SIRE511 : LINUX AND BIOINFORMATICS DATA SKILLS

Fundamental Linux Part V:

5th week: Shell scripting part II

1/10/2024

Kwanrutai Mairiang, Ph.D

Arrays

- An array is a data container comprised of two parts including keys and values.

Create indexed or associative arrays using `declare` command

Syntax:

1). Bash indexed array: the keys of array are ordered integers.

```
declare -a array_name

array_name=(value1 value2)
```

2). Bash associative array: the keys of array are strings.

```
declare -A array_name

array_name=(["key1"]="value1" ["key2"]="value2")
```

Arrays

Access values of an array

1) Access all data in the array

```
${array_name[@]}
```

2). Show all index of the array

```
${!array_name[@]}
```

3). Access to the data of the index **n** of the array

```
${array_name[n]}
```

4). Show the length of the array

```
${#array_name[@]}
```

5). Remove both index and data at the index **n**

```
unset array_name[n]
```

6). Add new data to the array at the index **n**

```
array_name[n]=”new_value”
```

Sort number in array

```
# Declare the index array
$ declare -a arr_num
# Assign number to array
$ arr_num=(1 3 5 2 4 8 6 7)
# Sort the number in array
$ IFS=$'\n' sorted=($(sort -n <<<"${arr_num[*]}"))
$ unset IFS
# Print the sorted array
$ echo ${sorted[@]}
```

- IFS=$'\n' sets the Internal Field Separator to a newline, enabling correct splitting of elements in the array.
- ${arr_num[*]} expands the elements to a single string, separated by IFS
- <<< is a “here-string” that passes the expanded array as input to sort
- sort -n performs a numeric sort on the elements of the expanded array
- sorted is a variable where the sorted contents will be stored
- Unset to restore the original IFS value

https://www.baeldung.com/linux/sort-bash-arrays

Arithmetic operators

- Arithmetic operator is a mathematical function that used to perform an arithmetic operation. The following 11 arithmetic operators are supported by bash.

- Double parentheses can be used to specify arithmetic operation in Bash.

Syntax:

((expression))

Operator	Name	Description	Example
+	Addition	It adds two operands	x=$((10+3)) Result: x = 13
-	Subtraction	It subtracts the second operand from the first one	x=$((10-3)) Result: x = 7
*	Multiplication	Multiply two operands	x=$((10*3)) Result: x = 30
/	Division	Divide first operand from second operands and return quotient	x=$((10/3)) Result: x = 3

Arithmetic operators

Operator	Name	Description	Example
**	Exponentiation	The second operand raised to the power of the first operand.	x=$((10**3)) Result: x = 1000
%	Modulo	Divide the first operand from the second operand and return the remainder	x=$((10%3)) Result: x = 1
+=	Increment by constant	Increment value of the first operand with a given constant value	x=10 ((x+=3)) Result: x=13
-=	Decrement by constant	Decrement value of the first operand with a given constant value	x=10 ((x-=3)) Result: x=7
*=	Multiply by constant	Multiply value of the first operand with a given constant value	x=10 ((x*=3)) Result: x=30
/=	Divide by constant	Divide value of the first operand with a given constant value and return the quotient	x=10 ((x/=3)) Result: x=3
%=	Remainder by dividing with constant	Divide value of the first operand with a given constant value and return the remainder	x=10 ((x%=3)) Result: x=1

Arithmetic operators

calculation.sh

```
#!/bin/bash

echo "10 + 3 = $((10+3))”
echo "10 + 3 = $((10–3))”
echo "10 + 3 = $((10*3))”
echo "10 + 3 = $((10/3))”
a=$((10%3))
echo "10 % 3 = $a"

x=10
echo "x = $x"
echo "x%=3 then x = $((x%=3))”

b=$((x/=3))
echo "x/=3 then x = $b"
```

Please revise the given command and execute it in the command line.

`$((x%=3))` → `$((x%3))`

`b=$((x/=3))` → `b=$((x/3))`

Script Input (STDIN)

- Command line arguments
- Read command

Command line arguments

The arguments are input that necessary for processing the script. The command line arguments are passed in a positional way.

Syntax:

```
./bash_script.sh arg1 arg2 arg3..
```

where `arg1 = $1  arg2 = $2  arg3 = $3`

Command line arguments

Special variable	Detail
`$0`	Name of bash script
`$1 … $n`	Positional argument indicated from 1 to n.
`$@`	All arguments that are passed in to the script
`$#`	The total number of arguments passed to script
`$?`	The exit status of the most recently run process
`$$`	The process ID of the current script

Practical: Command line arguments

cmdAgrument1.sh

```
#!/bin/bash

echo "My name is $1"
echo "I'm $2 year old."
```

cmdAgrument2.sh

```
#!/bin/bash
i=1
for user in "$@"
do
        echo "Username: $i. $user";
        i=$((i+=1))
done
```

Read command

A read command is built-in command that takes the user input into a variable.

Syntax:

```
read OPTIONS ARGUMENT
```

Try Read command

1). Save the user input into a specified variable

```
     read input
     echo $input
```

2).Split the user input into different variables by adding multiple argument

```
     read var1 var2
     echo $var1
     echo $var2
```

3). Piping: pipe a standard output from one command and pass it as an input for the other command

```
echo Red Black | (read var1 var2; echo $var1 $var2)
```

Condition statement

Comparison operators

A condition statement is used for decision making in any programing language. Bash scripting also use this statement for making some decisions in an automated task.

Comparison operators

Operator	Syntax	Description
-eq	INTEGER1 -eq INTEGER2	Return true if two numbers are equal
-ne	INTEGER1 -ne INTEGER2	Return true if two numbers are not equal
-lt	INTEGER1 -lt INTEGER2	Return true if integer1 less than integer2
-gt	INTEGER1 -gt INTEGER2	Return true if integer1 greater than integer2
==	STRING1 == STRING2	Return true if STRING1 is equal to STRING2
!=	STRING1 != STRING2	Return true if STRING1 is not equal to STRING2
!	! EXPRESSION	Return true if the expression is false
-d	-d FILE	Check the existence of a directory
-e	-e FILE	Check the existence of a file
-r	-r FILE	Check the existence of a file and read permission
-w	-w FILE	Check the existence of a file and write permission
-x	-x FILE	Check the existence of a file and execute permission

If statement

- The basic if statement contains one level of condition and action.
- The syntax consisting of **`if`** follow by **`EXPRESSION`** in square brackets.
- If the **`EXPRESSION`** is true, **`then`** **`ACTION`** will be performed. The statement ends with **`fi`**.
- One **`if`** statement can contain one (single condition) or more expressions (multiple conditions).

If statement

1). Single condition

Syntax:

```
if [ EXPRESSION ]; then
ACTION
fi
```

Check if input number is less than 100

singleCond.sh

```
#!/bin/bash

#Get input number from user input
echo "Enter a number"
read n

#Check if input number less than 100
if [ $n -lt 100 ]; then
echo "$n is less than 100"
fi
```

If statement

2). Multiple conditions

Multiple conditions in “if statement” need BOOLEAN operator for joining between conditions.

Operator	Symbol	Description
AND	&&	Return TRUE when both Expression_1 and Expression_2 are TRUE
OR	||	Return TRUE when one of Expression_1 or Expression_2 is TRUE

If statement

2). Multiple conditions

Syntax:

AND operator

```
if [ EXPRESSION_1 ] && [ EXPRESSION_2 ];
then
ACTION
fi
```

OR operator

```
if [ EXPRESSION_1 ] || [ EXPRESSION_2 ];
then
ACTION
fi
```

If statement

2). Multiple conditions

Check if input number is between 1 and 10

multipleCond.sh

```
#!/bin/bash

#Get input number from user input
echo "Enter a number"
read n

#Check if input number is greater than 1
and less than 10
if [ $n -gt 1 ] && [ $n -lt 10 ]; then
echo "$n is number between 1 and 10 "
fi
```

If-else statement

This pattern of conditional statement is used to execute one action with a true condition and the other action with a false condition.

Syntax:

```
if [ EXPRESSION ]; then
ACTION_1
else
ACTION_2
fi
```

If-else statement

ifelse.sh

Check if input number is already in “users” array

```bash
#!/bin/bash

declare -A users

users=([10]="Harry Potter"
[15]="Hermione Granger"
[21]="Ron Weasley"
[28]="Kwanrutai Mairiang")

echo "Please enter your registeration number"
read num

if [ -n "${users[$num]}" ]; then
   printf '%s is already registered\n' "${users[$num]}"
else
   echo "Please register for the meeting"
fi
```

If..elif..else statement (if-else in ladder)

- This pattern of conditional statement is used for a series of conditions.
- The set of **ACTION** in **if** statement is executed, when the **EXPRESSION** is TRUE.
- If there is no TRUE **EXPRESSION**, the **ACTION** in **else** statement will be executed.

If..elif..else statement (if-else in ladder)

Check grade using the input score

ifelseLadder.sh

```
#!/bin/bash

echo "Enter the score"
read s

if (( $s >= 85 )); then
echo "Grade – A"
elif (( $s < 85 && $s >= 75 )); then
echo "Grade – B"
elif (( $s < 75 && $s >= 65 )); then
echo "Grade – C"
elif (( $s < 65 && $s >= 55 )); then
echo "Grade – D"
else
echo "Grade – F"
fi
```

Syntax:

```
if [ EXPRESSION_1 ]; then
ACTION_1
elif [ EXPRESSION_2 ]; then
ACTION_2
…
else
ACTION_3
Fi
```

Nested if statement

This pattern of conditional statement is used when one condition is true, then the next condition is checked. Two example syntax are shown below.

Syntax 1

if the **EXPRESSION_1** is true, then another expression, **EXPRESSION_2** is checked. If **EXPRESSION_2** also true, **ACTION** will be executed.

```
if [ EXPRESSION_1 ]; then
     if [ EXPRESSION_2 ]; then
     ACTION
     fi
fi
```

Nested if statement: Syntax 1

Check if input number is between 1 and 10 using nested if condition

nested1.sh

```
#!/bin/bash

#Get input number from user input
echo "Enter a number"
read n

#Check if input number is greater than 1 and less than 10
if [ $n -gt 1 ]; then
        if [ $n -lt 10 ]; then
        echo "$n is number between 1 and 10"
        fi
fi
```

Nested if statement

Syntax 2

if **EXPRESSION_1** is true, then the **ACTION_1** will be performed. But, if **EXPRESSION_1** is false, the **EXPRESSION_2** in **else** will be checked. If **EXPRESSION_2** is true, the **ACTION_2** will be executed.

```
if [ EXPRESSION_1 ]; then
ACTION_1
else
      if [ EXPRESSION_2 ]; then
      ACTION_2
      fi
fi
```

Nested if statement: Syntax 2

nested2.sh

```
#!/bin/bash

declare -A users

users=(["Harry"]="Harry Potter"
["Hermione"]="Hermione Granger"
["Ron"]="Ron Weasley"
["Kwanrutai"]="Kwanrutai Mairiang")

echo "Please enter your name"
read name

if [[ -n "${users[$name]}" ]]; then
     echo "Is '${users[$name]}' your Name–Surname? (y/n)"
     read check
     if [ $check == y ]; then
          printf '%s is already registered\n' "${users[$name]}"
     else
          echo "Please register for the meeting"
     fi
else
     echo "Please register for the meeting"
fi
```

Check if input name is already in “users” array

For loop

For loop

- For loop is used for iterating item in the list of items.
- An item from each round is assigned to the variable which is then used to perform any action in loop.
- The syntax of "**For** loop "consisting of **LIST** of data and variable (**ITEM**).
- The list of items can be a series of strings separated by spaces, a range of numbers, output of a command, an array.
- For loop starts with **do** and ends with **done**.

Syntax:

```
for ITEM in [LIST]
do
     ACTION
done
```

For loop: Loop over a series of strings

For loop over series of string: Sunday … Saturday

forloop1.sh

```
#!/bin/bash

count=0
for day in Sunday Monday Tuesday Wednesday Thursday Friday Saturday
do
        echo "Day $((count+=1)) = $day"
done
```

For loop: Loop over a number range

1). Loop over the specified range, {START..END}, of numbers

```
#!/bin/bash
for i in {1..5}
do
	echo "Number: $i"
done
```

forloop2.sh

For loop over specified range of number 1 to 5

2). Loop over the specified range with increment, {START..END..INCREMENT}

```
#!/bin/bash
for i in {0..10..2}
do
	echo "Number: $i"
done
```

forloop2_2.sh

For loop over specified range of number 0 to 10 with increment 2

Loop over array elements

Use for loop for iterating item in array.

forLoop3.sh

```bash
#!/bin/bash

wkday=(Monday Tuesday Wednesday Thursday Friday)

echo "Loop over items in array:"
for i in ${wkday[@]}
do
        echo $i
done

echo -e "\nLoop over index of items in array"
```

For loop over item in array

Loop over output of a command

The following example showing how to iterate filename with specific extension in current folder.

For loop over the list of files with extension “.sh”

```
#!/bin/bash

for file in *.sh
do
        echo $file
done
```

forLoop4.sh

While loop

While loop

- Another type of loop is while loop. While loop will iterate while the specified condition is true.
- While loop is useful when exact times for looping is not known.
- The syntax of "**While**" loop contains **CONDITION** that made the loop keep iterate. Then, **UPGRADE CONDITION** until condition becomes false for stopping the iteration.

Syntax:

```
while [ CONDITION ]
do
    ACTION
    UPGRADE_CONDITION    Ex.((number ++))
Done
```

While loop: Loop over condition

Loop and print out the number from 1 to 5

whileLoop.sh

```
#!/bin/bash

count=1
while [ $count -le 5 ]
do
        echo "Number: $count"
        ((count++))
done
```

While loop: Reading file using while loop

Read data or file from standard input

whileReadFile.sh

```
#!/bin/bash

while read line
do
     #Print out each line in file or input data
     echo $line
#Get filename or data from standard input
done < "${1:-/dev/stdin}"
```

Pipe 5 lines of data from “primer.txt” to Bash script:

```
$ head -n 5 primer.txt | ./whileReadFile.sh
```

Add a Directory to PATH Linux

- **Add to PATH Temporarily**
  - Temporarily addding a directory to PATH affects the current terminal session only. When user close the terminal, the directory is removed
  - Use “export” command to temporarily add directory to PATH.

```
$ export PATH=“path/to/directory:$PATH”
```

- **Add o PATH Permanently**
  - Open the .bashrc file using a text editor

```
$ nano .bashrc
```

  - Paste the ”export” syntax to the end of file.

```
$ export PATH=“path/to/directory:$PATH”
```

  - Save and exit.