Scripting with Bash and Python

Compact Course @ Max-Planck

Tobias Neckel

February 16 - 26, 2015
General Information

Organization

• About 1.5 hours lecture
  • Slides and live demonstrations
• About 1.5 hours tutorials
  • We suggest tasks to practise
  • You can work on whatever you want
  • I’ll try to answer any questions
• Breaks :-)}
General Information

Organization

• About 1.5 hours lecture
  • Slides and live demonstrations
• About 1.5 hours tutorials
  • We suggest tasks to practise
  • You can work on whatever you want
  • I’ll try to answer any questions
• Breaks :-)

Outline

• 4 days Bash
• 5 days Python
• One day off in second week
General Information

About this course

• Scripting in Bash and Python (and beyond)
• Your background?
• Prior knowledge?
• Feedback!!
General Information

About this course

• Scripting in Bash and Python (and beyond)
• Your background?
• Prior knowledge?
• Feedback!!

More information

• Slides and tutorials on website:
Contents

Bash (ca. 4 days):

- Overview, most important basics:
  - Files and directories: ls, dir, cd, mkdir, find, touch, ...
  - Variables, manipulation of variables, arrays, special variables
  - Control structures, functions, parameters

- Shell configuration
  - Aliases and variables
  - Configuration examples for bash (bashrc)
  - Pipes and wildcards
Bash (ca. 4 days):

- Advanced topics
  - Regular expressions
  - String-manipulation, regular expressions: grep, cut, sed, head, ...
- Operating on files
- Grouping, subshells
- awk
- Working remote
  - ssh, scp, rsync, unison, nx ...
- Process management
- Visualization with gnuplot
  - Parsing, preprocessing data
  - Interaction with shell tools
Contents (cont.)

Scripting with Python . . . and beyond (ca. 5 days)

- Overview
  - Interpreted vs. compiled programming languages
  - Python command shell vs. executable Python programs, IPython
  - Where to use it and where not
- Data types and control structures
  - Objects, type declarations
  - Numeric types, sequences (list, tuple, dict, strings), . . .
  - Control structures: if, for, break, while, . . .
  - Arithmetic operations
- Functions and parameters
- String handling: strip, trim, split, ...
- File I/O
- Objects, classes, modules, and packages
Scripting with Python ... and beyond (ca. 5 days)

- Object oriented programming
  - A short introduction to OOP
  - OOP in Python
- Common and useful libraries
  - os, sys
  - Regular expressions: re
  - Command-line options: optparse
  - math, random
  - GUIs with tkinter
  - ...
- Scientific Computing in Python
  - NumPy (vectors, matrices, ...)
  - Matplotlib
  - Calling gnuplot
Requirements

Bash

• Bash as shell (yes, tcsh works, too, but slightly different)
• Editor of your choice (vim, emacs, . . .)
• Collection of standard tools
• gnuplot
• If you are using Windows, we recommend to install cygwin full/extended
Requirements

Bash

- Bash as shell (yes, tcsh works, too, but slightly different)
- Editor of your choice (vim, emacs, ...)
- Collection of standard tools
- gnuplot
- If you are using Windows, we recommend to install cygwin full/extended

Python

- Python 2.7.x, as not all packages have been ported to Python 3.X
- Packages:
  - Scipy (which should include IPython as interactive Python shell; if not, get IPython)
  - PyLab/Matplotlib
- Gnuplot
Part I

Introduction
Some Types of Programming Languages

• Declarative programming
  • Functional (Lisp, Scheme, make, ...)
  • Logical (Prolog)
  • Other (Sql, ...)

• Imperative programming
  • Procedural (C, Fortran, Python, Pascal, ...)
  • Modular (Bash, ...)
  • Object-oriented (C++, Java, Python, ...)
Scripting vs. Compiled Languages

Scripting languages
- Bash, Python, Perl, postscript, matlab/octave, ...
- Interactive mode
- Few optimisations
- Easy to use
- No binary files (hardly useful for commercial software)

Compiled languages
- C, C++, Fortran, ...
- Efficient for computational expensive tasks
- Source code is compiled to binary code

Other
- Java
Shell vs. Python

**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
• ...
General Programming Rules

• Comments
General Programming Rules

- Comments
- Comments
- Comments
General Programming Rules

• Comments
• Comments
• Comments
General Programming Rules

• Comments
• Comments
• Comments
• Problem $\Rightarrow$ algorithm $\Rightarrow$ program
• Modular programming
• Generic where possible, specific where necessary
• ...
Part II

Bash Basics
Some Basic Commands

The first interactive example:

```bash
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):

```bash
#!/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 '
```

Tobias Neckel: Scripting with Bash and Python
Compact Course @ Max-Planck, February 16 - 26, 2015
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
#!/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

```bash
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

```bash
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

```bash
  test 1 -eq 1
  [ 1 -eq 1 ]
  [[ 1 -eq 2 || 1 -eq 1 ]]```
#!/bin/bash
# encryption program
clear
which $EDITOR
if [ $? != "0" ]
then
echo "Which editor would you like to use?"
read EDITOR
fi
$EDITOR plaintext  # gedit does not work! detached from console!
gpg -a --no-options -o cryption.gpg -c plaintext
shred -u plaintext
clear
cat cryption.gpg
shred -u cryption.gpg
exit
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

```bash
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`

Tobias Neckel: Scripting with Bash and Python
Command Substitution

Allows output of a command to replace command itself

```
$(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
#!/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 - case and select

```bash
#!/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

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
<th>Command</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>man</td>
<td>mkdir</td>
<td>bla</td>
<td>$?</td>
</tr>
<tr>
<td>$STR</td>
<td>continue</td>
<td>rmdir</td>
<td>clear</td>
</tr>
<tr>
<td>#!</td>
<td>shell builtin</td>
<td>$RANDOM</td>
<td>break</td>
</tr>
<tr>
<td>let</td>
<td>emacs</td>
<td>ls</td>
<td>#$</td>
</tr>
<tr>
<td>$3</td>
<td>touch</td>
<td>echo &quot;$STR&quot;</td>
<td>${#a}</td>
</tr>
<tr>
<td>$a -le $b</td>
<td>chgrp</td>
<td>if</td>
<td>a = 3</td>
</tr>
<tr>
<td>mv</td>
<td>a=$((3+5))</td>
<td>expr</td>
<td>test</td>
</tr>
<tr>
<td>$$</td>
<td>((a++))</td>
<td>cp</td>
<td>pwd</td>
</tr>
<tr>
<td>script suffix</td>
<td>cd</td>
<td>echo</td>
<td>for</td>
</tr>
<tr>
<td>chmod</td>
<td>$*</td>
<td>$PS3</td>
<td></td>
</tr>
<tr>
<td>esac</td>
<td>$@$</td>
<td>u+x</td>
<td></td>
</tr>
<tr>
<td>${STR:-&quot;x&quot;}</td>
<td>echo '$_STR'</td>
<td>info</td>
<td></td>
</tr>
<tr>
<td>apropos</td>
<td>rm</td>
<td>vi</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>- - help</td>
<td>a=$[3+5]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>