Unix Scripting

Week3

Agenda

- Introduction to Shell Scripting
  - Loops
  - File descriptor
  - Multiple Commands
    - Use () vs {}
    - Shell Level

Loops in shell scripting

- while
- for loop
  - for…in
  - for
  - for(…;…;)
- until

While loop

- In most **computer programming** languages, a **while loop** is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition.
- The while loop is used to repeat a section of code an unknown number of times until a specific condition is met.

While

- The second type of looping command to be described in this chapter is the while. The format of this command is
- `while condition`
  - `do`
    - `command`
    - `command ...`
  - `done`

Example

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

Observation: What does the following program do?

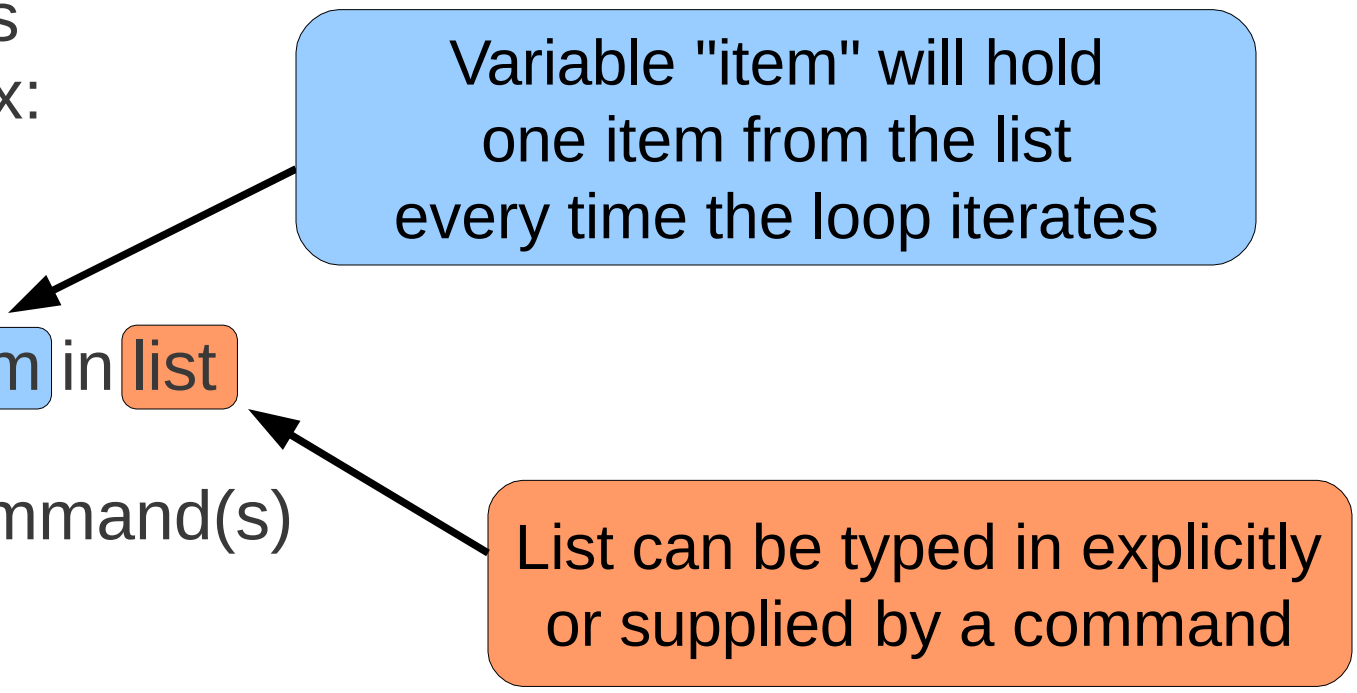
```
while [ "$#" -ne 0 ] do
   echo "$1"
shift
done
```

** *The shift command is one of the Bourne shell built-ins that comes with Bash. This command takes one argument, a number. The positional parameters are shifted to the left by this number, N. The positional parameters from N+1 to $# are renamed to variable names from $1 to $# - N+1.*

Using Loops

- A for loop is a very effective way to repeat the same command(s) for several arguments such as file names
  Syntax:

- for item in list
  do
  command(s)
  done



for loop: using range

- *for number in {start..end..step}*
  - `for number in {1..10}`
  - The curly brackets {} basically denotes a range, and the range, in this case, is 1 to 10 (the two dots separate the start and end of a range).
- To loop between 0 and 100 but only show every tenth number
  - `for number in {0..100..10}`

Example

```
for number in {0..100..10}
do
echo $number
done
```

A More Traditional Looking For Loop

- You can, however, write a for loop in a similar style to the C programming language
  - *for((initialization; condition;alteration))*
- Example

```
for ((number=1;number < 10;number++))
do
   echo $number
done
```

Activity: Using your own word, explain what does the following scripts do?

```
for addr in $(cat ~/addresses)
do
        mail -s ”Newsletter” $addr < ~/spam/newsletter.txt
done
```

```
for count in 3 2 1 'BLAST OFF!!!'
do
        sleep 1
        echo $count
done
```

```
for id in $(seq 1 1000)
do
        mkdir student_$id
done
```

until

- ***until*** control structure, loop until test becomes true (0 return code), the opposite of ***while***
- input=until ["$input" = end]
- do
  - echo -n "Type something: "
  - read input
  - echo "You typed: '$input' "
- done

Using exec to assign a file descriptor (fd) to file

- In the Bash shell environment, every process has three files opened by default.
  - These are standard input, display, and error. The file descriptors associated with them are 0, 1, and 2 respectively.
- In the Bash shell, we can assign the file descriptor to any input or output file. These are called file descriptors.

File Descriptor…

For redirecting the output

- The syntax for declaring myfile.txt as output is as follows:
  - **exec fd > output.txt**
  - **exec 4 > output.txt**
- The syntax for closing the file-descriptor is as follows:
  - **exec fd>&-**
  - Example: exec 4>&-

For redirecting the input

- The syntax for declaring myfile.txt as input is as follows:
  - **exec fd < myfile.txt**
  - **exec 4 < myfile.txt**
- The syntax for closing the file-descriptor is as follows:
  - **exec fd<&-**
  - Example: exec 4<&-

Multiple Commands

- Besides piping, there are other ways that multiple commands may be placed in one line
  - `cmd1 ; cmd2; cmd3 ; …`
- Example:
  - `pwd; date;`
- What would be the output of the following:
  - `pwd; date; whoami > output.txt`

Multiple Commands

- You can perform “multiple command” using () or {} as follow:
  - (cmd1;cmd2;cmd3)
  - { cmd1;cmd2;cmd3; }
- Example:
  - `( pwd; date; whoami ) >output1.txt`
  - `{ pwd; date; whoami; }>output2.txt`
  - Any difference between above commands?

Let’s understand the difference of () vs {}

- Try to run the followings :
  - (m=“message1”; echo $m)
    - What do you see?
  - Run: echo $m
    - What do you get?
- Now, try to run the followings :
  - { n=“message2”; echo $m }
    - What do you see?
  - Run: echo $n
    - What do you get?

What is Shell Level

- When you run a command in a shell, it runs at the shell level.
- Within a shell, you can open another shell, which makes it a subshell of the shell that opened it.
- Therefore, the parent shell is considered the level 1 shell, and the child shell is a level 2 shell.

How to Display the Shell Level

- The way to tell which shell level you are running in is to use the $SHLVL variable.
  - echo $SHLVL

Why Is Shell Level Important?

- The shell level is important when thinking about the scope of variables within your scripts.
- Run the followings:
  - p=“Hello World”
  - echo $p
  - echo $SHLVL
  - bash
  - echo $p
  - echo $SHLVL
  - What have you observed?

difference of () vs {}

- () create a new shell level
- Each Shell Level has its own scope, and if you declare a variable in one level, you can't reach that variable in a different level.
- {} doesn't create a new shell level!

Lists

- AND list
  - list of statements separated by &&
  - statements will be executed till one fails, giving a non-zero exit status
  - Example:
    - [$# != 2] && echo "This command requires two arguments" >&2
- OR list
  - list of statements separated by ||
  - statements will be executed till one succeeds, giving a zero exit status
  - for example:
    - [! -f "$1"] || [! -r "$1"] || [! -d "$2"] || [! -w "$2"] || [! -x "$2"] || cp $1 $2