



Linux Shell Script

Linux Shell Script

Shell 맛보기

실행 환경

- OS 환경에서 shell 기본 위치

```
$ env | grep bash
SHELL=/bin/bash
$ which bash
/bin/bash
```

- env (또는 printenv) 를 사용하면 현재 시스템 환경변수를 볼 수 있다.
- set 를 사용하면 지역변수를 포함한 모든 변수/함수의 목록을 볼 수 있다.
- ~/.bashrc는 Bash 셸 실행에 필요한 환경 변수를 저장하고 있다.
- 환경 관련 파일을 수정하고 난 후 다음과 같이 환경 변수를 현재 셸에 적용해야 한다.

```
source ~/.bashrc
```

셸 메타문자

- 메타문자들은 특수한 의미를 가지는 특수 문자들이며 셸 메타문자들은 와일드카드라고도 한다.

셸 메타문자	의미
₩	뒤에 나오는 문자를 문자 그대로 해석한다.
&	백그라운드에서 실행하도록 한다.
;	명령을 구분한다.
\$	변수를 치환한다.
?	한 문자와 매칭한다.
[abc]	문자들 중 하나의 문자와 매칭한다. 예) a, b 또는 c
[!abc]	문자들 중 하나의 문자도 매칭하지 않는다. 예) a, b 또는 c가 아니다.
(cmds)	명령 그룹화, 서브셸을 생성한다.
{cmds}	명령 그룹화, 서브셸을 생성하지 않는다.
newline	명령 종료
* [] ?	파일명 확장을 위한 셸 메타문자들

셸 메타문자 - 백슬래시(\)

- 인터프리터로부터 하나의 문자를 인용부호화 하거나 이스케이프한다.
- 작은 따옴표가 있다면 인터프리트되지 않는다.
- 달러(\$) 기호, 백쿼터(` `)를 보호한다.
- 큰 따옴표로 둘러싸여지면 인터프리터로부터 백슬래시를 보호한다.

echo where are you from\?

백슬래시는 물음표(?)에 의한 파일명 치환으로부터 셸을 보호

where are you from?

**# echo 시작 라인이며 **

> 다음 라인이다.

시작 라인이며 다음 라인이다.

echo '\\$10.00'

작은 따옴표 안의 모든 문자들은 상수로 취급되며 백슬래시는 문자 그대로 인식

\\$10.00

echo "\$10.00"

큰 따옴표로 둘러싸여진 문자들 중 백슬래시는 변수 치환을 위한 인터프리터 해석으로부터 달러(\$) 기호를 보호

\$10.00

echo 'Don\'t you need \$10.00?'

#백슬래시는 작은 따옴표 안에서 인터프리트되지 않기 때문에

>

셸은 3개의 작은 따옴표로 해석하므로 작은 따옴표 수가 매칭되지 않고

> '

다음 라인의 프롬프트를 보여주며 작은따옴표가 입력되기를 기다린다.

Don't you need 0.00?

셸 메타문자 – 작은 따옴표(' ')

- 앞뒤로 매칭이 되어야 하며 인터프리터로부터 모든 메타문자를 보호한다.
- 작은 따옴표를 출력하기 위해서는 큰따옴표로 둘러싸거나 백슬래시로 이스케이프해야 한다.

echo '안녕하세요?

> 예, 안녕하세요?

> 예.'

안녕하세요?

예, 안녕하세요?

예.

여기서 작은따옴표는 매칭되지 않기 때문에 다음 라인을 보여주고

작은 따옴표를 입력할 때까지 기다린다.

echo Don't you need '\$10.00?'

Don't 에서 어포스트로프(')는 백슬래시에 의해 이스케이프된다.

Don' you need \$10.00?

echo '무엇을 공부하고 계세요?,"리눅스 쉘 스크립트요.'"

#작은 따옴표는 문자열에서 큰 따옴표를 보호한다.

무엇을 공부하고 계세요?,"리눅스 쉘 스크립트요."

셸 메타문자 – 큰 따옴표(" ")

- 변수와 명령 치환을 허용
- 셸에 의해 인터프리트되지 않도록 특수 메타문자들을 보호

```
# name=홍길동
```

```
# name 로컬 변수에 홍길동 할당
```

```
# echo "안녕하세요? $name님, 만나서 만가워요."
```

```
# echo 명령과 큰따옴표를 사용하여 name 변수의 값을 출력
```

```
안녕하세요? 홍길동님, 만나서 만가워요.
```

```
# echo "안녕하세요? $name님, 현재 시간은 $(date)입니다."
```

```
안녕하세요? 홍길동님, 현재 시간은 2018. 06. 13. (수) 03:52:49 KST입니다.
```

출력

- echo와 printf 는 원하는 내용을 화면에 출력한다.
- 아래 스크립트는 모두 동일한 내용을 출력한다. 쌍따옴표에서도 환경 변수는 제대로 해석된다.

```
$ env | grep PWD  
$ echo $PWD  
$ echo "$PWD"  
$ printf "$PWD"
```

- 파일로 출력하려면 출력 Redirection (>)을 사용한다.

```
printf "현재 경로:%s \n홈 경로:%s" $PWD $HOME > $HOME/sample.txt
```

```
$ printf "%8s %-15.3s:\n" first second third fourth fifth sixth
```

- > 은 새로 파일을 작성할 때만 가능하며, 존재하는 파일에 내용을 추가할 경우에는 >> 을 사용한다.
- 표준 출력 디스크립터를 활용해야 할 때도 있다. #2는 stderr 이고 #1은 stdout에 해당한다.

```
$ cat sample.txt > /dev/null 2>&1
```

- 위에서 /dev/null은 Bit-Bucket으로 무한한 양의 데이터를 받을 수 있는 가상 장치이다. 저장할 필요가 없을 때 사용한다. 이 장치로 stderr와 stdout에 해당하는 내용이 모두 출력된다.
- 모니터에 출력하기 위해서는 /dev/tty로 보내야 한다. 임시 파일을 사용하고자 할 때는 tmp\$\$ 와 같이 사용한 후 저장했다가 파일로 일괄 저장한다.

표준 출력

- 대부분의 커맨드 라인 프로그램들은 결과를 모니터에 출력한다.
- 이를 표준 출력이라고 말하며, 파일 디스크립터 숫자 값으로 1로 표기한다.
- 표준 출력을 '>' 문자를 사용하여 파일로 저장할 수 있다.
(> : 출력 리다이렉션 문자)

```
[root@RHEL7-SVR ~]# ls
anaconda-ks.cfg      script.txt
initial-setup-ks.cfg  shelltest.sh
[root@RHEL7-SVR ~]# ls > ls.txt
[root@RHEL7-SVR ~]# cat ls.txt
anaconda-ks.cfg
initial-setup-ks.cfg
ls.txt
script.txt
shelltest.sh
```

표준 입출력 파일 디스크립터

표준 입출력	파일 디스크립터 숫자
표준 입력(stdin) - 키보드	0
표준 출력(stdout) - 모니터	1
표준 에러(stderr) - 모니터	2

표준 출력

test1.txt 파일의 내용을 **ls.txt** 파일에 '>>' 문자를 사용하여 리다이렉션 하여 내용 추가

```
[root@RHEL7-SVR ~]# cat > test1.txt
English Only
^C
[root@RHEL7-SVR ~]# cat ls.txt
English Only
[root@RHEL7-SVR ~]# cat test1.txt >> ls.txt
[root@RHEL7-SVR ~]# cat ls.txt
anaconda-ks.cfg
initial-setup-ks.cfg
ls.txt
script.txt
shelltest.sh
English Only
```

표준 출력

linuxer.txt 파일이 존재하지 않을 때 출력을 수행하면 표준 에러 메시지를 모니터에 출력한다.

```
[root@RHEL7-SVR ~]# cat linuxer.txt
```

```
cat: linuxer.txt: No such file or directory
```

존재하지 않는 **linuxer.txt** 파일을 **cat** 명령으로 출력하면 표준 에러 메시지도 모니터에 출력하는데, 이 때 표준 출력 내용만 **/dev/null** 디바이스로 리다이렉션 했기 때문에 표준 에러 메시지는 모니터에 출력된다.

```
[root@RHEL7-SVR ~]# cat linuxer.txt > /dev/null
```

```
cat: linuxer.txt: No such file or directory
```

2>&1 : 표준 출력(1)으로 표준 에러(2)도 전달하라고 지정했기 때문에, 존재하지 않는 파일을 출력하기 위해 사용한 **cat** 명령에 의한 에러 메시지도 **/dev/null** 디바이스로 전달되어 모니터에는 아무런 메시지도 출력되지 않는다.

```
[root@RHEL7-SVR ~]# cat linuxer.txt > /dev/null 2>&1
```

표준 출력

존재하지 않는 **linuxer.txt** 파일을 삭제하려고 했기 때문에 모니터에 에러 메시지를 출력한다.

```
[root@RHEL7-SVR ~]# rm linuxer.txx
```

```
rm: cannot remove 'linuxer.txx': No such file or directory
```

rm 명령의 출력값이 **rm_error.txt**에 저장되는데, 이 때 **rm** 명령 결과의 표준 에러(2)도 표준 출력(1)으로 리다이렉션하도록 하였으므로 에러 메시지가 **rm_error.txt** 파일에 저장되며 모니터로는 출력되지 않는다.

```
[root@RHEL7-SVR ~]# rm linuxer.txt > rm_error.txt 2>&1
```

```
[root@RHEL7-SVR ~]# cat rm_error.txt
```

```
cat: rm: No such file or directory
```

```
cat: _error.txt: No such file or directory
```

입력

- 입력은 출력과 반대로 < 를 이용한다.
- {} 는 명령어에 범위를 입력할 수 있게 한다.

```
$ printf "%s\n" {{1..3},{a..c}}  
$ printf "%s\n" {A..D}_Grade  
$ printf "%s\n" A1{00..9..3}C
```

- [] 도 범위를 포함하는 파일을 검색할 때 유용[3]하지만 입력에는 아무런 기능을 하지 못한다.
- ~ (tilde) 는 사용자의 홈 디렉토리를 표시한다. printf "%s\n" ~Jacob
- 임시파일을 사용해야 하는 경우에 아래와 같이 입출력 리디렉션을 이용할 수 있다.

```
$ ./cli.sh < (ls -l) > (pr -Tn)
```

- 명령행 인수를 처리하거나 확인해야 할 때 셸 스크립트 내에서 getopt을 사용한다.
- 사용자 입력을 읽을 때는 read 를 사용한다. 입력받은 값의 기본 변수는 \$REPLY이다.

표준 입력

- 표준 입력은 키보드로부터 데이터를 입력 받는 것을 말한다.
- 파일로부터 입력을 받을 수도 있는데, 이런 경우에는 '<' 문자(less than sign 또는 left angle bracket) 를 사용한다.
- 파일 디스크립터 숫자 값으로 0을 사용한다.

sort 명령으로 파일의 내용을 정렬하여 ls.txt 파일에 입력한다.

```
[root@RHEL7-SVR ~]# sort < ls.txt
```

```
anaconda-ks.cfg
```

```
English Only
```

```
initial-setup-ks.cfg
```

```
ls.txt
```

```
...
```

sort 명령으로 정렬한 파일의 내용을 '>' 출력 리다이렉션을 사용하여 다른 파일로 저장한다.

```
[root@RHEL7-SVR ~]# sort < ls.txt > sorted_ls.txt
```

```
[root@RHEL7-SVR ~]# cat sorted_ls.txt
```

```
anaconda-ks.cfg
```

```
English Only
```

```
initial-setup-ks.cfg
```

```
ls.txt
```

```
...
```

변수

- 변수의 지정과 변수의 정의는 간단하다.
- 아래 결과에서 `-r` 을 지정한 NAME은 read only 이기 때문에 값이 변경되지 않고, NAME1의 값은 변경된다.

```
$ FILE="$HOME/sample.txt" ; cat $FILE
$ declare -r NAME="James Dean" ; NAME1="James Dean1"          # -r 옵션을 사용해 readonly로 선언
-bash: NAME1: readonly variable
[root@rhel7 ~]# declare -r NAME="James Dean2" ; NAME1="James Dean3"
-bash: declare: NAME: readonly variable
[root@rhel7 ~]# echo $NAME
James Dean
[root@rhel7 ~]# echo $NAME1
James Dean3
```

- 변수 이름은 반드시 알파벳 또는 "_"로 시작해야 하며 몇 가지 키워드는 변수명으로 사용할 수 없다.
- 지역변수로만 사용하려면 `local` 을 사용한다.

```
local IFS=.
```

변수

- 스크립트의 실행 결과값을 지정할 때 또는 명령의 결과를 치환할 때는 백쿼터 `` 또는 \$()을 이용한다.

```
$ message="java path is $(which java)"
$ echo $message
java path is /usr/bin/java
```

- \$() 은 명령어 실행결과에서 newline을 제거하는데 "\$ ()" 와 같이 사용하면 newline을 제거하지 않는다.
- \$() 은 서브 셸을 사용하기 때문에 자동으로 실행 중인 셸에 정의되어 있는 모든 함수들을 상속받는다.
- \$(()) 는 명령어 치환이 아니라 수치 연산이다. 변수명에 \$을 붙일 필요가 없다.

```
$ first=1
$ second=12
$ echo "sum is $((first+second))"
sum is 13
```

- \${ } 을 사용해 변수를 제대로 파악하기 어려운 경우 구분한다.

```
$ printf "%s\n" "${first}_${last}"          # {} 을 제거하면 first_를 변수명으로 인식한다.
```


변수

- 변수 분할을 위해 추가적인 방법을 사용한다.
 - `${변수명#표현식}` : 시작부터 처음 일치되는 부분을 제거
 - `${변수명##표현식}` : 시작부터 마지막 일치되는 부분을 제거
 - `${변수명%표현식}` : 끝에서부터 처음 일치되는 부분을 제거
 - `${변수명%%표현식}` : 끝에서부터 마지막 일치되는 부분을 제거

```
$ NAME2="James Dean"
$ printf "%s\n" "${NAME2%${NAME2#*m}} ${NAME2#*m}"
Jam es Dean
```

- `${ }` 확장을 이용해서 파일의 전체 경로에서 파일 이름만 안전하게 가져올 수도 있고, 에러를 출력하게 할 수도 있고, Substring을 가져올 수도 있다.

```
$ printf "%s\n" ${FILE_NAME##*/}
$ printf "%s\n" ${NAME1:-"Tom Riddle"}
$ printf "%s\n" ${NAME:5:5}
$ printf "%s\n" ${NAME1//?/*}
```

- 위 두번째 줄에서 `:-` 는 빈 값과 정의되지 않은 변수를 체크한다. `-` 만 있으면 정의되지 않았을 때 체크한다. 또한 `- 대신 =` 을 사용하면 변수를 치환한다. `- 대신 ?` 을 사용하면 오류 메시지를 표시한다. `+` 는 값이 있을 때 변경한다.
- `${#variable}` 은 변수의 길이를 표시한다.
- 대소문자 변환에서 `${var^[a-z]}`, `${var^^[a-z]}`, `${var,,}`, `${var,,[A-Z]}` 를 사용할 수 있다.

변수

- 배열을 다음과 같이 표현 가능하다. declare -A 변수명 은 미리 배열로 선언할 필요가 있을 때 사용한다.

```
$ province=(경기도 충청남도 대전광역시)
```

```
$ printf "%s\n" "${province[@]}"
```

```
경기도
```

```
충청남도
```

```
대전광역시
```

```
$ printf "%s\n" "${province[@]:2:1}"
```

```
대전광역시
```

```
$ printf "%s\n" "${province[2]}"
```

```
대전광역시
```

```
$ printf "%s\n" "${#province[@]}"
```

```
3
```

```
$ printf "%s\n" "${!province[@]}"
```

```
0
```

```
1
```

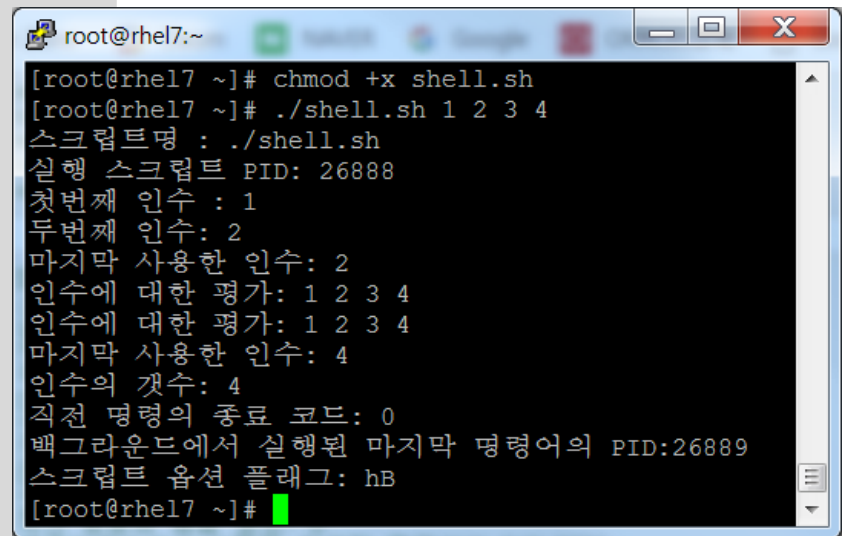
```
2
```

지정 파라미터

- 사용자가 지정하는 변수와 달리 셸 스크립트에서 미리 지정된 파라미터가 있다.
- 아래 내용을 shell.sh 로 저장하고 실행한다.

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

```
echo "스크립트명 :" $0
echo "실행 스크립트 PID:" $$
echo "첫번째 인수 :" $1
echo "두번째 인수:" $2
echo "마지막 사용한 인수:" $_
echo "인수에 대한 평가:" $*
echo "인수에 대한 평가:" @$
echo "마지막 사용한 인수:" $_
echo "인수의 갯수:" $#
sleep 1&
echo "직전 명령의 종료 코드:" $?
echo "백그라운드에서 실행된 마지막 명령어의 PID:"$!
echo "스크립트 옵션 플래그:" $-
```



```
root@rhel7:~
[root@rhel7 ~]# chmod +x shell.sh
[root@rhel7 ~]# ./shell.sh 1 2 3 4
스크립트명 : ./shell.sh
실행 스크립트 PID: 26888
첫번째 인수 : 1
두번째 인수: 2
마지막 사용한 인수: 2
인수에 대한 평가: 1 2 3 4
인수에 대한 평가: 1 2 3 4
마지막 사용한 인수: 4
인수의 갯수: 4
직전 명령의 종료 코드: 0
백그라운드에서 실행된 마지막 명령어의 PID:26889
스크립트 옵션 플래그: hB
[root@rhel7 ~]#
```

지정 파라미터

- 함수
 - declare -F 를 사용하면 현재 정의된 함수를 볼 수 있다.
 - unset -f 함수명 을 이용해 함수를 제거할 수 있다.
- 연산
 - 수치 연산을 위한 기본 방법은 (()) 을 사용한다.
 - 변수 지정을 위해서는 let 을 사용한다.

```
let "i=100*3"
```

- 명령어 확인
 - type은 bash 셸 내장 명령어 인지 알려준다.
 - 외부 명령이면 어디에 있는지 알려준다.

```
$ type -a source
source is a shell builtin
```

```
$ type -a bash
bash is /usr/bin/bash
```

실행

- 다중 실행이 필요할 때 몇 가지 방법이 있다.

```
$ FILE1="$FILE" ; printf $FILE1
sample.txt
$ FILE2="$FILE1" || printf $FILE2

$ FILE3="$FILE3" && printf $FILE3
printf: usage: printf [-v var] format [arguments]
```

- ; 과 || 은 앞선 명령어에 오류가 있어도 다음 명령어가 실행되는 점에서 동일하지만, ; 는 순차적으로 실행되는 반면에 || 그렇지 않다는 점이 다르다.
- 따라서 ||에서 앞 명령문에 지역 변수를 지정하더라도 다음 명령문에서 가져올 수 없다.
- && 은 어떤 명령문이라도 오류가 있으면 오류를 표시하고 중지한다.

파이프 (|)

- 앞 명령문에서 실행된 결과를 가지고 작업해야 할 때가 파이프(|)를 사용한다.
- 명령어들의 조합, 연결하여 사용할 때 '|' (vertical bar) 문자를 사용하여 두 명령어를 연결해 주면 앞에서 실행한 명령의 결과값을 뒤에 적은 명령어의 입력으로 사용하게 된다.
- 즉, 파이프로 연결된 하나의 표준 출력을 다른 명령의 표준 입력으로 사용하는 것이다.

\$ du -h --max-depth 1 sort -hr	#하위 디렉토리의 크기를 역순으로 보여준다.
\$ cat /etc/wgetrc uniq wc -l	#중복(빈줄)을 제거하고 몇줄이 있는지 보여준다.
\$ ls -l grep txt	#조회 결과에서 txt가 들어간 줄을 보여준다.
\$ find . -type f -print wc -l	#현재 디렉토리 하위의 파일 수를 계산한다.

- 파이프에 붙여서 자주 사용하는 것은 다음과 같다.
 - pr : 페이지 단위 보기
 - head : 처음 몇 줄만 출력
 - tail : 마지막 몇 줄만 출력
 - cut : 구분 기호로 분할하여 특정 필드 표시
 - paste : 두 파일의 각 행을 지정한 구분 기호로 결합. 혹은 한 파일의 여러 행을 지정한 구분 기호로 결합
 - tr : 문자의 치환

필터

- 파이프(|)에는 여러 가지 필터를 사용할 수 있다.
- 필터는 표준 입력을 받아서 이 필터로 연산을 한 다음, 그 결과를 표준 출력으로 보낸다.
- 필터의 종류

필터 프로그램	설명
sort	표준 입력에 대해 정렬을 수행하여 그 결과를 표준 출력으로 출력
uniq	표준 입력으로부터 정렬된 데이터를 받아서 중복된 항목을 제거하고 출력
grep	표준 입력으로부터 받은 라인 단위의 데이터로부터 지정한 문자 패턴을 가지고 있는 라인을 찾아서 출력
fmt	표준 입력으로부터 텍스트를 읽고 형식화된 텍스트를 표준 출력으로 출력
pr	표준 입력으로부터 텍스트를 입력 받아서 페이지 단위로 데이터를 잘라서 출력
head	입력된 파일에서 앞의 10개의 라인만 출력
tail	입력된 파일에서 마지막 10개의 라인만 출력
tr	입력된 문자를 변경(대/소문자)하거나, 반복, 삭제하여 출력
sed	스트림 에디터로써 tr 명령보다 다양한 문자 변경을 사용할 수 있다.
awk	강력한 필터

```
[root@RHEL7-SVR ~]# du -h * | sort -nr > $HOME/script.txt
```

```
[root@RHEL7-SVR ~]# ls
```

사진 서식 음악 다운로드 anaconda-ks.cfg script.txt

공개 문서 비디오 바탕화면 initial-setup-ks.cfg

```
[root@RHEL7-SVR ~]# cat script.txt
```

4.0K initial-setup-ks.cfg

4.0K anaconda-ks.cfg

.....

0 script.txt

<해석>

du : 용량 확인 명령

| (파이프) : 결과값을 다음 명령으로 연결하여 다음 명령의 아규먼트로 사용하도록 한다.

sort : 알파벳 또는 숫자를 기준으로 정렬 (-n 옵션 : 숫자값 기준으로 오름차순 정렬 -r 옵션 : 반대의 순서)

> (출력 리다이렉션) : 앞의 명령 결과를 다음에 나오는 파일명으로 저장

Linux Shell Script

Shell Script 의 이해

셸(Shell)과 셸 스크립트(Shell Script)

셸(Shell) 이란

- 운영체제에서 제공하는 명령을 실행하는 프로그램
- 운영체제(OS) 관리하에 있는 파일, 프린팅, 하드웨어 장치, 어플리케이션과의 인터페이스를 제공
- 즉, 운영체제에서 제공하는 각종 명령들을 셸 인터페이스에서 실행하면 운영체제가 그 명령에 해당하는 일을 수행하게 된다.

셸 스크립트(Shell Script) 란

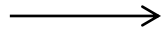
- 인터프리터로서 리눅스 시스템에서 지원하는 명령어들의 집합을 묶어서 프로그램화 한 것
- 기본 명령어들과 함께 if 문, test 문 또는 loop 문 등의 셸 내장명령어(built-in)들을 사용하기도 한다.
- 셸 스크립트는 시스템 관리자의 시스템 관련 작업이나 반복적인 작업들에 있어서 유용하게 사용된다.

본 문서의 실습 진행 방법

1. "예" 에 있는 스크립트 파일 명으로 박스 내용으로 스크립트 작성
2. 스크립트에 권한 부여
3. 스크립트 실행

- 설명
- 예 : test_sharp

```
#!/bin/bash  
스크립트 내용
```



1. # vi test_sharp
내용 작성
2. # chmod +x test_sharp
3. # ./test_sharp

셸 스크립트(Shell Script)의 실행 방법

```
[root@RHEL7-SVR ~]# vi shelltest.sh
#!/bin/bash
echo "This is shell test!"

[root@RHEL7-SVR ~]# sh shelltest.sh

This is shell test!

[root@RHEL7-SVR ~]# bash shelltest.sh

This is shell test!

You have new mail in /var/spool/mail/root

[root@RHEL7-SVR ~]# ./shelltest.sh
-bash: ./shelltest.sh: Permission denied

[root@RHEL7-SVR ~]# chmod +x shelltest.sh

[root@RHEL7-SVR ~]# ./shelltest.sh

This is shell test!
```

셸 스크립트의 실행 방법

1.

/bin/sh 스크립트 파일명

또는

/bin/bash 스크립트 파일명

2.

./스크립트 파일명

이 방법은 실행할 스크립트가 현재 셸에 접근한
사용자에 대해 실행 권한이 주어져야 한다.

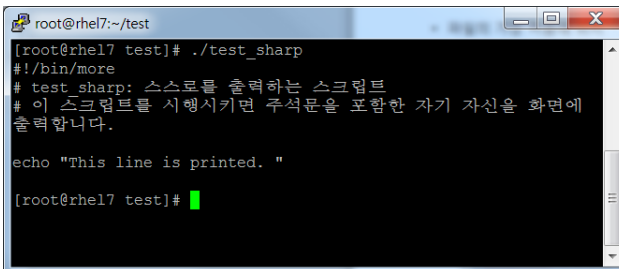
기본적으로 셸 스크립트에 실행 권한 퍼미션을
부여한 다음 실행하는 것이 바람직하다.

스크립트 파일의 구성 요소 - #!

- 매직 넘버
- 파일의 가장 처음에 위치
- 스크립트를 실행할 프로그램 지정
 - 각 셸마다 제공하는 스크립트 언어의 문법이 조금씩 다르므로, 이 스크립트를 실행할 셸을 지정해 주어야 올바르게 실행됨
 - 셸이 아닌, 다른 실행 가능한 명령을 지정해 주어도 됨
- 예 : test_sharp

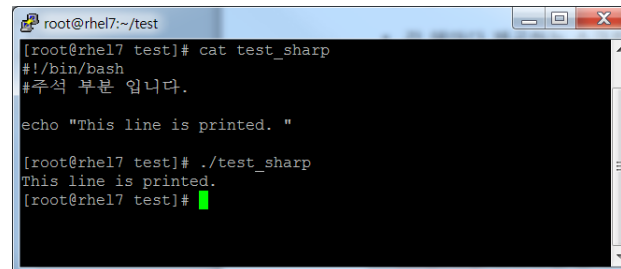
```
#!/bin/more
# test_sharp: 스스로를 출력하는 스크립트
# 이 스크립트를 실행시키면 주석문을 포함한 자기 자신을 화면에 출력합니다.

echo "This line is printed. "
```



```
root@rhel7:~/test
[root@rhel7 test]# ./test_sharp
#!/bin/more
# test_sharp: 스스로를 출력하는 스크립트
# 이 스크립트를 실행시키면 주석문을 포함한 자기 자신을 화면에
출력합니다.

echo "This line is printed. "
[root@rhel7 test]#
```



```
root@rhel7:~/test
[root@rhel7 test]# cat test_sharp
#!/bin/bash
#주석 부분입니다.

echo "This line is printed. "

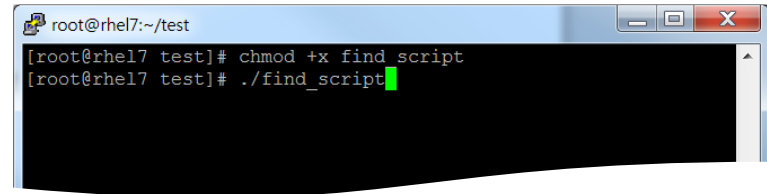
[root@rhel7 test]# ./test_sharp
This line is printed.
[root@rhel7 test]#
```

스크립트 파일의 구성 요소 - 셸 명령

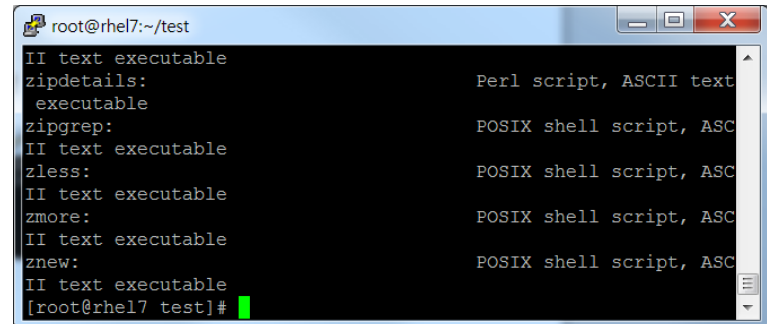
- 셸이 실행할 수 있는 모든 명령어 사용 가능
- 여러 명령을 반복 수행해야 할 때 스크립트 파일로 저장하여 실행
- 예 : find_script

```
#!/bin/bash
# find_script : /bin, /usr/bin 에 있는 셸 스크립트 검색

cd /bin
file * | grep "script"
cd /usr/bin
file * | grep "script"
```



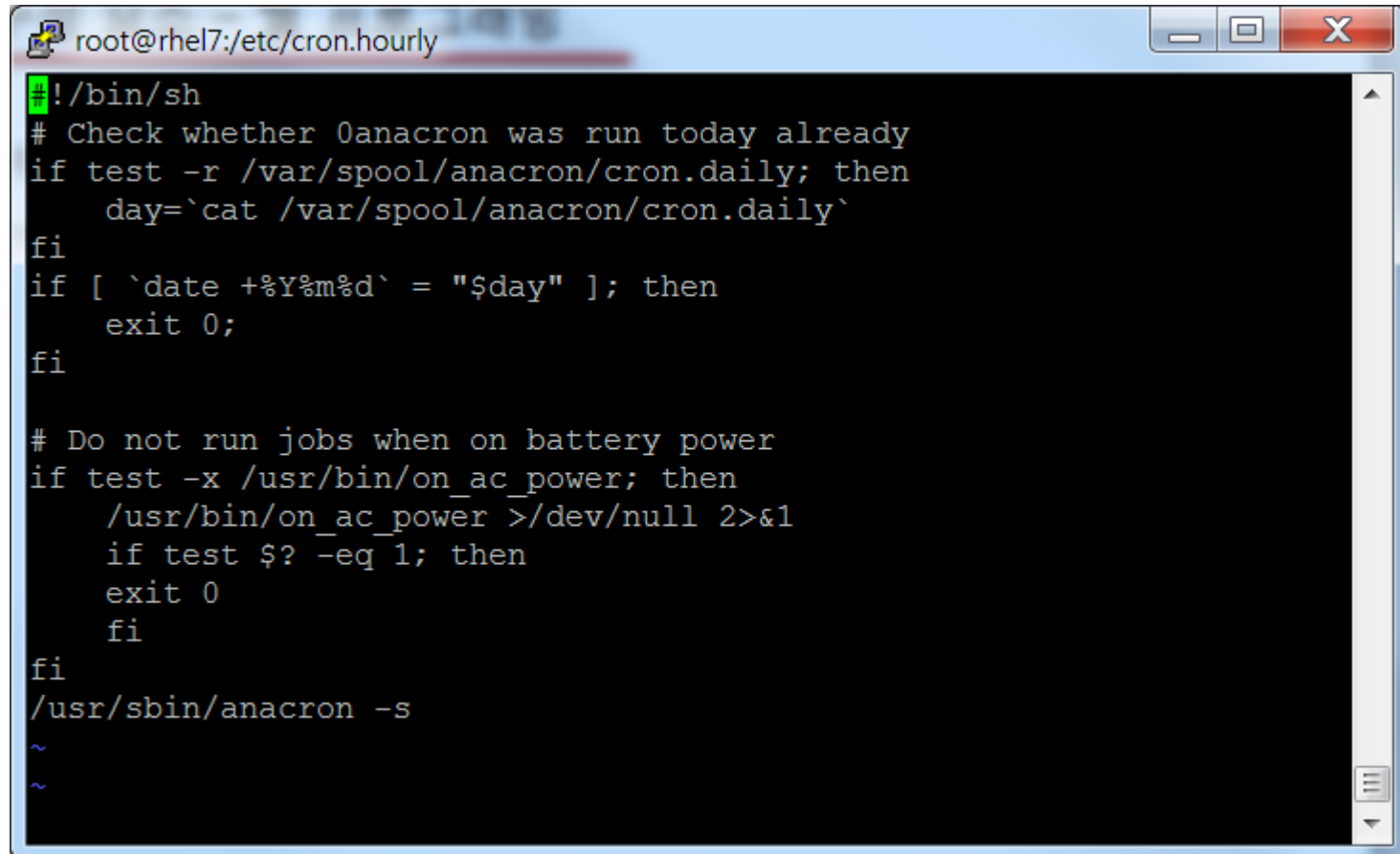
```
root@rhel7:~/test
[root@rhel7 test]# chmod +x find_script
[root@rhel7 test]# ./find_script
```



```
root@rhel7:~/test
II text executable
zipdetails: Perl script, ASCII text
executable
zipgrep: POSIX shell script, ASCII text
II text executable
zless: POSIX shell script, ASCII text
II text executable
zmore: POSIX shell script, ASCII text
II text executable
znew: POSIX shell script, ASCII text
II text executable
[root@rhel7 test]#
```

스크립트 파일의 구성 요소 - 셸 프로그래밍

- 각 셸이 제공하는 프로그램을 위한 구문
- 셸 변수, 인자처리, 각종 연산자, 제어문 등 포함

A terminal window titled 'root@rhel7:/etc/cron.hourly' with standard window controls. The terminal displays a shell script for anacron. The script starts with a shebang line, followed by a comment and a conditional check for anacron's daily log. It then checks if the current date matches the log entry. Another conditional block checks if the system is on battery power and skips jobs if so. Finally, it runs the anacron command with the -s flag.

```
#!/bin/sh
# Check whether 0anacron was run today already
if test -r /var/spool/anacron/cron.daily; then
    day=`cat /var/spool/anacron/cron.daily`
fi
if [ `date +%Y%m%d` = "$day" ]; then
    exit 0;
fi

# Do not run jobs when on battery power
if test -x /usr/bin/on_ac_power; then
    /usr/bin/on_ac_power >/dev/null 2>&1
    if test $? -eq 1; then
        exit 0
    fi
fi
/usr/sbin/anacron -s
~
~
```

Linux Shell Script

Shell 변수 사용하기

셸 변수

- 변수 : 프로그램에서 처리하는 다양한 정보를 저장하는 곳
- 종류
 - 로컬 변수 - 변수가 생성된 셸에만 한정된다.
 - 환경변수(글로벌 변수=전역변수) - 모든 셸에서 사용가능

- 지정 방법

변수명 = 값

- 셸 변수 표현식 (‘:’ 을 사용하지 않으면 널값을 갖고 있어도 변수가 정의된 것으로 간주)

형식	의미
#name	name의 값으로 대체
\$(name)	name의 값으로 대체. 변수 이름이 다른 구문과 인접해 있을 때 사용
\${name:-word}	name이 정의되어 있으면 그 값을, 그렇지 않으면 word값 사용
\${name:=word}	name이 정의되지 않았거나, 널 이라면 word를 대입하고 그 값 사용
\${name:+word}	name이 정의되어 있고, 그 값이 널이 아니라면 word 값을 사용
\${name:?word}	name이 정의되어 있고 널문자가 아니면 그 값을 사용. 그렇지 않으면 word 출력 후 종료

셸 변수 사용하기 – 사용 예

[root@rhel7 test]# test="Shell Test"	-- test 변수에 "Shell Test" 문자열 저장
[root@rhel7 test]# echo \$test	-- test 변수 값 출력
Shell Test	
[root@rhel7 test]# echo \${test:-word}	-- test 변수가 정의되어 있으므로 해당 값 출력
Shell Test	
[root@rhel7 test]# echo \${test1:-word}	-- test1 변수가 없으므로 문자열 word 출력
word	
[root@rhel7 test]# echo \${test:=word}	-- test 변수가 정의되어 있으므로 해당 값 출력
Shell Test	
[root@rhel7 test]# echo \${test1:=word}	-- test1 변수가 없으므로 word를 그 값으로 저장
word	
[root@rhel7 test]# echo \${test1:=word1}	-- test1 변수가 없으므로 위에서 저장되어 있으므로 그 값 출력
word	
[root@rhel7 test]# echo \${test:+word}	-- test 변수가 정의되어 있고 내용이 아니므로 word 출력
word	
[root@rhel7 test]# echo \${test:?word}	-- test 변수가 정의되어 있으므로 해당 값 출력
Shell Test	
[root@rhel7 test]# ^test^test2	-- test2 변수가 없으므로 world 출력 후 스크립트 종료
echo \${test2:?word}	
-bash: test2: word	

셸 변수 문자열 처리

- 변수의 값이 문자열일 때 문자열 내 패턴을 찾아 일부분을 제거하는 표현식

표현식	기능
<code>\${variable%pattern}</code>	variable 값의 뒤부터 패턴과 일치하는 첫 번째 부분을 찾아서 제거
<code>\${variable%%pattern}</code>	variable 값의 뒤부터 패턴과 일치하는 가장 큰 부분을 찾아서 제거
<code>\${variable#pattern}</code>	variable 값의 앞부터 패턴과 일치하는 첫 번째 부분을 찾아서 제거
<code>\${variable##pattern}</code>	variable 값의 앞부터 패턴과 일치하는 가장 큰 부분을 찾아서 제거

셸 변수 문자열 처리 - %,

- % : 뒤에서부터 패턴과 일치하는 최소 부분을 제거(%%는 최대부분)
- %% 사용 시 지정한 패턴이 변수 값 중간에 있다면, 패턴 이후에 임의의 값이 나올 수 있다는 표시로 '*'을 지정한다.
- 예 :

```
[root@rhel7 test]# path1="/usr/bin/local/bin"
```

```
[root@rhel7 test]# echo ${path1%/bin}
```

```
/usr/bin/local
```

```
[root@rhel7 test]# echo ${path1%%bin*}
```

```
/usr/
```

- # : 앞에서부터 패턴과 일치하는 최소 부분을 제거(##은 최대부분)

```
[root@rhel7 test]# path2="/home/user1/.profile"
```

```
[root@rhel7 test]# echo ${path2#/home}
```

```
/user1/.profile
```

```
[root@rhel7 test]# echo ${path2##*/}
```

```
.profile
```

셸 변수 문자열 처리 - #변수

- #변수 : 변수에 저장된 문자의 개수 출력
- 예 :

```
[root@rhel7 test]# echo ${#path1}
```

```
18
```

```
[root@rhel7 test]# echo ${#path2}
```

```
20
```

명령 행 인자 처리

- 명령 행 인자

스크립트를 실행할 때 인자로 주어진 값

- 위치 매개 변수

명령 행 인자를 저장하는 스크립트 변수

인자의 위치에 따라 이름이 정해 짐

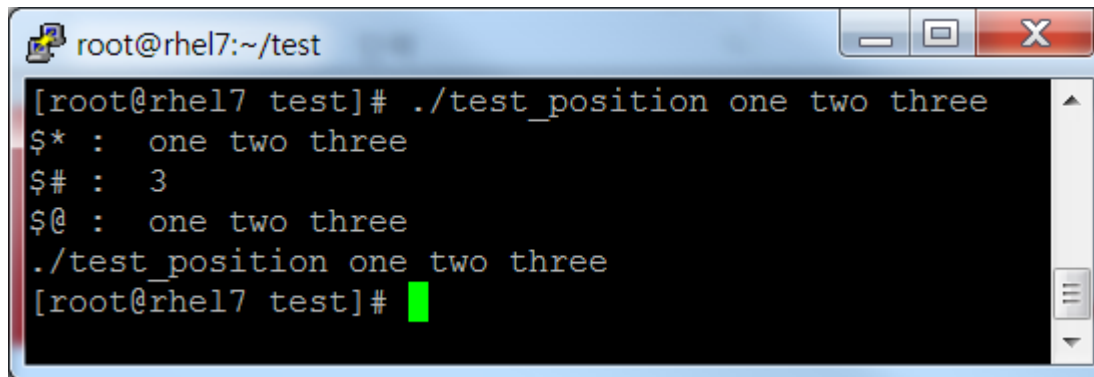
명령행 인자	의미
\$0	셸 스크립트의 이름
\$1 - \$9	명령 행에 주어진 첫 번째부터 9번째까지 인자
\$(10)	10번째 인자
\$#	전체 인자 개수
\$*	모든 인자
\$@	\$*과 같은 의미
"\$*"	"\$1 \$2 \$3"
"\$@"	"\$1" "\$2" "\$3"
\$?	최근 실행된 명령의 종료값

명령 행 인자 처리

- 예 : test_position

```
#!/bin/bash
# test_position

echo '$* : ' $*
echo '$# : ' $#
echo '$@ : ' $@
echo $0 $1 $2 $3
```



```
root@rhel7:~/test
[root@rhel7 test]# ./test_position one two three
$* :  one two three
$# :  3
$@ :  one two three
./test_position one two three
[root@rhel7 test]#
```

셸 특수문자 및 명령 처리

- 인용 부호 : 셸 특수문자의 의미를 없애기 위해 사용

인용 부호	기능	사용법
작은 따옴표 (' ')	모든 특수문자들이 해석되는 것을 막음	<code>\$echo '\$test'</code> <code>\$test</code>
큰 따옴표 (" ")	변수나 명령의 대체만 허용	<code>\$echo "\$test"</code> Shell Test
역슬래시 (\)	단일 문자가 해석되는 것을 막음	<code>\$echo \ \$test</code> <code>\$test</code>

- 명령 대체 : 명령 실행 결과를 문자열로 변환

기호	사용법
백쿼터 (` `)	<code>\$ echo `date`</code> Sunday, April 15, 2012 11:05:06 AM KST
\$(명령)	<code>\$ echo \$(date)</code> Sunday, April 15, 2012 11:15:11 AM KST

사용자로부터 입력 받기 - read

- 셸 내장 명령으로 터미널이나 파일로부터 입력 처리
- 사용 형식

형식	의미
read x	표준입력에서 한 행을 입력 받아 x에 저장
read first last	표준입력에서 한 행을 입력 받아 첫 번째 단어를 first에 저장하고 나머지 모두를 last에 저장
read -p prompt	prompt를 출력하고 입력을 기다린다. 입력된 값은 REPLY 변수에 저장

사용자로부터 입력 받기 - read

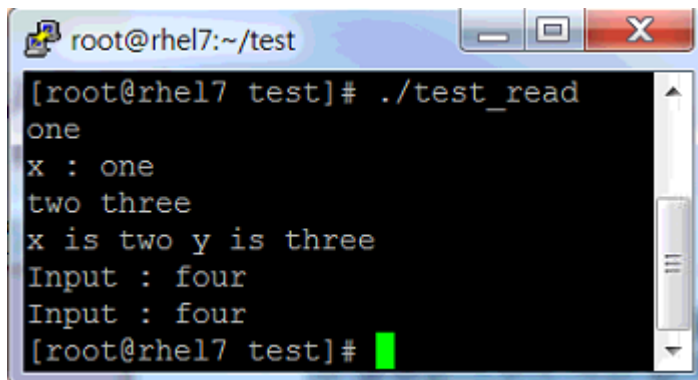
- 예 : test_read

```
#!/bin/bash
# 키보드 입력 처리를 테스트 하는 스크립트

read x                                # 아무 메시지 없이 사용자 입력을 기다림
echo "x : $x"                         # 사용자가 임의의 값을 입력하면 출력

read x y                             # 첫 단어는 x, 나머지는 y에 저장
echo "x is $x y is $y"               # x, y 값 출력

read -p "Input : "                   # Input : 을 출력한 후 입력 기다림
echo "Input : $REPLY"                # $REPLY에 자동 저장된 입력값 출력
```



A terminal window titled 'root@rhel7:~/test' showing the execution of the script. The prompt is '[root@rhel7 test]# ./test_read'. The output is as follows:

```
[root@rhel7 test]# ./test_read
one
x : one
two three
x is two y is three
Input : four
Input : four
[root@rhel7 test]#
```

문서를 통한 입력 – here 문서

- 표준 입력을 사용자로부터 직접 받아들이지 않고 자동 처리
- TERMINATOR 가 입력될 때까지 기술된 부분을 키보드 입력으로 처리
- 키보드 입력의 종료 문자로 사용하는 EOF(^D) 문자를 파일 안에서 사용할 수 없기 때문에 입력 종료를 나타내는 문자열을 지정하여 사용
- 예 : test_here

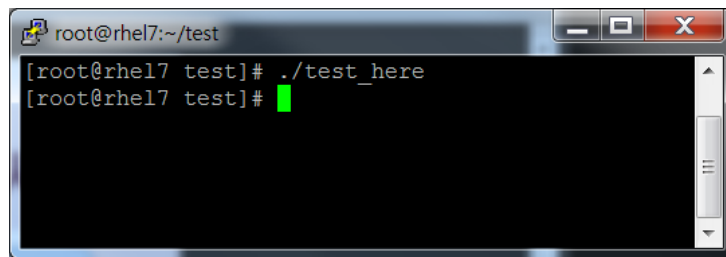
```
#!/bin/bash
```

```
# here 문서 테스트. testuser에게 메일을 보냄
```

```
mail testuser << END
```

```
This is a test mail for here document
```

```
END
```



```
[testuser@rhel7 mail]$ tail /var/spool/mail/testuser
```

```
To: testuser@rhel7.localdomain
```

```
User-Agent: Heirloom mailx 12.5 7/5/10
```

```
MIME-Version: 1.0
```

```
Content-Type: text/plain; charset=us-ascii
```

```
Content-Transfer-Encoding: 7bit
```

```
Message-Id:
```

```
<20190311203131.DA61F253766@rhel7.localdomain>
```

```
From: root@rhel7.localdomain (root)
```

```
This is a test mail for here document
```

Linux Shell Script

연산자

연산자

- 프로그램에서 자료를 처리하는 방법
- 산술 연산자, 비교 연산자, 논리 연산자, 비트 연산자 제공
- 수치 연산자 사용시 let 또는 () 사용해야 함

연산자	의미	사용 예
-	음수(단항연산)	-5
!	논리 부정(not)	((! x < y))
~	비트 반전 (not)	~y
* / %	곱셈, 나눗셈, 나머지 연산	let y=3 * 5
+ -	덧셈, 뺄셈	let x=x+1
<< >>	비트왼쪽 시프트, 비트오른쪽 시프트	((y = x << 3))
<= >= < > == !=	비교 연산	((x < y))
& ^	비트 AND, XOR, OR 연산	let "z = x ^ y"
&&	논리 AND, OR	((x<y x==3))
=	변수 값 지정	let z=1
*= /= %= += -= <<= >>= &= ^= =	단축 연산	let z+=1

연산자

- 예

```
$ a=5
```

```
$ echo $a
```

```
5
```

```
$ let a = 20
```

```
-bash: let: =: syntax error: operand expected (error token is "=")
```

```
$ let "a = 20"
```

```
$ echo $a
```

```
20
```

```
$ (( a = 30 ))
```

```
$ echo $a
```

```
30
```

```
$ a=$a*5
```

```
$ echo $a
```

```
30*5
```

```
$ echo $((5*6))
```

```
30
```

```
$ echo $(( ! 2 + 3 * 4 ))
```

```
12
```

```
$ echo $(( 2 << 1 ))
```

```
4
```

```
$ echo $(( 3 ^ 5 ))
```

```
6
```

----> let에서 공백을 사용 못함

----> 공백을 포함하려면 " " 사용해야 함

----> (())에서는 공백 사용 가능

----> let이나 (())을 사용하지 않으면 문자열로 처리

----> 계산 결과를 바로 출력할 수 있음

----> 우선순위에 따라 ! 먼저 수행. 2는 0이 됨.

----> 왼쪽 shift는 *2와 같음. 2번 shift는 *4

----> XOR 연산 결과

Linux Shell Script

정규 표현식과 패턴 검색

정규 표현식

- 정규 표현식 :

검색에서 사용할 매칭되는 같은 문자들의 패턴

- 메타 문자 :

정규 표현식에서는 문자 그대로의 의미 이상으로 해석되는 '메타문자'라고 부르는 문자들의 집합을 사용

연산자	효과
.	모든 문자 1개와 일치
?	앞에 존재하는 문자가 있을 수도, 없을 수도 있을 경우 사용
*	앞에 존재하는 문자가 0번 혹은 그 이상 반복되는 문자를 찾을 때 사용
+	앞에 존재하는 문자가 1번 혹은 그 이상 반복되는 문자를 찾을 때 사용
[]	대괄호 사이에 존재하는 문자들 중 하나에 일치
[a-z]	a부터 z까지 모든 영문자 소문자와 일치
^	대괄호 사이에 존재할 때는 부정 [^a], 대괄호 밖에서는 문자 열의 시작과 일치
\$	^와 반대로 문자열의 끝과 일치할 경우
{N}	정확히 N번 일치
{N,}	N번 또는 그 이상 일치
{N, M}	적어도 N번 일치하지만 M번 일치를 넘지 않음
\b	단어 끝의 공백 문자열
\B	단어 끝이 아닌 곳에서의 공백 문자열
\w<	단어 시작에서의 공백 문자열을 의미. \w<linux : linux 문자열로 시작하는 단어를 포함한 라인(vi, grep)
\w>	단어 끝에서의 공백 문자열을 의미. linux\w> : linux 문자열로 끝나는 단어를 포함한 라인

정규 표현식

- 확장 브래킷

브래킷	의미
[[:alnum:]]	[A-Za-z0-9] 알파벳 문자와 숫자로 이루어진 문자열
[[:alpha:]]	[A-Za-z] 알파벳 문자
[[:blank:]]	[\\x09] 스페이스와 탭
[[:cntrl:]]	컨트롤 제어 문자
[[:digit:]]	[0-9] 숫자
[[:graph:]]	[!~] 공백이 아닌 문자(스페이스, 제어 문자들을 제외한 문자)
[[:lower:]]	[a-z] 소문자
[[:print:]]	graph와 유사하지만 스페이스 문자를 포함
[[:punct:]]	[!-/:-@[-'{-~] 문장 부호 문자
[[:space:]]	[\\t\\n\\f] 모든 공백 문자(newline 줄바꿈, 스페이스, 탭)
[[:upper:]]	[A-Z] 대문자
[[:xdigit:]]	16진수에서 사용할 수 있는 숫자

vi 에서 정규 표현식을 사용한 검색

- vi에서 ESC키를 누르고 /검색할 문자열 형태를 입력하고 엔터를 누르면 검색 가능
- /enabled\$: enabled로 끝나는 문자열 검색
- /...세 : 4개 문자로 구성된 문자열 중 마지막 문자가 "세"로 끝나는 문자열 검색
- /o*ve : o로 시작되는 문자부터 ve로 끝나는 모든 문자열 검색
- /[L]ove : Love, love

Linux Shell Script

grep 패턴검색

grep

- 입력되는 파일에서 주어진 패턴 목록과 매칭되는 라인을 검색한 다음 표준 출력으로 검색된 라인을 복사해서 출력
- 정렬 관련 옵션을 사용하면 정렬해 출력할 수 있음
- grep의 검색 범위는 메모리 제한을 넘어가지 않는 범위에서 입력 라인의 제한이 없으며 하나의 라인 안의 전체적인 문자들도 매칭할 수 있음
- 입력 파일의 마지막 바트가 newline이 아니라면 grep은 작업을 수행한다.
- grep 메타문자
 - ^ : 라인의 시작
 - \$: 라인의 끝
 - . : 하나의 문자 매칭
 - * : 문자가 없거나 그 이상의 문자들이 매칭
 - [] : []안의 문자 중 하나라도 매칭
 - [^] : []안의 문자 중 하나도 매칭되지 않는 문자

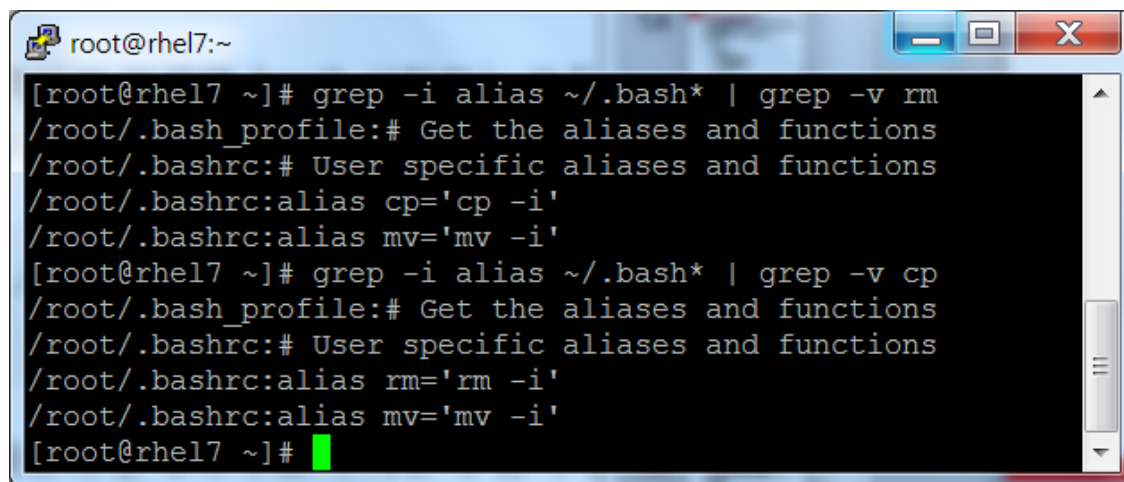
```
grep [옵션] [패턴] [파일명]
```

grep

- 일반 옵션
 - -b : 검색된 라인에 블록 번호를 붙여서 출력
 - -c : 매칭된 라인을 디스플레이하지 않고 매칭된 라인의 수를 출력
 - -h : 파일명은 출력하지 않음
 - -i : 패턴에서 사용되는 문자열에서 대소문자를 모두 검색
 - -l : 패턴에 의해 매칭된 라인이 하나라도 있는 파일의 이름만 출력. 출력시 각 파일명은 newline으로 분리
 - -n : 매칭된 라인을 출력할 때 파일상의 라인 번호를 함께 출력
 - -s : 조용히 진행. 에러 메시지를 출력하지 않음
 - -v : 패턴과 매칭되지 않는 라인만 출력
 - -w : \mathbb{W} <과 \mathbb{W} >로 둘러싸인 하나의 단어를 표현식으로 검색

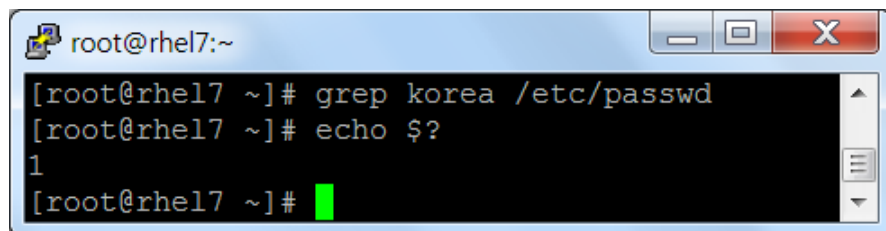
grep 예

- 아래 예에서 /root 디렉토리 아래에 .bash로 시작하는 모든 파일에서 alias라는 문자열을 검색하는데, 이 때 대소문자를 구분하지 않고 모두 검색하기 위해 -i 옵션을 사용하고 있다. 그리고 파이프를 연결하여 결과 값 중 rm 또는 cp 문자열을 포함하지 않는 줄을 출력하기 위해 -v 옵션을 사용하였다.



```
root@rhel7:~  
[root@rhel7 ~]# grep -i alias ~/.bash* | grep -v rm  
/root/.bash_profile:# Get the aliases and functions  
/root/.bashrc:# User specific aliases and functions  
/root/.bashrc:alias cp='cp -i'  
/root/.bashrc:alias mv='mv -i'  
[root@rhel7 ~]# grep -i alias ~/.bash* | grep -v cp  
/root/.bash_profile:# Get the aliases and functions  
/root/.bashrc:# User specific aliases and functions  
/root/.bashrc:alias rm='rm -i'  
/root/.bashrc:alias mv='mv -i'  
[root@rhel7 ~]#
```

- 아래 예에서는 검색한 내용이 존재하지 않기 때문에 검색에 실패했으므로 종료 상태값은 1 이다.



```
root@rhel7:~  
[root@rhel7 ~]# grep korea /etc/passwd  
[root@rhel7 ~]# echo $?  
1  
[root@rhel7 ~]#
```

egrep

- grep의 확장으로서 추가적인 정규표현식 메타문자들을 사용할 수 있음
- egrep에 추가된 메타문자들
 - + : + 앞의 문자 중 하나 이상이 매칭되는 문자
 - ? : 바로 앞의 문자 하나가 없거나 하나가 매칭되는 문자
 - a|b : a 또는 b와 매칭되는 문자(or)
 - () : 문자 그룹

```
# vi testfile
```

Seoul	kimLee	50.5	80.5	50.2
Incheon	Hong	91.5	50.3	60.5
Daejun	Bak	30.2	76.4	88.6
Daegu	Kim root	80.8	50.6	40.9
Ulsan	Lee	80.6	85.3	56.8
Busan	Kang Hong	85.6	91.7	58.3

egrep

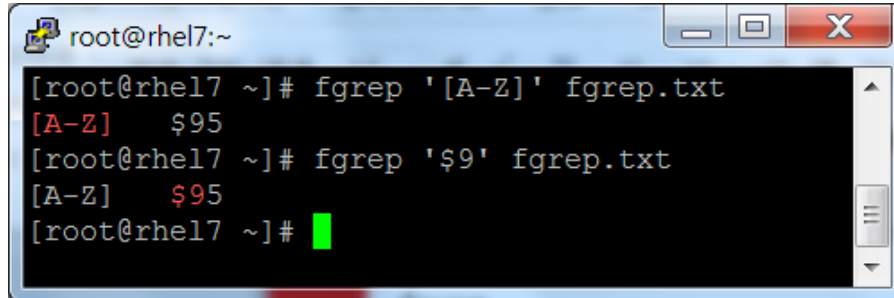
```
root@rhel7:~  
[root@rhel7 ~]# vi testfile  
[root@rhel7 ~]# cat testfile  
Seoul kimLee 50.5 80.5 50.2  
Incheon Hong 91.5 50.3 60.5  
Daejun Bak 30.2 76.4 88.6  
Daegu Kim root 80.8 50.6 40.9  
Ulsan Lee 80.6 85.3 56.8  
Busan Kang Hong 85.6 91.7 58.3  
[root@rhel7 ~]# egrep 'Kim|Kang' testfile  
Daegu Kim root 80.8 50.6 40.9  
Busan Kang Hong 85.6 91.7 58.3  
[root@rhel7 ~]#  
[root@rhel7 ~]# egrep '9+' testfile  
Incheon Hong 91.5 50.3 60.5  
Daegu Kim root 80.8 50.6 40.9  
Busan Kang Hong 85.6 91.7 58.3  
[root@rhel7 ~]#  
[root@rhel7 ~]# egrep '8\.[0-9]' testfile  
Seoul kimLee 50.5 80.5 50.2  
Daejun Bak 30.2 76.4 88.6  
Daegu Kim root 80.8 50.6 40.9  
Ulsan Lee 80.6 85.3 56.8  
Busan Kang Hong 85.6 91.7 58.3  
[root@rhel7 ~]#  
[root@rhel7 ~]# egrep '(Dae)+' testfile  
Daejun Bak 30.2 76.4 88.6  
Daegu Kim root 80.8 50.6 40.9  
[root@rhel7 ~]#
```

```
root@rhel7:~  
[root@rhel7 ~]# egrep 'K(i|a)' testfile  
Daegu Kim root 80.8 50.6 40.9  
Busan Kang Hong 85.6 91.7 58.3  
[root@rhel7 ~]#  
[root@rhel7 ~]# egrep 'sa|u' testfile  
Seoul kimLee 50.5 80.5 50.2  
Daejun Bak 30.2 76.4 88.6  
Daegu Kim root 80.8 50.6 40.9  
Ulsan Lee 80.6 85.3 56.8  
Busan Kang Hong 85.6 91.7 58.3  
[root@rhel7 ~]#
```


fgrep

- Fixed grep of Fast grep
- grep 과 유사하지만 정규 표현식 메타문자들은 사용할 수 없기 때문에 특수 문자 및 \$ 문자들은 문자 그대로 인식한다.

```
# vi fgrep.txt
[A-Z]      $95
B          99
C          66
```



```
root@rhel7:~
[root@rhel7 ~]# fgrep '[A-Z]' fgrep.txt
[A-Z] $95
[root@rhel7 ~]# fgrep '$9' fgrep.txt
[A-Z] $95
[root@rhel7 ~]#
```

- 위 예제의 두 번째 명령에서 사용한 '\$9' 패턴은 fgrep 명령에서 \$문자를 문자 그대로 인식하기 때문에 fgrep.txt 파일의 첫 번째 줄을 검색하여 출력해 준다.

Linux Shell Script

awk

awk

- 데이터를 조작하고 리포트를 생성하기 위해 사용하는 언어이다.
- 리눅스에서 사용하는 awk는 GNU 버전의 gawk로 심볼릭 링크되어있다.
- 간단한 연산자를 명령라인에서 사용할 수 있으며, 큰 프로그램을 위해 사용될 수 있다.
- awk는 데이터를 조작할 수 있기 때문에 셸 스크립트에서 사용되는 필수 툴이며, 작은 데이터베이스를 관리하기 위해서도 필수이다.
- awk 프로그래밍 형식

```
awk 'pattern' filename
awk '{action}' filename
awk 'pattern {action}'
filename
```
- 파일 이름을 지정하지 않으면 키보드 입력에 의한 표준 입력을 받는다.
- 입력된 라인들의 데이터들을 공백 또는 탭을 기준으로 분리해 \$1부터 시작하는 각각의 필드 변수로 분리해 인식한다.

파일로부터의 입력

```
$ vi awkfile
```

```
홍 길동 3324 5/11/96 50354  
임 걱정 5246 15/9/66 287650  
이 성계 7654 6/20/58 60000  
정 약용 8683 9/40/48 365000
```

- '길동'을 포함하고 있는 라인 출력

```
[root@rhel7 ~]# awk '/길동/' awkfile
```

```
홍 길동 3324 5/11/96 50354
```

- 공백을 기준으로 분리되는 필드 중 왼쪽부터 첫 번째로 나오는 필드(\$1)를 출력

```
[root@rhel7 ~]# awk '{print $1}' awkfile
```

```
홍  
임  
이  
정
```

- 시작 문자가 '홍'으로 시작되는 라인을 찾아서 첫 번째 문자와 두 번째 문자 출력

```
[root@rhel7 ~]# awk '/홍/{print $1, $2}' awkfile
```

```
홍 길동
```

명령어로부터의 입력

- 명령어로부터 입력을 받기 위해서 '|' 파이프를 사용할 수 있다.

```
[root@rhel7 ~]# df
```

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/sda3	15717376	8391948	7325428	54%	/
devtmpfs	918632	0	918632	0%	/dev
tmpfs	933632	144	933488	1%	/dev/shm

\$1

\$2

\$3

\$4

\$5

\$6

```
[root@rhel7 ~]# df | awk '$4 > 1000000'
```

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/sda3	15717376	8390880	7326496	54%	/

명령어로부터의 입력

- 명령어로부터 입력을 받기 위해서 '|' 파이프를 사용할 수 있다.

```
[root@rhel7 ~]# df
```

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/sda3	15717376	8391948	7325428	54%	/
devtmpfs	918632	0	918632	0%	/dev
tmpfs	933632	144	933488	1%	/dev/shm

\$1

\$2

\$3

\$4

\$5

\$6

```
[root@rhel7 ~]# df | awk '$4 > 1000000'
```

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/sda3	15717376	8390880	7326496	54%	/

awk 동작 원리

- 다음은 awk 파일이다.

홍	길동	3324	5/11/96	50354
임	걱정	5246	15/9/66	287650
이	성계	7654	6/20/58	60000
정	약용	8683	9/40/48	365000

\$1 \$2 \$3 \$4 \$5

```
[root@rhel7 ~]# awk '{print $1, $3}' awkfile
홍 3324
임 5246
이 7654
정 8683
```

1. awk는 파일 또는 파이프를 통해 입력 라인을 얻어와서 \$0 이라는 내부 변수에 라인을 입력해 둔다. 각 라인은 레코드라고 부르는데, 기본적으로 newline에 의해 구분된다.
2. 다음으로 라인은 공백을 기준으로 각각의 필드나 단어로 나누어진다. 각 필드는 번호가 매겨진 변수에 저장되고 \$1부터 시작한다.
3. awk 는 FS라고 부르는 필드 분리자로 가장 먼저 공백(탭)을 할당받는다. 만약 필드가 콜론(:)이나 대시(-)와 같은 다른 문자에 의해 분리된다면 새로운 필드 분리자로 FS의 값을 변경할 수 있다.
4. awk는 화면에 필드를 출력할 때 print 함수를 사용한다.
5. 옆의 결과를 보면 홍과 3324 사이에 공백이 있다. 이는 명령어에 콤마(,) 가 있기 때문인데, 콤마는 기본으로 공백을 할당받는 출력필드 분리자(OFS) 내장 변수와 매핑되어 있기 때문이다.
6. awk가 화면에 출력하고 나면 다음 라인이 호출되고 \$0으로 저장된다. 이 때 앞에서 변수 \$0에 저장되었던 라인은 덮어쓰기가 된다. 또 다시 공백을 기준으로 필드가 분리되고 처리가 진행된다. 이와 같은 프로세스는 파일의 모든 라인이 처리되기 전까지 계속 반복된다.

print 함수

- awk 명령의 액션 파트는 중괄호({ })로 묶어준다.
- 액션이 지정되지 않고 패턴이 매칭된다면 awk는 매칭된 라인을 모니터에 출력하는 기본 액션을 수행한다.
- print 함수는 포매팅이 필요없이 간단히 출력하는 데 사용된다.
- 좀 더 복잡한 포매팅은 printf, fprintf 함수를 사용하도록 한다.
- print 함수는 {print} 형식으로 awk의 액션 부분에 사용될 수 있다.
- print 함수는 아규먼트로 변수와 계산된 값 또는 문자열 상수를 받는다.
- 문자열은 큰따옴표(" ")로 둘러싸야 한다.
- 콤마(,)는 아규먼트들을 분리하는 데 사용된다.
- 만약 콤마를 사용하지 않으면 아규먼트들은 서로 연결되어 버린다.
- 콤마는 기본값으로 공백을 가지는 OFS의 값을 검사한다.

```
[root@rhel7 ~]# date
2019. 03. 13. (수) 15:04:50 KST
[root@rhel7 ~]# date | awk '{print "Today is " $4 "요일" "\n현재 시간 : "$5}'
Today is (수)요일
현재 시간 : 15:12:06
```


print 함수

- print 함수의 이스케이프 문자

Escape 문자	의미
\b	백스페이스
\f	폼피드
\n	newline 다음 줄
\r	캐리지 리턴
\t	탭
\047	8진수 47
\c	c는 문자를 대표한다.

- OFMT 변수
 - 숫자를 출력할 때 숫자 포맷 제어할 경우 사용. 간단히 printf를 사용할 수도 있지만, OFMT를 지정할 수 있다.
 - default는 %.6g로 소수점 6자리

```
$ awk 'BEGIN{OFMT="%.2f"; print 1.23412}'  
> 1.23
```

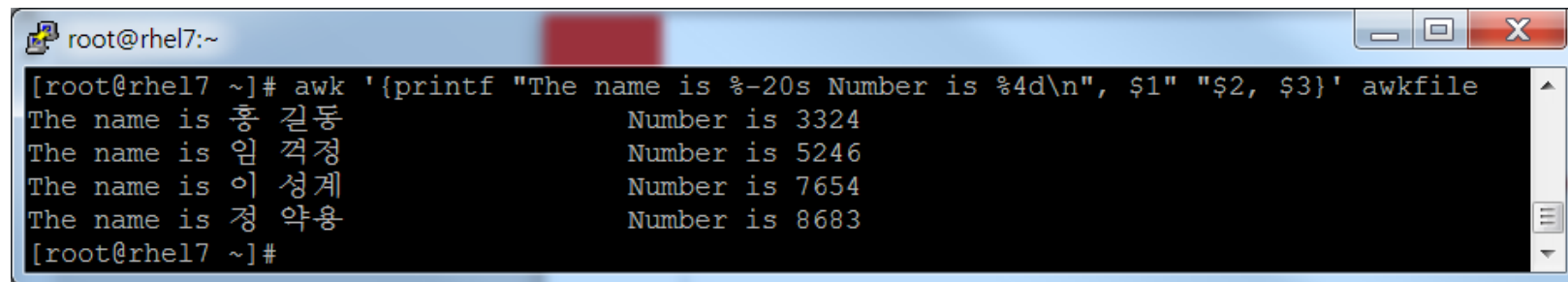
printf 함수

- printf 포맷 문자

변환 문자	정의
c	문자
s	문자열
d	10진수
ld	Long 10진수
u	Unsigned 10진수
lu	Long unsigned 10진수
x	16진수
lx	Long 16진수
o	8진수
lo	Long 8진수
e	지정한 노테이션(표기)에서 실수
f	실수
g	e 또는 f를 사용한 실수로 적어도 공백을 가진다.

printf 함수

- printf 함수는 print 보다 포매팅된 깔끔한 출력을 제공한다.
- printf 함수는 C언어의 printf 문장처럼 표준 출력으로 포매팅된 문자열을 리턴한다.
- printf 문장은 포맷 지시자와 변경자 등의 제어 문자열을 가지고 있다.
- 제어 문자열은 콤마로 분리된 표현식의 목록을 따른다.
- print 함수와 다르게 printf는 newline을 제공하지 않기 때문에 newline이 요구되면 이스케이프 문자"\n"을 사용해야 한다.
- % 기호와 포맷 지정자를 위해 아규먼트를 주어야 한다.
- 문자 % 기호를 출력하기 위해서는 %를 두번(%%) 사용하면 된다.

A terminal window titled 'root@rhel7:~' showing the execution of an awk command. The command uses printf to format output from a file named 'awkfile'. The output shows four lines, each with a name and a number, formatted with specific width and alignment specifiers. The terminal window has a standard Linux desktop interface with window control buttons (minimize, maximize, close) in the top right corner.

```
[root@rhel7 ~]# awk '{printf "The name is %-20s Number is %4d\n", $1, "$2, $3}' awkfile
The name is 홍 길동           Number is 3324
The name is 임 걱정           Number is 5246
The name is 이 성계           Number is 7654
The name is 정 약용           Number is 8683
[root@rhel7 ~]#
```

printf 함수 예

- printf 문자열 포맷에서 -가 붙으면 좌측에서 시작되고 기본형이면 우측에서 시작된다.

```
[root@rhel7 ~]# echo "Linux" | awk '{printf "|%-15s|\n",$1}'
```

```
|Linux      |
```

```
[root@rhel7 ~]# echo "Linux" | awk '{printf "|%15s|\n",$1}'
```

```
|          Linux|
```

- “홍길동” 문자를 %-20s 포맷을 사용하여 20개의 문자를 좌측에서 시작하도록 출력

```
[root@rhel7 ~]# awk '{printf "The name is %-20s number is %4d\n",$1 "$2,$3"}
```

```
awkfile
```

```
The name is 홍길동           number is 3324
```

```
The name is 임걱정           number is 5246
```

```
The name is 이성계           number is 7654
```

```
The name is 정약용           number is 8683
```

awk -f 옵션

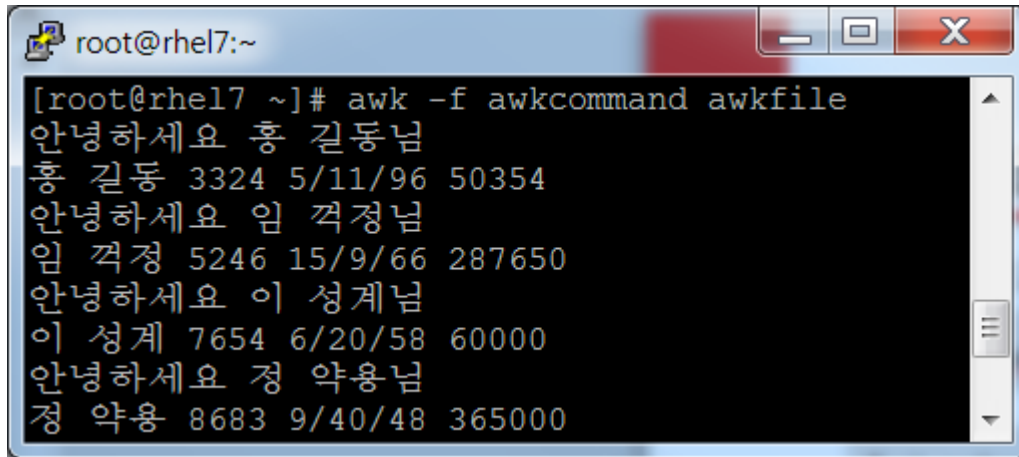
- awk 액션과 명령이 파일에 작성되어 있다면 -f 옵션을 사용

```
awk -f [awk 명령파일] [awk 명령을 적용할 텍스트 파일]
```

```
$ vi awkcommand
```

```
{print "안녕하세요 " $1, $2"님"}
```

```
{print $1, $2, $3, $4, $5}
```



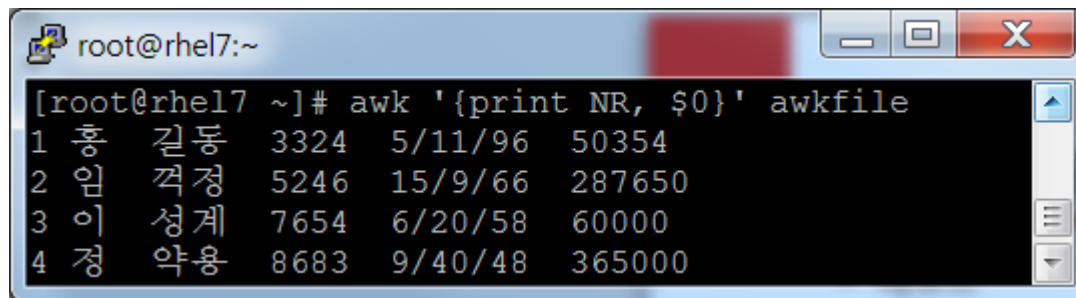
A terminal window titled 'root@rhel7:~' showing the execution of the command `awk -f awkcommand awkfile`. The output displays a greeting followed by five columns of data: name, ID, date, and two numerical values.

```
[root@rhel7 ~]# awk -f awkcommand awkfile
안녕하세요 홍 길동님
홍 길동 3324 5/11/96 50354
안녕하세요 임 걱정님
임 걱정 5246 15/9/66 287650
안녕하세요 이 성계님
이 성계 7654 6/20/58 60000
안녕하세요 정 약용님
정 약용 8683 9/40/48 365000
```

레코드와 필드

- 레코드

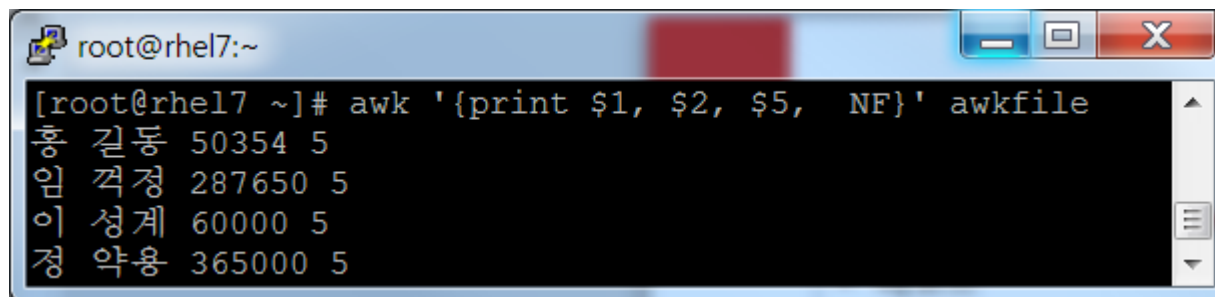
- awk는 입력 데이터를 볼 수 없지만 포맷 또는 구조는 볼 수 있다.
- 레코드라고 불리는 각 라인은 newline으로 분리
- NR 변수 : 각 레코드들의 번호는 awk의 빌트인 변수 NR에 저장된다.
- 레코드가 저장된 다음 NR의 값은 하나씩 증가한다



```
root@rhel7:~  
[root@rhel7 ~]# awk '{print NR, $0}' awkfile  
1 홍길동 3324 5/11/96 50354  
2 임걱정 5246 15/9/66 287650  
3 이성계 7654 6/20/58 60000  
4 정약용 8683 9/40/48 365000
```

- 필드

- 각 레코드는 디폴트로 공백이나 탭으로 분리된 필드라는 워드로 구성된다.
- NF에 필드의 수를 유지하며 라인당 100개의 필드를 가질 수 있다.



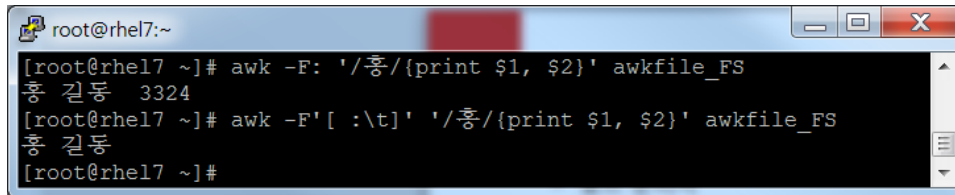
```
root@rhel7:~  
[root@rhel7 ~]# awk '{print $1, $2, $5, NF}' awkfile  
홍길동 50354 5  
임걱정 287650 5  
이성계 60000 5  
정약용 365000 5
```

레코드와 필드

- 필드 분리자
 - 빌트인 변수 FS는 입력 필드 분리자의 값을 가지고 있다.
 - default는 공백과 탭.
 - FS 값을 변경하기 위해선 -F를 사용하며 -F 다음에 오는 문자가 새로운 필드 분리자가 된다.

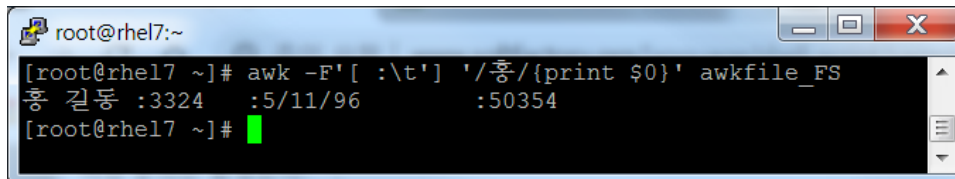
```
$ vi awkfile_FS
```

```
홍 길동 :3324 :5/11/96 :50354
임 걱정 :5246 :15/9/66 :283502
```



```
root@rhel7:~
[root@rhel7 ~]# awk -F: '{print $1, $2}' awkfile_FS
홍 길동 3324
[root@rhel7 ~]# awk -F'[ :\\t]' '{print $1, $2}' awkfile_FS
홍 길동
[root@rhel7 ~]#
```

- -F'[:\\t]' : -F 옵션은 브라켓([]) 안에서 정규표현식을 사용할 수 있는데, 위의 예제에서 공백이나, 콜론(:), 탭을 만나면 이 문자를 필드 분리자로 인식한다. 여기서 작은 따옴표를 사용한 것은, 쉘의 메타문자로 인식하지 않도록 하기 위함이다.



```
root@rhel7:~
[root@rhel7 ~]# awk -F'[ :\\t]' '{print $0}' awkfile_FS
홍 길동 :3324 :5/11/96 :50354
[root@rhel7 ~]#
```

- \$0 변수 : 레코드를 저장하고 있으므로 '홍' 문자열이 검색된 라인을 모두 출력한다.

awk 와 정규 표현식

- 정규 표현식은 슬래시로 둘러싸인 문자들로 구성된 패턴

```
[root@rhel7 ~]# awk '/이 성계 /' awkfile
이 성계 7654 6/20/58 60000
```

- awk에서 지원하는 메타문자

awk 메타 문자	의미
^	문자열의 시작과 매칭
\$	문자열의 끝과 매칭
.	문자 한 개와 매칭
*	문자가 없거나 그 이상과 매칭
+	하나의 문자 또는 그 이상과 매칭
-	문자가 없거나 하나와 매칭
[ABC]	A,B,C 문자셋 중 하나의 문자만 매칭
[^ABC]	A,B,C 문자셋 중 매칭되는 문자가 하나도 없음
[A-Z]	A에서 Z까지의 범위에서 매칭되는 문자가 있음
A B	A또는 B 문자 매칭
(AB)+	AB 문자셋이 하나 이상 매칭 (예)AB, ABAB, ABABAB
₩*	아스테리스크(*) 문자와 매칭
&	검색 문자열에서 검색된 문자열로 대체할 때 사용

awk 와 정규 표현식

- 정규 표현식은 슬래시로 둘러싸인 문자들로 구성된 패턴

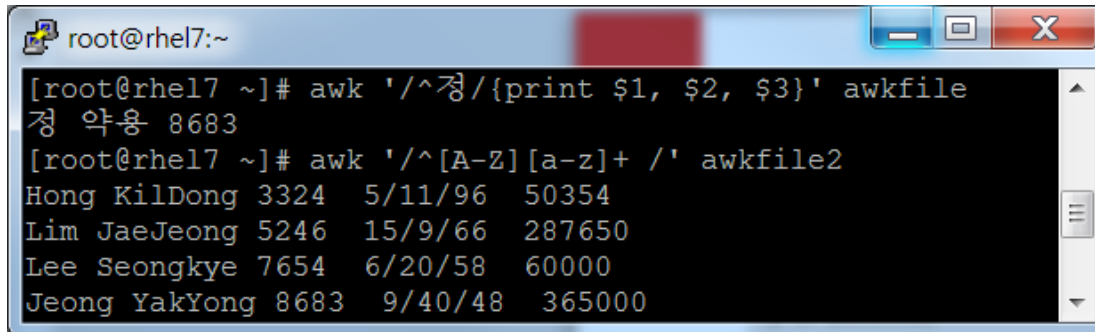
```
$ vi awkfile2
```

```
Hong KilDong 3324 5/11/96 50354
```

```
Lim JaeJeong 5246 15/9/66 287650
```

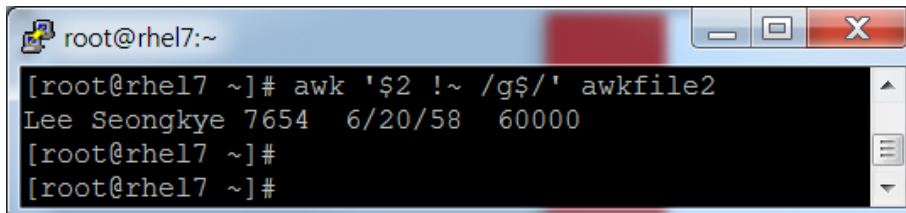
```
Lee Seongkye 7654 6/20/58 60000
```

```
Jeong YakYong 8683 9/40/48 365000
```



```
root@rhel7:~  
[root@rhel7 ~]# awk '/^정/{print $1, $2, $3}' awkfile  
정 약용 8683  
[root@rhel7 ~]# awk '/^[A-Z][a-z]+ /' awkfile2  
Hong KilDong 3324 5/11/96 50354  
Lim JaeJeong 5246 15/9/66 287650  
Lee Seongkye 7654 6/20/58 60000  
Jeong YakYong 8683 9/40/48 365000
```

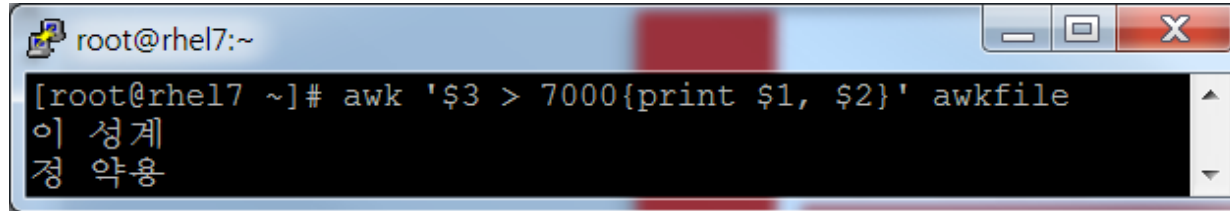
- match 연산자(~) : 표현식과 매칭되는 것이 있는지 검사하는 연산자
- 2번 필드가 g로 끝나지 않는 라인 출력



```
root@rhel7:~  
[root@rhel7 ~]# awk '$2 !~ /g$/' awkfile2  
Lee Seongkye 7654 6/20/58 60000  
[root@rhel7 ~]#  
[root@rhel7 ~]#
```

비교 표현식

- 어떤 상태가 참일 때만 액션이 수행



```
root@rhel7:~  
[root@rhel7 ~]# awk '$3 > 7000{print $1, $2}' awkfile  
이 성계  
정 약용
```

- 조건 표현식
 - 표현식을 검사하기 위해 ?와 :를 사용
 - if/else가 하는 역할과 같은 결과를 의미한다.

```
$ awk '{max=($1 > $2) ? $1 : $2; print max}' filename  
if $1 > $2:  
    max = $1  
else:  
    max = $2
```

- 산술 연산자
 - 계산을 통해 필터링 가능
- 논리 연산자
 - && : AND 연산
 - || : OR 연산
 - ! : NOT 연산

```
$ awk '$3 > $5 && $3 <= 100' filename
```

BEGIN 패턴

- awk가 입력 파일의 라인들을 처리하기 이전에 실행되며 액션 블록 앞에 놓인다.
- 입력 파일 없이 테스트할 수 있고, 빌트인 내장 변수(OFS, RS, FS)들의 값을 변경할 경우 사용한다.

```
[root@rhel7 ~]# cat awkfile
홍 길동 3324 5/11/96 50354
임 걱정 5246 15/9/66 287650
이 성계 7654 6/20/58 60000
정 약용 8683 9/40/48 365000

[root@rhel7 ~]# awk 'BEGIN{FS=":"; OFS="Wt"; ORS="WnWn"}{print $1, $2, $3}' awkfile
홍 길동 3324 5/11/96 50354

임 걱정 5246 15/9/66 287650

이 성계 7654 6/20/58 60000

정 약용 8683 9/40/48 365000
```

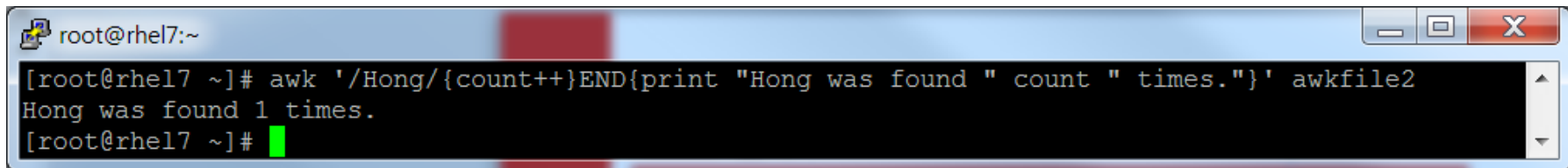
- 위 예제는 입력 파일이 처리되기 전에 필드 분리자(FS)가 콜론(:)으로 설정되고, 출력 필드 분리자(OFS)가 탭으로 설정되며, 출력 레코드 분리자(ORS)가 두 개의 newline으로 설정된다.

END 패턴

- 어떤 입력 라인과도 매칭되지 않는다.
- 하지만 END 패턴과 연관된 액션들을 실행한다.
- END 패턴은 입력의 모든 라인이 처리되고난 후에 처리된다.
- BEGIN만 사용할 경우엔 아규먼트 파일명을 적지 않아도 되지만 END 블록을 사용할 경우엔 반드시 아규먼트 파일을 적어야 한다.

```
[root@rhel7 ~]# awk 'END{print "The number of records is " NR}' awkfile
The number of records is 4
```

- 위의 예에서 END 블록은 awk가 파일 처리를 완료한 다음 실행된다. NR 값은 마지막 레코드를 읽은 다음 몇 번째 라인인지 보여주는 숫자값이다.

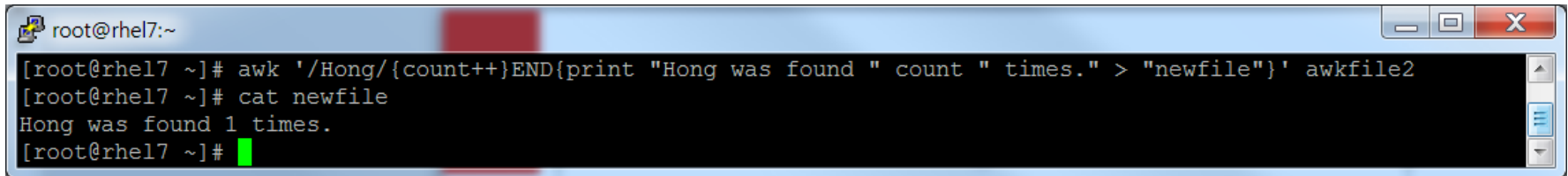
A terminal window titled 'root@rhel7:~' with standard window controls. The command entered is `awk '/Hong/{count++}END{print "Hong was found " count " times."}' awkfile2`. The output is `Hong was found 1 times.` followed by a new prompt line `[root@rhel7 ~]#` with a green cursor.

```
root@rhel7:~  
[root@rhel7 ~]# awk '/Hong/{count++}END{print "Hong was found " count " times."}' awkfile2  
Hong was found 1 times.  
[root@rhel7 ~]#
```

- awkfile2 입력 파일에서 Hong 패턴을 포함하고 있는 모든 입력 라인을 카운트하기 위해 count 변수를 사용한다. 입력 라인들이 읽혀질 때 END 블록은 count의 마지막 결과값을 포함하고 있는 Hong Was found 1 times 문자열을 출력하기 위해 실행된다.

awk 리다이렉션

- awk결과를 리눅스 파일로 리다이렉션할 경우 쉘 리다이렉션 연산자를 사용한다.
- 파일명은 큰따옴표로 묶어야 한다.
- '>' 심볼이 사용될 때 파일이 오픈되고 쓰여진다.

A terminal window titled 'root@rhel7:~' showing the execution of an awk command with redirection. The command is: [root@rhel7 ~]# awk '/Hong/{count++}END{print "Hong was found " count " times." > "newfile"}' awkfile2. The output is: [root@rhel7 ~]# cat newfile
Hong was found 1 times.
[root@rhel7 ~]#

```
root@rhel7:~  
[root@rhel7 ~]# awk '/Hong/{count++}END{print "Hong was found " count " times." > "newfile"}' awkfile2  
[root@rhel7 ~]# cat newfile  
Hong was found 1 times.  
[root@rhel7 ~]#
```

- getline 함수
 - 표준 입력, 파이프, 현재 처리되고 있는 파일로부터 입력을 읽기 위해 사용
 - 입력의 다음 라인을 가져와 NF, NR, FNR 빌트인 변수를 설정
 - 레코드가 검색되면 1을 리턴하고, 파일의 끝이면 0을 리턴. 에러가 발생하면 -1을 리턴

```
[root@rhel7 ~]# awk 'BEGIN{"date" | getline d; print d}'  
2019. 03. 13. (수) 18:39:49 KST
```

- 위 예는 date 함수를 실행하고 결과를 파이프로 연결한 다음,getline으로 얻어온 값을 사용자 정의형 변수인 d로 할당하고 d의 값을 출력한다.

```
[root@rhel7 ~]# awk 'BEGIN{"date " | getline d; split(d, year); print year[1]}'  
2019.
```

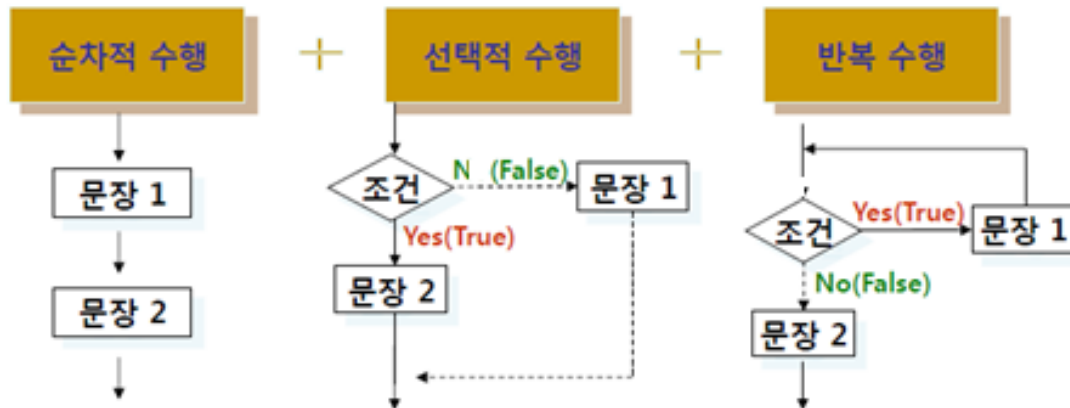
- date 명령 실행 → 결과 값이 d 변수로 할당 → split 함수에 의해 d 변수에서 year 배열 생성 → year 배열의 첫번째 요소 출력

Linux Shell Script

제어문

제어문

- 프로그램내의 문장 실행 순서를 제어하는 것
- 선택적 실행문
 - 프로그램 실행문을 조건에 따라 선택적으로 실행
 - if, select
- 반복 실행문
 - 프로그램 실행문을 정해진 횟수나 조건에 따라 반복 실행
 - while, do , for



선택적 실행문 - if~then~else

- 주어진 조건의 참, 거짓여부에 따라 명령 실행
- 조건 명령을 실행하여 그 실행 값이 0이 아닌 값이면 then 다음의 명령을 실행하고, 0이면 else 다음의 명령을 실행한다.

```
if 조건명령
then
    명령
[ else
    명령 ]
fi
```

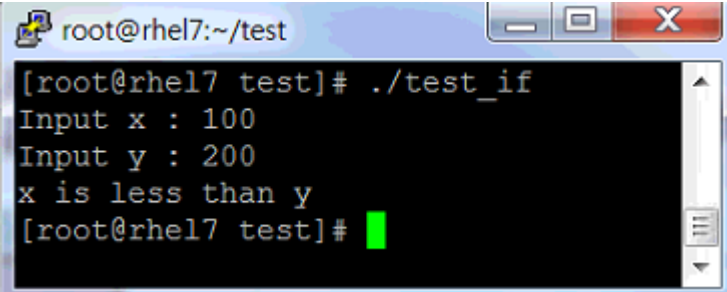
- 예 : test_if

```
#!/bin/bash
# test_if : if 문을 테스트하는 스크립트

echo -n "Input x : "
read x                # x 값을 입력 받음

echo -n "Input y : "
read y                # y 값을 입력 받음

if (( x < y ))
then
    echo "x is less than y"
else
    echo "y is less than x"
```

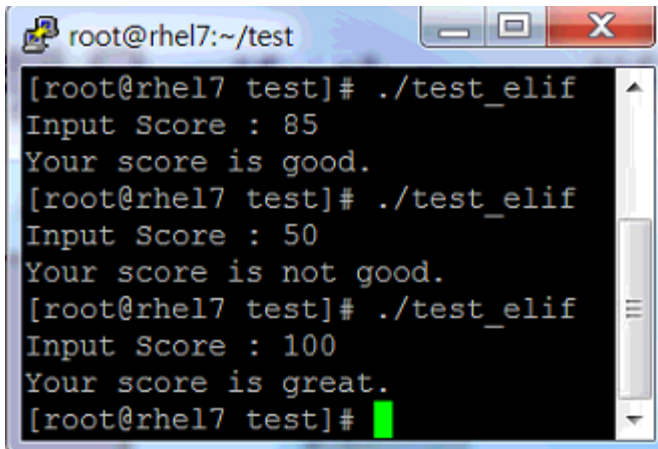


A terminal window titled 'root@rhel7:~/test' showing the execution of the script. The prompt is '[root@rhel7 test]# ./test_if'. The output shows 'Input x : 100', 'Input y : 200', and 'x is less than y'. The prompt returns to '[root@rhel7 test]#' with a green cursor.

선택적 실행문 - if~then~elif ~ else

- 조건이 실패할 때 새로운 분기 명령 실행

```
if 조건명령1
then
    명령
elif 조건명령2
then
    명령
else
    명령
fi
```



```
root@rhel7:~/test
[root@rhel7 test]# ./test_elif
Input Score : 85
Your score is good.
[root@rhel7 test]# ./test_elif
Input Score : 50
Your score is not good.
[root@rhel7 test]# ./test_elif
Input Score : 100
Your score is great.
[root@rhel7 test]#
```

- 예 : test_elif

```
#!/bin/bash
# test_elif: if-elif 문 테스트

echo -n "Input Score : "
read score

if (( $score > 90 ))
then
    echo "Your score is great. "
elif (( $score >= 80 ))
then
    echo "Your score is good. "
else
    echo "Your score is not good. "
fi
```

조건 테스트 - 문자열 연산자

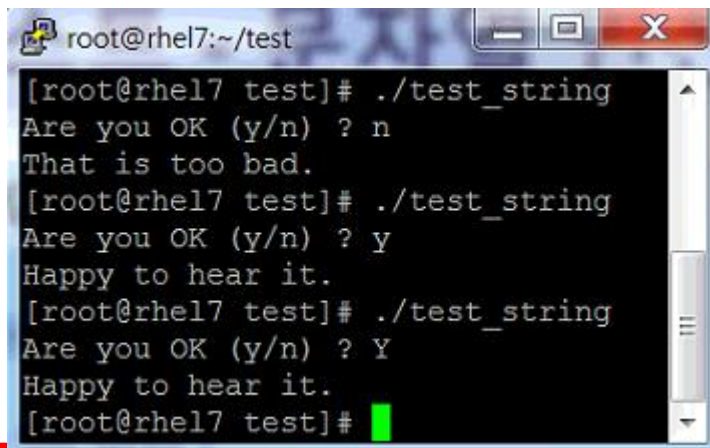
- 조건 명령에 사용하는 문자열 연산자
- 내장 명령 [[]] 사용

문자열 연산자	동작
string = pattern string == pattern	string 이 pattern 과 일치 = 연산자 양쪽에 공백
string != pattern	string 이 pattern 과 일치하지 않음
string	string 이 널이 아님
-z string	string 의 길이가 0
-n string	string 의 길이가 0 이 아님
-l string	string 의 길이

조건 테스트 - 문자열 연산자

- 예 : test_string

```
#!/bin/bash
# test_string: 문자열 테스트 스크립트
echo -n "Are you OK (y/n) ? "
read ans                                # ans 변수에 값 저장
if [[ $ans = [Yy]* ]]                  # y로 시작하는 문자열인가
then
    echo Happy to hear it.             # y로 시작하면
else
    echo That is too bad.              # y로 시작하지 않으면
fi
```



```
root@rhel7:~/test
[root@rhel7 test]# ./test_string
Are you OK (y/n) ? n
That is too bad.
[root@rhel7 test]# ./test_string
Are you OK (y/n) ? y
Happy to hear it.
[root@rhel7 test]# ./test_string
Are you OK (y/n) ? Y
Happy to hear it.
[root@rhel7 test]#
```

조건 테스트 – test 플래그 (1/3)

- 파일 관련 테스트 (1/2)

test 플래그	기능
-a file	파일이 존재
-e file	파일이 존재
-L file	심볼릭 링크 파일
-O file	사용자가 file 의 소유자
-G file	파일의 그룹 ID 가 스크립트의 그룹 ID 와 같음
-S file	소켓 파일
-r file	읽기 가능
-w file	쓰기 가능
-x file	실행 가능
-b file	블록장치 특수파일
-c file	문자장치 특수파일
-d file	디렉토리 파일
-p file	파이프 파일
-u file	setuid 권한 부여 파일
-g file	setgid 권한 부여 파일
-k file	sticky bit 접근 권한 부여 파일
-s file	파일의 크기가 0 이 아님

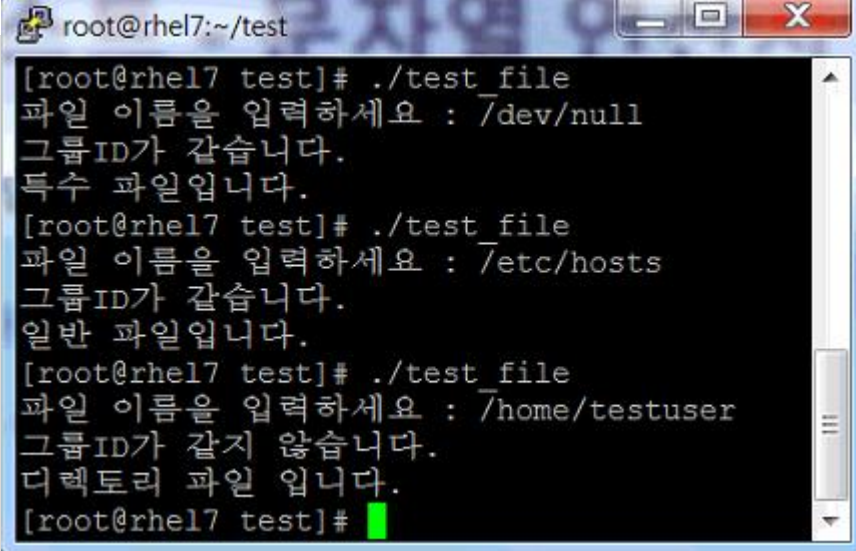
조건 테스트 – test 플래그 (3/3)

- 예 : test_file

```
#!/bin/bash
# test_file: 파일 연산자 테스트

echo -n "파일 이름을 입력하세요 : "
read file
if [[ -G $file ]]
then
    echo 그룹ID가 같습니다.
else
    echo 그룹ID가 같지 않습니다.
fi

if [[ ! -a $file ]]
then
    echo 파일이 존재하지 않습니다. 파일 이름을 다시 확인하세요.
elif [[ -f $file ]]
then
    echo 일반 파일입니다.
elif [[ -d $file ]]
then
    echo 디렉토리 파일 입니다.
else
    echo 특수 파일입니다.
fi
```



```
root@rhel7:~/test
[root@rhel7 test]# ./test_file
파일 이름을 입력하세요 : /dev/null
그룹ID가 같습니다.
특수 파일입니다.
[root@rhel7 test]# ./test_file
파일 이름을 입력하세요 : /etc/hosts
그룹ID가 같습니다.
일반 파일입니다.
[root@rhel7 test]# ./test_file
파일 이름을 입력하세요 : /home/testuser
그룹ID가 같지 않습니다.
디렉토리 파일 입니다.
[root@rhel7 test]#
```

선택적 실행문 - case 문 (1/2)

- 주어진 변수의 값에 따라 실행할 명령 따로 지정
- 변수의 값이 value1 이면 value1부터 ;;을 만날 때까지 명령 실행
- 값의 지정에 특수기호, | (or 연산자) 사용 가능
- 일치하는 값이 없으면 기본값인 * 부터 실행

```
case 변수 in
value1)
    명령 ;;
value2)
    명령 ;;
*)
    명령 ;;
esac
```

선택적 실행문 - case 문 (2/2)

- 예 : test_case

```
#!/bin/bash
# case 테스트 스크립트
```

echo -n 명령을 선택하세요 :

```
read cmd
```

```
case $cmd in
```

```
[0-9])
```

```
    date
```

```
    ;;
```

```
"cd" | "CD")
```

```
    echo $HOME
```

```
    ;;
```

```
[aA-C]*)
```

```
    pwd
```

```
    ;;
```

```
*)
```

```
    echo Usage : 명령을 선택하세요
```

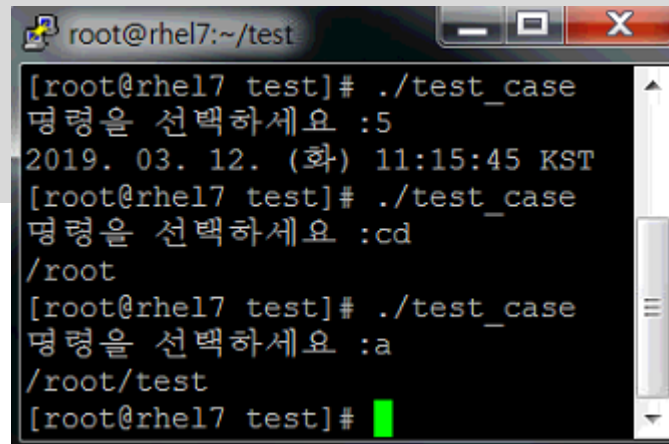
```
    ;;
```

```
esac
```

0부터 9까지 임의의 숫자

cd 또는 CD

소문자 a, 대문자 A,B,C로 시작하는 임의의 문자열



```
root@rhel7:~/test
[root@rhel7 test]# ./test_case
명령을 선택하세요 :5
2019. 03. 12. (화) 11:15:45 KST
[root@rhel7 test]# ./test_case
명령을 선택하세요 :cd
/root
[root@rhel7 test]# ./test_case
명령을 선택하세요 :a
/root/test
[root@rhel7 test]#
```

반복 실행문 - for

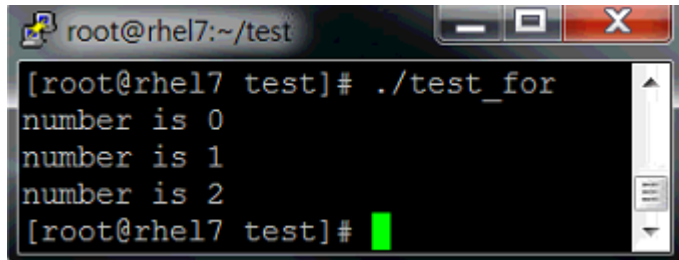
- 리스트 안의 각 값들에 대해 지정한 명령을 순차 실행
- list 대신 \$(<file)을 사용하면 외부 파일의 내용을 입력으로 받아서 처리

```
for 변수 in list
do
    명령
done
```

- 예 : test_for

```
#!/bin/bash
# test_for: for 테스트 스크립트

for num in 0 1 2
do
    echo number is $num
done
```

A terminal window titled 'root@rhel7:~/test' showing the execution of the script './test_for'. The output displays 'number is 0', 'number is 1', and 'number is 2' on separate lines, followed by a green cursor at the prompt '[root@rhel7 test]#'.

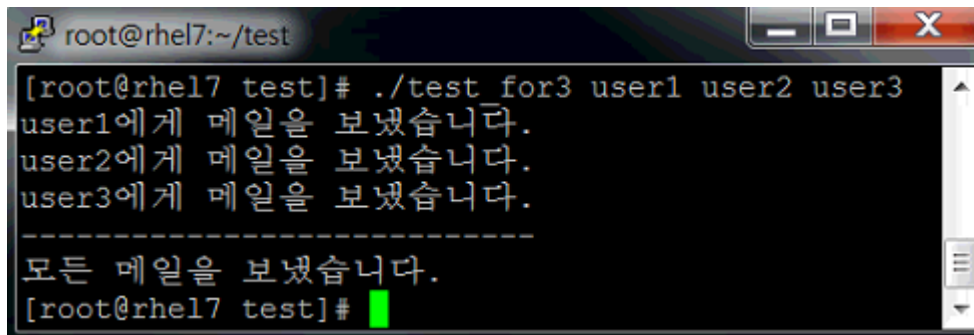
```
root@rhel7:~/test
[root@rhel7 test]# ./test_for
number is 0
number is 1
number is 2
[root@rhel7 test]#
```


반복 실행문 - for

- 명령행 인자 처리 가능
- 예 : test_for3

```
#!/bin/bash
# test_for3: 명령행 인자 처리

for person in $*
do
    mailx $person < letter
    echo ${person}에게 메일을 보냈습니다.
done
echo -----
echo 모든 메일을 보냈습니다.
```



```
root@rhel7:~/test
[root@rhel7 test]# ./test_for3 user1 user2 user3
user1에게 메일을 보냈습니다.
user2에게 메일을 보냈습니다.
user3에게 메일을 보냈습니다.
-----
모든 메일을 보냈습니다.
[root@rhel7 test]#
```

반복 실행문 - while

- 조건 명령이 정상 실행되는 동안 명령 반복

```
while 조건명령
do
    명령
done
```

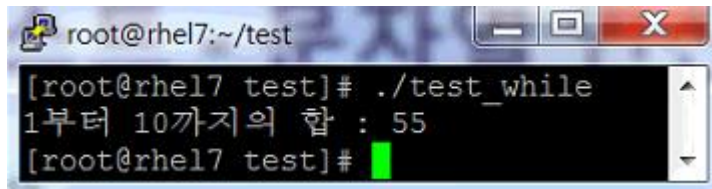
- 예 : test_while

```
#!/bin/bash
# test_while: while을 이용해 1부터 10까지 합을 구하는 스크립트

count=1 sum=0

while (( count <= 10 ))
do
    (( sum += count ))
    let count+=1
done

echo 1부터 10까지의 합 : $sum
```

A terminal window titled 'root@rhel7:~/test' showing the execution of the script. The prompt is '[root@rhel7 test]# ./test_while', followed by the output '1부터 10까지의 합 : 55'. The prompt returns to '[root@rhel7 test]#' with a green cursor.

```
root@rhel7:~/test
[root@rhel7 test]# ./test_while
1부터 10까지의 합 : 55
[root@rhel7 test]#
```

반복 실행문 - until

- 조건 명령이 정상 실행되는 동안 명령 반복

```
until 조건명령
do
    명령
done
```

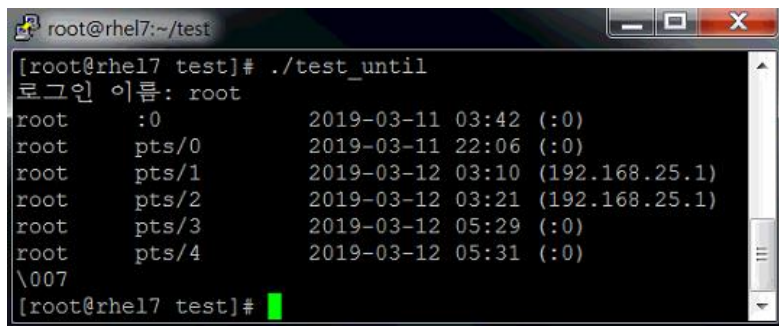
- 예 : test_until

```
#!/bin/bash
# test_until: 지정한 사용자가 로그인하면 알리는 스크립트

echo -n "로그인 이름: "
read person

until who | grep $person
do
    sleep 5
done

# 입력 안내문 출력
# 유저 이름을 person에 저장
# > /dev/null
# 유저가 접속 중이 아니면 5초 쉼
```



```
root@rhel7:~/test
[root@rhel7 test]# ./test_until
로그인 이름: root
root      :0          2019-03-11 03:42 (:0)
root      pts/0       2019-03-11 22:06 (:0)
root      pts/1       2019-03-12 03:10 (192.168.25.1)
root      pts/2       2019-03-12 03:21 (192.168.25.1)
root      pts/3       2019-03-12 05:29 (:0)
root      pts/4       2019-03-12 05:31 (:0)
\007
[root@rhel7 test]#
```

반복 실행문 - select

- 메뉴를 생성할 수 있는 반복 실행문
- list에 지정한 항목을 자동으로 선택 가능한 메뉴로 만들어 화면에 출력해줌
- 사용자는 각 항목에 자동 부여된 번호를 선택
- 사용자 입력은 select와 in 사이에 지정된 변수에 저장
- 보통 case 문과 결합하여 입력 값 처리

```
select 변수 in list  
do  
  명령  
done
```

- 변수 : 사용자 입력 값 저장
- list : 자동메뉴 생성목록
- 명령 : 일반적으로 CASE문 사용

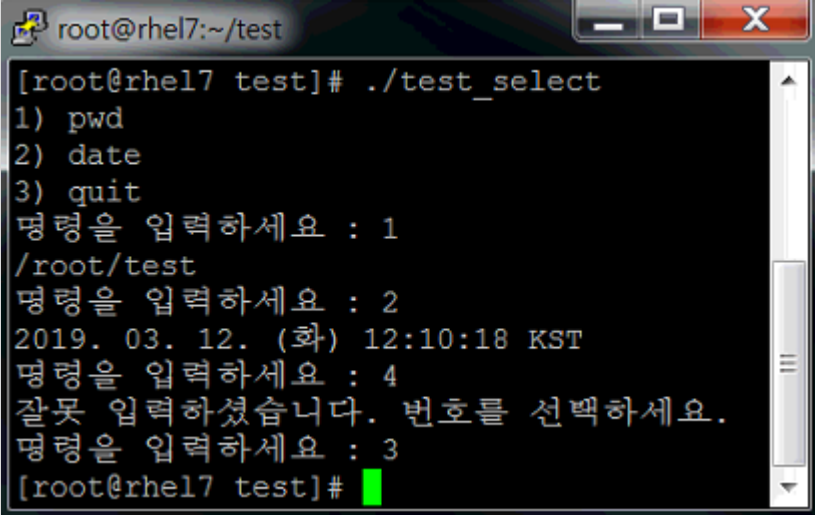
반복 실행문 - select

- 예 : test_select

```
#!/bin/bash
# test_select: 사용자 입력에 따라 pwd,date 명령실행
```

```
PS3="명령을 입력하세요 : "
```

```
select cmd in pwd date quit
do
    case $cmd in
        pwd)
            pwd
            ;;
        date)
            date
            ;;
        quit)
            break
            ;;
        *)
            echo 잘못 입력하셨습니다. 번호를 선택하세요.
            ;;
    esac
done
```



```
root@rhel7:~/test
[root@rhel7 test]# ./test_select
1) pwd
2) date
3) quit
명령을 입력하세요 : 1
/root/test
명령을 입력하세요 : 2
2019. 03. 12. (화) 12:10:18 KST
명령을 입력하세요 : 4
잘못 입력하셨습니다. 번호를 선택하세요.
명령을 입력하세요 : 3
[root@rhel7 test]#
```

루프 제어문 - continue

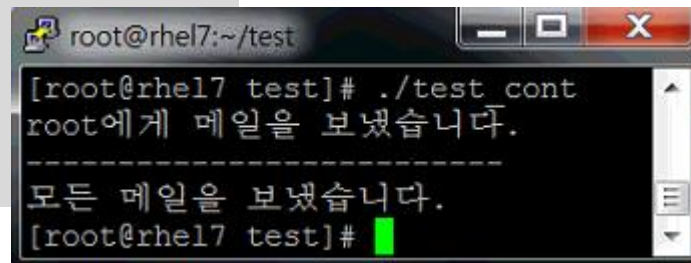
- 루프 안에서 사용
- 이후 실행 순서를 무시하고 루프의 처음으로 돌아가는 명령
- 숫자를 지정하면 중첩된 루프 안에서 특정 루프의 처음으로 돌아갈 수 있음
- 예 : test_cont

```
#!/bin/bash
# test_cont: continue 테스트

for person in $(< list)    # `cat list`와 동일
do
    if [[ $person == user2 ]]
    then
        continue          # user2 이면 건너뛰
    fi

    mailx -s "continue test" $person < letter
    echo ${person}에게 메일을 보냈습니다.
done

echo -----
echo 모든 메일을 보냈습니다.
```

A terminal window titled 'root@rhel7:~/test' showing the execution of the script. The prompt is '[root@rhel7 test]# ./test_cont'. The output shows 'root에게 메일을 보냈습니다.' followed by a separator line '-----' and then '모든 메일을 보냈습니다.'. The prompt returns to '[root@rhel7 test]#'.

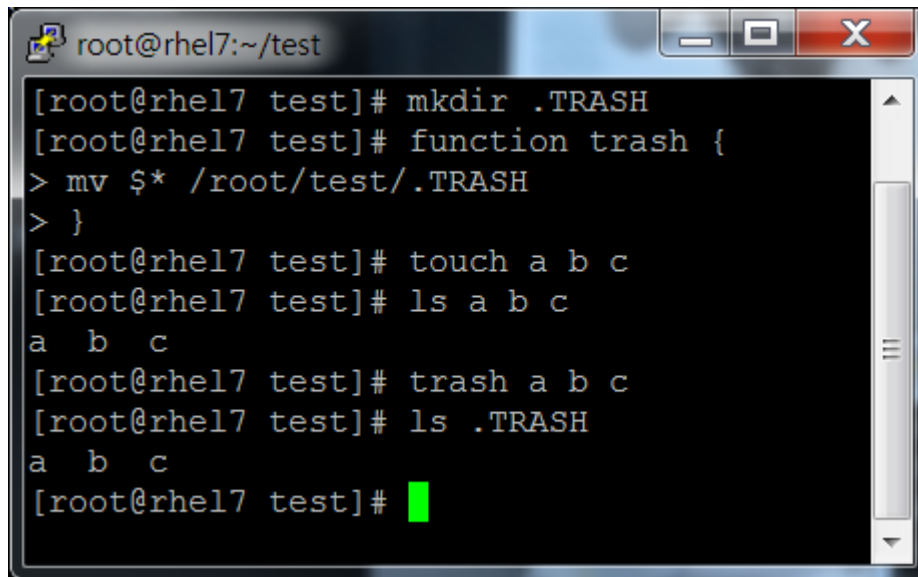
```
root@rhel7:~/test
[root@rhel7 test]# ./test_cont
root에게 메일을 보냈습니다.
-----
모든 메일을 보냈습니다.
[root@rhel7 test]#
```

함수

- 함수 : 하나의 목적으로 사용되는 명령들의 집합
- Alias와의 차이점
 - 조건에 따라 처리 가능
 - 인자 처리 가능

```
function 함수이름
{
    명령들
}
```

- 예 : trash



```
root@rhel7:~/test
[root@rhel7 test]# mkdir .TRASH
[root@rhel7 test]# function trash {
> mv $* /root/test/.TRASH
> }
[root@rhel7 test]# touch a b c
[root@rhel7 test]# ls a b c
a b c
[root@rhel7 test]# trash a b c
[root@rhel7 test]# ls .TRASH
a b c
[root@rhel7 test]#
```

정의된 함수 확인

```
# typeset -f
.....
trash()
{
    mv $* /root/test/.TRASH
}
```

함수의 종료 - return

- 함수 종료 조건
 - 함수 본문 안의 마지막 문장 실행
 - return 문 실행

```
return [n]
```

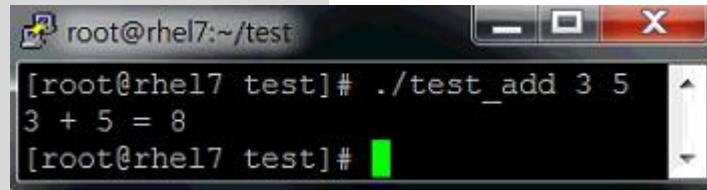
- 지정한 값이 함수의 종료값으로 \$?에 저장됨
- 예 : test_add

```
#!/bin/bash
# 함수 리턴값 테스트

function sum
{
    typeset sum

    (( sum = $1 + $2 ))
    return $sum
}

sum $1 $2
echo $1 + $2 = $?
```

A terminal window titled 'root@rhel7:~/test' showing the execution of a script. The prompt is '[root@rhel7 test]# ./test_add 3 5'. The output is '3 + 5 = 8'. The next prompt is '[root@rhel7 test]#', followed by a green cursor. The window has standard Linux window controls (minimize, maximize, close) in the top right corner.

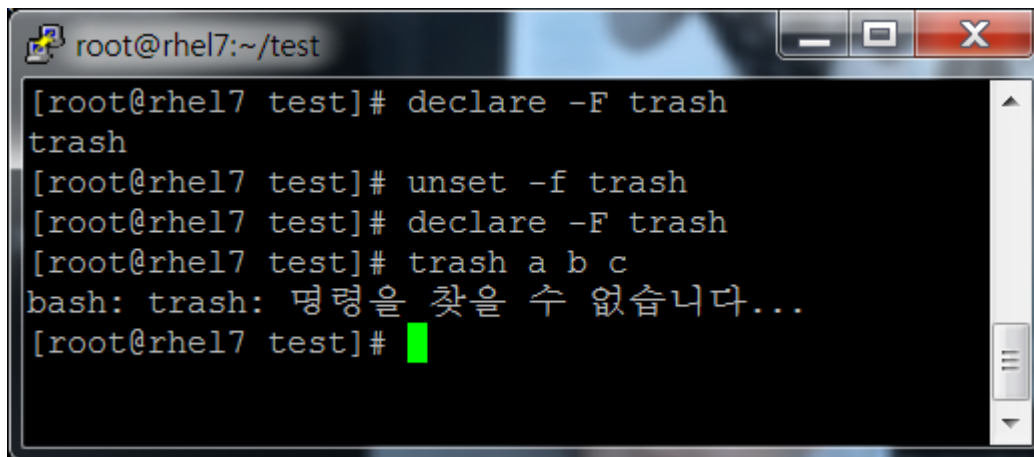
```
root@rhel7:~/test
[root@rhel7 test]# ./test_add 3 5
3 + 5 = 8
[root@rhel7 test]#
```


함수의 목록과 제거 – declare, unset

- declare : 함수 목록과 함수 정의를 하는 명령
- declare -F : 정의되어 있는 함수명 출력
- declare -f : 정의된 함수 확인

```
# typeset -f  
  
.....  
trash()  
{  
    mv $* /root/test/.TRASH  
}
```

- unset -f 함수명 : 현재 쉘에서 정의한 함수 제거



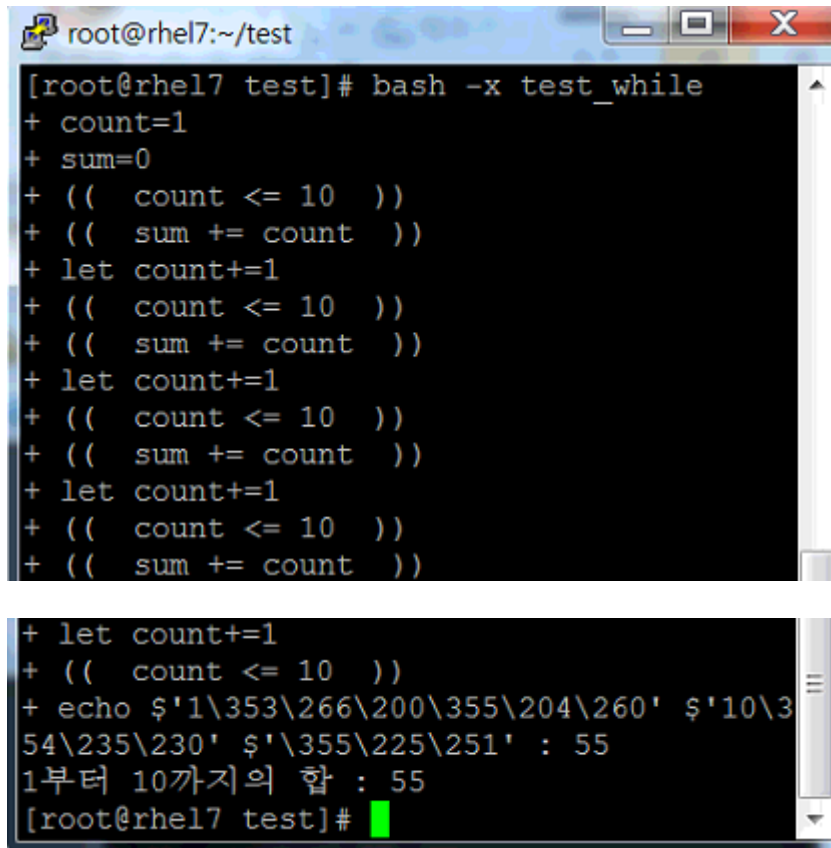
```
root@rhel7:~/test  
[root@rhel7 test]# declare -F trash  
trash  
[root@rhel7 test]# unset -f trash  
[root@rhel7 test]# declare -F trash  
[root@rhel7 test]# trash a b c  
bash: trash: 명령을 찾을 수 없습니다...  
[root@rhel7 test]#
```

디버깅

- 스크립트 실행도중 발생한 오류 수정 방법
- 구문 오류
 - 셸이 실행도중 구문오류가 발생한 라인번호 출력
 - 실행 오류
 - 오류 메시지 없이 실행이 안되거나 비정상 종료
 - 오류 수정 방법
 - `bash -x`, `trap`

디버깅 : bash -x

- 가장 간단한 스크립트 실행 오류 수정방법
- 스크립트의 각 행이 실행될 때마다 화면에 출력됨



```
root@rhel7:~/test
[root@rhel7 test]# bash -x test_while
+ count=1
+ sum=0
+ (( count <= 10 ))
+ (( sum += count ))
+ let count+=1
+ (( count <= 10 ))
+ (( sum += count ))
+ let count+=1
+ (( count <= 10 ))
+ (( sum += count ))
+ let count+=1
+ (( count <= 10 ))
+ (( sum += count ))
+ let count+=1
+ (( count <= 10 ))
+ echo $'1\353\266\200\355\204\260' $'10\354\235\230' $'\355\225\251' : 55
1부터 10까지의 합 : 55
[root@rhel7 test]#
```

디버깅 : trap

trap 명령 시그널

- 지정한 시그널이 스크립트로 전달될 때마다 지정한 명령 실행
- 스크립트의 명령이 한 줄씩 실행될 때마다 DEBUG
- 시그널이 스크립트로 전달됨
- DEBUG 시그널을 받을 때마다 원하는 변수값 출력 가능
=> 스크립트가 실행되는 도중 변수값 확인
- 예 : test_trap

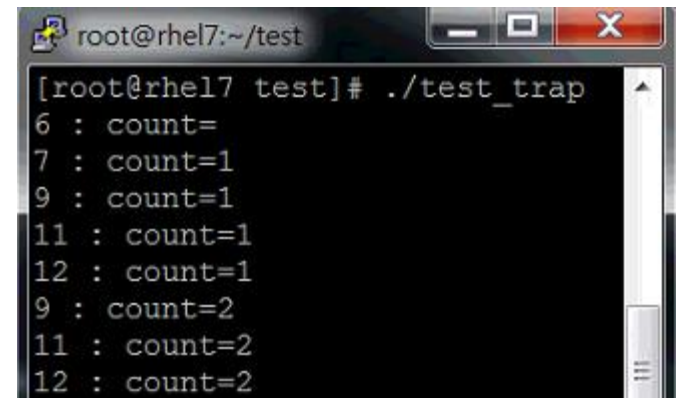
```
#!/bin/bash
# test_trap: trap 테스트 스크립트

trap 'echo "$LINENO : count=$count " ' DEBUG

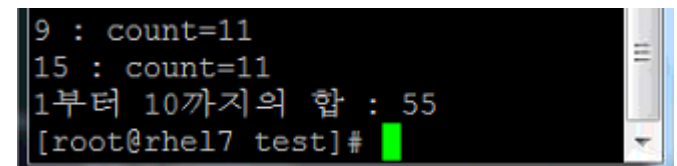
count=1
sum=0

while (( count <= 10 ))
do
    (( sum += count ))
    let count+=1
done

echo 1부터 10까지의 합 : $sum
```



```
root@rhel7:~/test
[root@rhel7 test]# ./test_trap
6 : count=
7 : count=1
9 : count=1
11 : count=1
12 : count=1
9 : count=2
11 : count=2
12 : count=2
```



```
9 : count=11
15 : count=11
1부터 10까지의 합 : 55
[root@rhel7 test]#
```