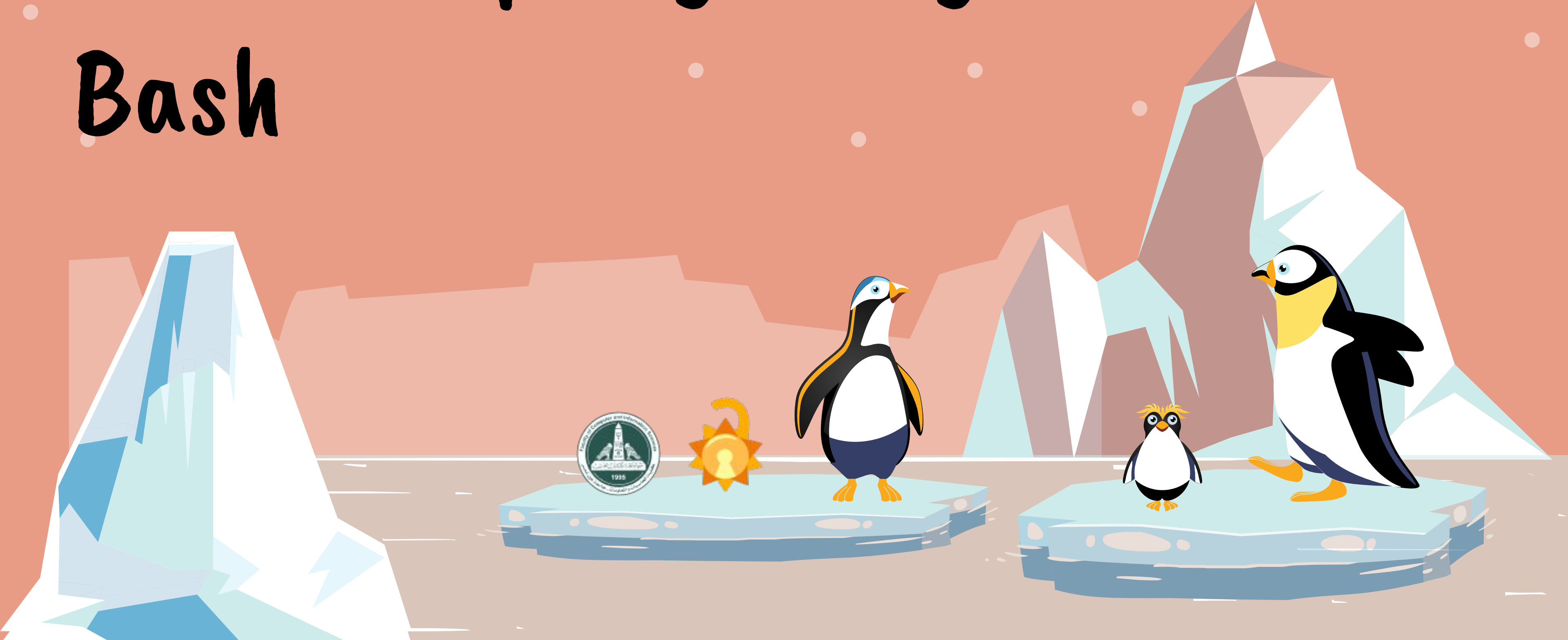
Shell Scripting using Bash



Agenda

01 What is a shell script

02 How to write a script

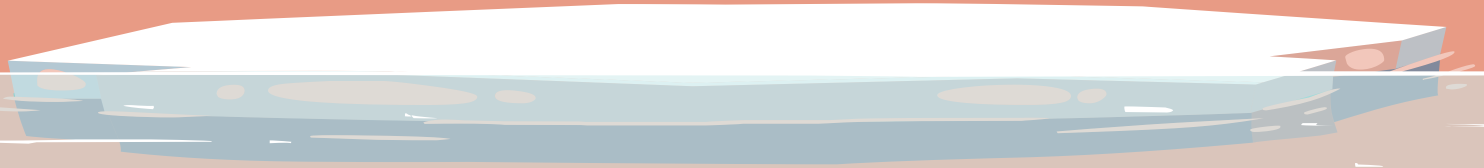
03 How to execute a script

04 Variables, input and math

05 Booleans

06 Flow control and functions

What is a Shell Script

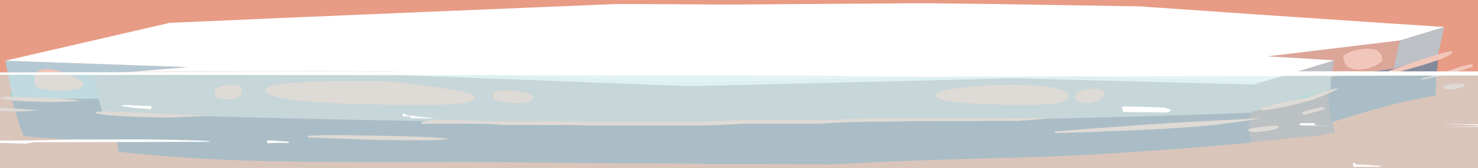


- Shell: a software that takes commands from the user and makes the operating system execute them.

- Script: a file containing multiple commands that get execute sequentially when the script is run.

- Bash: the Bash Shell (Bourne Again Shell) is one of many shells.

Writing a script



The Shebang

- The “#!/path/to/shell” at the first line of a script is called a Shebang and is used to tell the operating system what shell it should use to execute the commands

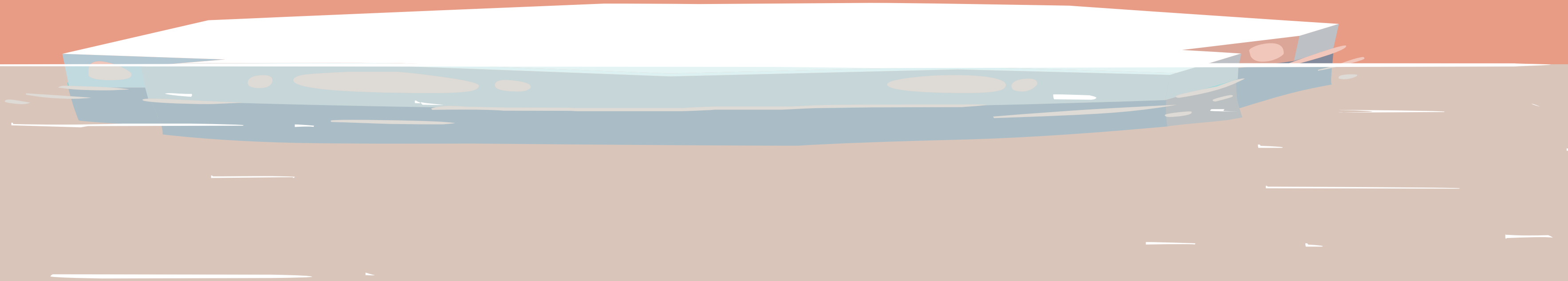
- By typing this line you do not need to specify the shell at the command line before executing the file

- This should be the first line to write in a script

Rest of The script

- The rest of the script is just shell commands that you would normally write in you terminal
- Each command is placed on a separate line

How to run a script



By writing the path to the script

- You could execute a script by writing the relative or absolute path to that script
- For that to work the script must have execute permission
- By default, files in linux do not have execute permission so you will have to give the script execute permission using “chmod u+x script.sh”

Calling bash to execute the script

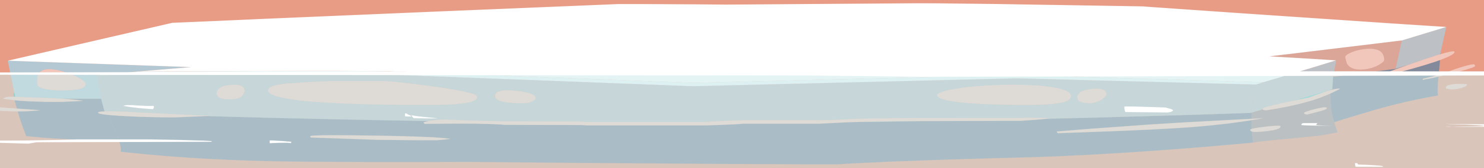
- Another method is to write “bash <nameOfScript>” in your terminal
- This will execute the script even if it does not have execute permission

The PATH variable

- The PATH is a environment variable (we will learn about variables in more depth later on) that contains paths to directories where bash looks for scripts or binaries when commands are called

- By placing the directory of your script in your PATH, you could call the script from anywhere in your system by simply only writing the name of the script in your terminal. (you no longer need to write the path)

Variables, Input and Math



1. Variables

- They are created by the user (think of them like variables in c++)
- Their name are case sensitive.
- They can only contain letters numbers and underscores and they CANNOT start with a number
- When assigning values to new variables their must be NO space around the equality sign
- You should not name variable in all caps as they can overwrite variables that should no be overwritten

1. Variables

- When you want to reference the value of a variable we put the "$" before the variable name (this is called parameter expansion)
- Unlike languages like c++, a variable doesn't need to be declared before being used in bash and so you can reference a variable that doesn't exist and the code will run fine and without error. When expanded the non existent variable will be substituted by an empty string
- Due to the previously discussed property in bash you should be careful to not create sneaky bugs

1.2. " " vs ' '

- Double quotes: expressions, variable etc.. inside double quotes are interpreted by bash

  Ex: echo “$x” ⟶ 2

- Single quote: expressions, variables and other symbols are not interpreted and they are treated as literals

  Ex: echo ‘$x’ ⟶ $x

2. Input

- You could make your script take input from the terminal using two ways:
    1. Positional parameters
    2. The “read” command

2.1. Input (Positional parameters)

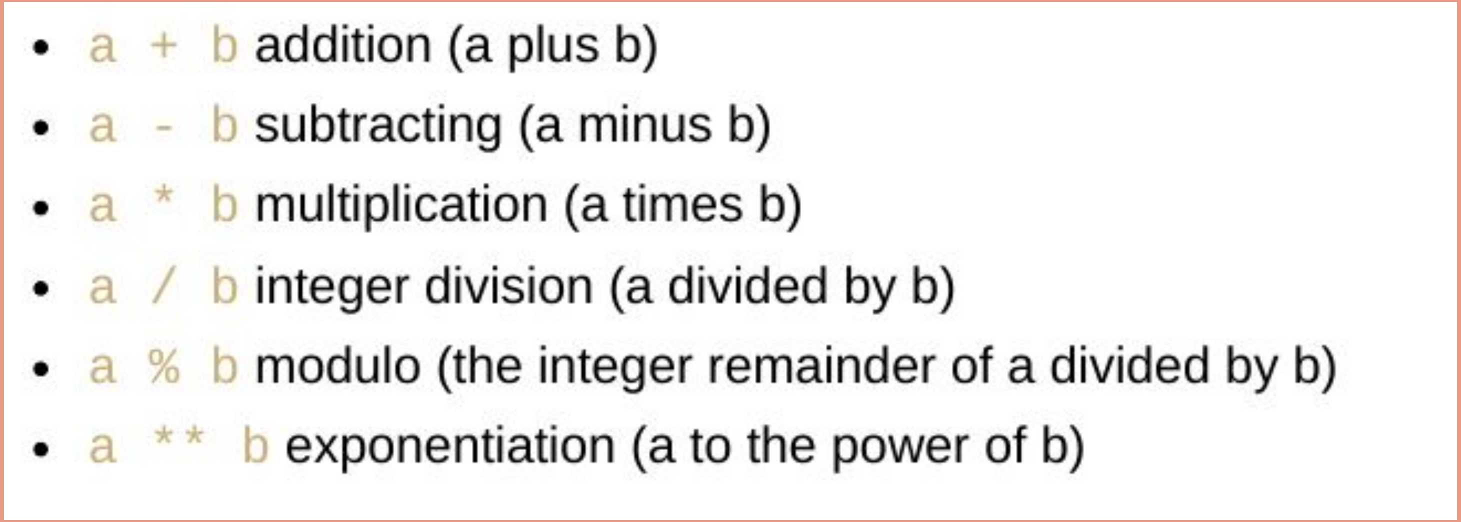
- They are variable used to store and later reference arguments given to a script
- It is denotes using “$n” where n is the position of the argument starting from 1
- If n is more than a single digit number then it should be expanded with curly braces like this: “${11}

2.2. Input (“read” command)

- The “read” command takes the input from the terminal and stores it in the specified variable.
- It automatically places quotes around the input
- Using the “-p” option allows you to prompt the user along with taking input

3. Arithmetic in bash

- The built in arithmetic operations of bash are:



- a + b addition (a plus b)
- a - b subtracting (a minus b)
- a * b multiplication (a times b)
- a / b integer division (a divided by b)
- a % b modulo (the integer remainder of a divided by b)
- a ** b exponentiation (a to the power of b)

3.1. Arithmetic evaluation

- To make bash evaluate an arithmetic expression and give you its value we use the following syntax:

  ```
  $(( $a + $b ))
  ```

- This method does not deal with floating point arithmetic

3.2. The "expr" command

- This command evaluates an expression and returns its value
- Ex: expr 2 + 7 → 9
- This also does not deal with floating point arithmetic
- Note: there must be spaces around the operator and operands when using "epxr"

3.3. The "let" keyword

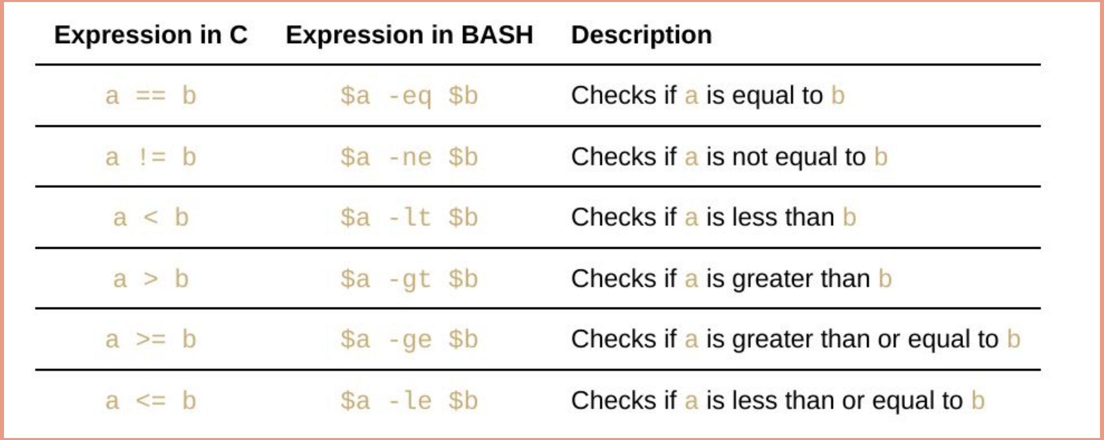
- This makes bash evaluate and expression and set it equal to a varibale
- Ex: let a=2+7 ; echo &a → 9
- This also does not deal with floating point arithmetic

3.4. Dealing with floating point numbers

- To deal with floating point numbers we use a tool called bc but we will not discuss it in this session (google is your friend here)

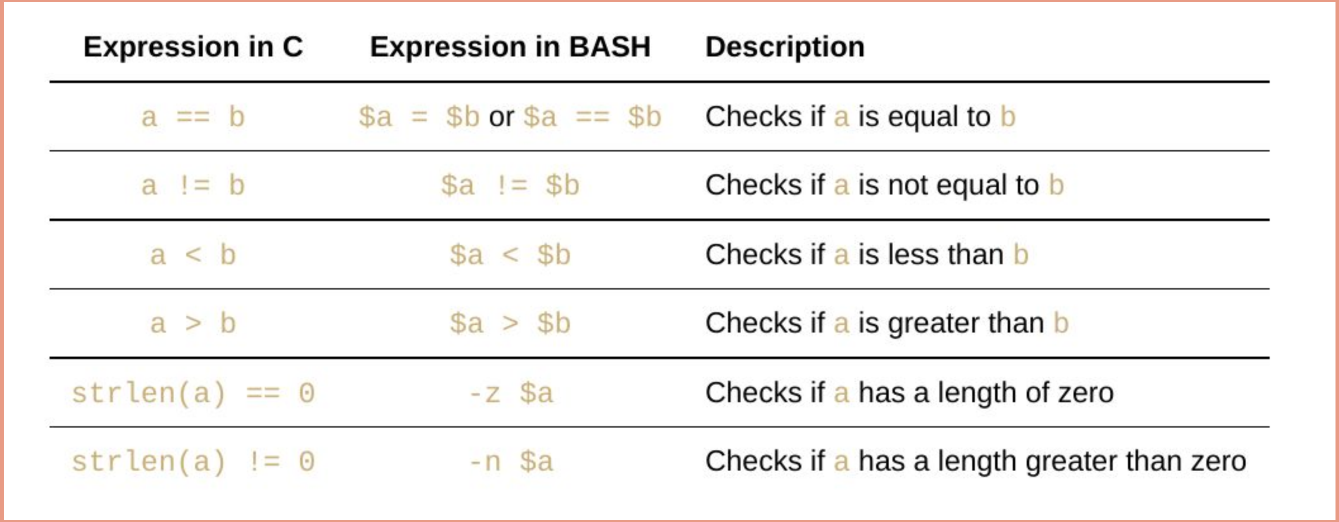
Comparing numericals

- numerical boolean expression in bash should be surrounded by [[ ]] and it MUST HAVE SPACE AROUND IT TO SEPARATE IT FROM THE SQUARE BRACKETS



Expression in C	Expression in BASH	Description
`a == b`	`$a -eq $b`	Checks if a is equal to b
`a != b`	`$a -ne $b`	Checks if a is not equal to b
`a < b`	`$a -lt $b`	Checks if a is less than b
`a > b`	`$a -gt $b`	Checks if a is greater than b
`a >= b`	`$a -ge $b`	Checks if a is greater than or equal to b
`a <= b`	`$a -le $b`	Checks if a is less than or equal to b

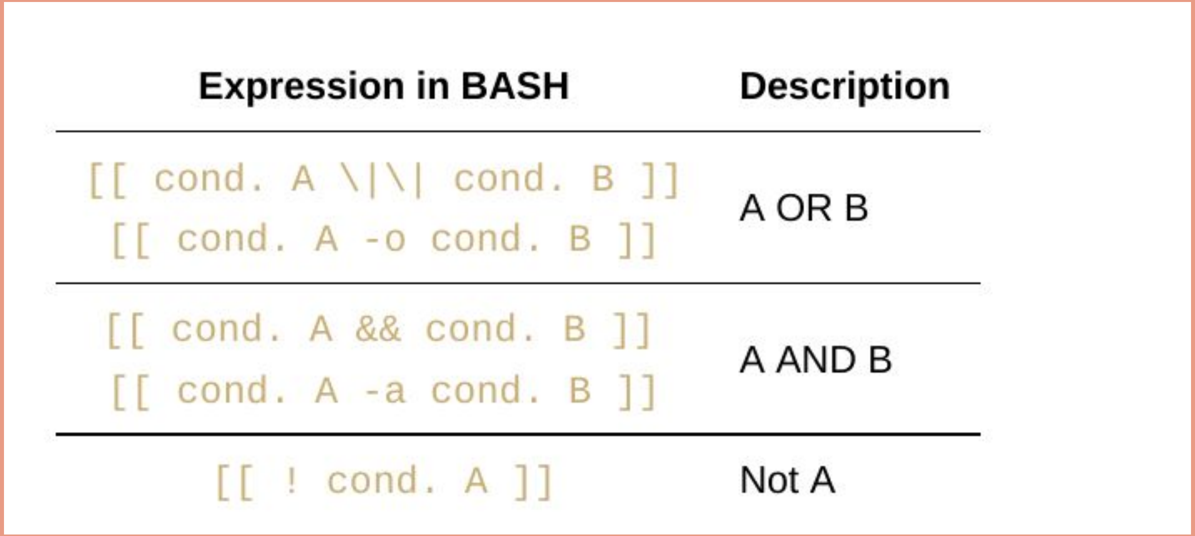
Comparing Strings



Expression in C	Expression in BASH	Description
`a == b`	`$a = $b` or `$a == $b`	Checks if a is equal to b
`a != b`	`$a != $b`	Checks if a is not equal to b
`a < b`	`$a < $b`	Checks if a is less than b
`a > b`	`$a > $b`	Checks if a is greater than b
`strlen(a) == 0`	`-z $a`	Checks if a has a length of zero
`strlen(a) != 0`	`-n $a`	Checks if a has a length greater than zero

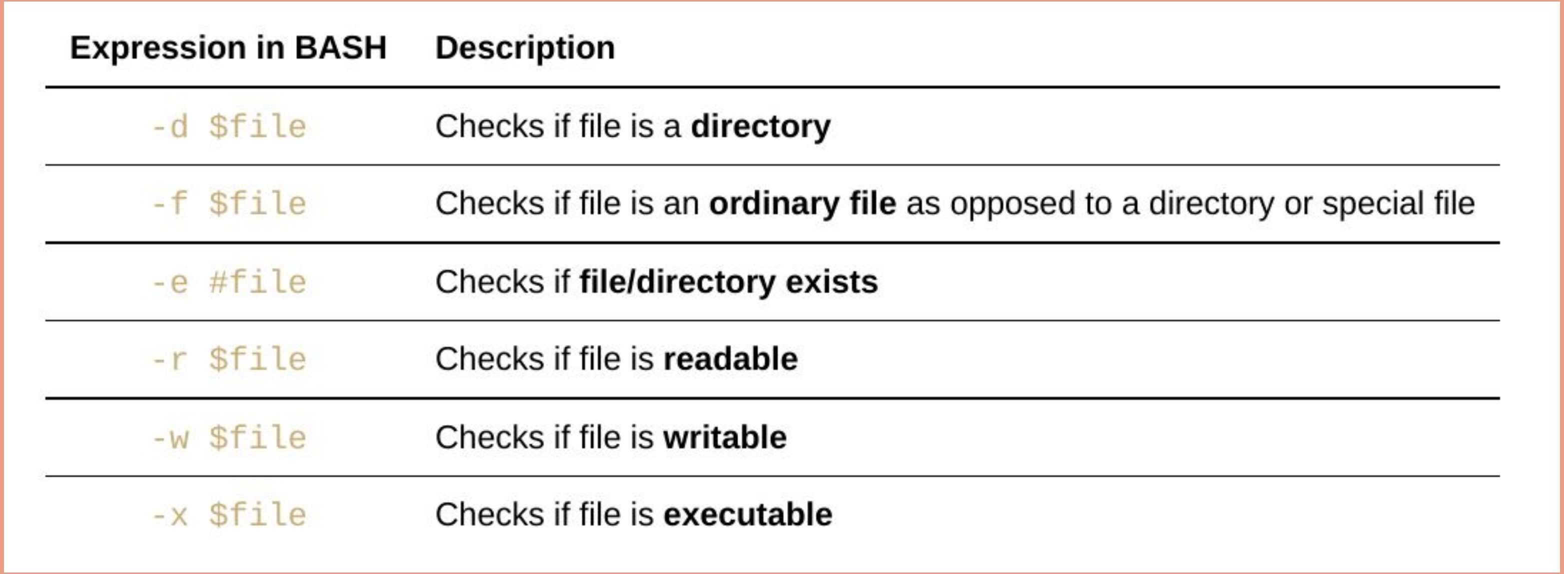
- Comparisons of “>” and “<” are done using lexicographic ordering
- We still place these inside [[ ]] and have space separating brackets from expression

Boolean AND & OR



Expression in BASH	Description
[[cond. A || cond. B]]	A OR B
[[cond. A -o cond. B]]	
[[cond. A && cond. B]]	A AND B
[[cond. A -a cond. B]]	
[[! cond. A]]	Not A

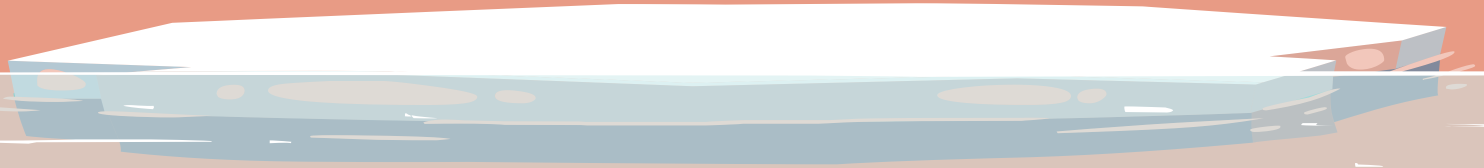
File conditions



Expression in BASH	Description
`-d $file`	Checks if file is a **directory**
`-f $file`	Checks if file is an **ordinary file** as opposed to a directory or special file
`-e #file`	Checks if **file/directory exists**
`-r $file`	Checks if file is **readable**
`-w $file`	Checks if file is **writable**
`-x $file`	Checks if file is **executable**

Note: the [[ ]] is still required.

Flow control and functions



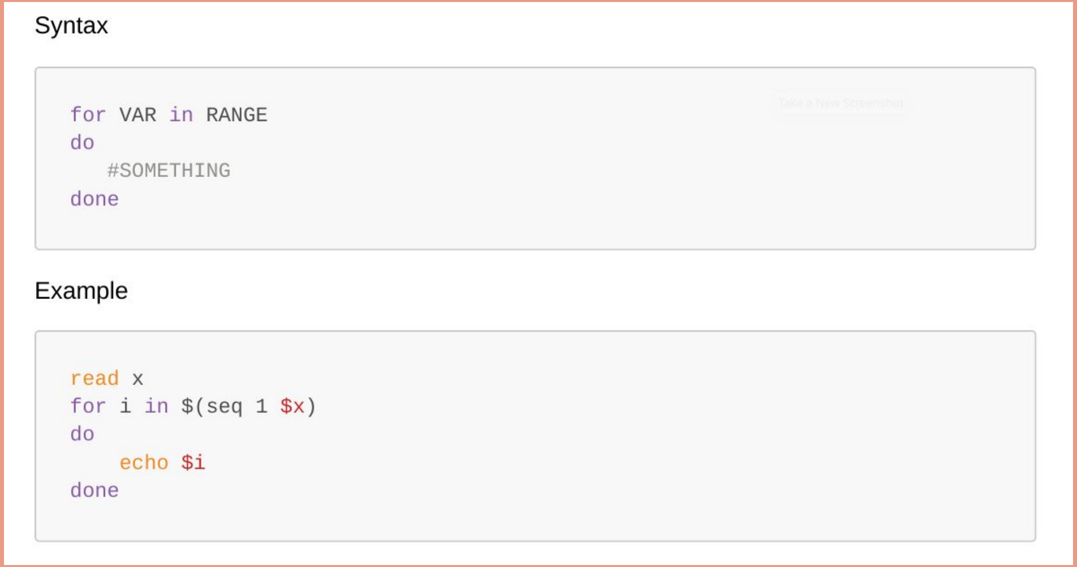
If, else if ladder

```
if [[ $x = "String" ]]
  then
      echo 1
elif [[ $x = "String 2" ]]
  then
      echo 2
else
      echo 3
fi
```

Case statements

```
case $1 in
    start)
        echo starting
        ;;
    stop)
        echo stopping
        ;;
    restart)
        echo restarting
        ;;
    *)
        echo don\'t know
        ;;
esac
```

For loops


Syntax

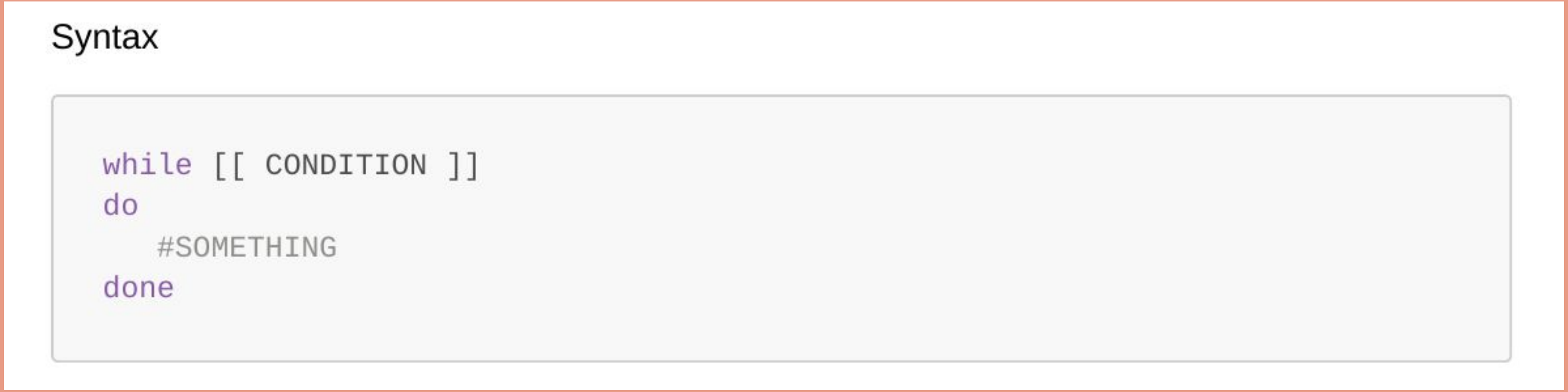
```
for VAR in RANGE
do
   #SOMETHING
done
```

Example

```
read x
for i in $(seq 1 $x)
do
    echo $i
done
```

- For loops could also be written in c style using (( )) or by using bash syntax of {START..END} or {START..END..INCREMENT}

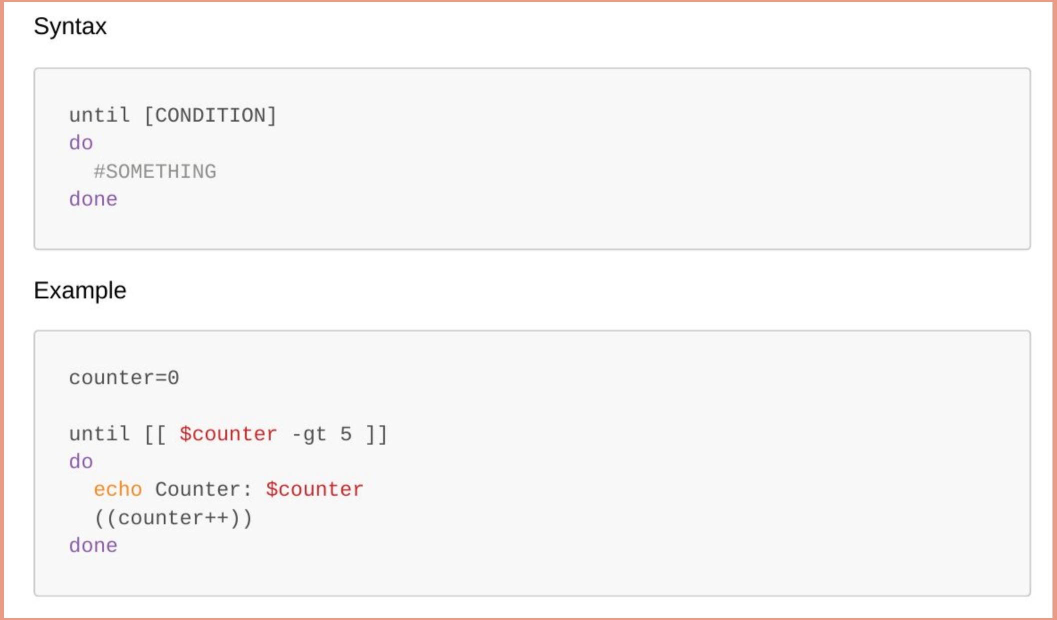
While Loop



Syntax

```
while [[ CONDITION ]]
do
   #SOMETHING
done
```

Until Loops



Syntax

```
until [CONDITION]
do
  #SOMETHING
done
```

Example

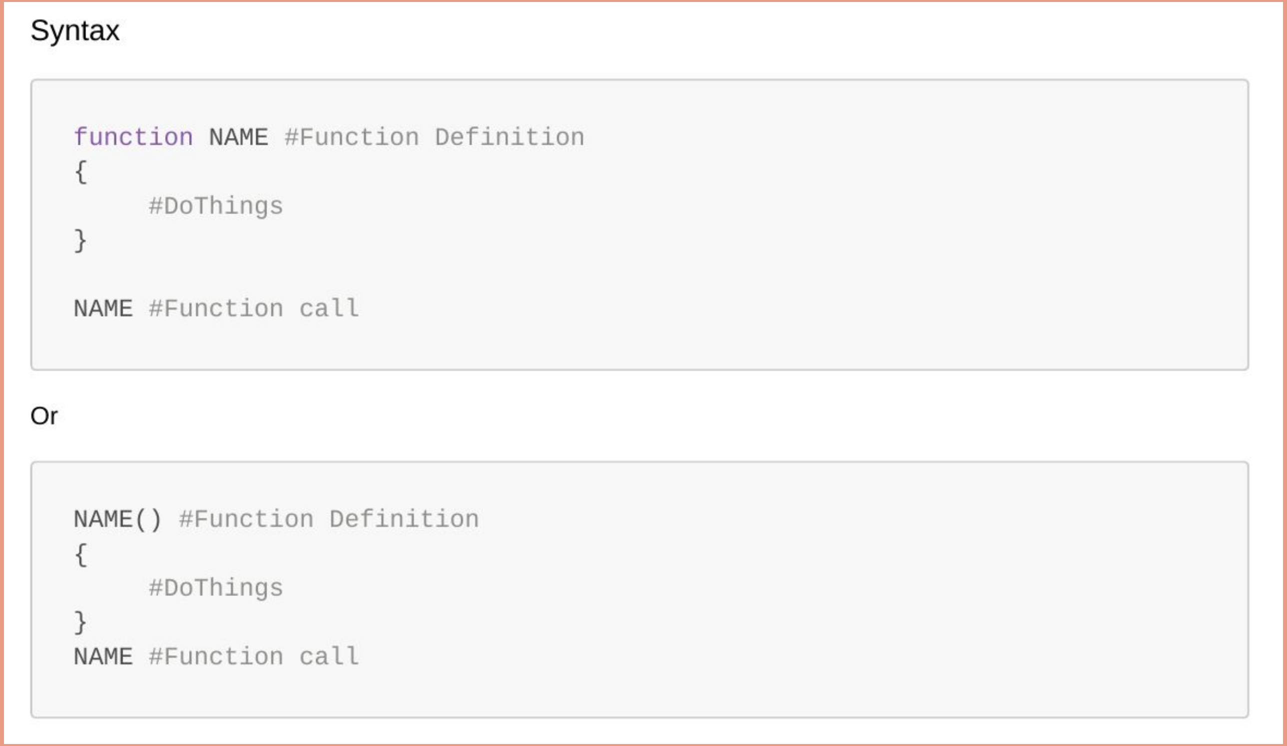
```
counter=0

until [[ $counter -gt 5 ]]
do
  echo Counter: $counter
  ((counter++))
done
```

- Until loops keep going as long as the condition evaluates to true(while loops are the opposite)

Break and continue statements

- Break: breaks out of a loop midway
- Continue: goes back to the beginning of the loop without continuing the rest of the loop's body

Functions



Syntax

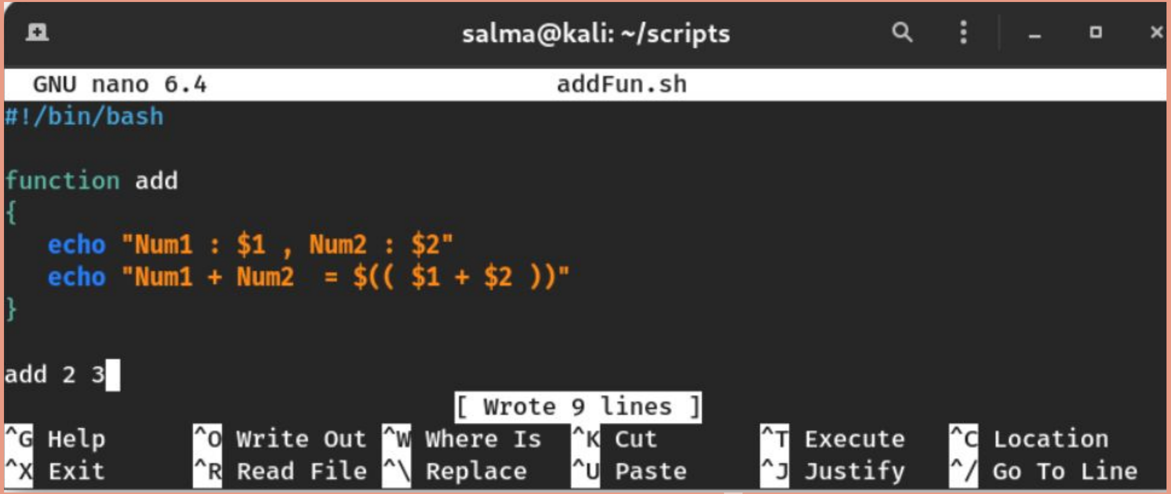
```
function NAME #Function Definition
{
     #DoThings
}

NAME #Function call
```

Or

```
NAME() #Function Definition
{
     #DoThings
}
NAME #Function call
```

Functions (passing arguments)

- To use arguments passed to a function as variables inside the function we refer to the arguments as "$n" where n is the order of the argument in the arguments provided
- ex:



```
 GNU nano 6.4                    addFun.sh
#!/bin/bash

function add
{
   echo "Num1 : $1 , Num2 : $2"
   echo "Num1 + Num2  = $(( $1 + $2 ))"
}

add 2 3
                         [ Wrote 9 lines ]
^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify   ^/ Go To Line
```

The fork bomb function

- It is a function that keeps calling itself infinitely until it eats up all the memory of the machine and causes it to freeze
- The style of attacks are called denial-of-service (DoS) attacks
- After a fork bomb starts execution on a machine the only way to stop it may be to restart the machine
- *NEVER TRY THE FORK BOMB ON YOUR MACHINE*

Thanks!