge
   - “p >& q” Merges output from stream p with stream q
   - “p <& q” Merges input from stream p with stream q

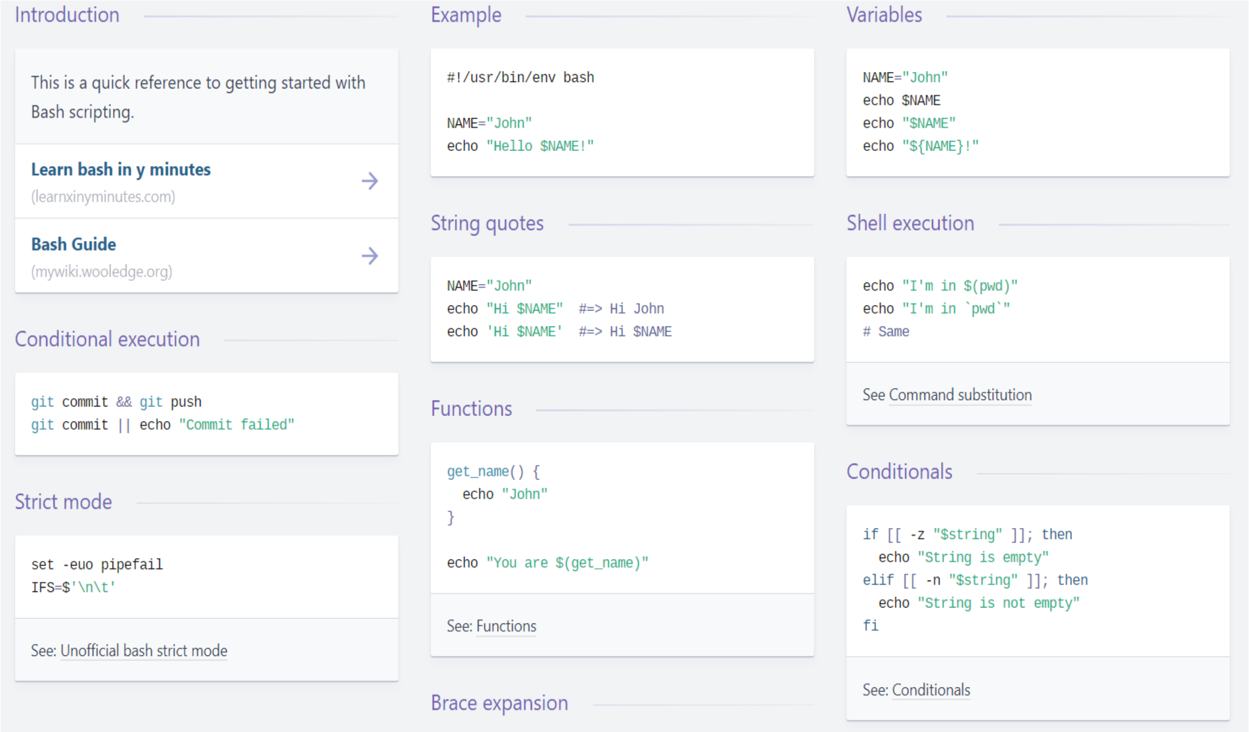
```
#!/bin/bash
# Author : Vijaykumar S Biradar
# Script follows here:
#with and without standard output eg
echo hello everyone > without_standardoutput.txt
cat without_standardoutput.txt
#redirects standard output to with_standardoutput.txt; This is also works same as above
echo hello everyone 1> with_standardoutput.txt
cat with_standardoutput.txt
echo thanks for reading1 >> without_standardoutput.txt
cat without_standardoutput.txt
echo thanks for reading2 >> without_standardoutput.txt
cat without_standardoutput.txt
echo thanks for reading3 > without_standardoutput.txt
cat without_standardoutput.txt
#redirects stdout to with_standardoutput.txt
echo thanks for reading1 1>> with_standardoutput.txt
cat listings.txt
echo thanks for reading2 1>> with_standardoutput.txt
cat listings.txt
echo thanks for reading3 1> with_standardoutput.txt
cat with_standardoutput.txt

#standard error eg
ls without_standardoutput.txt with_standardoutput.txt without_standarderror.txt
cat without_standarderror.txt
ls without_standardoutput.txt with_standardoutput.txt without_standarderror.txt 2>
with_standarderror.txt
cat with_standarderror.txt

#standard input eg
cat > with_standardinput.txt <<EOF
This is Shell Scripting Course. We are covering IO Redirections.
Thanks.
EOF
cat with_standardinput.txt
```

https://devhints.io/bash

## Bash scripting cheat sheet



### 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

### Variables

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

### Shell execution

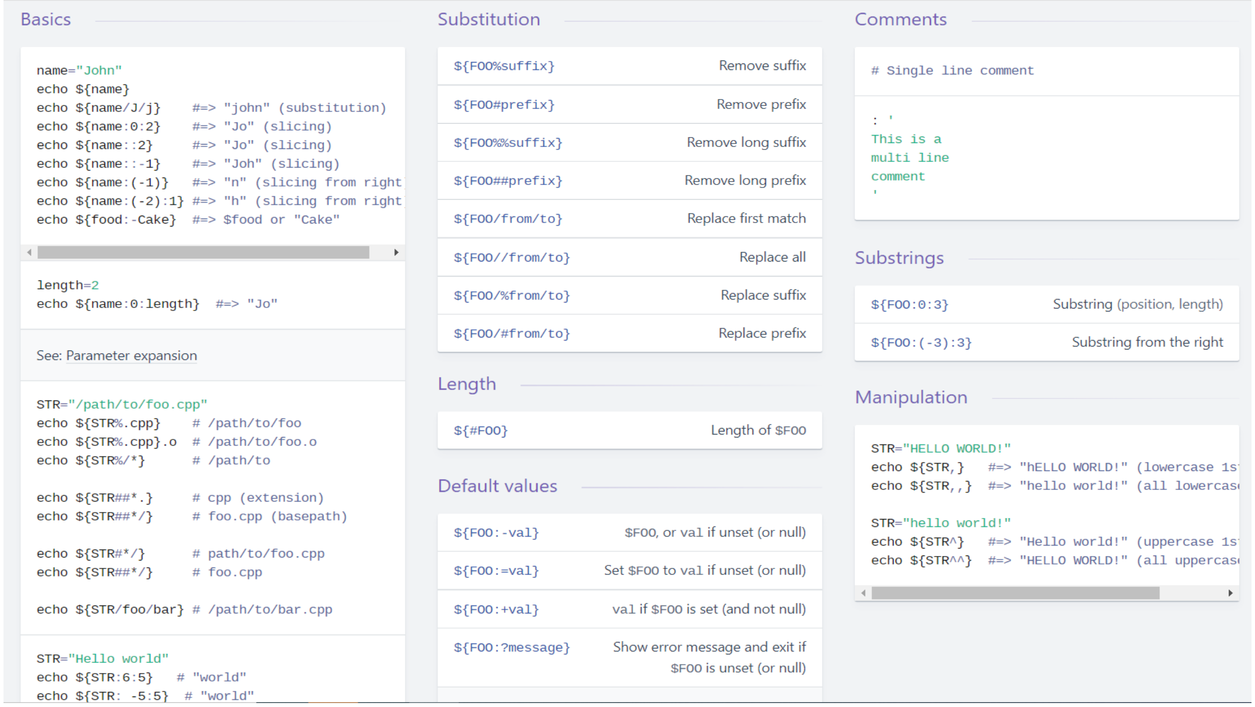
```
echo "I'm in $(pwd)"
echo "I'm in `pwd`"
# Same
```

See Command substitution

### Conditionals

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

See: Conditionals



### 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"
```

### 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) |

### Comments

```
# Single line comment
```

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

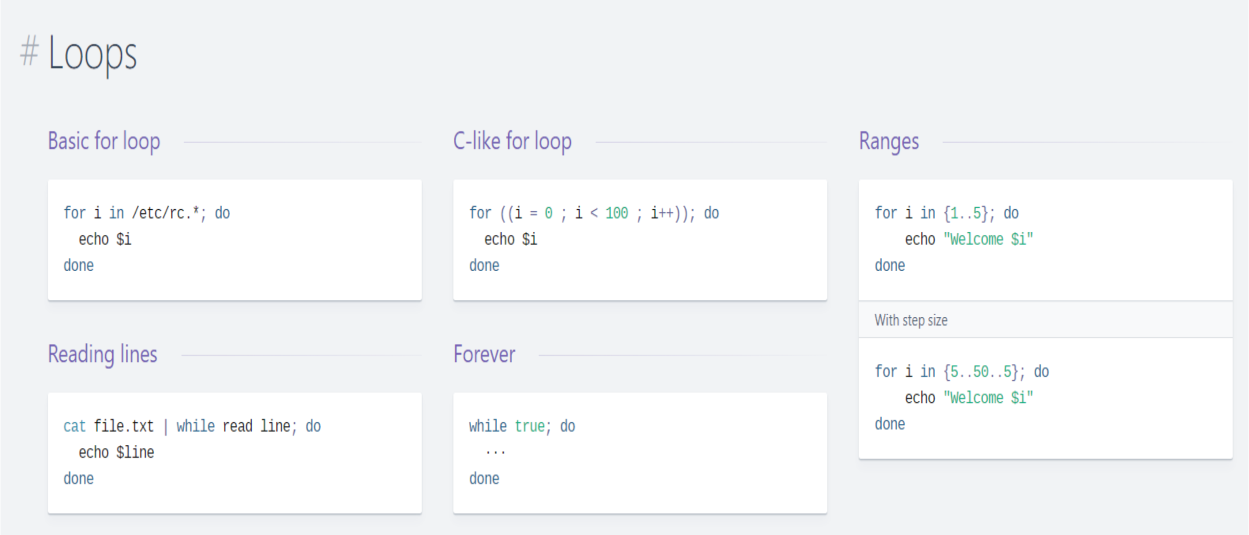
### Substrings

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

### Manipulation

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

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



# 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

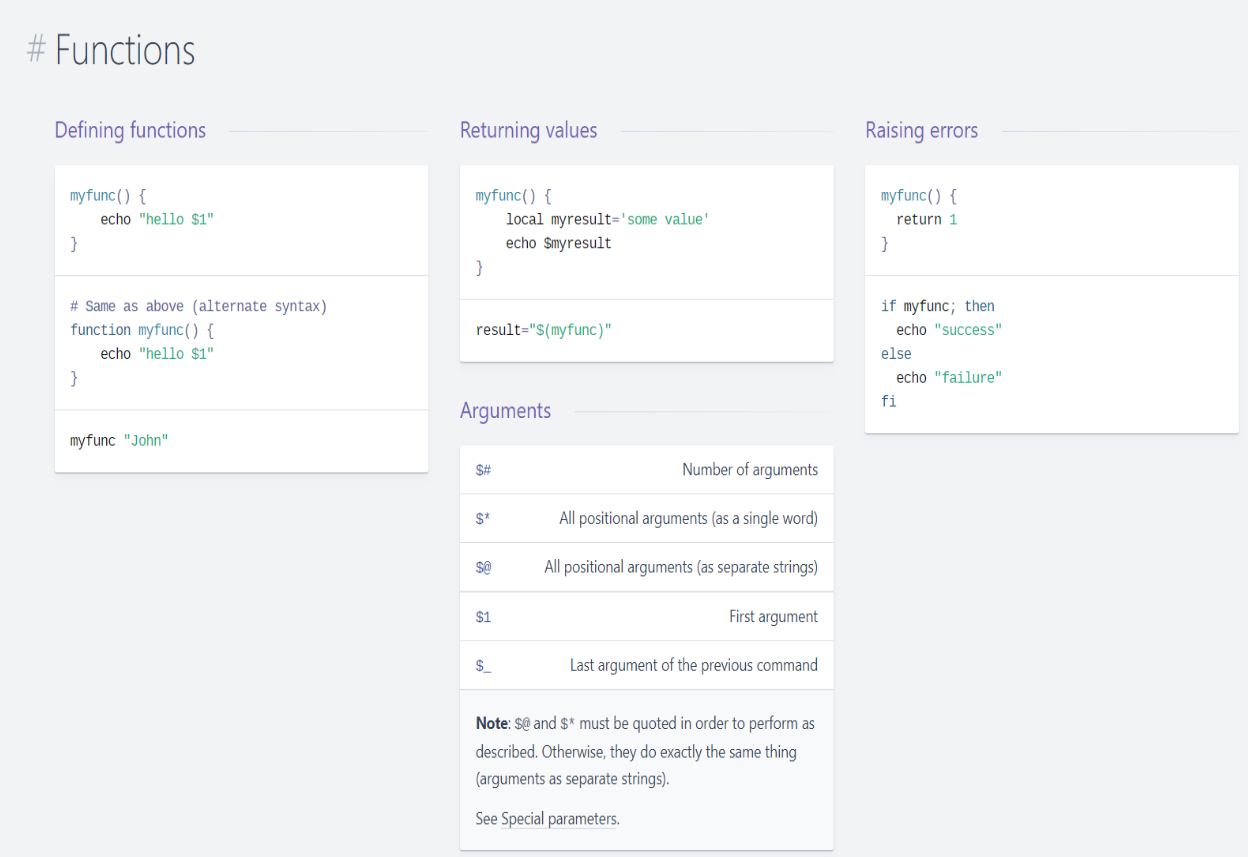
```
for i in {5..50..5}; do
    echo "Welcome $i"
done
```

### 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 positional arguments (as a single word) |
| `$@` | All positional arguments (as separate strings) |
| `$1` | First argument |
| `$_` | Last argument of the previous command |

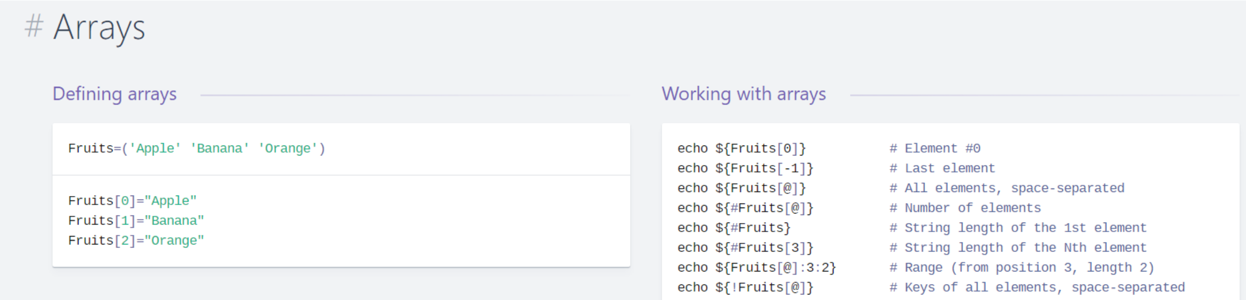
**Note**: `$@` and `$*` must be quoted in order to perform as described. Otherwise, they do exactly the same thing (arguments as separate strings).

See Special parameters.

### Raising errors

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

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



# # Arrays

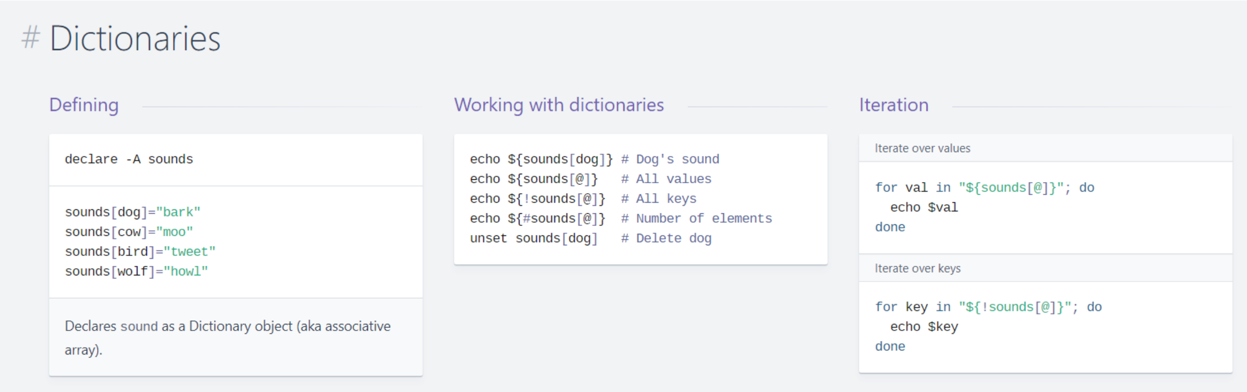
### Defining arrays

```
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
```



# # 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
```

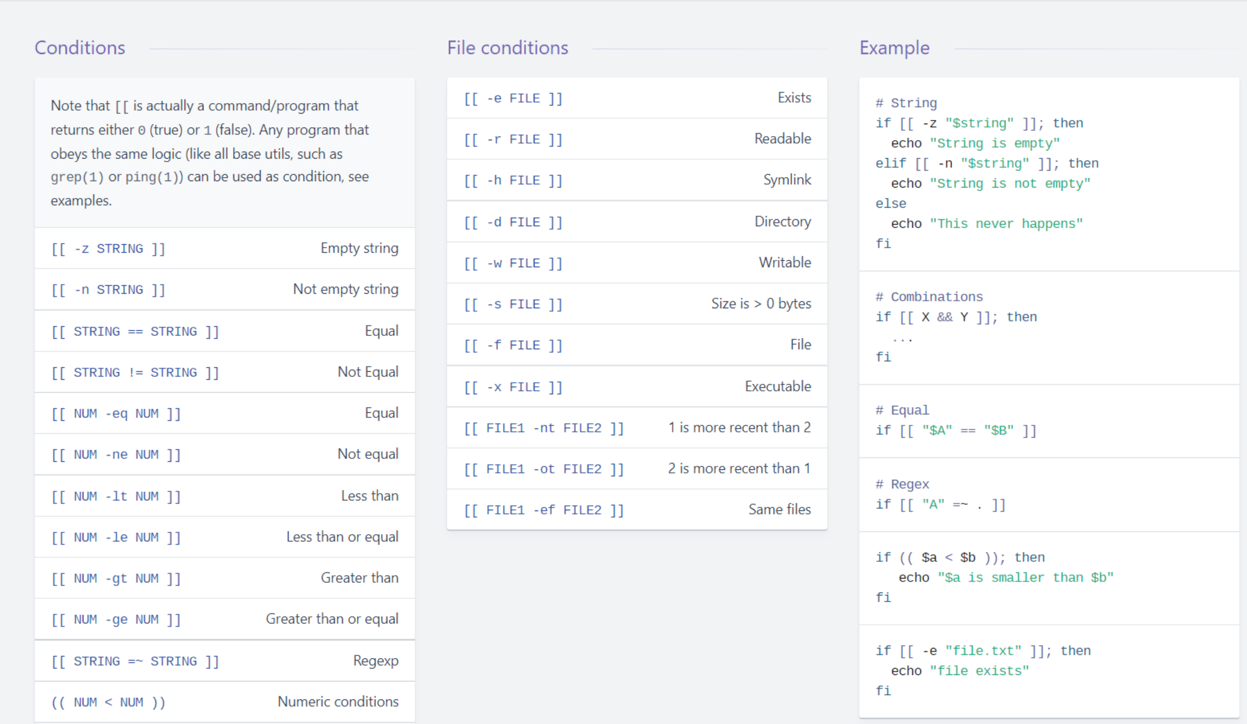
### Iteration

Iterate over values

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

Iterate over keys

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



### 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 |

### 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
```