Tutorial 7: Bash

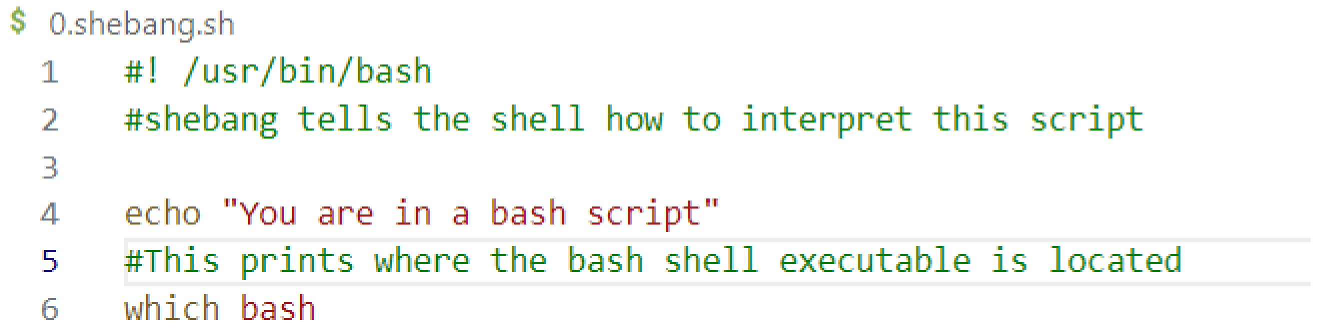
CS 104

Spring 2025

TA: Kritin Gupta

Introduction to Bash

- Bash is a scripting language. You write a series of commands in a “.sh” file to automate some tasks.
- The script you write runs in the shell as if you were typing commands one after the other.
- It supports many programming language features of variables, loops, conditionals and functions



$ 0.shebang.sh

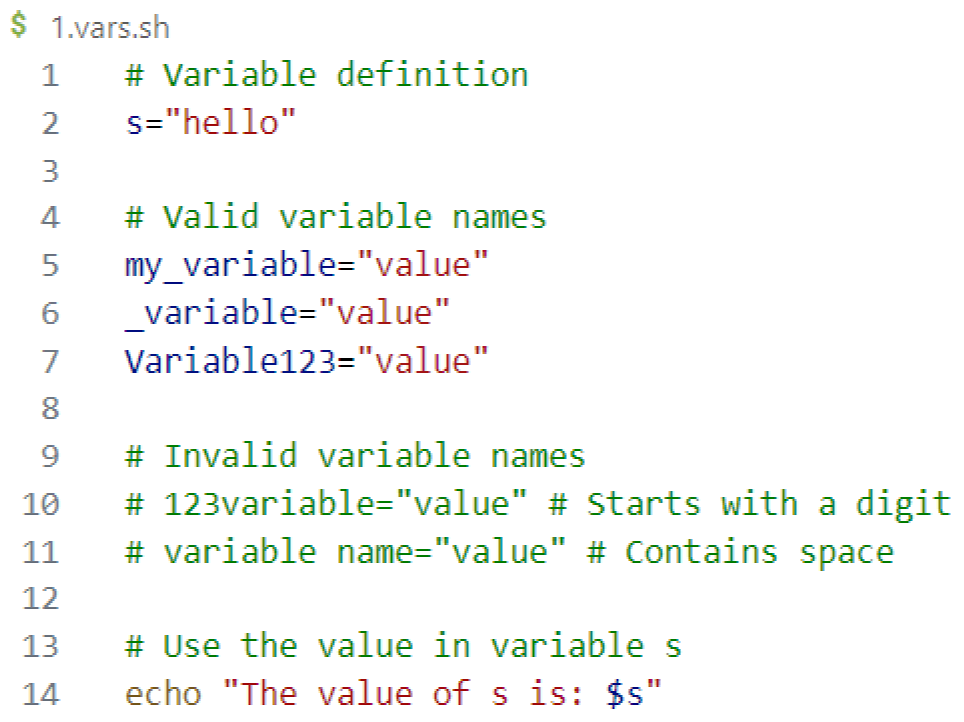
```bash
#! /usr/bin/bash
#shebang tells the shell how to interpret this script

echo "You are in a bash script"
#This prints where the bash shell executable is located
which bash
```

Note the use of shebang. If you were running a python script, you could write /usr/bin/python. Or, you could skip this entirely and run the script as bash “script name”

Variables in bash

- Variables are strings in bash by default, though you can also have integers.
- Declare/assign a variable: var=“hello”
- Use a variable: ${var}
- $(…) is command substitution, whereas ${…} is substituted by variable value.

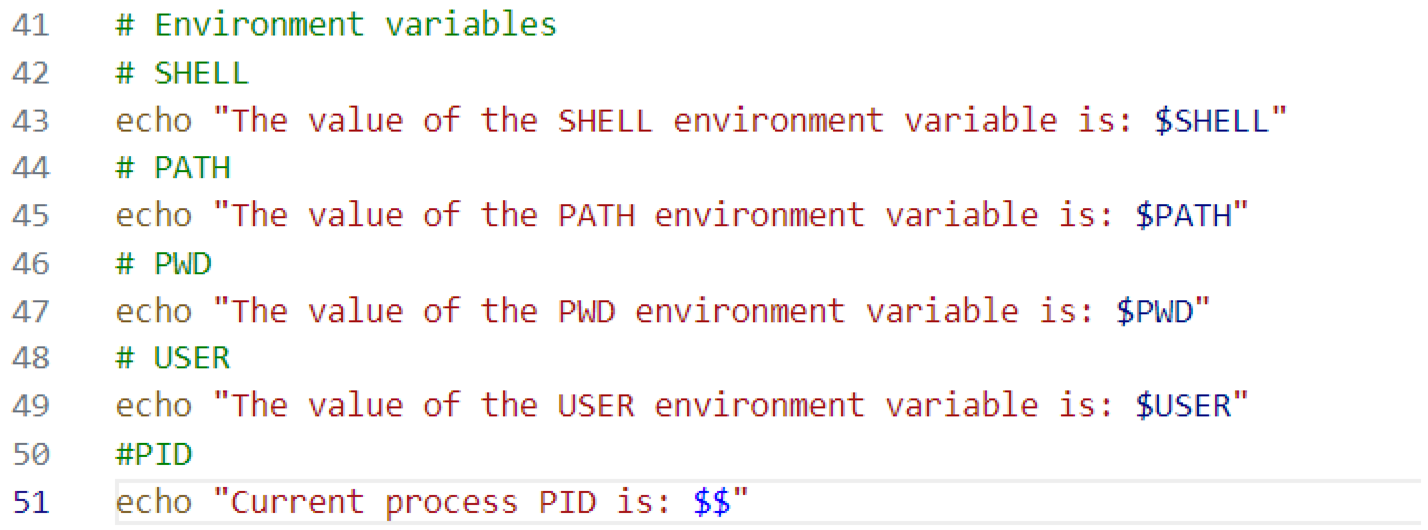
- Remember: No spaces around “=“


$ 1.vars.sh

```bash
# Variable definition
s="hello"

# Valid variable names
my_variable="value"
_variable="value"
Variable123="value"

# Invalid variable names
# 123variable="value" # Starts with a digit
# variable name="value" # Contains space

# Use the value in variable s
echo "The value of s is: $s"
```

Environment Variables

- These are some variables that store configurations and settings. These are some useful variables available to all processes. You can see all environment variables using env.
- These can be used as normal variables



```
# Environment variables
# SHELL
echo "The value of the SHELL environment variable is: $SHELL"
# PATH
echo "The value of the PATH environment variable is: $PATH"
# PWD
echo "The value of the PWD environment variable is: $PWD"
# USER
echo "The value of the USER environment variable is: $USER"
#PID
echo "Current process PID is: $$"
```

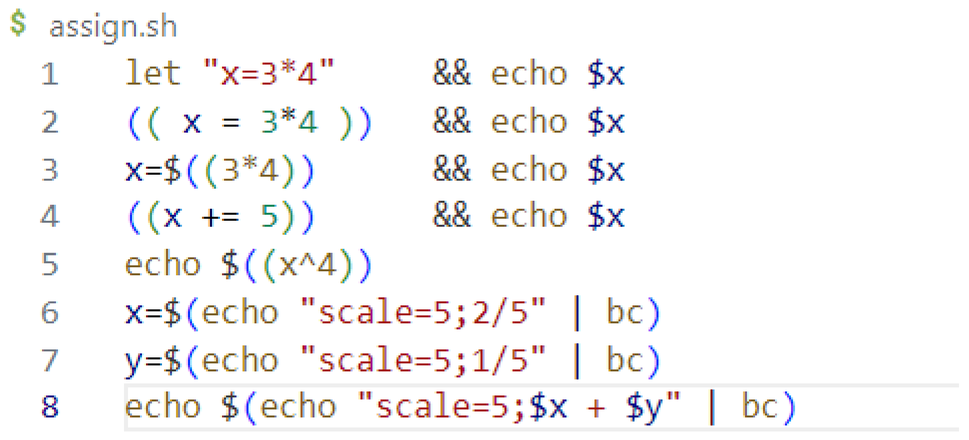
Arrays

- Arrays in bash can be declared as `arr=(val1 val2 val3)`
- i-th index element can be accessed as `${arr[i]}`
- All elements can be accessed as `${arr[*]}`
- Length of the array can be accessed as `${#arr[*]}`
- Elements can be added using `+=` and removed using `unset arr[i]`

- Declarative arrays (like python dictionary or C++ maps) are declared as `declare –A arr`
- Here, the keys are strings and are accessed as `${arr[“key”]}`
- All keys can be accessed as `${!arr[*]}`
- Elements can be added by `arr[“new-key”]=“new-val”` and removed using `unset arr[“key”]`

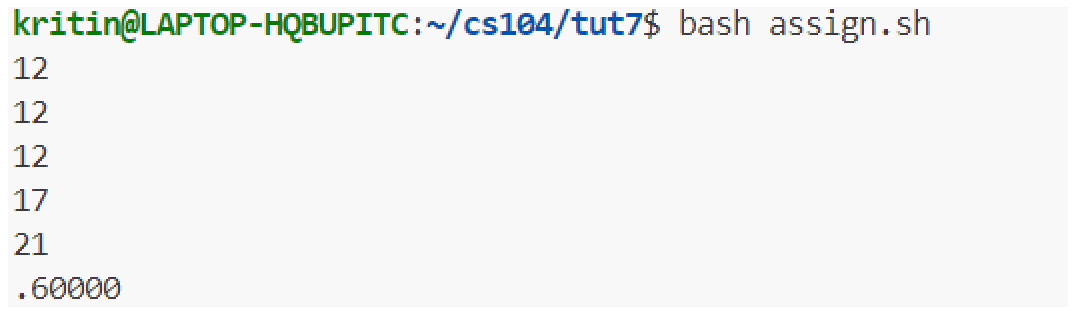
Arithmetic

- Assign value to integer variables using let or ((…))
- Use $((…)) for computing the value of an arithmetic expression
- Bash supported arithmetic operators: Add(+), Sub(-), Mul(*), Div(/), Mod(%), Exp(**)
- Bitwise Operators: And(&), Or(|), Not(~), XOR(^), Left Shift(<<), Right Shift(>>)
- Assignment operators: =, {+,-,*,/,%}=
- Use bc for floating point arithmetic



$ assign.sh

```bash
let "x=3*4"      && echo $x
(( x = 3*4 ))    && echo $x
x=$((3*4))       && echo $x
((x += 5))       && echo $x
echo $((x^4))
x=$(echo "scale=5;2/5" | bc)
y=$(echo "scale=5;1/5" | bc)
echo $(echo "scale=5;$x + $y" | bc)
```



```
kritin@LAPTOP-HQBUPITC:~/cs104/tut7$ bash assign.sh
12
12
12
17
21
.60000
```

Conditionals

Syntax:

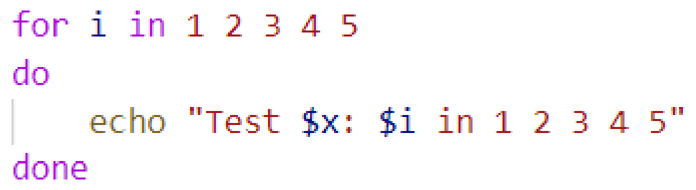
```
if CONDITION; then
        #commands
elif CONDITION; then
        #commands
else
        #commands
fi
```

- The condition can be any command or function call. If the condition exited with a return value of 0, it is evaluated as true and false otherwise.
- The most common CONDITION is the test command or [[…]]
- See man test for all the options you can give it.
- [[…]] is preferred over […] as it is more modern and much easier to use.
- This allows for arithmetic expression, string comparison and file checking

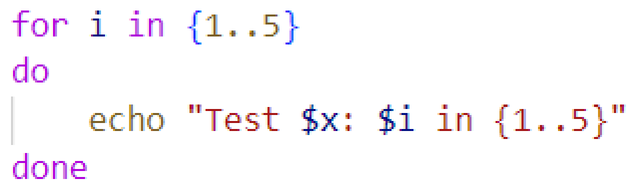
Loops

```
for variable in list; do
        #commands
done

while CONDITION; do
        #commands
done

until CONDITION; do
        #commands
done
```

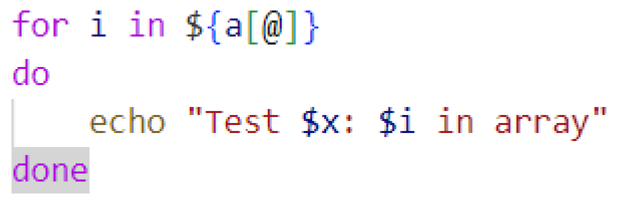
- For while and until loops, CONDITION is similar to if command. While breaks when CONDITION is false and until breaks when it is true.



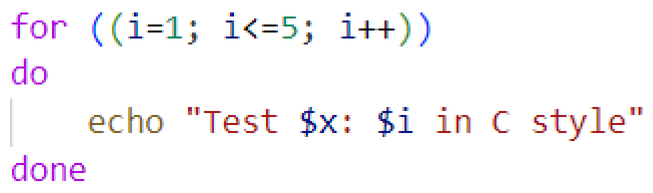
```
for i in 1 2 3 4 5
do
    echo "Test $x: $i in 1 2 3 4 5"
done
```



```
for i in {1..5}
do
    echo "Test $x: $i in {1..5}"
done
```



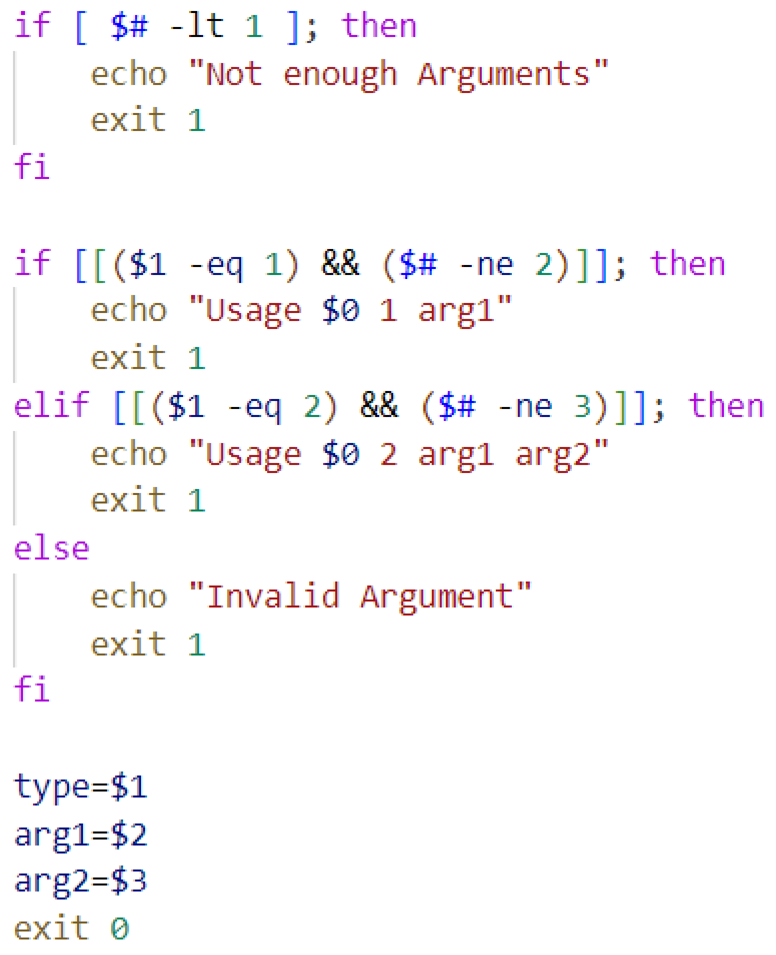
```
for i in ${a[@]}
do
    echo "Test $x: $i in array"
done
```



```
for ((i=1; i<=5; i++))
do
    echo "Test $x: $i in C style"
done
```

Command Line Arguments

- A bash script can be given command line arguments like `./script.sh arg1 arg2 arg3`
- The script can read these arguments in the following variables:
  - $#: Number of arguments
  - $0: script name
  - $1, $2, $3…: first, second, third argument respectively
  - $*: all arguments as a string
  - $@: all arguments as an array



```
if [ $# -lt 1 ]; then
    echo "Not enough Arguments"
    exit 1
fi

if [[($1 -eq 1) && ($# -ne 2)]]; then
    echo "Usage $0 1 arg1"
    exit 1
elif [[($1 -eq 2) && ($# -ne 3)]]; then
    echo "Usage $0 2 arg1 arg2"
    exit 1
else
    echo "Invalid Argument"
    exit 1
fi

type=$1
arg1=$2
arg2=$3
exit 0
```

Functions

- Functions can be declared as `func_name(){…}` and called as `func_name arg1 arg2`
- Arguments can be accessed inside the function using `$1`, `$2` etc.
- Function can return a value using echo, and calling script can get it using command substitution
- Return statement is for returning exit code only, which can be read by `$?`

- Local variables: exist within the scope of function only. (use local keyword)
- Global variables: available to all functions in script.
- Exported variables: available to child scripts as well.

```
let x=1
export x
```

File IO

- Reading:

```
if [ -f $file ]; then
    while read line; do
        echo $line
    done < $file
fi
```

- Writing:

```
cat << END >> $file
This text will be appended to the file $file
This will go on till I type END
END
```

Thank You