CS 45, Lecture 4
Shell Scripting

Winter 2023
Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Administrivia

- Assignment 1 is due tonight. Reach out if you don't think you will be able to finish it in time.
- Assignment 2 will go out tonight! It covers shell scripting and text editors.
- Lecture recordings
- Thank you for the feedback! 🙏

What we know

In Lecture 3, we learned how to:

- Use shell commands to manipulate and analyze data
- Write regular expressions
- Run more complex shell commands such as `grep`, `sort`, `uniq`, `xargs`

What we will learn today

In today's lecture, we will learn how to:

- Write shell scripts

What is Shell Scripting?

We've seen how to execute commands in the shell and pipe multiple commands together.

Sometimes, we want to run many, many commands together and/or make use of control flow expressions such as conditionals and loops.

That's where shell scripting comes in.

What is Shell Scripting?

A ***shell script*** is a text file that contains a sequence of commands for a UNIX-based operating system.

It is called a script because it combines a sequence of commands—that would otherwise have to be typed into a keyboard one at a time—into a single script.

```
#!/bin/bash

function gpio()
{
    local verb=$1
    local pin=$2
    local value=$3

    local pins=($GPIO_PINS)
    if [[ "$pin" -lt ${#pins[@]} ]]; then
        local pin=${pins[$pin]}
    fi

    local gpio_path=/sys/class/gpio
    local pin_path=$gpio_path/gpio$pin
```

What is Shell Scripting?

Most shells have their own scripting language, each with its own variables, control flow, and syntax.

What makes shell scripting different from other scripting languages is that it is optimized for performing shell-related tasks.

Creating command pipelines, saving results into files, and reading from standard input are baked into in shell scripting, making it easier to use compared to other scripting languages.

Basics of Bash Scripting

Bash scripting refers to writing a script for a bash shell (Bourne Again SHell).

Basics of Bash Scripting

Bash scripting refers to writing a script for a bash shell (Bourne Again SHell).

You can check what shell you are using by running `ps -p $$`

Basics of Bash Scripting

Bash scripting refers to writing a script for a bash shell (Bourne Again SHell).

You can check what shell you are using by running `ps -p $$`

If you are on Linux, your default shell should be a bash shell. If you are on macOS or Windows, your shell may be different but this shouldn't cause an issue given that your shell will still know how to "speak" bash.

Your Very First Script

Let's write a super simple shell script that says hello!

Your Very First Script

Here is a super simple bash script called `hello.sh`:

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

echo "Hello world!"
```

Your Very First Script

Here is a super simple bash script called `hello.sh`:

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

echo "Hello world!"
```

The **shebang** is the very first line of a script

Shebang

The **shebang**, also called a sharp exclamation, is the very first line of a script.

It is the combination of the pound symbol (`#`) and an exclamation mark (`!`).

The shebang is used to specify the interpreter that the given script will be run with. In our case, we indicate that we want a `bash` interpreter (i.e. a bash shell). If you want to run your script with a `zsh` shell, you simply change the shebang.

Shebang

A note about shebangs:

There are a number of different ways to write your shebang such as `#!/usr/bin/env bash` and `#!/bin/bash`

We recommend that you always use the former as it increases the portability of your script. The `env` command tells the system to resolve the bash command wherever it lives in the system, as opposed to just looking inside of `/bin`

Running (Your Very First Script)

You can always run a shell script by simply prepending it with a shell interpreter program:

```
sh hello.sh
```

```
bash hello.sh
```

```
zsh hello.sh
```

Running (Your Very First Script)

You can always run a shell script by simply prepending it with a shell interpreter program:

```
sh hello.sh
bash hello.sh
zsh hello.sh
```

Interpreter

Running (Your Very First Script)

You can also run a script by turning it into an executable program and then running it.

Running (Your Very First Script)

You can also run a script by turning it into an executable program and then running it.

First, turn the program into an executable using `chmod` (change mode):

```
chmod +x hello.sh
```

Running (Your Very First Script)

You can also run a script by turning it into an executable program and then running it.

First, turn the program into an executable using `chmod` (change mode):

```
chmod +x hello.sh
```

Makes the program e**x**ecutable

Running (Your Very First Script)

You can also run a script by turning it into an executable program and then running it.

First, turn the program into an executable using `chmod` (change mode):

```
chmod +x hello.sh
```

Then run the program:

```
./hello.sh
```

Bash Scripting: Variables

To assign variables, use the following:

```
x=foo
```

Bash Scripting: Variables

To assign variables, use the following:

```
x=foo
```

You can access the value of x using the following:

```
$x
```

Bash Scripting: Variables

To assign variables, use the following:

```
x=foo
```

You can access the value of x using the following:

```
$x
```

Note: you cannot use `x = foo` (with spaces) because it is interpreted as trying to run a program `x` with two arguments: `=` and `foo`.

Bash Scripting: Strings

Next, we can define strings.

If we want to define a string literal, we will use single quotes:

```
'$x'
```

Bash Scripting: Strings

Next, we can define strings.

If we want to define a string literal, we will use single quotes:

```
'$x'
```

If we want to define a string that allows substitution, we will use double quotes:

```
"$x"
```

Bash Scripting: Strings

Here's the difference in behavior:

```
x=foo
echo '$x'
# prints $x
```

```
x=foo
echo "$x"
# prints foo
```

Your Very First Script

Let's use a variable in `hello.sh`:

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

greeting="Hello world!"
echo $greeting
```

Bash Scripting: Control Flow

Like other programming languages, bash scripts also have control flow directives such as `if`, `for`, `while`, and `case`.

Bash Scripting: Control Flow

Like other programming languages, bash scripts also have control flow directives such as `if`, `for`, `while`, and `case`.

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

if [ CONDITION ]
then
    # do something
fi
```

Bash Scripting: Control Flow

Like other programming languages, bash scripts also have control flow directives such as `if`, `for`, `while`, and `case`.

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

num=101
if [ $num -gt 100 ]
then
    echo "That's a big number!"
fi
```

Bash Scripting: Control Flow

Like other programming languages, bash scripts also have control flow directives such as `if`, `for`, `while`, and `case`.

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

num=101
if [ $num -gt 100 ] && [ $num -lt 1000 ]
then
    echo "That's a big (but not a too big) number!"
fi
```

Bash Scripting: Control Flow

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

if [ CONDITION ]
then
    # do something
elif [ CONDITION ]
then
    # do something else
else
    # do something totally different
fi
```

Bash Scripting: Control Flow

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

num=101
if [ $num -gt 100 ]
then
    echo "That's a big number!"
elif [ $num -gt 1000 ]
then
    echo "That's a huge number!"
else
    echo "That's a small number."
fi
```

Bash Scripting: Control Flow

Like other programming languages, bash scripts also have control flow directives such as `if`, `for`, `while`, and `case`.

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

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

Bash Scripting: Control Flow

Like other programming languages, bash scripts also have control flow directives such as `if`, `for`, `while`, and `case`.

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

num=0
while [ $num -lt 100 ]
do
    echo $num
    num=$((num+1))
done
```

Bash Scripting: Control Flow

Like other programming languages, bash scripts also have control flow directives such as `if`, `for`, `while`, and `case`.

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

for VARIABLE in {1..N}
do
    # do something
done
```

Bash Scripting: Control Flow

Like other programming languages, bash scripts also have control flow directives such as `if`, `for`, `while`, and `case`.

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

num=0
for i in {1..100}
do
    echo $num
    num=$((num+1))
done
```

Bash Scripting: Exercise

Exercise 1: Write a shell script called `num_loop.sh` that loops through every number `1` through `20` and prints each number to standard output. The script should also conditionally print `I'm big!` for every number larger than `10`.

Bash Scripting: Exercise

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

for i in {1..20}
do
    echo $i
    if [ $i -gt 20 ]
    then
        echo "I'm big!"
    fi
done
```

Bash Scripting: Arguments

Let's take a look at how we might use command line arguments to make our `big_num.sh` script a little more interesting.

Bash Scripting: Arguments

Let's take a look at how we might use command line arguments to make our `big_num.sh` script a little more interesting.

In bash, the variables `$1` - `$9` refers to the arguments to a script.

Bash Scripting: Arguments

Let's take a look at how we might use command line arguments to make our `big_num.sh` script a little more interesting.

In bash, the variables `$1` - `$9` refers to the arguments to a script.

```
adrazen@ayelet-computer ~ % sh my_script.sh ayelet
```

Bash Scripting: Arguments

Let's take a look at how we might use command line arguments to make our `big_num.sh` script a little more interesting.

In bash, the variables `$1` - `$9` refers to the arguments to a script.

```
adrazen@ayelet-computer ~ % sh my_script.sh ayelet
```

This is `$1`

Bash Scripting: Arguments

Let's take a look at how we might use command line arguments to make our `big_num.sh` script a little more interesting.

In bash, the variables `$1` - `$9` refers to the arguments to a script.

The variable `$0` refers to the name of the script.

```
adrazen@ayelet-computer ~ % sh my_script.sh ayelet
```

This is `$1`

Bash Scripting: Arguments

Let's take a look at how we might use command line arguments to make our `big_num.sh` script a little more interesting.

In bash, the variables `$1` - `$9` refers to the arguments to a script.

The variable `$0` refers to the name of the script.

```
adrazen@ayelet-computer ~ % sh my_script.sh ayelet
```

This is `$0` (my_script.sh)

This is `$1` (ayelet)

Bash Scripting: Arguments

Let's assign `num` to be the first argument when calling the script.

```
adrazen@ayelet-computer ~ % sh big_num.sh 102
```

Bash Scripting: Arguments

Let's assign `num` to be the first argument when calling the script.

```
adrazen@ayelet-computer ~ % sh big_num.sh 102
```

This is `$1`

Bash Scripting: Arguments

Let's assign `num` to be the first argument when calling the script.

```
adrazen@ayelet-computer ~ % sh big_num.sh 102
```

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

num=101
if [ $num -gt 100 ]
then
    echo "That's a big number!"
fi
```

Bash Scripting: Arguments

Let's assign `num` to be the first argument when calling the script.

```
adrazen@ayelet-computer ~ % sh big_num.sh 102
```

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

num=$1
if [ $num -gt 100 ]
then
    echo "That's a big number!"
fi
```

Bash Scripting: Functions

We can also define functions!

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh
```

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

make_and_enter() {
    # calls mkdir (including parent directories)
    # calls cd
}
```

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh
```

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

make_and_enter(directory_name) {
    mkdir -p directory_name
    cd directory_name
}
```

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh
```

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

make_and_enter(directory_name) {
    mkdir -p directory_name
    cd directory_name
}
```

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh
```

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

make_and_enter() {
    mkdir -p "$1"
    cd "$1"
}
```

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh
```

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

make_and_enter() {
    mkdir -p "$1"
    cd "$1"
}

make_and_enter new_folder
```

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh
```

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

make_and_enter() {
    mkdir -p "$1"
    cd "$1"
}

make_and_enter new_folder
```

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh
```

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

make_and_enter() {
    mkdir -p "$1"
    cd "$1"
}

make_and_enter new_folder
```

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh
```

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

make_and_enter() {
    mkdir -p "$1"
    cd "$1"
}

make_and_enter new_folder
```

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh my_folder
```

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

make_and_enter() {
    mkdir -p "$1"
    cd "$1"
}

make_and_enter new_folder
```

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh my_folder
```

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

make_and_enter() {
    mkdir -p "$1"
    cd "$1"
}

make_and_enter new_folder
```

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh my_folder
```

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

make_and_enter() {
    mkdir -p "$1"
    cd "$1"
}

make_and_enter $1
```

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh my_folder
```

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

make_and_enter() {
    mkdir -p "$1"
    cd "$1"
}

make_and_enter $1
```

Bash Scripting: Functions

We can also define functions!

```
adrazen@ayelet-computer ~ % sh mcd.sh my_folder
```

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

make_and_enter() {
    mkdir -p "$1"
    cd "$1"
}

make_and_enter $1
```

Bash Scripting: Exercise

Exercise 2: Write a shell script called `my_folder.sh` that takes in two arguments: your name (e.g. `ayelet`) and your name with the `.txt` ending (e.g. `ayelet.txt`). The script should call a function that creates a folder by the name of the first argument (e.g. `ayelet`) and then create a file inside by the name of the second argument (e.g. `ayelet.txt`).

For my name, my function would create a folder named `ayelet` and a file named `ayelet.txt` inside of `ayelet`.

Bash Scripting: Exercise

```
adrazen@ayelet-computer ~ % my_folder.sh ayelet ayelet.txt
```

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

make_my_folder() {
    mkdir "$1"
    cd "$1"
    touch "$2"
}

make_my_folder $1 $2
```

Bash Scripting: Return Values

The notion of **exit codes** allows for verifying the success or failure of a previous command.

Bash Scripting: Return Values

The notion of **exit codes** allows for verifying the success or failure of a previous command.

An exit code or return value is the way scripts or commands can communicate with each other about how execution went.

Bash Scripting: Return Values

The notion of **exit codes** allows for verifying the success or failure of a previous command.

An exit code or return value is the way scripts or commands can communicate with each other about how execution went.

A return value of 0 means that everything went OK. A return value other than 0 means that an error occurred.

Bash Scripting: Return Values

The notion of **exit codes** allows for verifying the success or failure of a previous command.

An exit code or return value is the way scripts or commands can communicate with each other about how execution went.

A return value of 0 means that everything went OK. A return value other than 0 means that an error occurred.

`$?` provides the return value from the most recently executed command

Bash Scripting: Return Values

If you ever need a placeholder for a command that succeeds or fails, you can use the true and false commands.

Bash Scripting: Return Values

If you ever need a placeholder for a command that succeeds or fails, you can use the true and false commands.

`true` is a command that does nothing except return an exit status of 0.

`false` is a command that does nothing except return an exit status of 1.

Bash Scripting: Return Values

```
adrazen@ayelet-computer ~ % sh success_or_failure.sh
```

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

result=$(($RANDOM % 2))
if [ $result -eq 0 ]
then
    true
    echo "$?"
else
    false
    echo "$?"
fi
```

Bash Scripting: Return Values

Return values are useful if you want to conditionally execute commands based on the execution of the previous command.

Bash Scripting: Return Values

Return values are useful if you want to conditionally execute commands based on the execution of the previous command.

In addition to using if-statements, we can also conditionally execute commands using `&&` and `||` .

Bash Scripting: Return Values

Return values are useful if you want to conditionally execute commands based on the execution of the previous command.

In addition to using if-statements, we can also conditionally execute commands using `&&` and `||`.

```
true && echo "Print if things went well!"
# prints "Print if things went well!"
```

Bash Scripting: Return Values

Return values are useful if you want to conditionally execute commands based on the execution of the previous command.

In addition to using if-statements, we can also conditionally execute commands using `&&` and `||`.

```
true && echo "Print if things went well!"
# prints "Print if things went well!"
```

```
false && echo "Print if things went well!"
# no output
```

Bash Scripting: Exercise

Exercise 3: Write a shell script called `file_checker.sh` that checks if a file exists or not. The script take in a file name as an argument and try to run `cat` on that file. The script should then check the exit code of the `cat` command to determine if the file exists or not. If the file exists, the script should print `File exists!`. If the file does not exist, the script should print `File does not exist!`.

Bonus: change the script to suppress the actual output of `cat` and only include your script's output (e.g. `File exists!` or `File does not exist!`).

Bash Scripting: Exercise

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

cat $1
if [ $? -eq 0 ]
then
    echo "File exists!"
else
    echo "File does not exist!"
fi
```

Bash Scripting: Exercise

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

cat $1 &> /dev/null
if [ $? -eq 0 ]
then
    echo "File exists!"
else
    echo "File does not exist!"
fi
```

Bash Scripting: Exercise

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

cat $1 && echo "File exists!"
cat $1 || echo "File does not exist!"
```

Bash Scripting: Exercise

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

cat $1 &> /dev/null && echo "File exists!"
cat $1 &> /dev/null || echo "File does not exist!"
```

Bash Scripting: Command Substitution

Command substitution is another useful feature of bash scripting.

You might want to run a command and then use its output as a variable to some other piece of code.

Bash Scripting: Command Substitution

Command substitution is another useful feature of bash scripting.

You might want to run a command and then use its output as a variable to some other piece of code.

Example:

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

for element in $(ls ~/Desktop)
    do
        echo "Desktop contains file named $element"
    done
```

Bash Scripting: Extra Syntax

Bash scripting has some specific syntax that is worth calling out.

If you're ever stuck, look something up 😁

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ condition ]
then
    # do something
fi
```

```
if [[ condition ]]
then
    # do something
fi
```

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ condition ]
then
    # do something
fi
```

```
if [[ condition ]]
then
    # do something
fi
```

What's the difference?

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ condition ]
```

```
if [[ condition ]]
```

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ condition ]
```

```
if [[ condition ]]
```

Single brackets are a reference to the the `test` command

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ condition ]
```

Single brackets are a reference to the the `test` command

```
if [[ condition ]]
```

Double brackets are bash specific. (Also works for zsh)

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ 1 < 2 ]
then
    echo "Correct!"
fi
```

```
if [[ 1 < 2 ]]
then
    echo "Correct!"
fi
```

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ 1 < 2 ]
then
    echo "Correct!"
fi
```

```
if [[ 1 < 2 ]]
then
    echo "Correct!"
fi
```

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ 1 < 2 ]
then
    echo "Correct!"
fi
```

```
if [[ 1 < 2 ]]
then
    echo "Correct!"
fi
```

```
2: No such file or directory
```

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ 1 < 2 ]
then
    echo "Correct!"
fi
```

```
if [[ 1 < 2 ]]
then
    echo "Correct!"
fi
```

```
2: No such file or
directory
```

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ 1 < 2 ]
then
    echo "Correct!"
fi
```

```
2: No such file or
directory
```

```
if [[ 1 < 2 ]]
then
    echo "Correct!"
fi
```

```
Correct!
```

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ condition ]
```

```
if [[ condition ]]
```

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ condition ]
```

```
if [[ condition ]]
```

In general, single brackets are recognized by more scripting languages and are POSIX compliant. (Won't work with `sh` interpreter unless linked to bash.)

Bash Scripting: [vs [[

When you have an if-statement, you need to encapsulate the condition. You can do this in two ways:

```
if [ condition ]
```

```
if [[ condition ]]
```

In general, single brackets are recognized by more scripting languages and are POSIX compliant. (Won't work with `sh` interpreter unless linked to bash.)

Double brackets are less portable, but they align with what you would expect from high level coding languages. You can use comparison operators such as `<` or `>` and logical operators such as `&&` or `||`.

Bash Scripting: Comparison

In order to compare **numbers** in a bash script, use the following:

`a -eq b` for checking if `a` is equal to `b`

Bash Scripting: Comparison

In order to compare **numbers** in a bash script, use the following:

`a -eq b` for checking if `a` is equal to `b`

`a -ne b` for checking if `a` is not equal to `b`

Bash Scripting: Comparison

In order to compare **numbers** in a bash script, use the following:

`a -eq b` for checking if `a` is equal to `b`

`a -ne b` for checking if `a` is not equal to `b`

`a -gt b` for checking if `a` is greater than `b`

Bash Scripting: Comparison

In order to compare **numbers** in a bash script, use the following:

`a -eq b` for checking if `a` is equal to `b`

`a -ne b` for checking if `a` is not equal to `b`

`a -gt b` for checking if `a` is greater than `b`

`a -ge b` for checking if `a` is greater than or equal to `b`

Bash Scripting: Comparison

In order to compare **numbers** in a bash script, use the following:

`a -eq b` for checking if `a` is equal to `b`

`a -ne b` for checking if `a` is not equal to `b`

`a -gt b` for checking if `a` is greater than `b`

`a -ge b` for checking if `a` is greater than or equal to `b`

`a -lt b` for checking if `a` is less than `b`

Bash Scripting: Comparison

In order to compare **numbers** in a bash script, use the following:

`a -eq b` for checking if `a` is equal to `b`

`a -ne b` for checking if `a` is not equal to `b`

`a -gt b` for checking if `a` is greater than `b`

`a -ge b` for checking if `a` is greater than or equal to `b`

`a -lt b` for checking if `a` is less than `b`

`a -lt b` for checking if `a` is less than or equal to `b`

Bash Scripting: Comparison

In order to compare **strings** in a bash script, use the following:

`s1 = s2` for checking if `s1` is equal to `s2`

Bash Scripting: Comparison

In order to compare **strings** in a bash script, use the following:

`s1 = s2` for checking if `s1` is equal to `s2`

`s1 != s2` for checking if `s1` is not equal to `s2`

Bash Scripting: Comparison

In order to compare **strings** in a bash script, use the following:

`s1 = s2` for checking if `s1` is equal to `s2`

`s1 != s2` for checking if `s1` is not equal to `s2`

`s1 < s2` for checking if `s1` is less than `s2` by lexicographical order

Bash Scripting: Comparison

In order to compare **strings** in a bash script, use the following:

- `s1 = s2` for checking if `s1` is equal to `s2`
- `s1 != s2` for checking if `s1` is not equal to `s2`
- `s1 < s2` for checking if `s1` is less than `s2` by lexicographical order
- `s1 > s2` for checking if `s1` is greater than to `s2` by lexicographical order

Bash Scripting: Comparison

In order to compare **strings** in a bash script, use the following:

`s1 = s2` for checking if `s1` is equal to `s2`

`s1 != s2` for checking if `s1` is not equal to `s2`

`s1 < s2` for checking if `s1` is less than `s2` by lexicographical order

`s1 > s2` for checking if `s1` is greater than to `s2` by lexicographical order

`-n s1` for checking if `s1` has a length greater than 0

Bash Scripting: Comparison

In order to compare **strings** in a bash script, use the following:

`s1 = s2` for checking if `s1` is equal to `s2`

`s1 != s2` for checking if `s1` is not equal to `s2`

`s1 < s2` for checking if `s1` is less than `s2` by lexicographical order

`s1 > s2` for checking if `s1` is greater than to `s2` by lexicographical order

`-n s1` for checking if `s1` has a length greater than 0

`-z s1` for checking if `s1` has a length of 0

Bash Scripting: Arithmetic

To do arithmetic, we need to follow bash syntax.

Bash Scripting: Arithmetic

To do arithmetic, we need to follow bash syntax.

To add two numbers `1` and `2`, and then assign to a variable `a`:

```
a=$((1+2))
```

Bash Scripting: Arithmetic

To do arithmetic, we need to follow bash syntax.

To add two numbers `1` and `2`, and then assign to a variable `a`:

```
a=$((1+2))
```

You can also use the `let` keyword:

```
let a=1+2
```

Bash Scripting: Arithmetic

To do arithmetic, we need to follow bash syntax.

To add two numbers `1` and `2`, and then assign to a variable `a`:

```
a=$((1+2))
```

You can also use the `let` keyword:

```
let a=1+2
```

You can use the expr keyword:

```
a=$( expr 1 + 2 )
```

Bash Scripting: Exercise

Exercise 4: Write a shell script called `timely_greeting.sh` that greets you based on the current time. The script should call the `date` command, extract the current hour (look into using `%H`) and then print the following greeting based on the time.

If it is between 5AM (05:00) and 12PM (12:00): `Good morning!`

If it is between 12PM (12:00) and 6PM (18:00): `Good afternoon!`

If it is between 6PM (18:00) and 5AM (5:00): `Good night!`

Bash Scripting: Exercise

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

time=$(date +%H)
if [ $time -gt 5 ] && [ $time -lt 12 ]
then
    echo "Good morning!"
elif [ $time -gt 12 ] && [ $time -lt 18 ]
then
    echo "Good evening!"
elif [ $time -gt 18 ] && [ $time -lt 5 ]
then
    echo "Good night!"
fi
```

Advanced Running

You can turn your shell script into a "command" by moving it to `~/bin`. For example if you have a script called `hello`, you could do the following:

```
adrazen@ayelet-computer ~ % mv hello ~/bin/
```

You can then run the command by just calling `hello`:

```
adrazen@ayelet-computer ~ % hello
```

Note: this probably won't work yet on your computer but we will learn about it in a later lecture.