



5. Shell Scripts

5.1 Syntax

5.1.1 Data Structures: Constants and Variables

Bash

```
1 echo "What would you like to talk about?"
2 read issue
3 echo "Interesting, so you're keen to talk about $issue. Does your interest in this
   topic originate in your childhood?"
```

5.1.2 Control Structures

5.1.2.1 Conditions

`>` / `<` compares the *lexicographical* order of strings, so 11 would be sorted before 2. This is typically not what we want.

We would also have to escape `>` / `<`, either by prefixing them with `\` or by using double brackets (`[[a > b]]`). `[a > b]` would actually return the exit code of writing to a file named b.

5.1.2.2 Loops and Case Matching

Bash

```
1 count=10
2
3 i=1
4 result=1
5 current=1
6 prev=1
7 while [ $((i++)) -lt $count ]; do
8     echo $result
9     current=$result
10    result=$((result+prev))
11    prev=$current
12 done
```

Bash

```
1 declare -a order=(3 7 4 5 6 8)
2 declare -A shapeCorners=(
3     [3]=Triangle
4     [4]=Rectangle
5     [5]=Pentagon
6     [6]=Hexagon
7     [7]=Heptagon
8     [8]=Octagon
9 )
10 correct=0
11 for i in ${order[@]}; do
12     read -p "What is a regular shape with $i corners called? " response
13     if [ $response = ${shapeCorners[$i]} ]; then ((correct++)); fi
14 done
15 echo "Right: $correct, wrong: $(( ${#order[@]} - $correct ))"
```

Bash

```
1 declare -a order=(3 7 4 5 6 8)
2 correct=0
3 for i in ${order[@]}; do
4     read -p "What is a regular shape with $i corners called? " response
5     case $response in
6         "Triangle") if [ $i -eq 3 ]; then ((correct++)); fi ;;
7         "Rectangle") if [ $i -eq 4 ]; then ((correct++)); fi ;;
8         "Pentagon") if [ $i -eq 5 ]; then ((correct++)); fi ;;
9         "Hexagon") if [ $i -eq 6 ]; then ((correct++)); fi ;;
10        "Heptagon") if [ $i -eq 7 ]; then ((correct++)); fi ;;
11        "Octagon") if [ $i -eq 8 ]; then ((correct++)); fi ;;
12    esac
13 done
14 echo "Right: $correct, wrong: $(( ${#order[@]} - $correct ))"
```

This lists all files in a directory and prints their names. The `*` is treated specially by bash where a filename could be meant. It's called "filename expansion" and we'll look at it a bit later.

5.1.3 Program Structures: Methods

- `binary 10`
- It checks whether `$base` is empty or zero
- Because that's not the input the [program expects. We could alternatively use `[$base = "-h" -o $base = "--help"]`, though.

5.2 Language Features

5.2.1 Builtin Parameters

If there are e.g. two parameters and the first one consists of two words (separated by a space, which by default is the first character in `$IFS`) - e.g. `"foo bar" baz`, then based on `$@`, one won't be able to tell whether there were

- three parameters, each of them one word
- two parameters, with the first one containing two words
- two parameters, with the second one containing two words
- one parameter containing of three words

`"$@"`, on the other hand, will keep the two parameters separate (in quotes).

Bash

```
1 for i in $*; do echo $i; done
2 echo "___"
3 for i in "$*"; do echo $i; done
4 echo "___"
5 for i in $@; do echo $i; done
6 echo "___"
7 for i in "$@"; do echo $i; done
```

The output of the above is the following for arguments `"a b" c`:

Output

```
a
b
c
___
a b c
___
a
b
c
___
a b
c
```

5.2.2 Expansions

5.2.2.1 Parameters

Bash

```
1 foo() {  
2   read -t 5 -p "Input: " x  
3   echo "Continuing with ${x:-my work}"  
4 }
```

5.2.2.2 Output

Bash

```
1 x=$(foo) && echo ${x:16}
```

5.2.2.3 Curly Braces

1. `mkdir -p Appendix\ {A,B,C,D}/Page\ {1..4}`
2. `rm -r Appendix\ {B,C}/Page\ 4`

Bash

```
1 for i in Appendix\ ?/Page\ {2,4}; do cat <<EOF > "$i/docker-compose.yml"  
2   version: '3.7'  
3   services:  
4     hello_world:  
5       image: alpine  
6       command: [/bin/echo, 'Hello world']  
7 EOF  
8 done;
```

5.2.3 Commands: Built-In & Chaining

Bash

```
1 for i in */; do  
2   pushd "$i";  
3   for j in */; do  
4     pushd "$j";  
5     test -f docker-compose.yml && docker-compose up;  
6     popd;  
7   done;  
8   popd;  
9 done;
```



6. Unix Tools

6.1 Information Gathering

6.1.1 find

Bash

```
1 find / -type f -mmin -60
```

Bash

```
1 find / -type f -mmin -60 2> /dev/null
```

Bash

```
1 find ~ -name "*.log"
```

6.2 Content

6.2.1 cat

Bash

```
1 cat <(ls /) <(ls ~)
```

6.2.2 tail

Bash

```
1 find / -type f -mmin -60 2> /dev/null > ./finds.txt &  
2 tail -f ./finds.txt
```

6.2.3 column

Bash

```
1 column -t -s , test.csv
```

6.2.4 wc

Both commands show the number of lines output by `ls`: the number of files in the current directory.

Additionally, `wc -l <(ls)` displays information about the file it reads, which is the file descriptor created by the output redirection, which temporarily holds the content produced by `ls`. On the other hand, `ls | wc -l` doesn't act on a file, so there is no file information to show.

6.3 Text Manipulation

6.3.1 sort

Bash

```
1 sort -u <(ls /) <(ls ~)
```

6.3.2 grep

Bash

```
1 find / -type f -mmin -60 2>&1 | grep -v -e "Permission denied" -e "Operation not permitted"
```

Note that we have to redirect `stderr` into `stdout` since we're dealing with error messages. Alternatively, we can use `|&` to pipe both `stdout` and `stderr` into `grep`.

6.3.3 cut

Bash

```
1 cut -d : -f 1,3 /etc/passwd
```

Bash

```
1 rev file.txt | cut -c 1-4 | rev
```

6.3.4 awk

Bash

```
1 ps | awk '/bash/ { print $1 }' | xargs kill -9
```

—

Bash

```
1 awk 'BEGIN{FS=":"}{ print $3 ": " $1}' /etc/passwd
```

Bash

```
1 awk 'BEGIN{print "<html>\n  <body>\n"}{ print "    <a href=\"mailto:\"$1\">\"$1\"</a><br />"}END{print "\n  </body></html>"}' email.lst > email.html
```

6.3.5 sed

Bash

```
1 sed -n '/^a.*z$/p' /usr/share/dict/words
```

6.4 Miscellaneous

6.4.1 seq

Bash

```
1 seq 3 3 100 | grep -v $(seq 7 7 100 | xargs -I % echo "-e %")
```

Bash

```
1 for i in $(seq 3 3 100); do test $((i % 7)) -gt 0 && echo $i; done;
```

6.4.2 curl

Bash

```
1 curl -o /dev/null --silent --ipv4 --head --write-out '%{remote_ip}:%{remote_port} ->
  %{http_code}\n' https://www.ost.ch
```