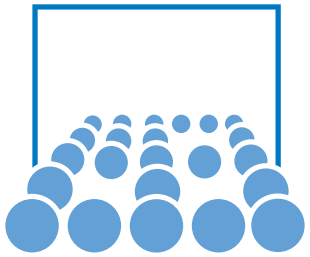
Scripting with Bash

Compact Course @ Max-Planck

Tobias Neckel & Severin Reiz

March 19 – 22, 2019



Shell vs. Python

Shell (Bash)

- Seperate 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
- ...

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
```

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

Hello World!

The first non-interactive example (hello.sh):

```
#!/bin/bash

echo "Hello World!"
```

- Convention: shell script suffix `.sh`
- #! sha-bang (#: sharp, !: bang)
- Sha-bang is followed by interpreter (try `#!/bin/rm`)
- echo is a shell builtin
- ”Hello World!” is (as almost everything) a string
- hello.sh has to be executable (chmod)

Variables

```
#!/bin/bash
STR="Hello World!"
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}`

Variables

```
#!/bin/bash
STR="Hello␣World!"
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="Hello␣␣␣␣␣World!"
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[*]}
```

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!)

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
```

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
```

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 ]]
```

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
```

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

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

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?
```

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 perfom 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
```

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]	