Tutorial 2 – Bash & Scripting

MICB405 – BIOINFORMATICS – 2021W-T1

17 SEPTEMBER 2021

AXEL HAUDUC

The Unix Philosophy

1. Make each program do one thing well. To do a new job, build a new program rather than complicate old programs.
2. The output of any program should easily become the input for another.

Utilities review

cd (change directory)

ls (list files) (-l) (long)

pwd (print working directory)

cp (copy)

rm (remove) (-r) (-recursive)

mv (move)

cat (concatenate)

echo (print content you provided as argument)

More utilities…

grep (global regular expression print)

less (read text files)

head/tail (print first/)

wc (wordcount) (-l, -w, -m) (-lines, -word, character)

sort (sort lines)

uniq (return unique lines)

chmod (change mode a.k.a. permissions)

mkdir (make directory)

`awk` command

Less Unix-y than previous commands

- More of a multi-tool for dealing with data files
- Not short for anything – does file processing based on lines & columns
- Follows the format:
  - **`awk ‘yourFilter { yourAction }’ yourFile`**
  - **`awk ‘NR >= 2 { print $0 }’ mtcars.tsv`**

Special characters

`>` (redirect)

`>>` (append)

`|` (pipe) (not 1, l, or I)

`;` (separate command)

`&&` (AND) `||` (OR)

`*` (wildcard)

In-terminal text editing

`nano` = easiest to use & sufficient for most purposes

- Example: **`nano filename.txt`**
- This makes a new file, or edits an existing one of that name

```
Ctrl + o = overwrite (save)
Ctrl + x = exit
```

Scripting

Helps repeat repetitive tasks

Scripts the form of a text file that can be “executed” when you need them

Script headers

All script files need a header to indicate how the script file should be run

First line specifies the program that will interpret the rest of the script

```
#!/bin/bash
```

- Program should interpret following text using Bash
- Other programs = other headers, if run by calling the script directly

Script variables

We can create a variable and assign it a value with **`RESULTS_DIR="results/"`**

- Note that spaces matter when setting Bash variables. Do not use spaces around the equal sign

Variables

To access a variable’s value, we use a dollar sign in front of the variable’s name.

Suppose we want to create a directory for a sample’s alignment data, called <sample>_aln/, where <sample> is replaced by the sample’s name.

```
SAMPLE=“Individual_2A”
mkdir “${SAMPLE}_data/”
```

This will create a file with the name “Individual_2A_data”

Command line arguments

```
grep -c “string” yourFile
```

```
bash mySearchScript.bash ./folder/file
```

Arguments added after calling a script are assigned to the default variables $1, $2, $3, and so forth… within the script

`for` loops

In bioinformatics, most of our data is split across multiple files.

Many processing pipelines need a way to apply the same workflow on each file, taking care to keep track of sample names.

Looping over files with Bash's for loop = simplest way to accomplish this

Three essential parts to creating a pipeline to process a set of files:

1. Selecting which files to apply the commands to
2. Looping over the data and applying the commands
3. Keeping track of the names of any output files created

for loops

```
#!/bin/bash

for A_FILE in /home/username/*
do
    ACTION on $A_FILE
    ANOTHER ACTION on $A_FILE
done
```

`for` looping through directories

Creating **`for`** loops that loop through an entire directory

```
#!/bin/bash
for FOO in /home/ahauduc_mb20/test_directory/*
do
     head $FOO > ${FOO}.head.and.tail.txt
     tail $FOO >> ${FOO}.head.and.tail.txt
done
```

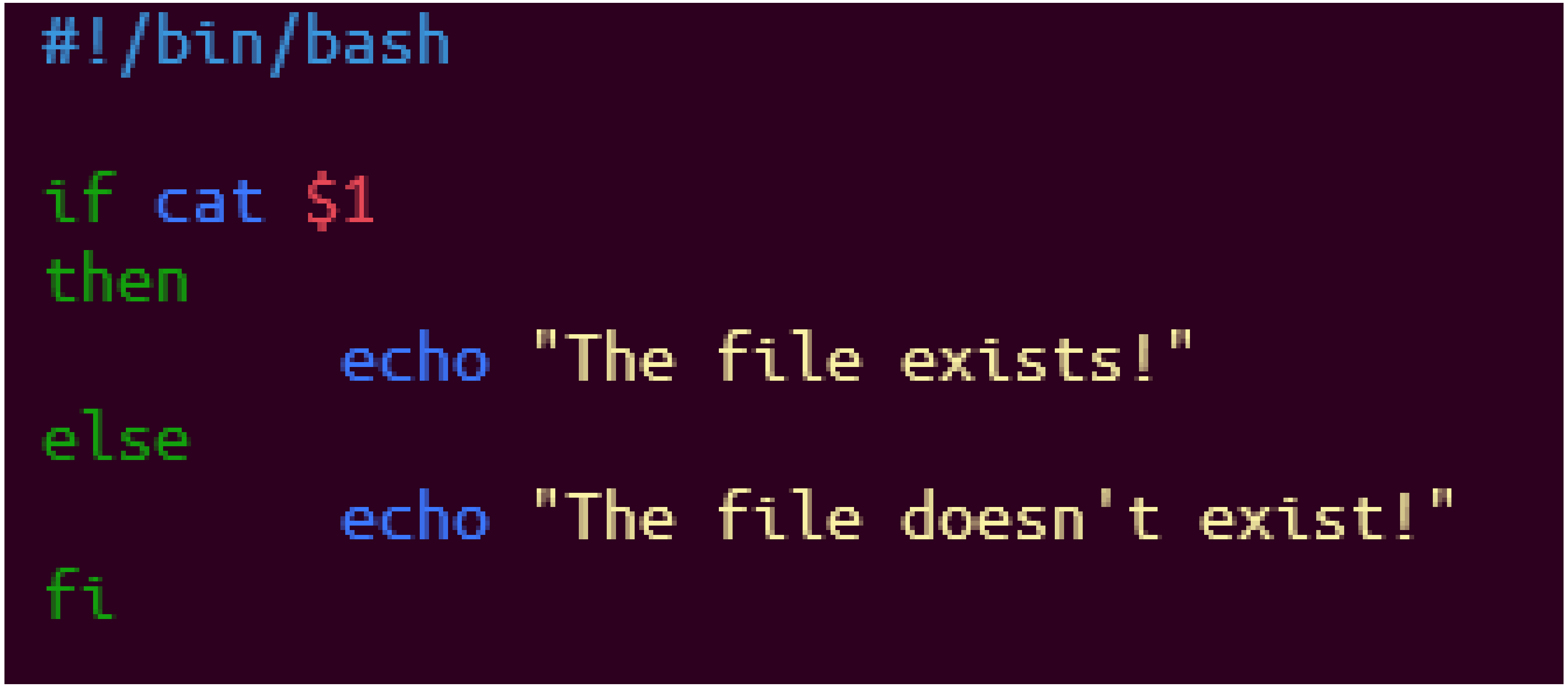
`if` statements

Use if you want to perform commands on a subset of files, or only if an action meets certain conditions.

The basic syntax is:

```
if [command is true]
then
     DO THIS
else
     DO THAT
fi
```

`if` statements



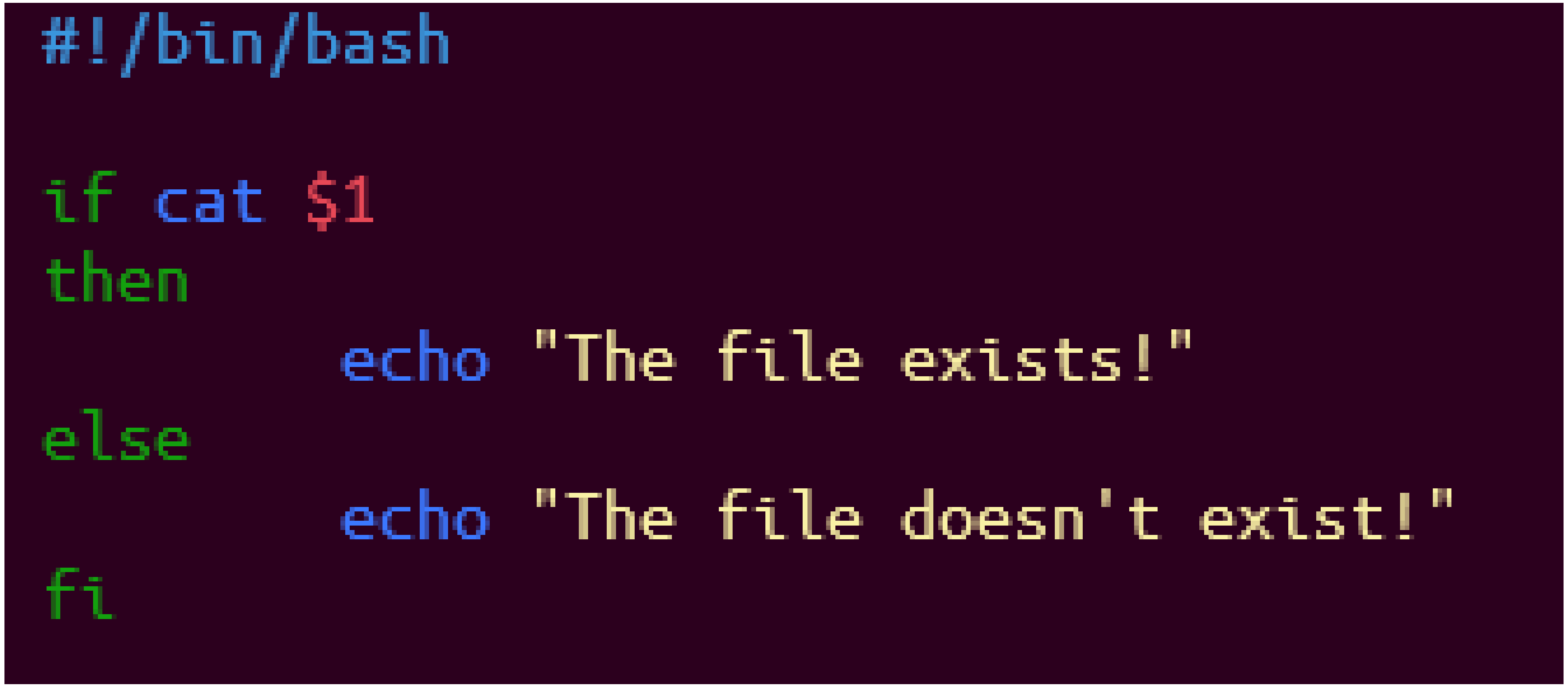
```
#!/bin/bash

if cat $1
then
        echo "The file exists!"
else
        echo "The file doesn't exist!"
fi
```

Return codes

`0`	=	Command executed successfully
`1+`	=	Command did not execute successfully

- There can by many different error types denoted by specific numbers

Return codes are invisible



```
#!/bin/bash

if cat $1
then
	echo "The file exists!"
else
	echo "The file doesn't exist!"
fi
```

`test` statements: **`[[ *statement* ]]`**

Like other programs, **`test`** exits with either 0 or 1.

Test statements can be included at the beginning of the if program to make producing the right return code easier

`test` supports numerous helpful comparisons you might need

`test` summary

String/integer	Description
`-z str`	String str is null
`str1 = str2`	str1 and str2 are identical
`str1 != str2`	str1 and str2 are different
`int1 -eq -int2`	Integers int1 and int2 are equal
`int1 -ne -int2`	int1 and int2 are not equal
`int1 -lt -int2`	int1 is less than int2
`int1 -gt -int2`	int1 is greater than int2
`int1 -le -int2`	int1 is less than or equal to int2
`int1 -ge -int2`	int1 is greater than or equal to int2

`test` summary for files/directories

File/directory expression	Description
-d dir	dir is a directory
-f file	file is a file
-e file	file exists
-r file	file is readable
-w file	file is writable
-x file	file is executable

if Statements + **test** Statements

Combining test with if statements is simple:

```
#!/bin/bash
if [[ -f some_file.txt ]]
then
     ACTION TO PERFORM
else
     ACTION TO PERFORM
fi
```

- Note the spaces around and within the brackets: these are required.

Putting it all together…

for looping through directories

```
#!/bin/bash
for foo in /home/axel/Documents/data/*
do
      if [[ -f ${foo} ]]
      then
            head ${foo} > ${foo}.head.and.tail.txt
            tail ${foo} >> ${foo}.head.and.tail.txt
      else
            echo “${foo} is not a file to summarize!”
      fi
done
```