University of Zurich[UZH]

URPP Evolution in Action

Next-Generation Sequencing 2 – Advanced Course

Basic bash scripting

Dr. Heidi E.L. Lischer
University of Zurich
Switzerland

13 May, 2015

What is a bash script?

- Mainly a set of commands that can be executed in the terminal
  - Text file with commands
  - Should begin with the shebang line (`#!/bin/bash`)
  - Executed line by line → new line = new command
- Run same command with different parameters
- Create small programs or pipelines
- Bash script files end with .sh
- Example: script.sh

```
#!/bin/bash
#this is a comment
echo "Hello World"
```

→ shebang line
→ A comment (ignored)
→ Print something in terminal

```
chmod +x script.sh
./script.sh
Hello World
```

→ Give execution permission
→ Run script

Write to a file

- “>” will store the output of a command to a file

```
#!/bin/bash
#this is a comment
echo "Hello World" > log.txt
echo "How are you" > log.txt
```

→ Print «Hello World» to log.txt file

→ Will overwrite the above log.txt file

- “>>” will append the output of a command to the file

```
#!/bin/bash
#this is a comment
echo "Hello World" > log.txt
echo "How are you" >> log.txt
```

→ Print «Hello World» to log.txt file

→ Append «How are you» to log.txt

Variables

Variables: names for values

- Store data and configuration options
- Make scripts shorter and easier
- Easier to change values
- Create: name followed by "=" and the value
- Call: put "$" in front of the name,
  enclose the name in {} if directly followed by something else
- Example:

```
#!/bin/bash
a=/home/user
echo "My home folder is $a"
echo "My data folder is ${a}/data"
```

→ Create variable with name a

```
My home folder is /home/user
My data folder is /home/user/data
```

Arrays

- Arrays are variables containing multiple values
- Create: name followed by “=()“, values are space separated
- Values can be accessed by their index (number) starting from 0
- Example:

```
#!/bin/bash
col=(red blue yellow)
echo ${col[*]}
col[3]=green
echo ${col[*]}
echo "The sun is ${col[2]}"
echo ${#col[*]}
```

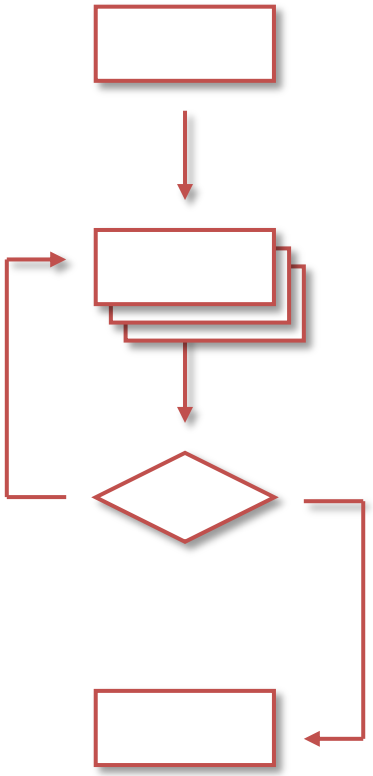
→ Create array with name col
→ Print all elements of the array
→ Set 4. element in the array
→ Return 3. element of the array
→ Returns the number of elements in the array

```
red blue yellow
red blue yellow green
The sun is yellow
4
```

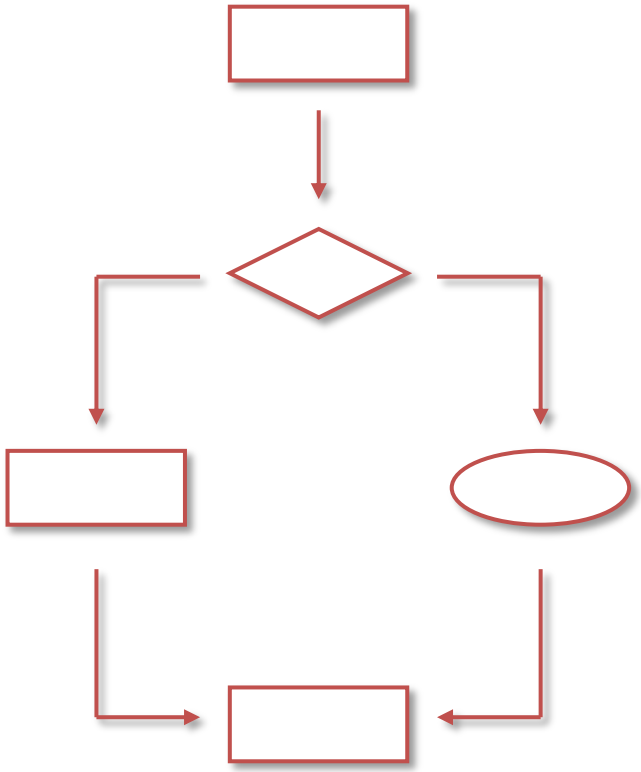
Flow control

- Real power of programs comes from:

repetition



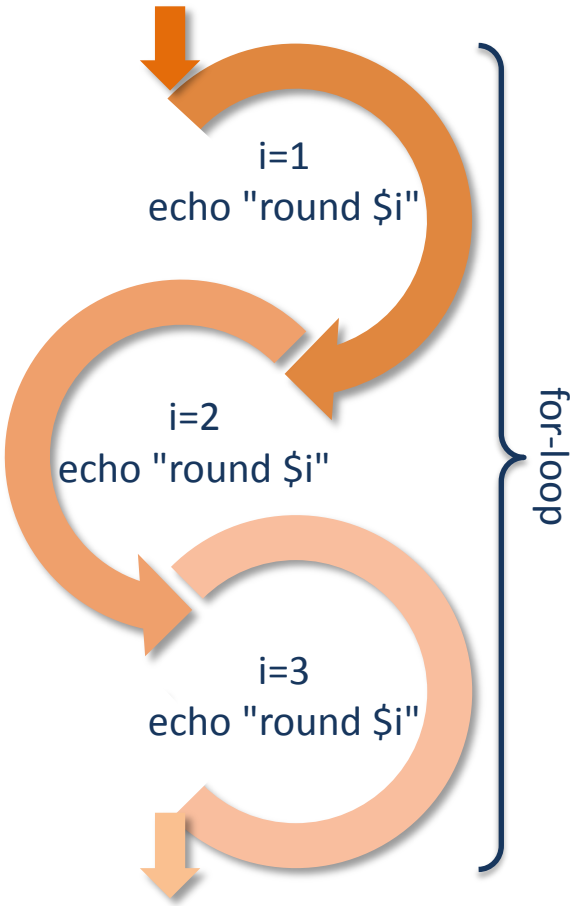
selection



For loop

```
for VARIABLE in 1 2 3
do
  command1
done
```

- Can be used to repeat certain tasks



- Examples:

```
#!/bin/bash
for i in 1 2 3
do
  echo "round $i"
done
```

```
#!/bin/bash
for i in {1..3}
do
  echo "round $i"
done
```

```
round 1
round 2
round 3
```

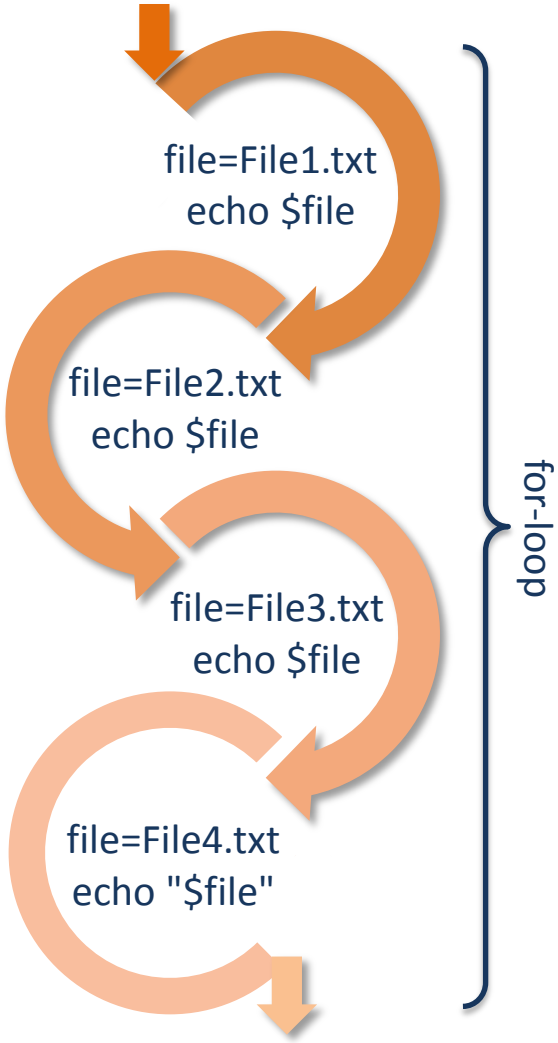
For loop

- Examples:

Go through all files in a folder:

```
#!/bin/bash
for file in /home/user/*
do
  echo $file
done
```

```
File1.txt
File2.txt
File3.txt
File4.txt
```



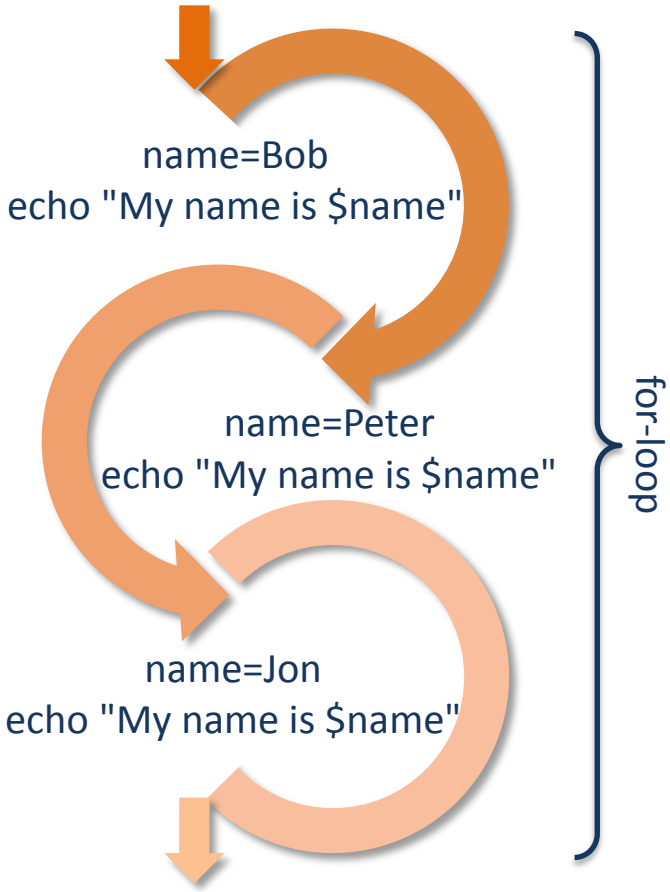
For loop

- Examples:

Go through all items of an array:

```
#!/bin/bash
names=(Bob Peter Jon)
for name in ${names[*]}
do
  echo "My name is $name"
done
```

```
My name is Bob
My name is Peter
My name is Jon
```



If statement

- If statements can be used to check for something and do something else depending on the outcome of the check → make choices

```
if [[ someting ]]
then
  command1
elif [[ something ]]
then
  command2
else
  command3
fi
```

  - Can have any number of `elif` clauses (including none)
  - `else` clause is optional
  - Always tested in order
    → if one test is true, its block of statements is executed and no other branch is tested

If statement

Boolean operators:

- **-e FILE**: True if file exists

```
#!/bin/bash
if [[ -e log.txt ]]
then
  echo "log file exist"
else
  echo "log file doesn't exist"
fi
```

If statement

Boolean operators:

- **-e FILE**: True if file exists
- **STRING = STRING**: True if first string is identical to the second
- **STRING != STRING**: True if first string is not identical to the second
- **STRING < STRING**: True if first string sorts before the second
- **STRING > STRING**: True if first string sorts after the second

```
#!/bin/bash
name=Bob
if [[ $name = "Rod" ]]
then
  echo "Your name is Rod"
else
  echo "Your name is not Rod"
fi
```

If statement

```
#!/bin/bash
a=10
if [ $a -eq 7 ]
then
  echo "Not equal to 7"
else
  echo "You guessed $a"
fi
```

Boolean operators:

- **-e FILE**: True if file exists
- **STRING = STRING**: True if first string is identical to the second
- **STRING != STRING**: True if first string is not identical to the second
- **STRING < STRING**: True if first string sorts before the second
- **STRING > STRING**: True if first string sorts after the second
- **INT -eq INT**: True if both integers are identical
- **INT -ne INT**: True if integers are not identical
- **INT -lt INT**: True if first integer is less than the second
- **INT -gt INT**: True if first integer is greater than the second
- **INT -le INT**: True if first integer is less than or equal to the second
- **INT -ge INT**: True if first integer is greater than or equal to the second

If statement

```
#!/bin/bash
a=Bob
if [[ $a = "Rob" || $a = "Tod" ]]
then
  echo "Your name is Rod or Tod"
else
  echo "Your name is not Rod or Tod"
fi
```

Boolean operators:

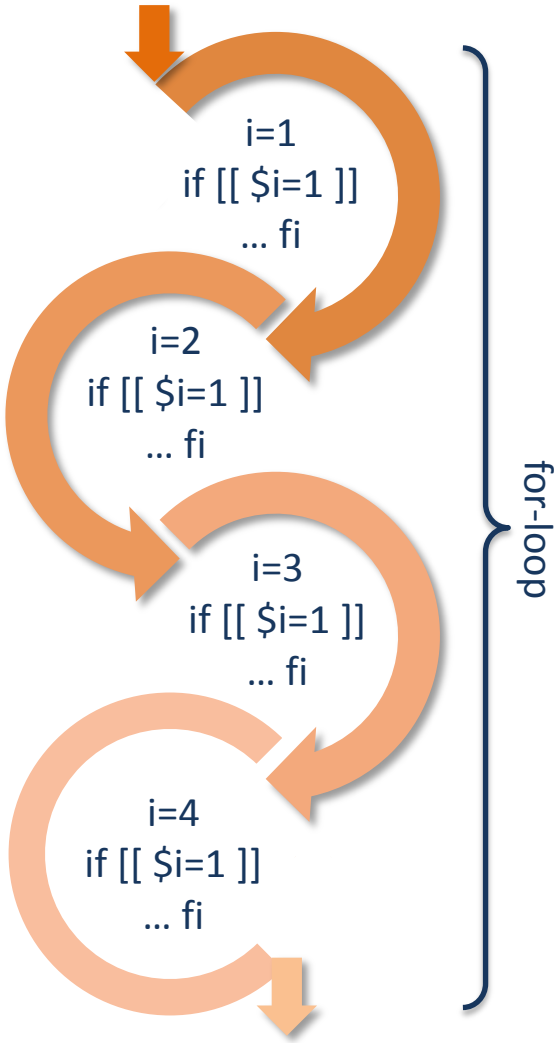
- **-e FILE**: True if file exists
- **STRING = STRING**: True if first string is identical to the second
- **STRING != STRING**: True if first string is not identical to the second
- **STRING < STRING**: True if first string sorts before the second
- **STRING > STRING**: True if first string sorts after the second
- **INT -eq INT**: True if both integers are identical
- **INT -ne INT**: True if integers are not identical
- **INT -lt INT**: True if first integer is less than the second
- **INT -gt INT**: True if first integer is greater than the second
- **INT -le INT**: True if first integer is less than or equal to the second
- **INT -ge INT**: True if first integer is greater than or equal to the second
- **! EXPR**: Inverts the result of the expression (logical NOT)
- **EXPR && EXPR**: True if both expressions are true (logical AND)
- **EXPR || EXPR**: True if either expression is true (logical OR)

If statement

- Example:

```
#!/bin/bash
for i in {1..4}
do
  if [[ $i = 1 ]]
  then
    echo "first round"
  elif [[ $i = 2 ]]
  then
    echo "second round"
  else
    echo "round $i"
  fi
done
```

```
first round
second round
round 3
round 4
```



Run commands in parallel

- To run a command in the background add & at the end of the line

```
command1 &
command2 &
wait
```

→ wait until commands are finished

- This can be used to easily parallelise jobs within a for-loop

```
for i in {1..10}
do
  command &
done
wait
```

→ Don't forget the wait after the for-loop to make sure that commands are finished before doing the next step

→ **Attention: this will need 10 threads**

Run commands in parallel

- To limit the number of threads one can implement a count variable, which makes sure that only a limited number of threads are started:

```
count=0
for i in {1..10}
do
  command &
  let count+=1
  if [[ $((count%4)) -eq 0 ]]
   then wait
  fi
done
wait
```

→ call wait if count is a multiple of 4, thus only 4 threads are started at once

`$((count%4))`: brackets needed to make sure that expression is interpreted as mathematical operators
(% → modulo, returns the *remainder* of the division)

Functions

- Functions are blocks of commands
- Arguments can be passed to functions and be accessed by $1 (first argument), $2 (second argument),…
- Example:

```
#!/bin/bash
sum() {
  echo "$1 + $2 = $(($1 + $2))"
}

sum 1 4
sum 8 7
```

Set up function

Call function

Call function

```
1 + 4 = 5
8 + 7 = 15
```

Special characters

- **#**: Comment character, anything afterwards on that line is ignored
- **$**: Expansion character (see variables)
- **"text"**: protect the text inside from being split into multiple words or arguments
- **'text'**: protect the text inside from being split into multiple words or arguments, but prevents special meaning of all special characters
- ****: Escape character, prevents the next character from being used in any special sort of way
- **>** or **<**: Redirection characters, are used to modify (redirect) the input and/or output of a command
- **|**: Pipelines allow you to send the output of one command as the input to another command

```
#!/bin/bash
name=Bob
echo "I am $name"
echo 'I am $name'
echo "I am \$name"
```

```
I am Bob
I am $name
I am $name
```

Special characters

- **[[expression]]**: evaluates the conditional expression as a logical statement (true/false)
- **{ commands; }** : Command grouping, commands inside the braces are treated as though they were only one command (e.g.: see functions)
- **((expression))**: Inside the parentheses, operators such as +, -, * and / are seen as mathematical operators
- **$((expression))**: Comparable to the above, but this expression is replaced with the result of its arithmetic evaluation

```
#!/bin/bash
a=4
b=6
echo "The average is (( (a+b)/2 ))"
echo "The average is $(( (a+b)/2 ))"
```

```
The average is (( (a+b)/2 ))
The average is 5
```

Acknowledgment

Tutorials:

- http://www.allaboutlinux.eu/bash-script-for-beginners/
- http://www.howtogeek.com/67469/the-beginners-guide-to-shell-scripting-the-basics/
- http://bash.cyberciti.biz/guide/Main_Page
- http://www.arachnoid.com/linux/shell_programming.html
- http://www.tldp.org/LDP/Bash-Beginners-Guide/html/

- http://mywiki.wooledge.org/BashGuide

→ There are a lot of other online tutorials available and forums discussing diverse kinds of topics!