

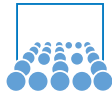
Shell vs. Python

Scripting with Bash

Compact Course @ Max-Planck

Tobias Neckel & Severin Reiz

March 19 – 22, 2019



Shell (Bash)

- Separate program for each simple task
- Gluing together programs with a script
- Not really a full programming language
- Powerful tools available
- Suitable for small tools (1-100 lines of code)

Python

- Full programming language
- Large number of libraries available
- Intuitive naming conventions
- Suitable for almost any task

Use of Shell Scripts

Where to use them:

- System administration
- Automating everyday terminal tasks
- Searching in (and manipulating) ASCII-Files
- ...

Where not to use them:

- Lots of mathematical operations
- Computational expensive tasks
- Large programs which need structuring and modularisation
- Arbitrary file access
- Data structures
- Platform-independent programs
- ...

Part I

Bash Basics

Some Basic Commands

The first interactive example:

```
cd
mkdir bash_course
cd bash_course
touch hello.sh
chmod u+x hello.sh
<editor> hello.sh
```

File manipulation

- Files: touch, ls, rm, mv, cp
- Directories: cd, mkdir, rmdir, pwd
- Access with chmod: read, write and execute
- Editors: vi, emacs, gedit, nedit, ...
- Documentation: man, info, apropos, --help

Variables

```
#!/bin/bash
STR="HelloWorld!"
echo $STR
```

- NO WHITESPACE in assignments
- STR: name of the variable used for assignment
- \$STR: reference to the value
- \$STR is a short form of \${STR}

Quoting

```
#!/bin/bash
STR="HelloUUUUWorld!"
echo $STR
echo "$STR"
echo '$STR'
```

Operations

```
a=3+5 # does not work
a=`expr 3 + 5` # does work
a=`expr 3+5` # does not work
let "a=3+5" # does work
let "a=3+5" # does work
a=$((3+5)) # does work
((a++)) # does work; result?
a=${3+5} # does work
```

- No direct mathematical operations (everything is a string)
- 'command' is used to call a program
- expr: evaluate expressions
- Only integer operations possible
- Operators for expr: comparison (<, ...) and arithmetics (+, -, *, /)
- Operators for let: comp., arith., +=, ..., bitwise and logical

Arrays

```
#!/bin/bash
arr[0]=1
arr[1]=$((${arr[0]}*2))
arr[2]=$[${arr[1]}*2]
arr[5]=32

echo ${arr[0]}
echo ${arr[1]}
echo ${arr[2]}
echo ${arr[3]}
echo ${arr[4]}
echo ${arr[5]}
echo ${#arr[*]}
echo ${arr[*]}
```

T. Neckel: Scripting with Bash
Compact Course @ Max-Planck, March 19 – 22, 2019



TUM

Special Variables

within a script

- \$? Exit status variable
- \$\$ process ID
- \$0, \$1, \$2, ... command line parameters
- \$* all command line parameters (single string)
- @\$ all command line parameters (one string per parameter)
- \$# number of command line parameters
- \${#(variable)} string length of the variable value
- ...
- shift n shift all command line parameters to the left by n (first n parameters are lost!)

Environment Variables

```
env
```

T. Neckel: Scripting with Bash
Compact Course @ Max-Planck, March 19 – 22, 2019



TUM

Control Structures – if

```
if [ 1 = 1 ]
then
  echo "Hello World!"
fi
```

- Usually (most languages), something is done if test is true
- Usually, 0 is false, non-zero values are true
- In bash, test gives a return value
- Return value 0 means no error

```
test 1 = 2 # equals [ 1 = 2 ]
test 1 = 1
```

T. Neckel: Scripting with Bash
Compact Course @ Max-Planck, March 19 – 22, 2019



TUM

Control Structures – test

- Usage: test EXPRESSION or [EXPRESSION]
- Exits with the status determined by the given expression
- And: EXPRESSION1 -a EXPRESSION2
- Or: EXPRESSION1 -o EXPRESSION2
- Negation: ! EXPRESSION
- String comparison: =, != (test 1 = 1 is string comparison!)
- Integer comparison: -eq, -ge, -gt, -le, -lt, -ne
- File comparison: -ef, -nt, -ot
- File test with single operand: -e, -d, ...
- More details: man test

```
test 1 -eq 1
[ 1 -eq 1 ]
[[ 1 -eq 2 || 1 -eq 1 ]]
```

T. Neckel: Scripting with Bash
Compact Course @ Max-Planck, March 19 – 22, 2019



TUM

First Example

```
#!/bin/bash
rand=${RANDOM%2}
if [ $# -eq 0 ]
then
  echo "Usage: $0 <0_or_1>"
else
  if [[ $1 -ne 0 && $1 -ne 1 ]]
  then
    echo "parameter has to be 0 or 1"
  elif [ $1 -eq $rand ]
  then
    echo "won"
  else
    echo "lost"
  fi
fi
```

T. Neckel: Scripting with Bash
Compact Course @ Max-Planck, March 19 – 22, 2019



TUM

Control Structures – loops

```
#!/bin/bash
for i in <list>
do
  <commands>
done
```

- List can be any possible list of elements
- seq -s <s> <x> produces a list of numbers from 1 to x with separator s (jot - 1 x on MAC)
- for i in {1..x} does the same (no variable expansion!)
- break/break (n): stop the loop (n levels of nested loops)
- continue/continue (n): continue with the next iteration
- Other loops: while [condition]; do command; done
- Other loops: until [condition]; do command; done

T. Neckel: Scripting with Bash
Compact Course @ Max-Planck, March 19 – 22, 2019



TUM

Functions

```
#!/bin/bash
function function_name {
  command
}

function_name () {
  command
}
```

- Both syntactic variants do the same
- The round brackets are NOT used for parameters
- Functions have to be defined before they are used
- Functions may not be empty (use :)
- Parameter passing to functions equal to programs

T. Neckel: Scripting with Bash
Compact Course @ Max-Planck, March 19 – 22, 2019



TUM

Functions with Parameters

```
function min {
  if [ $1 -lt $2 ]
  then
    return $1
  else
    return $2
  fi
}

a=`min 4 6`
echo $a # does not work
min 4 6
a=$? # works
min 500 600
a=$? # does not work! why?
```

T. Neckel: Scripting with Bash
Compact Course @ Max-Planck, March 19 – 22, 2019



TUM

Functions with Parameters (2)

```
function min {
  if [ $1 -lt $2 ]
  then
    echo $1
  else
    echo $2
  fi
}

a=$(min 4 6) # equals a=`min 4 6`
echo $a
```

Command Substitution

Allows output of a command to replace command itself

```
$(command)
```

```
`command`
```

- Both perform expansion by executing command
- Replacing command with standard output of command, trailing newlines deleted
- Not return code!
- May be nested: escape inner backquotes with backslashes

Control Structures – case and select

```
#!/bin/bash
echo "Hit a key, then hit return."
read Key
case "$Key" in
  [[:lower:]] ) echo "Lowercase letter";;
  [[:upper:]] ) echo "Uppercase letter";;
  [0-9] ) echo "Digit";;
  * ) echo "something else";;
esac
```

```
#!/bin/bash
PS3='Choose your favorite language: '
select language in "bash" "python" "brainfuck" "C++"
do
  echo "Your favorite language is $language."
  break
done
```

Partial List of Commands/Variables/... so far

man	mkdir	bla	\$?
\$STR	continue	rmdir	clear
#!	shell builtin	\$RANDOM	break
let	emacs	ls	\$#
\$3	touch	echo "\$STR"	\${#a}
\$a -le \$b	chgrp	if	a = 3
mv	a=\$((3+5))	expr	test
\$\$	((a++))	cp	pwd
script suffix	cd	echo	for
chmod	\$*	\$PS3	
esac	\$@	u+x	
\${STR:-"x"}	echo '\$STR'	info	
apropos	rm	vi	
read	-- help	a=\${3+5}	

Part II

Shell Config

Special Directories

- . current directory
- .. parent directory
- ~ own home directory
- ~user home directory of user
- - previous directory

```
cd /usr/local/bin
pwd
cd ~
pwd
cd -
pwd
cd ..
pwd
```

Aliases and Variables

alias - short form for any command

```
alias
alias ll='ls -l'
```

environment variables

- Variables are not only within scripts, but also in the shell
- By setting a variable, it is present in the current script/shell
- By exporting it (export VARIABLE=value), it is also present at all child processes (NOT at the parent)
- Many environment variables are already set by default

```
PATH=/home/neckel/bin/:$PATH; echo $PATH
env
```

local variables

- set shows all local and global variables and functions
- unset deletes a variable

Bash config

profile and rc

- especially aliases are used in every session
⇒ should not be defined each time
- You might even run arbitrary code on login or logout (e.g., clean up (kill jobs) on logout)
- profile is executed for login shells (everything with password or keys)
- bashrc is executed for interactive non-login shells (e.g., new terminal window)

Files

- /etc/profile
- /etc/bash.bashrc
- \$HOME/.bash_profile
- \$HOME/.bashrc
- \$HOME/.bash_logout

- `stdin/stdout`: Standard input/output for programs
- Most Linux programs read from `stdin` and write to `stdout`
- Pipes are used to redirect input and output
- `cmd1 | cmd2` connect the output of `cmd1` with the input of `cmd2`
- `cmd > file` redirect the output of `cmd` to file
- `cmd 2> file` redirect `stderr` of `cmd` to file
- `cmd > file 2>&1` redirect `stdout` to file and `stderr` to `stdout`
- `cmd &> file` redirect all output to file
- `cmd >> file` redirect the output of `cmd` and append it to file
- `cmd < file` use the content of file as `stdin` for `cmd`
- `cmd <<Endmark` Read from `stdin` until `Endmark` is inserted in a separate line
- `;` command separator

- Wildcards are expanded by the shell
- `*` – zero or more characters
- `?` – exactly one character
- `[abcd]` – one of the characters `a-d`
- `[a-d]` – same
- `[!a-d]` – any other character
- `{first,second}` – either `first` OR `second`
- Similar patterns exist in other contexts as well (compare regular expressions), but always a bit different. . . :-(

Part III

Bash Advanced: Regular Expressions and More

Regular Expressions???

The set of regular languages over an alphabet Σ and the corresponding regular expressions are defined recursively as follows:

- The empty language \emptyset is a regular language, and the corresponding regular expression is \emptyset .
- The empty string $\{\wedge\}$ is a regular language, and the corresponding regular expression is \wedge .
- For each a in Σ , the singleton language $\{a\}$ is a regular language, and the corresponding regular expression is a .
- If A and B are regular languages, and $r1$ and $r2$ are the corresponding regular expressions,
 - Then $A \cup B$ (union) is a regular language, and the corresponding regular expression is $(r1+r2)$
 - AB (concatenation) is a regular language, and the corresponding regular expression is $(r1r2)$
 - A^* (Kleene star) is a regular language, and the corresponding regular expression is $(r1^*)$

Regular Expressions!!!

What does all this mean to you, as a user? Absolutely nothing. As a user, you don't care if it's regular, nonregular, unregular, irregular, or incontinent. So long as you know what you can expect from it, you know all you need to care about.

Jeffrey Friedl, author of *Mastering Regular Expressions*

Did you ever...

- ... search for a character or string in a text file?
- ... use tab for auto-completion?
- ... use the `*` in a terminal for selecting a group of files?
- ...

Then you've already somehow used regular expressions.

Regular Expressions

- When searching for a string, exactly the given character sequence is searched
- Regular expressions: a powerful tool to find patterns
- Additionally to ordinary characters, which stand for themselves special characters are used which are interpreted in a special way.
- The exact syntax for special characters differs between different implementations
- Example to identify an email address:
`[^@]\+@.\+\.\ [^.] \+`
- regular expressions usually look very cryptic!
- But imagine you would have to write a normal program to identify an email address!
- Regular expressions can be used to find/replace groups of strings which can be described by a pattern

Special Characters for Regular Expressions

- `char` – A character matching itself
- `*` matches zero or more occurrences (greedy!) of the previous expression
- `\+` matches one or more occurrences (GNU extension)
- `\?` matches zero or one occurrence (GNU extension)
- `\{i\}` matches `i` occurrences
- `\{i,\}` matches `i` or more occurrences
- `\{i,j\}` matches `i` to `j` occurrences
- `.` matches an arbitrary character
- `^` matches the beginning of a string
- `$` matches the end of a string
- `[list]` matches a single character from the list (`[~list]` any character not in the list)
- `\n` matches newline
- `\(\)` defines a group, reuse via `\1` (1st group), `\2` (2nd group)
- `\|` allows for a logical OR

Character Classes

- `[:digit:]` 0 to 9 (alternative: `[0-9]`).
- `[:alnum:]` alphanumeric character 0-9 or A-Z or a-z.
- `[:alpha:]` character A-Z or a-z.
- `[:xdigit:]` Hexadecimal notation 0-9, A-F, a-f.
- `[:punct:]` Punctuation symbols, e.g. `. , " ' ? ! ; : # $ % & ()`
- `[:print:]` Any printable character.
- `[:space:]` whitespace (space, tab, ...).
- `[:upper:]` uppercase character A-Z (alternative: `[A-Z]`).
- `[:lower:]` lowercase character a-z (alternative: `[a-z]`).

Attention: Additional brackets are necessary! `[:digit:]` \equiv `[0 - 9]`

programs using regular expressions

- `grep` print lines matching a pattern
- `grep -i` case insensitive
- `grep -v` invert match
- `grep -n` additionally print line number
- `grep -r` work recursively
- `tr` translate: `echo "lower/UPPER" | tr "A-Z" "a-z"`
- `sed` stream editor
- `awk` pattern scanning and processing language
- `find` search for files in a directory hierarchy
- ...
- most editors can handle regular expressions

Operating on Files

less is more than more

- `more` displays the content of text files pagewise
- only downward-scrolling is possible
- `less` is an extended and more flexible `more`

outputting files

- `cat` outputs the content of a file to stdout (try `tac`)
- `cat` concatenates several files
- `head` outputs the first part of files
- `tail` outputs the last part of files

```
for i in `seq 30`; do echo $i >> temp.txt; sleep 1; \
done &
tail -f temp.txt
```

Operating on Files (2)

- `file` find out more about file type
- `cmp` Compare files byte by byte
- `diff` Compare files line by line (graphical: `kdiff3`)
- `tar` pack and compress files (try `tar -xzf` and `tar -czf`)
- `sort` sort lines of text files (and write to stdout)
- `wc` print newline, word, and byte counts for each file
- `cut` print selected parts of lines from each file

```
echo "1;2;3;4" | cut -b 4-5
echo "1;2;3;4" | cut -d ";" -f 3
cut -d ' ' -f 1 cut.txt
```

find

- Tool to retrieve recursively(!) files and directories depending on search criteria
- Perform operations on the files found
- In connection with other bash tools perform tasks such as
 - Find all files not older than two days
 - Find all `.txt` files containing the phrase "Hello World!"
 - Find all `.bak` files and remove them
- General syntax:


```
find [path...] [expression]
```

 An expression can be
 - Options, such as `--maxdepth n`
 - Tests, such as `-amin n` (last access less than `n` minutes ago)
 - Actions, such as `-delete`
 If path is not specified, assume `./`

find (2)

```
wget http://www5.in.tum.de/lehre/vorlesungen/\
progcourse/bashpython/ss19/Bash/story.tar.gz
tar -xzf story.tar.gz
```

Examples

```
#!/bin/bash
find -maxdepth 2 -name "*.txt"
find ./ -amin -500 -iname "*.TXT"
find -atime +1 -name "*.txt"
find -path "*folder1*"
find -size +400c -name "file*.*" -ls
find -name "*.txt" -printf "%h,%f\n"
find -name "file??.txt" -exec grep -H "Bilbo" {} \;
find -regex '.*[0-9][0-4].txt'
find -name "*.txt" -delete # take care!!
```

Executing Commands

- For executing a single command, it first has to be found!
- Either path is provided, or command is in `$PATH`
- `which`, `whereis` and `type`: find out about program type and path
- For commands with more than one line:

```
echo "This command prints a lot of useless text \
which does not fit into one line"
```

multiple commands

- One way to execute several commands: connecting IO via pipes
- `cmd1; cmd2; ...` executing several commands
- `cmd1 && cmd2 && ...` execute following commands if previous successful
- `cmd1 || cmd2 || ...` execute following commands if previous fails

Part IV

Working Remotely And More

Working remotely - ssh, scp, screen and unison

ping

- `ping hostname`
- Useful to find out whether some machine is down or has no connection.

ssh and scp - security by encryption

- Developed 1995 to allow secure connections between computers
- `ssh username@computername` log in onto `computername` as `username`
- `scp -r * username@computername:<path_rel_to_home>`
- `scp *.txt username@computername:<absolute_path>`
- `scp "username@computername:path/*.*" ./` – copy all remote files in folder `path` to current directory
 - Quotes necessary for remote wildcards

Synchronising data – Unison

- very nice to sync 2 locations
- GUI and backup support
- 2-way sync (including merge etc.)
- supports Windows and Linux
- needs gtk and ssh
- no active development any longer
- download and infos:
<http://www.cis.upenn.edu/~bcspierce/unison/>

Management of processes

- `nice -n <value> <command>` starts command with lower priority
- `renice <value> -p <processID>` sets priority to new value
- `top` shows processes with top computing time
- within `top`, `r` can renice a process
- `nohup` run a command immune to hangups
- `ps -ef` list all processes
- `pidof` find out the process id for a program name
- `kill <ID>` kills process with the given Id (-9 to force)
- `killall <name>` kills processes with given name
- `bg` and `fg` send programmes to background/foreground

Working remotely - screen

- during a typical session, many terminals are used
- if working remotely ⇒ opening many ssh connections
- for each login/logout, everything has to be closed/opened again
- `screen` solves all these problems by creating virtual terminals
 - `screen` creates a new terminal
 - `Ctrl-a d` Of `screen -d` detach the screen
 - `Ctrl-a D D` detach the screen and log out
 - `screen -r` reattach the screen
 - `Ctrl-a A` give a name to a terminal
 - `screen -t <title>` creates a new terminal with specified title
 - `Ctrl-a w` show the currently opened terminals
 - `Ctrl-a k` kill a terminal

Some more commands you might want to know

```
basename "this/is/the/path/to/a.file"
dirname "this/is/the/path/to/a.file"
time sleep 3
time for i in `seq 100000`; do a=${3*$i}; done
users
groups
du -sh
df -h
ln -s <file> <link>
```