CS45, Lecture 3: Shell Scripting

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Winter 2023

Contents

1 Lecture Overview

In Lecture 3, we learned how to use shell commands and pipelines to manipulate and analyze data. We also learned how to write regular expressions and how to incorporate these into tools such as `sed`. Finally, we learned how to run complex shell commands such as `grep`, `sort`, `uniq`, and `xargs`. In today's lecture we will learn how to write shell scripts and the syntax of shell scripts.

2 What is Shell Scripting?

We've already seen how to execute simple commands in the shell and pipe multiple commands together. Sometimes, we want to run many commands together and make use of control flow expressions such as conditionals and loops. This is where shell scripting comes in.

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.

Most shells have their own scripting language, each with variables, control flow, and its own syntax. (You may have heard of bash scripting as a popular scripting language. Bash scripting is a type of shell scripting.) 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 primitives in shell scripting, making it easier to use compared to other scripting languages. (For example, if you want to run the `cd` command in Python, you would need to import the `os` library and then call `chdir` from that library.)

3 Bash Scripting: Basic Mechanics

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, you may need to switch to a bash shell. On macOS run `exec bash` to launch a bash shell. On Windows, run `bash` to launch a bash shell.

3.1 Your Very First Script

Now that we have a bash shell, we can write our very first bash script, called `hello.sh`. Shell scripts normally end with the `.sh` ending to indicate that they are scripts. To run a script, you will type the following at your command line: `sh hello.sh`

This script will make use of the `echo` command that we learned about in Lecture 2.

```
#!/usr/bin/env bash
echo "Hello world!"
```

3.2 Shebangs

The first line in our script (`#!/usr/env/ bash`) is called a shebang, or sharp exclamation. It 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 our script to be run with a bash interpreter (i.e. a bash shell). If you want to run your script with a zsh shell, you would change your shebang to reflect as such.

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`.

Another example of a shebang that may be useful to many of you is using a shebang to write a python script. You may be familiar with writing a Python script and then running the script by calling `python3`. Yous can also create a Python script and specify that it should be run with Python using the shebang.

3.3 Running a Script

You can always run a shell script by simply prepending it with a shell interpreter program such as `sh hello.sh`, `bash hello.sh`, or `zsh hello.sh`.

You can also run a script by turning it into an executable program and then running it. First, you need to turn the program into an executable using the `chmod` (change mode) command. This command is used for changing the file permissions on a file, such as making the file executable. In our case, we will `chmod` with the `+x` argument to turn the script into an executable. You can do so as follows:
`chmod +x hello.sh`

To run the script, you would then simply run the program with `./hello.sh` (which is likely similar to how you have run other programs in the past.

4 Bash Scripting: Variables and Strings

4.1 Variables

Now that we have a basic script, let's talk about the mechanics of bash scripting. When writing a bash script, you can assign variables using the syntax `x=foo`. You can then access this variable using the syntax `$x`. One thing to be careful of is that when you assign a variable in a bash script, you should not add extra spaces. If you write `x = foo` then the line will be interpreted as running a program called `x` with the arguments `=` and `foo`. In general, shell scripts interpret the space character as an argument splitter.

4.2 Strings

We can also define strings in a bash script. If we want to define a string literal, we will use single quotation marks: `'$x'`. If we want to define a string that allows substitution, we will use double quotes: `"$x"`. The double quotes will allow for substitution of variables:

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

5 Bash Scripting: Control Flow Directives

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

5.1 If Statements

The syntax for writing an if statement in bash is as follows:

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

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

The condition we are interested in is denoted by `CONDITION`.

Let's take a look at an example of a bash script with an if statement:

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

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

In this script, we begin by assigning a variable `num` to be equal to 101 on line 3. We then check whether `num` is greater than 100. Notice that we use the syntax `$num` to access the value of `num`. We use the comparison operator `-gt` to compare `num` to `100`. In bash, comparisons for integers and strings are done differently. `-gt` is an integer comparison operator while `>` is a string comparison operator (that compares ASCII values). The use of the square brackets is actually a synonym for the `test` command, which tests the validity of a command or statement.

We can also write an if-statement with multiple conditions. In this example, we will check if `num` is both greater than `100` and less than `1000`.

```
1  #!/usr/bin/env bash
2
3  num=101
4  if [ $num -gt 100 ] && [ $num -lt 1000 ]
5  then
6    echo "That's a big (but not a too big) number!"
7  fi
```

Again, in this script we assign `num` to be 101. We first check if `num` is greater than `100`. We than add a second condition using the `&&` syntax. Our second condition checks if `num` is less than 1000.

We can also use `elif` (for else if) and `else` if we have multiple different blocks of code. Here is the syntax for using `if`, `elif`, and `else`:

```
1  #!/usr/bin/env bash
2
3  if [ CONDITION ]
4  then
5    # do something
6  elif [ CONDITION ]
7  then
8  # do something else
9  else
10 # do something totally different
11 fi
```

The above syntax should look fairly similar to the traditional `if` statement. Here is an example of code that uses `if`, `elif`, and `else`:

```
1  #!/usr/bin/env bash
2
3  num=101
4  if [ $num -gt 1000 ]
5  then
6    echo "That's a huge number!"
7  elif [ $num -gt 100 ]
8  then
9    echo "That's a big number!"
10 else
11   echo "That's a small number."
12 fi
```

5.2 While Loops

We can also add while loops to our bash scripts. The syntax for adding a while loop to a bash script is the following:

```
1  #!/usr/bin/env bash
2
3  while [ CONDITION ]
4  do
5    # do something
6  done
```

Again, the condition of interest is denoted as `CONDITION`.

Here is an example of a script that initializes `num` to `0` and continues looping until `num` reaches a value of `99`.

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

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

Notice how we access the variable `num` with the `$` syntax: `$num`. We use the `-lt` flag to compare `num` to `100`. When then print the value of `num` using the `echo` command. To increment the value of `num`, we use the double parentheses `(( .. ))` for arithmetic evaluation. Inside of the double parentheses, we can increment the value of `num` by 1.

5.3 For Loops

To declare a for loop in bash, we can declare either an index-based for loop or a range-based for-loop. To declare an index based for-loop, we will use the following syntax:

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

for VARIABLE in 1 2 3 .. N
do
  # do something
done
```

Perhaps, we are interested in implementing the above `while` loop as `for` loop:

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

num=0
for i in {0..99}
do
  echo $num
  num=$((num+1))
done
```

Notice that we define our iterator `i` and we set the bounds of the for-loop to be 0 and 99.

5.4 Exercise 1

Let's put our scripting expertise to use and write a bash script. You should write a 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`.

Solution: Here is one possible solution : `num=$1`:

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

for i in {1..20}
do
  echo $i
  if [ $i -gt 10 ]
  then
    echo "I'm big!"
  fi
done
```

6 Bash Scripting: Arguments and Functions

6.1 Arguments

Our `big_num.sh` script isn't very interesting as it will always print the same thing: "That's a big number!" Let's take a look at how we might use command line arguments to make this 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. For example, consider the following:
`adrazen@thinking-computer CS45 % sh my_script.sh ayelet`
In this case, the name of the script (`my_script.sh`) is defined by the variable `$0`. The first argument to the script (`ayelet`) is defined by the variable `$1`. In the case of our `big_number.sh` script, we can change the value of `num` to be dependent on the first argument that is passed in when the script is invoked. In other words, `num` will simply be the value of `$1`.

Let's assume that we invoke the script as follows:
`adrazen@ayelet-computer CS45 % sh big_num.sh 102`

In our script, we can replace the line `num=101` to be `num=$1`:

```
1 #!/usr/bin/env bash
2
3 num=$1
4 if [ $num -gt 100 ]
5 then
6   echo "That's a big number!"
7 fi
```

By changing line 3, we now set the value of `num` to be the first argument from the command line when invoking the script. This means the use could pass a different value every time they invoke the script.

6.2 Functions

We can also define functions in bash. Let's define a function for making a new directory and then entering that directory. (This is a pretty commaon thing that users want to do and there is actually a command called `mcd` in UNIX to allow users to do that.) We can use the commands `mkdir` and `cd` to achieve this. When running `mkdir`, we want to make sure that we also create any necessary intermediate directories. Therefore, we will want to make sure we pass the `-p` flag when calling `mkdir`, which creates all the intermediate directories on the path to the final directory that do not already exist.

When defining a function in bash, you may wonder how to pass arguments to the function. Let's take a look at our function so far and how you might think of passing arguments:

```
1 #!/usr/bin/env bash
2
3 make_and_enter(directory_name) {
4   mkdir -p directory_name
5   cd directory_name
6 }
```

Unfortunately, this won't work in bash. In bash, we will refer to arguments that are passed into a function based on their position. For instance, we might use the value of the very first argument and refer to it using `$1`.

```
1 #!/usr/bin/env bash
2
3 make_and_enter() {
4   mkdir -p "$1"
5   cd "$1"
6 }
```

When we want to invoke our function, we will use the following line:
`make_and_enter new_folder`

In this case, we are calling our function (i.e. `make_and_enter`) with the argument `new_folder`. Our script will look as follows:

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

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

make_and_enter new_folder
```

Now consider that we want the name of the new folder to be passed from the command line whenever the script is invoked. In that case, we would invoke the script as follows: `adrazen@ayelet-computer CS45 % sh mcd.sh my_folder` In order for the argument `new_folder` to be used inside of our function, we need to make sure that we pass it into the function. In this case, we need to pass the argument from the command line invocation to the function call, and then from the function call to the function body. Our final script would look as follows:

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

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

make_and_enter "$1"
```

6.3 Exercise 2

Let's try another exercise to solidify our function-writing and argument-passing skills. In this exercise, you should 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`.

Solution:

Here is possible solution to this problem:

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

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

make_my_folder $1 $2
```

7 Exit Codes and Command Substitution

7.1 Exit Codes

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 previous command. (For students who are familiar with C/C++, you may notice that the `main()` function always returns an `int` and that this `int` is often 0. The `int` returned by `main()` is the exit code for the program.)

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.

Below is an example of a script that randomly generates either 0 or 1 and runs a either the `true` or the `false` command based on this random value. The script then checks the return value of the previous command. As you can tell, this script is a bit contrived but it demonstrates how you might use the return value of the previous command as an input to the next command.

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

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

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 `||`.

7.2 Exercise 3

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!`).

Solution:

Below is one possible solution:

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

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

To suppress the output of `cat`, you should modify the script as follows:

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

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

7.3 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.

Here is an example of a script that uses command substitution:

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

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

8 Bash Scripting: Other Syntax

There is plenty of other syntax to keep in mind when it comes to bash scripting. Here are a few other syntax details for bash.

8.1 Comparisons

Bash draws a distinction between comparisons for numbers and comparisons for strings. 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 -le b` or checking if a is less than or equal to b

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 > s1` for checking if s1 is greater than 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

8.2 Exercise 4

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!

Solution: Here is one possible solution:

```
#!/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
```