Shell Scripting Exercises

Shell Scripting in 8 Hours

Shell Scripting

In 8 Hours

For Beginners
Learn Coding Fast

Ray Yao

Shell Scriptin
Shell Scriptin
Sh

Shell Scripting

In 8 Hours

For Beginners
Learn Coding Fast

Ray Yao

About the Author: Ray Yao

Certified PHP engineer by Zend, USA

Certified JAVA programmer by Sun, USA

Certified SCWCD developer by Oracle, USA

Certified A+ professional by CompTIA, USA

Certified ASP**.** NET expert by Microsoft, USA

Certified MCP professional by Microsoft, USA

Certified TECHNOLOGY specialist by Microsoft, USA

Certified NETWORK+ professional by CompTIA, USA

www **.** amazon **.** com/author/ray-yao

Prefac e

“Linux Shell Scripting in 8 Hours & Exercises” covers all essential Linux Shell language knowledge. You can learn complete primary skills of Linux Shell Scripting fast and easily.
The book includes more than 60 practical examples for beginners and includes exercises for the college exam, the engineer certification exam, and the job interview exam.

Note:

This book is only for beginners, it is not suitable for experienced programmers.

Table of Content

Recommended Books

Hour 1

What is Linux Shell?

The shell is the interface between the user and Linux; every command you type at the prompt is interpreted by the shell and passed to the Linux kernel.

The shell is a command-language interpreter with its own built-in shell command collection. In addition, the shell can be called by any other valid Linux utilities and application programs in the system.

Whenever you type a command, it is interpreted by the Linux shell.

For example:

The command of printing current working directory (PWD) is contained within Linux bash.

The copy commands (cp) and the move commands (rm) are two single programs that exist in a directory on the file system.

The shell first checks to see if the command is an internal command, and then checks to see if it is an application, either of Linux's own utilities, such as ls and rm, or of a purchased commercial program, such as xv and ghostview.

Another important feature of the shell is that it is an interpreted programming language that supports most data structures, such as loops, functions, variables, and arrays.

Shell Scripting

A Shell script is a scripting program written for the Shell**.**

Shell Working Environment

Shell scripting, like Java, C++ programming, requires only a text editor to write code and a script interpreter to interpret execution

Various Linux Shell

Linux has different types of shells, such as**:**

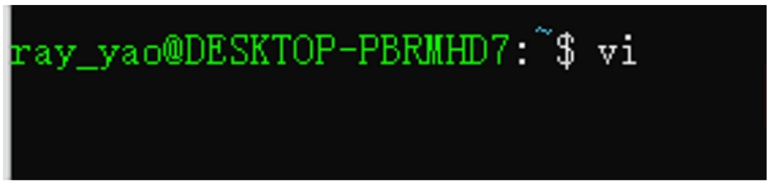
- Bourne Shell (/bin/sh)
- Bourne Again Shell (/bin/bash)
- C Shell (/usr/bin/csh)
- K Shell (/usr/bin/ksh)
- Shell for Root (/sbin/sh)

This book talks about Bash (Bourne Again Sell)**.** Bash is the default Shell on most Linux systems**.**

The First Shell Scripting

Example 1.1

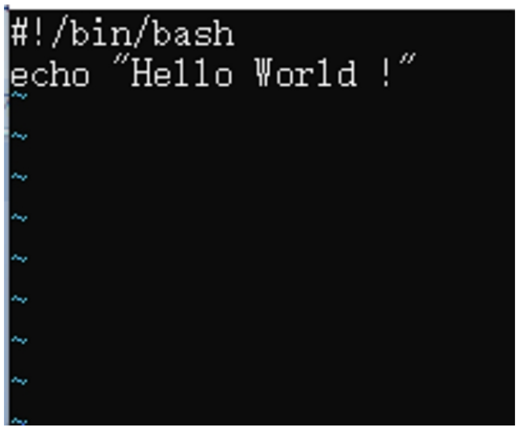
1. Open the Vi text editor with a "**vi** " command.



```
ray_yao@DESKTOP-PBRMHD7:~$ vi
```

2. Press "**i** " key, enter the Insert Mode.
3. Type the following code in the Vi editor:

```
#! /bin/bash
echo "Hello World ! "
```



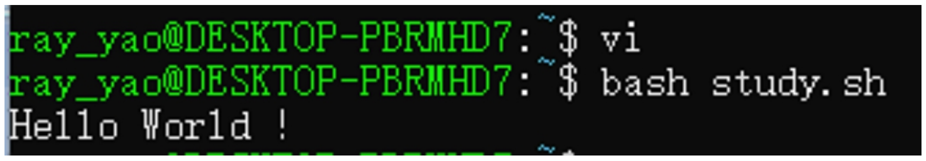
```
#!/bin/bash
echo "Hello World !"
~
~
~
~
~
~
~
~
~
```

4. Press the "**Esc** " key to quit the Insert Mode.
5. Type "**:wq study.sh** " to save the file. The file name is "study. sh", the extension name is ". sh".
6. Run this Shell program with the following commands:

bash study.sh

Output:

Hello World!



```
ray_yao@DESKTOP-PBRMHD7:~$ vi
ray_yao@DESKTOP-PBRMHD7:~$ bash study.sh
Hello World !
```

Explanation:

"vi" is a command to open the Vi editor.

"i" key is used to enter the Insert Mode in Vi editor.

Insert Mode can be used for writing and editing Shell Scripts.

"#! /bin/bash" let Shell system know that using Bash to interpret the scripting.

"echo "Hello World ! "" shows the content "Hello World! "

"Esc" key is used to quit the Insert Mode in Vi editor.

": wq file_name" saves the Shell file, and quit from Vi.

"bash study. sh" executes the Shell file "study. sh".

"bash myFile.sh" executes the Shell file "myFile.sh".

Shell Comment

```
#
```

"#" symbol is used for the Shell comments.

Shell interpreter always ignores the Shell comments

Example 1.2

```
#! /bin/bash
echo "Hello World! "     # output Hello World!
echo "Hi, Friends! "     # output Hi, Friends!
```

Output:

```
Hello World!
Hi, Friends!
```

Explanation:

"# output Hello World ! " is a comment of Shell scripting.

"# output Hi, Friends ! " is a comment of Shell scripting.

Multi-Line Comments

```
: <<EOF
comment 1
comment 2
comment 3
...
EOF
```

"**:** <<EOF……EOF" is used for multi-line comments.

Example 1.3

```
#! /bin/bash
echo "Hello World! "
:<<EOF
This is a "Hello World" program.
This is a Shell scripting.
This is a sample of multi-line comment.
......
EOF
echo "This is a sample of multi-line comment. "
```

Output:

```
Hello World!
This is a sample of multi-line comment.
```

Explanation:

"**:<<EOF……EOF** " is used for multi-line comments.

Shell Variables

1. The syntax to define a variable is as follows**:**

```
var=“value”
```

Note**:** There must be no Spaces before and after the equal sign**!**

2. To use a variable, you need to use “$” before a variable**.**

```
$var
```

Example 1.4

```
#! /bin/bash
book ="Scala in 8 Hours"      #  define a variable
echo  $book               # use the variable
```

Output:

```
Scala in 8 Hours
```

Explanation:

When defining a variable, make sure that no spaces before or after the equal sign**.**

Variable Name

The rule of the variable name is as follows:

- Only English letters, numbers, and underscores can be used for variable naming.
- The first character of the variable name cannot begin with a number.
- There can be no spaces in the middle of a variable name.
- Variable names cannot be punctuated.
- Bash keywords cannot be used for variable names.

The Valid Variable Name:

_myVariable

myVariable001

my_Varialbe

The Invalid Variable Name:

001Variable

echo

my*Varialbe

Read-Only Variable

The “readonly” command can define a variable as a read-only variable, whose value cannot be changed.

```
readonly  variable
```

Example 1.5

```
#! /bin/bash
myVar="C# in 8 Hours"
readonly myVar
myVar="Scala in 8 Hours"       # change the variable value
```

Output: (an error message as follows:)

```
test. sh: line 4: myVar: readonly variable
```

Explanation:

“**readonly** myVar” sets myVar as read only.

The output is an error message.

After myVar becomes a read-only variable, any change in the value of myVar will cause an error.

Remove Variable

The “unset” command can delete variables**.**

```
unset variable
```

Example 1.6

```
#! /bin/sh
myVar="C++ in 8 Hours"
unset myVar       # remove a variable
echo $myVar
```

Output:

(nothing)

Explanation:

After a variable is removed, the variable no longer exists, and its value no longer exists**.** So you don't see any output**.**

Input to Variable

The value of a variable can be input from keyboard by a user.

```
read  variable
```

Example 1.7

```
#! /bin/sh
echo "Please input your name. "
read myInput        # the input value will be stored in myInput
echo "Hi! Your name is: $myInput"
# Assume that I will input my name :   Ray Yao
```

Output:

```
Please input your name.
Ray Yao
Hi! Your name is Ray Yao
```

Explanation:

Assume that I input my name: Ray Yao

"**read** myInput" stores the input value to "myInput"

"echo "Hi! Your name is: $myInput"" shows the input value.

Hour 2

Shell String (1)

The string is enclosed in single quotes.

```
str='This is a string'
```

- Any characters in the single quotes will be printed in its original format.
- The string variables in the single quotes are invalid.

Example 2.1

```
#! /bin/sh
var=Andy
str='Hello, $var'        # single-quote string
echo $str
```

Output:

```
Hello, $var
```

Explanation:

'Hello, $var' is a single-quote string. Any character in the single quotes will be printed in its original format.

Shell String (2)

The string is enclosed in double quotes.

```
str="This is a string"
```

- Any characters in the double quotes will be printed by its returned value.
- The string variables in the double quotes are valid.

Example 2.2

```
#! /bin/sh
var=Andy
str="Hello, $var"        # double-quote string
echo $str
```

Output:

```
Hello, Andy
```

Explanation:

'Hello, $var' is a double-quote string. Any character in the double quotes will be printed by its returned value.

Connect Strings

We can use double quotes to connect two strings.

Example 2.3

```
#! /bin/sh
str1="Kotlin"
str2="in 8 Hours"
myString="$str1 $str2"     # connect two strings
echo $myString
```

Output:

```
Kotlin in 8 Hours
```

Explanation:

"$str1 $str2" connects two strings into one string by using double quotes.

String Length

We can get the length of the string by using “#” symbol**.**

```
#string
```

Example 2.4

```
#! /bin/sh
str="jQuery in 8 Hours"
echo ${#str }       # get the length of the string
```

Output:

```
17
```

Explanation:

“#str” can get the length of the string**.** The string length is 17 characters and spaces**.**

Extract Substring

We can extract a substring from a string.

```
string : start : length
```

"start": the extraction starts at this index.

"length": the extraction size

Example 2.5

```
#! /bin/sh
str="DjanGo in 8 Hours"
echo ${str:4:16 }       # extract the substring
```

Output:

```
Go in 8 Hours
```

Explanation:

"str: 4: 16" extracts a substring starting at the index 4, the extracting length is 16.

About `expr…`

` expr…` is an expression, which is used for calculation.

```
` expr…`
```

Note: ` is a back quote, rather than a single quote. The ` key is on the most left side of the keyboard.

Example 2.6

```
#! /bin/sh
x=100
y=200
z=`expr $x + $y`        # an expression
echo "The sum is $z"
```

Output:

```
The sum is 300
```

Expression:

`expr $x + $y` is an expression, the calculated result will be assigned to the z.

Note: ` is a back quote, rather than a single quote. The ` key is on the most left side of the keyboard.

Search Index

We can find the index of the first occurrence of a character in a string**.**

```
` expr  index “string”  character`
```

Note**:** ` is a back quote, rather than a single quote**.**

“index” is a parameter, for searching the index of a character**.**

“character” is a parameter, indicating the character you specify**.**

Example 2.7

```
str="Kotlin in 8 Hours"
echo `expr index "$str" n`
```

Output:

```
6
```

Explanation:

“ **`expr index "$str" n`** ” finds the index of the first occurrence of the “n” character in the string**.**

Note**:** ` is a back quote, rather than a single quote**.**

Shell Parameters

We can pass parameters to a Shell script as it executes.
The syntax of the parameter is as follows:

```
$n
```

“n” represents a number.
$0 represents the file name.
$1 represents the first parameter.
$2 represents the second parameter.
$3 represents the third parameter.
……

Example 2.8

```
#! /bin/bash
echo "My file name is:$0 ";
echo "$1 in 8 Hours";
echo "$2 in 8 Hours";
echo "$3 in 8 Hours";
```

Save the file named “OK. sh”
Run the file using the following command:

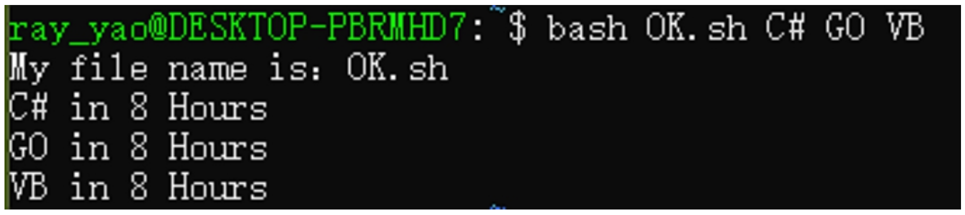
```
$ bash OK.sh  C#  GO  VB
```

Output:

```
My file name is: OK. sh
C# in 8 Hours
```

GO in 8 Hours

VB in 8 Hours



```
ray_yao@DESKTOP-PBRMHD7:~$ bash OK.sh C# GO VB
My file name is: OK.sh
C# in 8 Hours
GO in 8 Hours
VB in 8 Hours
```

Explanation:

"**bash OK.sh C# GO VB** " runs the file OK. sh, and pass four parameters to Shell script.

The zero parameter "OK. sh" passes to $0

The first parameter "C#" passes to $1.

The second parameter "GO" passes to $2.

The third parameter "VB" passes to $3.

Built-in Parameters

There are built-in parameters in Shell as follows**:**

Parameters	**Descriptions**
$0	Return the file name
$#	Return the total number of the parameters
$*	Return all parameters input by the user
$@	Return all parameters input by the user
$$	Return the current process ID number of the script
$ **!**	Return the last process ID number of the script
$?	Returning 0 indicates running successfully

Example 2.9

```
#! /bin/bash
echo "My file name is: $0 ";
echo "The total of the parameter is:$# ";
echo "All parameter input by the user is:$* ";
echo "The running result is:$? ";
```

Save the file named "yes. sh"

Run the file using the following command**:**

bash yes.sh R in 8 Hours

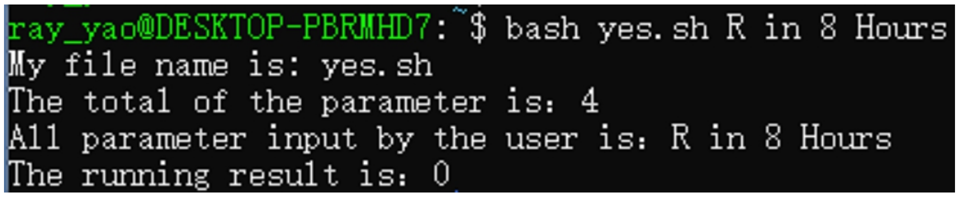
Output:

My file name is**:** yes**.** sh

The total of the parameter is**:** 4

All parameter input by the user is**:** R in 8 Hours

The running result is**:** 0



```
ray_yao@DESKTOP-PBRMHD7:~$ bash yes.sh R in 8 Hours
My file name is: yes.sh
The total of the parameter is: 4
All parameter input by the user is: R in 8 Hours
The running result is: 0
```

Explanation:

"**bash yes.sh R in 8 Hours** " runs the file yes**.** sh, and pass four parameters to Shell script**.**

$#   Return the total number of the parameters

$*   Return all parameters input by the user

$**?**   Returning 0 indicates the script running successfully, Returning other numbers indicates the script running failure**.**

Hour 3

Shell Array

An array is a particular variable, which can contain one or more values at a time.

Bash supports one-dimensional arrays, does not support multidimensional arrays, and does not limit the size of arrays.

1. The syntax to create an array:

```
array_name =
(value0  value1  value2…)
```

Example

myArray=(100 101 102 103)

Explanation:

The above code creates an array, the array name is "myArray", it has four elements, its index is 0, 1, 2, 3, its value is 100, 101, 102, 103. Note that index begins with zero.

2. Another syntax to create an array:

```
array_name[0]=value0
array_name[1]=value1
array_name[2]=value2
```

Example

```
color [0] = "red";
color [1] = "blue";
color [2] = "green";
```

Explanation:

Above code creates an array, the array name is “color”, and it has three elements: color [0], color [1], color [2]. Its indexes are 0, 1, and 2. Its values are red, blue, and green.

Note that index begins with zero.

Accessing the Elements

The syntax to access the elements of an array is as follows:

```
${array_name[index]}
```

Example 3.1

```
#! /bin/bash
my_array=(Rust in 8 Hours)       # create an array
echo "The first element is: ${my_array[0]} "
echo "The second element is: ${my_array[1]} "
echo "The third element is: ${my_array[2]} "
echo "The fourth element is: ${my_array[3]} "
```

Output:

```
The first element is: Rust
The second element is: in
The third element is: 8
The fourth element is: Hours
```

Explanation:

"my_array=(Rust in 8 Hours)" creates an array.

"${my_array[0]}" accesses the element at index 0.

"${my_array[1]}" accesses the element at index 1.

"${my_array[2]}" accesses the element at index 2.

"${my_array[3]}" accesses the element at index 3.

The value of my_array[0] is: Rust

The value of my_array[1] is: in

The value of my_array[2] is: 8

The value of my_array[3] is: Hours

Get All Elements

array_name[@] or array_name[*] can get all elements of an array.

```
array_name[@]
or
array_name[ * ]
```

Example 3.2

```
#! /bin/bash
my_array[0]=Java
my_array[1]=Ruby
my_array[2]=Html
my_array[3]=Rust
echo "All elements are: ${my_array[@] }"
echo "All elements are: ${my_array[*] }"
```

Output:

```
All elements are: Java  Ruby  Html  Rust
All elements are: Java  Ruby  Html  Rust
```

Explanation:

my_array[@] or my_array[*] can get all elements of the array.

Array Length

#array[@] or #array[*] can get the length of an array.

```
#array[@]
or
#array[ * ]
```

Example 3.3

```
#! /bin/bash
my_array=(PHP MySQL in 8 Hours)
length1=${#my_array[@] }
length2=${#my_array[*] }
echo $length1
echo $length2
```

Output:

```
5
5
```

Explanation:

#my_array[@] or #my_array[*] can get the length of the array.

Printf Command

“printf” is a command to format the output in the Shell.

```
printf  “type”  variables
```

“type” is a parameter used to format the output.

Typ e	Description
%b	Print it as a binary value
%d	Print it as a digital value
%f	Print it as a float-point value
%o	Print it as an octal value
%s	Print it as a string value
%x	Print it as a hexadecimal value

For example:

%10s means formatting as a string with a maximum width of 10 characters

%-10s means formatting as a string with a maximum width of 10 characters (- sign means left aligned, no - sign means right aligned)

%8. 2f means formatting as float-point value, 8 means 8 integers, 2 means 2 decimal places.

%-8. 2f means formatting as float-point value, 8 means 8 integers, 2 means 2 decimal places. (- sign means left aligned, no – sign means right aligned)

Example 3.4

```
printf "%-10s %-7s %-8s\n" Name Age Weight
```

Output:

```
Name      Age    Weight
```

Explanation:

"%-10s" is replaced by "Name", "Name" is formatted as a string with maximum 10 characters, left aligned.

"%-7s" is replaced by "Age", "Age" is formatted as a string with maximum 7 characters, left aligned.

"%-8s" is replaced by "Weight", "Weight" is formatted as a string with maximum 8 characters, left aligned.

"\n" is used to go to the next line

Example 3.5

```
#! /bin/bash
printf "%-10s %-7s %-8s\n" Name Age Weight
printf "%-10s %-7d %-5. 2f\n" Andy 18 91. 8843
printf "%-10s %-7d %-5. 2f\n" Tomy 19 95. 4675
printf "%-10s %-7d %-5. 2f\n" Rosy 17 80. 8563
```

Output:

```
Name      Age     Weight
Andy      18       91. 88
Tomy      19      95. 47
Rosy      17      80. 86
```

Explanation:

“printf "%-10s %-7d %-5. 2f\n" Andy 18 91. 8843” formatted three parameters “Andy”, “18”, “91. 8843”.

“%-10s” is replaced by “Andy”, “Andy” is formatted as a string with maximum 10 characters, left aligned.

“%-7d” is replaced by “18”, “18” is formatted as a digital type with maximum 7 numbers, left aligned.

“%-5. 2f” is replaced by “91. 88”, “91. 88” is formatted as a float-point type with maximum 5 integers, 2 decimal places, left aligned.

“\n” is used to go to the next line.

Escape Sequences

The " \ " backslash character can be used to escape characters.

\a outputs an alert

\n outputs the content to the next new line.

\r makes a return

\t makes a tab

\f outputs the content to the next page.

\' outputs a single quotation mark.

\" outputs a double quotation mark.

Example 3.6

```
#! /bin/bash
printf "%s %s \'%s\' \n" He said OK
printf "%d\t %d\n" 10 100
printf "%s\n %s %d %s\n"  C# in 8 Hours
```

Output:

```
He said 'OK'
10      100
C#
in 8 Hours
```

Explanation:

\'%s\' outputs the single quotation marks and a formatted string .

%d\t outputs a tap's space between two numbers

%s\n outputs following stings in the next line**.**

Hour 4

Operators

The Shell, like other programming languages, supports a variety of operators, including: Arithmetic operator, Relational operator, Boolean operator, String operator, File operator.

```
`expr…`
```

`expr…` is an expression evaluation tool. The ` key is a back quote, rather than a single quote.

Example 4.1

```
#! /bin/bash
sum=`expr 100 + 200`
echo "The sum is: $sum"
```

Output:

```
The sum is 300
```

Explanation:

The ` key is a back quote, rather than a single quote.

The complete expression must be included by the back quotes.

There must be a space between the expression and the operator. For example, "10+20" is wrong, "10 + 20" is correct.

if…then…else…fi

“if statement” executes code1 if a specified condition is true, executes code2 if the condition is false**.**

```
if condition
then
   # if true run this code1
else
    # if false run this code2
fi
```

Example 4.2

```
a=10
b=20
if [ $a == $b ]       # check if a is equal to b or not
then
  echo "a is equal to b"
else
  echo "a is not equal to b"
fi
```

Output:

```
a is not equal to b
```

Explanation:

[$a == $b] compares a and b, check if a is equal to b or not.

Note: About ==, it should be put between square brackets with spaces. For example: [$a==$b] is wrong, [$a == $b] is correct.

== is a comparison operator. If a is equal to b, return true. If a is not equal to b, return false.

In above "if statement", because the test condition returns false, the "else code2" is executed.

Therefore, the 'echo "a is not equal to b"' command is executed.

Arithmetical Operators

Operators	Running
+	add
-	subtract
*	multiply (note: *)
/	divide
%	get modulus
=	assignment
==	check equality
! =	check inequality

Example 4.3

```
#! /bin/bash
a=100
b=2

result=`expr $a + $b`
echo "a + b : $result"
result=`expr $a - $b`
echo "a - b : $result"
result=`expr $a \* $b`        # note \*
echo "a \* b : $result"
result=`expr $a / $b`
```

```
echo "a / b : $result"
result=`expr $a % $b`
echo "a % b : $result"
```

Output:

```
a + b : 102
a - b : 98
a \* b : 200
a / b : 50
a % b : 0
```

Explanation:

Operators	**Running**
+	add
-	subtract
*	multiply (note: *)
/	divide
%	get modulus

Example 4.4

```
a=100
b=200
if [ $a == $b ]       # if a is equal to b, return true
then
   echo "a is equal to b"
fi
if [ $a != $b ]       # if a is not equal to b, return true .
then
   echo "a is not equal to b"
fi
```

Output:

```
a is not equal to b
```

Explanation:

[$a **!** = $b] checks if a is not equal to b**.**

Note**:** There should be spaces between [ ] and **!** = **.**

[$a**!** =$b] is wrong**.** [$a **!** = $b] is correct**.**

Comparison Operators

Linux Shell has comparison operators as follows**:**

Operators	**Checks**
-eq	equal
-ne	not equal
-gt	greater than
-ge	greater or equal
-lt	less than
-le	less or equal

The comparison expression returns true or false**.**

Example 4.5

```
#! /bin/bash

a=100
b=200

if [ $a -eq $b ]        # check if a is equal to b
then
  echo "$a -eq $b ?   a is equal to b"
else
  echo "$a -eq $b ?   a is not equal to b"
fi
```

```
if [ $a -ne $b ]        # check if a is not equal to b
then
  echo "$a -ne $b ?   a is not equal to b"
else
  echo "$a -ne $b ?   a is equal to b"
fi

if [ $a -gt $b ]      # check if a is greater than b
then
  echo "$a -gt $b ?   a is greater than b"
else
  echo "$a -gt $b ?   a is not greater than"
fi

if [ $a -lt $b ]      # check if a is less to b
then
  echo "$a -lt $b ?   a is less than b"
else
  echo "$a -lt $b ?   a is not less than b"
fi

if [ $a -ge $b ]    # check if a is greater than or equal to b
then
  echo "$a -ge $b ?   a is greater than or equal to b"
else
  echo "$a -ge $b ?   a is less than b"
fi
```

```
if [ $a -le $b ]      # check if a is less than or equal to b
then
  echo "$a -le $b ?   a is less than or equal to b"
else
  echo "$a -le $b ?   a is greater than b"
fi
```

Output:

```
100 -eq 200 ?    a is not equal to b
100 -ne 200 ?    a is not equal to b
100 -gt 200 ?    a is not greater than
100 -lt 200 ?    a is less than b
100 -ge 200 ?    a is less than b
100 -le 200 ?    a is less than or equal to b
```

Explanation:

Operators	**Checks**
-eq	equal
-ne	not equal
-gt	greater than
-ge	greater or equal
-lt	less than
-le	less or equal

Boolean Operators

Operators	**Equivalent**
-a	and
-o	or
!	not

After using logical operators, the result will be true or false**.**

true -a true;
returns true;
	true -a false;
	returns false;
		false -a false;
		returns false;

true -o true;
returns true;
	true -o false;
	returns true;
		false -o false;
		return false;

! false;
returns true;
	! true;
	returns false;	

Example 4.6

```
#! /bin/bash

x=true
y=false

if [ $x -a $y  ]       # -a means “and”
then
   echo flase
fi
```

```
if [ $x -o $y  ]        # -o means "or"
then
   echo true
fi

if [ !$x ]        # ! means "not"
then
   echo flase
fi
```

Output:

```
flase
true
flase
```

Explanation:

true -a true;
returns true;
	true -a false;
	returns false;
		false -a false;
		returns false;

true -o true;
returns true;
	true -o false;
	returns true;
		false -o false;
		return false;

! false;
returns true;
	! true;
	returns false;	

Hour 5

String Operators

Operators	Checks
=	If two strings are equal, returns true .
! =	If two strings are not equal, returns true .
-z	If the length of a string is equal to zero, returns true .
-n	If the length of a string is not equal to zero, returns true
$	If the string is not empty, returns true .

Example 5.1

```
#! /bin/bash

a="Java"
b="Ruby"

if [ $a = $b ]      # check if a is equal to b
then
   echo "$a = $b ?  a is equal to b"
else
   echo "$a = $b ?  a isn't equal to b"
fi

if [ $a != $b ]      # check if a isn’t equal to b
then
```

```
   echo "$a ! = $b ?  a isn't equal to b"
else
   echo "$a ! = $b ?  a is equal to b"
fi

if [ -z $a ]      # check if the string length is 0
then
   echo "-z $a ?  The string length is 0"
else
   echo "-z $a ?  The string length isn't 0"
fi

if [ -n "$a" ]       # check if the string length isn’t 0
then
   echo "-n $a ?  The string length isn't 0"
else
   echo "-n $a ?  The string length is 0"
fi

if [ $a ]       # check if the string isn’t empty
then
   echo "$a ?  The string isn't empty"
else
   echo "$a ?  The string is empty"
fi
```

Output:

```
Java = Ruby ?  a isn't equal to b
Java ! = Ruby ?  a isn't equal to b
```

```
-z Java ?  The string length isn't 0
-n Java ?  The string length isn't 0
Java ?  The string isn't empty.
```

Explanation:

=	If two strings are equal, returns true .
! =	If two strings are not equal, returns true .
-z	If the length of a string is equal to zero, returns true .
-n	If the length of a string is not equal to zero, returns true
$	If the string is not empty, returns true .

File Operators

File operators are used to detect various properties of Linux files.

Operat or	Checks
-d file	If the object is a directory, return true .
-f file	If the object is a file, return true .
-p file	If the object is a pipe, return true .
-r file	If the object is a readable file, return true .
-w file	If the object is a writable file, return true .
-x file	If the object is an executable file, return true .
-e file	If the object is an empty file, return true .
-s file	If the object isn't an empty file, return true .
-b file	If the object is a block device file, return true .
-c file	If the object is a character device file, return true .

Example 5.2

# Assume that the file "myFile . sh" does **not** exist, but we still check the property of this file .

```
#! /bin/bash
file="/root/myFile. sh"
if [ -r $file ]     # check if the object is readable
then
   echo "The object is readable"
```

```
else
   echo "The object isn't readable"
fi

if [ -w $file ]     # check if the object is writable
then
   echo "The object is writable"
else
   echo "The object isn't writable"
fi

if [ -x $file ]   # check if the object is executable
then
   echo "The object is executable"
else
   echo "The object isn't executable"
fi

if [ -f $file ]    # check if the object is a file
then
   echo "The object is a file"
else
   echo "The object isn't a file"
fi

if [ -d $file ]   # check if the object is a directory
then
   echo "The object is a directory"
```

```
else
   echo "The object isn't a directory"
fi

if [ -s $file ]    # check if the object isn’t empty
then
   echo "The object isn't empty"
else
   echo "The object is empty"
fi

if [ -e $file ]     # check if the file is existing
then
   echo "The file is existing"
else
   echo "The file isn't existing"
fi
```

Output:

```
The object isn't readable
The object isn't writable
The object isn't executable
The object isn't a file
The object isn't a directory
The object is empty
The file isn't existing
```

Explanation:

Assume that there is no the file “myFile. sh” in directory “root”, but we still check the property of this file.

Let’s review the following file test operators:

-d file	If the object is a directory, return true
-f file	If the object is a file, return true
-r file	If the object is a readable file, return true .
-w file	If the object is a writable file, return true .
-x file	If the object is an executable file, return true .
-e file	If the object is an empty file, return true .
-s file	If the object isn’t an empty file, return true .

The result of the final check is “The file isn’t existing. ”

“echo” Command

“echo” command is used to output contents. In here, let’s talk about its special usages.

Output the Escape Characters.

```
#! /bin/sh
echo "\" C++ in 8 Hours\" "
```

Output:

```
“C++ in 8 Hours”
```

Output the Variable Value

```
#! /bin/sh
book ="C# in 8 Hours"
echo "$book is a great book"
```

Output:

```
C# in 8 Hours is a great book
```

Output the Contents in the Same Line

```
#! /bin/sh
echo -e "Go \c "         # -e    turn on the escape switch
echo "in 8 Hours"        # \c    output in the same line
```

Output:

```
Go in 8 Hours
```

Output to a File

```
#! /bin/sh
echo "PHP in 8 Hours" > myFile      # “>” redirect to
```

Output the Original Format

```
#! /bin/sh
echo ' He said: "OK! " '      # using single quotes
```

Output:

```
He said: "OK! "
```

Output the Executed Result

```
#! /bin/sh
echo Today is `date`
# using back quotes rather than single quotes
```

Output:

```
Today is Thu Aug 22 09: 01: 19 DST 2019
```

Show the Input by Users

```
#! /bin/sh
echo "Please type your name: "
read name       # "read" accepts the input by users
echo "My name is $name"
```

Output:

```
Please type your name:
Ray Yao
My name is Ray Yao
```

Hour 6

Test Command

The test command checks to see if a condition is true. It can be used to test three aspects: numerics, characters and files.

[ ] is used to execute the arithmetical calculation.

```
test  numerics / characters / files
```

Example 6.1

```
#! /bin/bash
num1=10
num2=20
if test $[num1] -eq $[num2]
then
    echo 'Two numbers are equal. '
else
    echo 'Two numbers are not equal. '
fi
```

Output:

```
Two numbers are not equal.
```

Explanation:

“test $[num1] -eq $[num2]” tests the equality of two numbers.

[ ] Symbol

[ ] is used to execute the arithmetical calculation.

In an expression, the two sides of the = cannot have spaces.

Example 6.2

```
#! /bin/bash
x=100
y=200
z=$[x + y]
echo "The result is: $z"
```

Output:

```
The result is: 300
```

Explanation:

[x + y] executes an arithmetical calculation.

Note: the two sides of the = cannot have spaces.

Test Equality

Test command can be used to test whether two strings are equal.

Example 6.3

```
#! /bin/bash
str1="Java"
str2="JavaScript"
if test $str1 = $str2
then
    echo 'Two strings are equal! '
else
    echo 'Two strings are not equal! '
fi
```

Output:

```
Two strings are not equal!
```

Explanation:

"test $str1 = $str2" tests the equality of two strings.

Test Existence

Test command can be used to test whether a file is existing.

Example 6.4

```
#! /bin/bash
# assume that theFile . sh is not existing .
if test -e /root/theFile.sh       # -e checks existence
then
    echo 'The file exists! '
else
    echo ' The file doesn't exist! '
fi
```

Output:

```
The file doesn't exist!
```

Explanation:

Assume that theFile. sh is not existing.

"test -e /root/theFile. sh" tests whether the file is existing.

Test Empty String

Test command can be used to test whether a string is empty.

Example 6.5

```
#! /bin/bash
# assume "$myString" has not been assigned a value yet .
if test -n "$myString"      # -n checks empty string
then
    echo "It's not an empty string. "
else
    echo "It's an empty string. "
fi
```

Output:

It's an empty string.

Explanation:

Because "myString" has not been assigned any value, it is an empty string.

Shell Function

A function is a code block that can repeat to run many times.
The syntax to define a function is as follows:

```
function_name(){
}
```

To call a function, use "function_name;"

```
function_name
```

Example 6.6

```
#! /bin/bash
myFunction() {     # define a function
    echo "jQuery in 8 Hours"
}
echo "Before calling the function. "
myFunction      # call the function
echo "After calling the function. "
```

Output:

```
Before calling the function.
jQuery in 8 Hours
After calling the function.
```

Explanation:

"myFunction(){   }" defines a function

"myFunction" calls the function.

Function with Arguments

When calling a function, you can pass parameters to the function body**.** For example, pass the first parameter to ${1}, pass the second parameter to ${2} **...**

Example 6.7

```
#! /bin/bash
myFunction(){
    echo "The first parameter is: ${1} ! "
    echo "The second parameter is: ${2} ! "
    echo "The third parameter is: ${3} ! "
    echo "The fourth parameter is: ${4} ! "
    echo "All parameters are: $* ! "
}
myFunction Kotlin in 8 Hour      # pass four parameters
```

Output:

```
The first parameter is: Kotlin !
The second parameter is: in !
The third parameter is: 8 !
The fourth parameter is: Hours !
All parameters are: Kotlin in 8 Hours !
```

Explanation:

${1} accepts the first parameter

${2} accepts the second parameter**.**

$* shows all parameters

"myFunction **Kotlin in 8 Hour** " calls the function and pass four parameters**.**

Return Values

"return" can return a value to the "**$?** "**.**
"**$?** " shows the return value**.** (In some machine, "**$?** " only shows the value from 0 to 255**.**)

function_name { **return** value }	
$?	# shows the return value

Example 6.8

```
#! /bin/bash
add(){      # define a function
x=100
y=155
z=`expr $x + $y`
return $z    # returns the value to $?
}
add       # call the function
echo "The sum of two numbers is $? "
```

Output: The sum of two numbers is 255

Explanation:

"return $z" returns the value to **$?.**
"**$?** " shows the return value**.**

Example 6.9

```
#! /bin/bash
myFunction(){
echo "Please enter the first number: "
read x        # assume entering 80
echo "Please enter the second number: "
read y        # assume entering 90
z=`expr $x + $y`
return $z      # returns a value to " $? "
}
myFunction
echo "The return value is $? "
```

Output:

```
Please enter the first number:
80
Please enter the second number:
90
The return value is 170
```

Explanation:

"return $z" returns a value to "**$?** "

"**$?** " shows the return value.

Hour 7

if .. then .. elif .. then .. else

"if**...** then...elif...then...else " runs this code if a condition is true and runs that code if the condition is false**.**

```
if condition1
then
    code1
elif condition2
then
    code2
else
    code3
fi
```

"elif" means "else...if..."

Example 7.1

```
#! /bin/bash
x=100
y=80
if [ $x == $y ]
then
   echo "x is equal to y"
elif [ $x -gt $y ]      # meet this condition
then
   echo "x is greater than y"
elif [ $x -lt $y ]
```

```
then
   echo "x is less than y"
else
   echo "n/a"
fi
```

Output:

```
x is greater than y
```

Explanation:

"elif" means "else…if…"

"elif [$x -gt $y]" tests if $x is greater than $y.

For Loop

"For loop" runs a block of code repeatedly by the specified items

```
for var in item1 item2 item3…
do
    code1
    code2
   …
done
```

Example 7.2

```
#! /bin/bash
fo r var in 1 2 3
do
    echo "The number is: $var"
done
```

Output:

```
The number is: 1
The number is: 2
The number is: 3
```

Explanation:

"**fo** r var **in** 1 2 3" repeatedly run the next code when var=1, var= 2, var=3**.**

Example 7.3

```
#! /bin/bash
for str in Rust in 8 Hours
do
    echo $str
done
```

Output:

```
Rust
in
8
Hours
```

Explanation:

"**for** str **in** Rust in 8 Hours" repeatedly run the next code when str=Rust, var=in, var=8,var=Hours**.**

While Loop

"while loop" loops through a block of code if the specified condition is true**.**

```
while condition
do
     code
done
```

Example 7.4

```
#! /bin/bash
counter=1
while(( $counter<=4 ))       # run 4 times
do
    echo $counter
    counter=`expr $counter + 1`
done
```

Output:

```
1
2
3
4
```

Explanation: ($counter<=4) is a test condition, if the condition is true, the code will loop at most four times**.**

"While Loop" statement can be used to input data continuously, but if you want to finish the input, please press <ctrl + d> to quit the program.

Example 7.5

```
#! /bin/bash
echo 'Please enter a book name: '
while read book      # accepts the input from user
do
   echo "OK! $book is a good book! "
   echo "If quit, please press <ctrl + d>. "
done
```

Output:

```
Please enter a book name:
PHP in 8 Hours
OK! PHP in 8 Hours is a good book!
If quit, please press <ctrl + d>.
```

Explanation:

"while read book" accepts the continuous input from the user. You can input data more than one time if you want.

Until Loop

"until loop" loops through a block of code if the specified condition is **false** , will quit the loop if the condition is **true.**

```
until condition
do
    code
done
```

Example 7.6

```
#! /bin/bash
n=0
until [ $n -ge 4 ]        # will quit the loop if true
do
   echo $n
   n=`expr $n + 1`
done
```

Output:

```
0
1
2
3
```

Explanation:

"until [$n -ge 4]" will quit the loop if $n is equal to 4**.**

Case Statement

“case statement” is a multi-select commend . The case statement matches a value to a pattern . If the match succeeds, executes the matching command .

```
case value in
pattern1) code1;;        # note using ) and ;;
pattern2) code2;;
pattern3) code3;;
pattern*)  code4;;      # when match nothing, run this .
esac
```

Example 7.7

```
#! /bin/bash
echo 'Please input a number between 1 to 4: '
read num
case $num in
   1)   echo 'You have input 1';;
   2) echo 'You have input 2';;
   3)   echo 'You have input 3';;
   4)   echo 'You have input 4';;
   *) echo 'Your number is not between 1 to 4';;
esac
```

# Assume we input a number 2 .

Output:

```
Please input a number between 1 to 4:
2
You have input 2
```

Explanation:

Assume we input a number 2, which matches the second case, therefore the second code is executed.

Break Command

"break" keyword is used to stop the running of a loop according to the condition.

```
break
```

Example 7.8

```
#! /bin/bash
num=0
while(( $num<=8))
do
if [ $num == 3 ]
then
break       # quit the while loop
fi
num=`expr $num + 1`
echo $num
done
```

Output:

```
1
2
3
```

Explanation:

If $num is 3, the program will run the “break” command, and quit the loop,

Continue Command

“continue” command is used to stop the current iteration, ignoring the following code, and then continue the next loop.

```
continue
```

Example 7.9

```
#! /bin/sh
nums="1 2 3 4 5 "
for n in $nums
do
if [ $n -eq 3 ]
then
continue        # run the next for loop
fi
echo $n
done
```

Output

```
1
2
4
5
```

Explanation:

Note**:** The output has no the 3 number**.**

If $n = 3, the program will run “continue” command, ignore the next command “echo $n”, and then run the next “for loop”**.**

Hour 8

Select Statement

“Select” statement allows the user to select one item to meet the condition, and run the related code**.**

“Select” statement usually works with “Case” statement**.**

```
select var in item1 item2 item3...
do
    case statement
done
```

Example 8.1

```
#! /bin/bash
Echo "Please input a number between 1 to 4. "
echo "If you input 4, the program will exit. "
select choice in 1 2 3 4 ;     # select one number
do
  case $choice in
     1)  echo "You choose 1";;
     2)  echo "You choose 2";;
     3)  echo "You choose 3";;
     4)  echo "You choose 4 to exit"
          break;;
     *)  echo "Enter error! "
          break;;
  esac
done
```

Output:

```
Please input a number between 1 to 4.
If you input 4, the program will exit.
1) 1
2) 2
3) 3
4) 4
#? 2
You choose 2
#? 4
You choose 4 to exit
```

Explanation:

Assume that we input 2 at the first time, input 4 at the second time.
"select choice in 1 2 3 4 ;" choose one of the 4 numbers.
"Select" statement usually works with "Case" statement.
#? is an input prompt, you can input one number.

I/O Redirection

Linux Shell system commands take input from your terminal and send the resulting output back to your terminal. A command usually reads data from standard input, and writes data to standard output.

The commands of the redirection are as follows:

Commands	**Descriptions**
command > file	redirect output to the file.
command < file	redirect input to the file.
command >> file	append output to the file.
n > file	redirect File n to the file
n >> file	append File n to the file
n >& m	combine the output file m and File n
n <& m	combine the input file m and File n
<< tag	take the content between tags as input.

"File n", "File m" means the file with file descriptor 0, 1 or 2.

0 means Standard Input. (STDIN)

1 means Standard Output. (STDOUT)

2 means Standard Error. (STDERR).

Output Redirection

Output redirection**:** Execute the command and store the output into the file by using “**>** ”

```
command > file
```

Example 8.2

```
echo "R in 8 Hours" > myFile
cat  myFile        # “cat” views the content of myfile
```

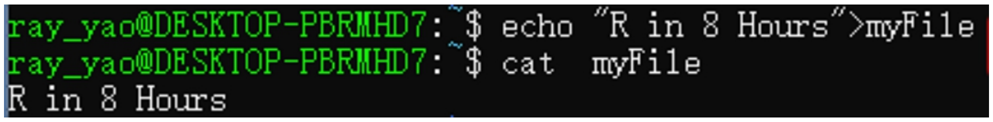
Output:

R in 8 Hours

Explanation:

“echo “R in 8 Hours” **>** myFile” redirects the output to myFile

You need to use “cat” command to view the contents of myFile**.**



```
ray_yao@DESKTOP-PBRMHD7:~$ echo "R in 8 Hours">myFile
ray_yao@DESKTOP-PBRMHD7:~$ cat  myFile
R in 8 Hours
```

Append Redirection

If you don't want the file content to be overwritten, please append the contents to the end of the file by using “**>>** ”

```
command >> file
```

Example 8.3

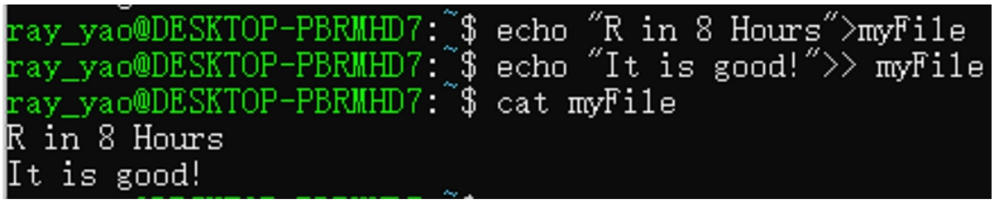
```
echo "R in 8 Hours" > myFile
echo "It is good! " >>  myFile
cat  myFile        # “cat” views the content of myfile
```

Output:

```
R in 8 Hours
It is good!
```

Explanation:

“echo “It is good**!** ”**>>** myFile” append the contents to myFile**.**



```
ray_yao@DESKTOP-PBRMHD7:~$ echo "R in 8 Hours">myFile
ray_yao@DESKTOP-PBRMHD7:~$ echo "It is good!">> myFile
ray_yao@DESKTOP-PBRMHD7:~$ cat myFile
R in 8 Hours
It is good!
```

Input Redirection

Input redirection**:** Get input from a file by using "**<** "**.**

```
command < file
```

Example 8.4

```
wc < myFile     # myFile is created on the previous page .
```

Output:

```
2 7 25
```

Explanation:

"wc < myFile" gets the input from myFile**.**

"wc" counts how many lines, words, characters in the file**.**

2**:** there two lines

7**:** there seven words**.**

25**:** there are characters**.** (including spaces)

myFile is the file created on the previous page**.**

File Descriptor

When each Linux command executes, they will open three files

1. **Standard input file** (stdin)**:** the file descriptor of stdin is 0, and the Shell program reads data from stdin by default**.**
2. **Standard output file** (stdout)**:** the file descriptor of stdout is 1, and Shell program writes data to stdout by default**.**
3. **Standard error file** (stderr)**:** the file descriptor of stderr is 2, and the Shell program writes error messages to the stderr stream**.**

Example:

Command < file redirects stdin to file

```
$ command < file
```

Example:

Command > file redirects stdout to file

```
$ command > file
```

Example:

Read the input from the infile file, write the output to the outfile**.**

```
$ command < infile > outfile
```

Here Document

Here Document is used to redirect the file contents to a command.

```
command << delimiter
    document
delimiter
```

The second delimiter must be on the most left, no characters in front, and no characters in the back.

Example 8.5

```
#! /bin/bash
cat << DOC          # DOC defines a Here Document
AngularJS in 8 Hours
JavaScript in 8 Hours
Visual Basics in 8 Hours
DOC
```

Output:

```
AngularJS in 8 Hours
JavaScript in 8 Hours
Visual Basics in 8 Hours
```

Explanation:

"**cat << DOC......DOC** " is a special redirecting mode, redirect the file contents to a command. "cat" shows the contents of the file.

"DOC" is a delimiter. You can specify other characters as a delimiter. "DOC……DOC" defines a Here Document.

Example 8.6

```
#! /bin/bash
wc << EOF       # wc counts the number of lines, word,
characters
AngularJS in 8 Hours
JavaScript in 8 Hours
Visual Basics in 8 Hours
EOF
```

Output:

```
3 13 68
```

Explanation:

"**wc << EOF……EOF** " redirects the document contents to "wc".

"wc" is used to count the numbers of lines, words and characters.

"EOF" is a delimiter, defines a Here Document.

3: means 3 lines

13: means 13 words.

68: means 68 characters, including spaces.

File Included

The Shell can include an external script file into the current file.

```
source   ./ external_file
```

"./ " locates the external file.

We will create two files as examples.

(1) The file1. sh is as follows:

Example 8.7

```
#! /bin/bash
url = www . amazon . com/author/ray-yao
```

# This is the file1 . sh

(2) The file2. sh is as follows :

Example 8.8

```
#! /bin/bash
source ./file1.sh      # include an external file
echo "The website of Ray Yao is:$url "
```

# This is the file2 . sh

Output:

The website of Ray Yao is:

www . amazon . com/author/ray-yao

Explanation:

"source ./ file1. sh" includes an external file "file1. sh".

In file1, defines a variable "url".

In file2, references the variable "$url".

Appendix 1
Shell Scripting Chart

Built-in Parameters Chart

There are built-in parameters in Shell as follows**:**

Parameters	**Descriptions**
$0	Return the file name
$#	Return the total number of the parameters
$*	Return all parameters input by the user
$@	Return all parameters input by the user
$$	Return the current process ID number of the script
$ **!**	Return the last process ID number of the script
$?	Returning 0 indicates running successfully

Printf Command Chart

“printf” is a command to format the output in the Shell**.**

```
printf  “type”  variables
```

“type” is a parameter used to format the output**.**

Type	**Description**
%b	Print it as a binary value
%d	Print it as a digital value
%f	Print it as a float-point value
%o	Print it as an octal value
%s	Print it as a string value
%x	Print it as a hexadecimal value

Escape Sequences Chart

The “ \ ” backslash character can be used to escape characters**.**

\a outputs an alert

\n outputs the content to the next new line**.**

\r makes a return

\t makes a tab

\f outputs the content to the next page**.**

\’ outputs a single quotation mark**.**

\” outputs a double quotation mark**.**

Comparison Operators Chart

Linux Shell has comparison operators as follows**:**

Operators	**Checks**
-eq	equal
-ne	not equal
-gt	greater than
-ge	greater or equal
-lt	less than
-le	less or equal

The comparison expression returns true or false**.**

Boolean Operators Chart

Operators	**Equivalent**
-a	and
-o	or
!	not

After using logical operators, the result will be true or false**.**

true -a true;
returns true;
	true -a false;
	returns false;
		false -a false;
		returns false;

true -o true;
returns true;
	true -o false;
	returns true;
		false -o false;
		return false;

! false;
returns true;
	! true;
	returns false;	

String Operators Chart

Operators	**Checks**
=	If two strings are equal, returns true **.**
! =	If two strings are not equal, returns true **.**
-z	If the length of a string is equal to zero, returns true **.**
-n	If the length of a string is not equal to zero, returns true
$	If the string is not empty, returns true **.**

File Operators Chart

File operators are used to detect various properties of Linux files.

Operator	**Checks**
-d file	If the object is a directory, return true
-f file	If the object is a file, return true
-p file	If the object is a pipe, return true
-r file	If the object is a readable file, return true .
-w file	If the object is a writable file, return true .
-x file	If the object is an executable file, return true .
-e file	If the object is an empty file, return true .
-s file	If the object isn’t an empty file, return true .
-b file	If the object is a block device file, return true .
-c file	If the object is a character device file, return true .

I/O Redirection Chart

Linux Shell system commands take input from your terminal and send the resulting output back to your terminal. A command usually reads data from standard input, and writes data to standard output.

The commands of the redirection are as follows:

Commands	**Descriptions**
command > file	redirect output to the file.
command < file	redirect input to the file.
command >> file	append output to the file.
n > file	redirect File n to the file
n >> file	append File n to the file
n >& m	combine the output file m and File n
n <& m	combine the input file m and File n
<< tag	take the content between tags as input.

Appendix 2 Exercises

Questions

Please choose the correct answer.

(01)

```
fill in      # fill in characters as a multi-line comment
comment 1
comment 2
comment 3
...
EOF
```

A. EOF B. #EOF C. <<EOF D. : <<EOF

(02)

```
#! /bin/sh
str1="Kotlin"
str2="in 8 Hours"
myString="$str1 fill in $str2"    # connect two strings
echo $myString
```

A. + B. – C. nothing D. concat

(03)

The syntax to access the elements of an array is **fill in** ?

A. $(array_name[index])
B. ${array_name[index]}
C. array_name(index)
D. array_name{index}

(04)

#**!** /bin/bash

sum= **fill in** **expr 100 + 200** **fill in**

echo "The sum is**:** $sum"

A**.** double quotes

B**.** single quotes

C**.** back quotes

D**.** nothing

(05)

Parameter "-s file" checks **fill in** ?

A**.** If the object isn't an empty file, return true**.**

B**.** If the object is an empty file, return true**.**

C**.** If the object is a block device file, return true**.**

D**.** If the object is a character device file, return true **.**

(06)

```
#! /bin/bash
if test fill in /root/myFile.sh       # checks myFile if exist .
then
    echo 'The file exists! '
else
    echo ' The file doesn't exist! '
fi
```

A. –o   B. –e   C. –n   D. –s

(07)

```
if  fill in condition fill in
then
    code1
else
    code2
fi
```

A. (   )   B. {   }   C. [   ]   D. nothing

(08)

```
# redirects the output to myFile
echo "R in 8 Hours"  fill in   myFile
cat  myFile
```

A. ->   B. >   C. in   D. to

(09)

```
#! /bin/sh
echo "Please input your name. "
fill in myInput      # the input value will be stored in myInput
echo "Hi! Your name is: $myInput"
```

A. input   B. var   C. $myInput=   D. read

(10)

```
#! /bin/sh
str="jQuery in 8 Hours"
echo fill in      # get the length of the string
```

A. str. length()   B. length(str)   C. ${#str}   D$[#str]

(11)

fill in can get all elements of an array.

A. array_name[@]

B. array_name[#]
C. array_name[%]
D. array_name[&]

(12)

Operators	**Running**
+	add
-	subtract
fill in	multiply
/	divide

A. * **B.** * **C.** X **D.** \x

(13)

```
#! /bin/sh
echo fill in "Go \c "        #  turn on the escape switch
echo "in 8 Hours"
```

A. –a **B.** –o **C.** –s **D.** –e

(14)

```
#! /bin/bash
if test fill in "$myString"      # checks empty string
```

```
then
    echo "It's not an empty string. "
else
    echo "It's an empty string. "
fi
```

A. –o　　B. –e　　C. –n　　D. –s

(15)

"until loop" loops through a block of code if the specified condition is **fill in** , will quit the loop if the condition is **fill in** .

A.	true	true
B.	true	false
C.	false	false
D.	false	true

(16)

If you don't want the file content to be overwritten, please append the contents to the end of the file by using "**fill in** " symbol.

A. >>　　B. ->　　C. =>　　D. >=

(17)

Parameters	**Descriptions**
fill in	Return the file name
$#	Return the total number of the parameters
$*	Return all parameters input by the user
$@	Return all parameters input by the user

A. $n   B. $0   C. $f   D. $%

(18)

```
#! /bin/bash
arr=(PHP MySQL in 8 Hours)
length= fill in      # get the length of the array
echo $length
```

A . $arr . length()   B . length($arr)   C . size($arr)   D . ${#arr[@]}

(19)

true -a true; returns true;	true -a false; returns false;	false -a false; returns false;
true -o true; returns **fill in** ;	true -o false; returns **fill in** ;	false -o false; return f **fill in** ;
! false;	! true;	

returns true;	returns false;	

A. true true true
B. false true false
C. true true false
D. false false true

(20)

Operators	**Checks**
-z	If the length of a string is equal to zero, returns true .
-n	If the length of a string is not equal to zero, returns true
fill in	If the string is not empty, returns true .

A. $ B. & C. @ D. %

(21)

```
#! /bin/bash
add(){        # define a function
x=100
y=155
z=`expr $x + $y`
return $z       # return the value
}
add         # call the function
echo "The sum is fill in "    # show the return value
```

A. $!   B. $@   C. $* $?

(22)

```
#! /bin/bash
echo 'Please input a number between 1 to 4: '
read num
case $num in
   1)   echo 'You have input 1' fill in
   2) echo 'You have input 2' fill in
   3)   echo 'You have input 3' fill in
   4)   echo 'You have input 4' fill in
   *) echo 'Your number is not between 1 to 4' fill in
esac
```

A. ;   B. ;;   C. ;;;   D. ;;;;

(23)

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

fill in   # counts the number of lines, word, characters
AngularJS in 8 Hours
JavaScript in 8 Hours
Visual Basics in 8 Hours
EOF

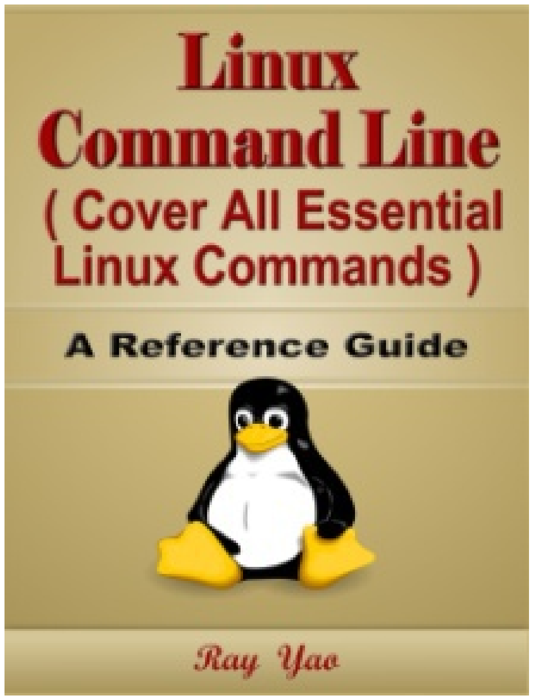
A. num << DOC
B. ls << DOC
C. wc << EOF
D. cat << EOF

(24)
#**!** /bin/bash
fill in ./file1.sh   # include an external file "file1 **.** sh"
echo "The website of Ray Yao is**:$url** "

A**.** include   B**.** import   C**.** resource   D**.** source

Answers

01. D	13. D
02. C	14. C
03. B	15. D
04. C	16. A
05. A	17. B
06. B	18. D
07. C	19. C
08. B	20. A
09. D	21. D
10. C	22. B
11. A	23. C
12. B	24. D


Linux Command Line (Cover All Essential Linux Commands)

A Reference Guide

Ray Yao

See You!