# 쉘 + 앤서블

# 쉘 기초

# 쉘 기초

**KSH**

KSH은 유닉스에서 많이 사용하는 쉘 중 하나. 보통 다음과 같은 유닉스 시스템에서 많이 사용한다.

BSD계열

Unix 계열(HP, AIX, Solaris)

# 쉘 기초

## KSH와 BASH 차이점

ksh와 bash의 큰 차이점은 실제로는 거의 없다.

하지만, Conflicts between ISO/IEC 9945 (POSIX) and the Linux Standard Base의 내용에 충돌이 되기 때문에 서로 호환이 되지 않는 부분이 있다. 예를 들어서 LSB에서는 /usr/xpg4/bin를 사용하지 않으며 또한 솔라리스도 이러한 POSIX규격을 따르지 않는다.

리눅스 및 솔라리스는 2004년도 기준으로 표준 사양을 변경하였다. 이를 통해서 리눅스 및 솔라리스 같은 유닉스와 스크립트 격차를 많이 줄었다.

# 쉘 기초

## KSH와 BASH 차이점

POSIX에서 명시한 echo명령어는 불행하기도UNIX System V, BSD 그리고 GNU기반에 따라서 각기 다르게 동작한다. 예를 들어서 GNU echo 표준은 UNIX System V 혹은 BSD와 다르게 동작한다.

그러한 이유로, echo명령어 대신 printf명령어 사용하는 것을 권장한다.

# 쉘 기초

## BASH 무엇인가?

1. 쉘은 사용자가 커널 및 프로그램 사이에서 서로 대화 할 수 있도록 해주는 도구.
2. 쉘은 ssh같은 도구가 아님. 컴퓨터와 대화하기 위한 도구
3. 쉘은 TTY가 아니다. 많은 사용자들이 오해하고 있는게 TTY == Shell라는 생각을 가지고 있음,
4. PuTTY, iTerm, xterm같은것 도구는 bash가 아님.
5. Bash는 명령어 라인 도구가 아님. Bash는 대화형 모드를 가지고 있으며 이를 통해서 프로그램들을 사용함. 예를 들어서 ls같은 명령어.

# 쉘 기초

## BASH 무엇인가?

거의 대다수의 리눅스는 bash가 설치되어 있음. 다른 말로 GNU/Linux는 거의 bash가 표준이라고 생각해도 됨.

다른 배포판 경우 예를 들어서 FreeBSD/NetBSD/OpenBSD 그 외 다른 유닉스들에서는 bash가 아닌 csh, tsh를 사용하기도 한다. 하지만, 그거와 상관 없이 bash사용이 가능하다.

# 쉘 기초

## BASH 무엇인가?

Bash 버전 확인하는 방법은 다음과 같다.

bash$ declare --p BASH
BASH="/usr/local/bin/bash"

chsh에서 bash로 변경하는 경우 아래와 같이 변경이 가능하다.

$ chsh -s /usr/local/bin/bash

# 쉘 기초

**BASH 무엇인가?**

리눅스 시스템에서 사용하는 쉘 목록은 아래에서 확인이 가능하다.

$ chsh -l

설치가 안되어 있으면

# yum install chsh -y

# 쉘 기초

## BASH 무엇인가?

## 프로토타입

bash기반으로 간단하게 스크립트 프로그램밍 빠르게 가능하다.
bash의 목표 중 하나가 “hack together”

## 대화형 관리 시스템

bash는 borne-style쉘 스타일로 리눅스 및 유닉스 에서 손쉽게 명령어를 사용할 수 있도록 해준다. 또한 bash스크립트는 자연스러운 구조를 가지고 있다.

# 쉘 기초

## BASH 무엇인가?

## 오토메이션

자주 사용하는 명령어를 스크립트를 간단하게 텍스트 파일로 만들어서 실행권한을 주어서 자동화 가능.

## 프로그램 상화작용

한 개 이상의 프로그램을 쉘 스크립트를 통해서 동시에 사용 할 수 있도록 해주며, 파일이나 혹은 표준 입출력을 통해서 데이터를 서로 전달하여 프로세싱 할 수 있다.

# 쉘 기초

**BASH 무엇인가?**

**필터링 그리고 입력 변환**

어떠한 프로그램은 동시에 데이터 처리에 대해서 고려하지 않고, 개발이 된 경우가 있다. 이러는 경우 bash는 다른 데이터 프로세싱 프로그램을 통해서 다시 데이터 재처리를 해준다.

예를 들어서 awk, sed같은 프로그램이 중간 컨버팅 기능을 해준다.

# 쉘 기초

**BASH 무엇인가?**

**유닉스 혹은 리눅스 파일 시스템 네비게이션**

어떠한 프로그램은 동시에 데이터 처리에 대해서 고려하지 않고, 개발이 된 경우가 있다. 이러는 경우 bash는 다른 데이터 프로세싱 프로그램을 통해서 다시 데이터 재처리를 해준다.

예를 들어서 awk, sed같은 프로그램이 중간 컨버팅 기능을 해준다.

# 쉘 기초

**BASH 무엇인가?**

**베이직 패턴 문자열 매칭**

bash는 문자열 패턴 매칭 기반의 기능을 가지고 있다. 이를 통해서 파일이름, 경로 이름을 파라메타로 확장이 가능하다.

# 쉘 기초

**BASH 무엇인가?**

**베이직 패턴 문자열 매칭**

bash는 문자열 패턴 매칭 기반의 기능을 가지고 있다. 이를 통해서 파일이름, 경로 이름을 파라메타로 확장이 가능하다.

# 쉘 기초

## BASH 무엇인가?

## 이동성

bash는 다양한 유닉스 시스템에 패키징이 되어있다. 이러한 이유로 POSIX를 쉘 스크립트를 표준적으로 지원하며, 각기 다른 런타임 환경에서 사용하는 경우 bash는 강력한 도구가 될 수 있다.

# 쉘 기초

## BASH 사용하기

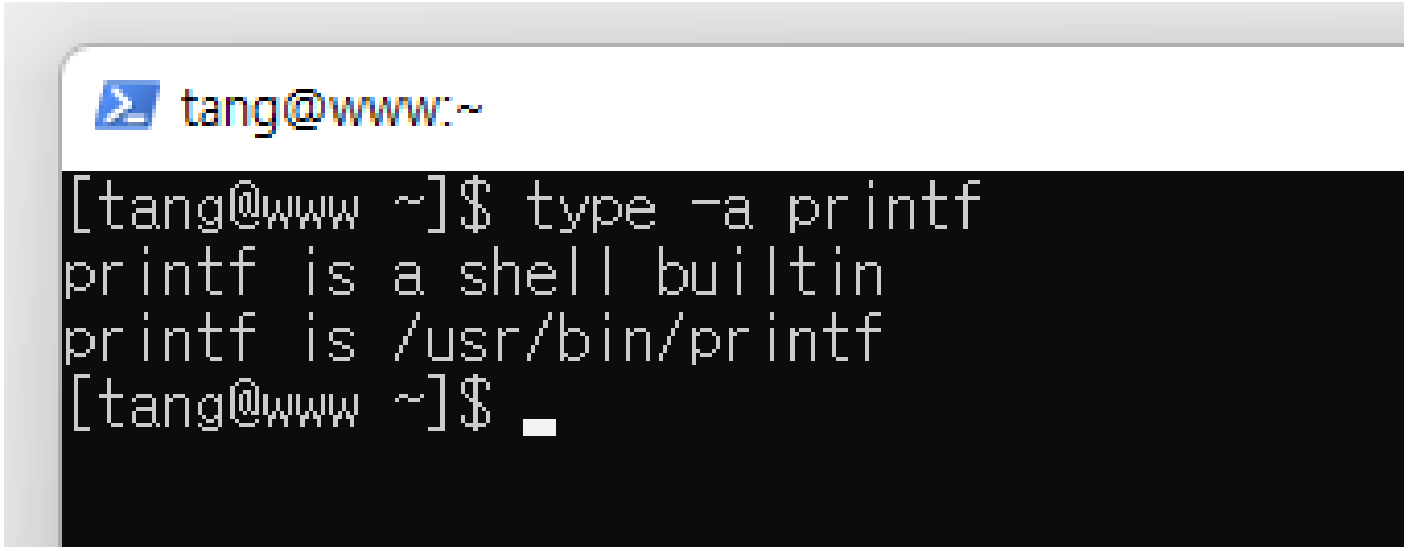
bash를 사용하다가 어려운 부분이 있으면 다음과 같은 방법으로 도움을 받을 수 있다.

bash$ help printf

bash$ man printf

# 쉘 기초

## BASH 사용하기



# bash 명령어 구조

# bash 명령어 구조

**간단한 명령어**

bash에서 명령어 사용 방법은 다음과 같다.

$ mkdir -p testdir/subdir

매우 간단하다! :”)

# bash 명령어 구조

## 간단한 명령어

하지만, 이렇게만 사용하지 않는다. 앞에서 이야기 하였던 쉘에서 사용하는 특수문자(metacharacters)가 있다. 예를 들어서 다음과 같다.

$ touch test file

$ ls –l

test

file

위와 같이 나온다. 여기에 좀 더 복잡하게 특수문자를 사용하면 다음처럼 결과가 나온다.

# bash 명령어 구조

## 간단한 명령어

$ touch report<SRV1>.txt

$ touch report;Final.txt

위의 명령어를 실행을 하면 올바르게 실행이 되지 않는다?

왜?

# bash 명령어 구조

짤

# bash 명령어 구조

## 간단한 명령어

몇가지 더 테스트 해본다.

$touch $$$likeMoney.txt

$ ls

12234likeMoney.txt

왜 이럴까요?

# bash 명령어 구조

**간단한 명령어**

이러한 문제를 해결 하기 위해서 쿼팅(quoting)을 사용해야 한다.
보통 다음과 같은 방법이 있다.

a-z, 0-9 혹은 \, ‘ ’, “ ”

# bash 명령어 구조

## 특수문자 해결 방법

$ touch report\<SRV1\>.txt

$ touch report\;Final.txt

$touch \$\$\$likeMoney.txt

# bash 명령어 구조

**그러면 만약 백슬래쉬(\)를 출력하는 방법?**

$ echo \\backslash\\

\backslash\

# bash 명령어 구조

**그러면 만약 백슬래쉬(\)를 출력하는 방법?**

$ echo backslash\

> hello

> SKT

backslashhelloSKT

# bash 명령어 구조

## 싱글쿼터

$ touch ‘report document’

$ touch ‘report<SRV1>.txt’

$ touch ‘report;Final.txt’

$ touch ‘$$$report.txt’

# bash 명령어 구조

**싱글쿼팅 줄내림(Single Quotes)**

$ echo ‘quotes

> foo

> bar’

quotes

foo

bar

# bash 명령어 구조

**그러면 어떻게 싱글쿼티 출력하지?**

$ echo ‘it\’s today’

$ echo ‘it’’s today’

$ echo ‘it’\’’s today’

어떤게 출력이 될까요?

# bash 명령어 구조

## 더블쿼티

$ echo “This is my shell: $SHELL”

$ echo ‘This is my shell: $SHELL’

$ echo “The variable: \$total”

$ echo “Back\\to\\back\\back\\slashes_world”

$ echo “hello $USER”’!!’

# bash 명령어 구조

**백틱(backticks)**

$ echo “backticks: \`\`\”

# bash 명령어 구조

## 복합적 쿼티

$ echo Hello, \ “$USER”’! Hell to the ‘”$HOSTNAME”’!’

# bash 명령어 구조

**Shell Exit Values**

간단하게…

$ echo $?

$ true ; echo $?

$ false ; echo $?

1: False

0: True

# bash 명령어 구조

**Shell Exit Values**

명령어가 없는 경우…

$ thereisnocommand ; echo $?

127

# bash 명령어 구조

## 명령어 연속 실행 시 오류 중지

여러 명령어를 수행하는 방법은 두 가지가 있다.

$ cd ; rm -r /testdir ; ls

$ cd / && rm -r /testdir ; ls

첫 번째 방법은 오류가 발생하여도 계속 연속적으로 명령어가 실행이 되며, 두 번째 방법은 명령어 수행 도중에 문제가 발생하면 중지.

# bash 명령어 구조

## 포그라운드/백그라운드

명령어 수행 시 포그라운드/백그라운드 실행은 다음과 같은 명령어로 실행이 가능하다.

$ sleep 10

$ sleep 10 &

# bash 명령어 구조

## 포그라운드/백그라운드

백 그라운드로 실행이 된 명령어를 확인 하려면 아래와 같은 명령어로 확인이 가능하다.

$ bg

# 기본 명령어

# 기본 명령어

## bash 기본 명령어

쉘에서 기본적으로 제공하는 내부 명령어(internal command)가 있다. 이 명령어를 통해서 좀 더 편하게 쉘 환경을 꾸밀 수 있다.

# 기본 명령어

## 빌트인 명령어

내부 명령어는 외부에 별도의 명령어 없이 실행이 가능한 내장형 명령어. 대표적인 명령어는 다음과 같다.

echo, type, source

# 기본 명령어

## 런타임 명령어

런타임 명령어(runtime)는 기본적으로 쉘 스크립트(bash)명령어 기반으로 구성이 되어 있다. 이러한 도구들은 보통 function(), alias로 구성이 되어 있으며, 자체적으로 실행은 불가능하며, 쉘을 통해서 실행이 가능하다.

# 기본 명령어

## 시스템 명령어

시스템 명령어는 파일 시스템에 독립적으로 구성이 되어 있는 프로그램이다. 이러면 명령어는 어떠한 패키지가 설치가 되어 있느냐에 따라서 다르다.

대표적인 명령어는 다음과 같다.

grep, ping, rm, ls, nmcli

# 기본 명령어

## 대표 명령어

type: 명령어 형식 확인. 쉘에 alias로 구성이 되어 있는 경우 alias를 출력한다.

echo: 문자열 출력

printf: echo와 비슷하지만 포멧팅 형식으로 출력

pwd: 쉘의 현재 위치 출력

cd: 쉘 위치 변경

set: 현재 쉘에 설정된 값을 출력

decleare: 변수나 함수(function)를 관리하는 명령어

test, [ , [[: 정규식 구성, 대괄호로 구성이 된다.

# 기본 명령어

## 대표 명령어

ls: 디렉터리 및 파일 출력

mv: 파일 이동 및 이름 변경

cp: 파일 복사

rm: 파일 및 디렉터리 제거, 다른 명령어로 rmdir이 있다.

grep: 문자열 검색하는 정규식 도구

cut: 행 단위로 자르는 명령어

wc: 줄 및 문자열 카운팅 도구

find: 파일이나 혹은 디렉터리를 특정 조건으로 찾는 도구

sort, uniq: 정렬 및 중복제거 하는 도구

# 기본 명령어

## type

bash에서 명령어 사용 방법이 궁금한 경우 아래와 같이 사용이 가능하다.

bash$ help type

type명령어를 사용하면 다음과 같이 출력한다.

bash $ type cp

cp is /usr/bin/cp

bash $ type ls

ls is aliased to `ls --color=auto'

bash $ help cp

-sh: help: no help topics match `cp'.  Try `help help' or `man -k cp' or `info cp'.

$ type for

for is a shell keyword

# 기본 명령어

## type

function()를 type으로 확인하기.

bash $ myfunc() { : ; }

bash $ type myfunc

myfunc is a function

myfunc()

{

  :

}

bash $ alias myalias=:

bash $ type myalias

myalias is aliased to `:’

# 기본 명령어

**type**

bash $ type -a true

true is a shell builtin

true is /bin/true

# 기본 명령어

**echo**

bash $ echo hello

hello

bash $ echo ‘Hello, ‘”$USER”\!

Hello, bashuser!

# 기본 명령어

**printf**

bash $ printf ‘%s\n’ ‘Hello!’

Hello!

이 명령어는 C의 printf와 흡사한 포멧팅 구조를 가지고 있다.

# 기본 명령어

**printf**

bash $ printf ‘%s\n’ -n

-n

bash $ string=-n

bash $ printf ‘%s\n’ “$string”

-n

# 기본 명령어

**printf**

$ printf ‘%s\n’ foo bar baz

foo

bar

baz

# 기본 명령어

**pwd**

현재 작업하는 위치를 확인 시 사용하는 명령어.

$pwd

/home/bash/working

$ ls secret/report.txt

$ ls report_review.txt

$ ls nonexistent

# 기본 명령어

**tildes**

경로 관련된 ~(틸드, tild)특수 문자. 이 문자를 통해서 홈 디렉터리로 이동이 가능하다. 틸드 path이외 다른 특수 문자 ‘.’, ‘..’등 있지만, 여기에서는 ~만 다룬다.

$ echo ~

$ echo ~/secret

# echo ~

# echo ~root/.ssh

# 기본 명령어

**tildes**

만약 존재하지 않는 경우 그냥 문자열로 출력이 된다.

$ echo ~nouser

~nouser

# 기본 명령어

**tildes**

특수 문자로 출력하고 싶은 경우 아래처럼 실행한다.

$ echo ~bash

/home/bash

$ echo \~bash ‘~bash’ “~bash”

~bash ~bash ~bash

# 기본 명령어

## cd

cd는 change directory의 약자, 이를 통해서 디렉터리 이동을 한다.  이와 관련된 쉘 변수는 $PWD 혹은 pwd명령어가 있다.

$ pwd

/home/bash

$ cd /tmp

$ pwd

/tmp

# 기본 명령어

**cd**

앞에서 언급하였던 .. 혹은 .를 통해서 디렉터리 이동이 가능함.

$ cd .

$ cd ..

$ pwd

# 기본 명령어

**pushd/popd**

cd 명령어와 같이 pushd, popd명령어로 다음처럼 사용이 가능하다.

$ sudo yum install httpd -y

$ pushd /var/www

$ pushd /opt/

$ push -n /usr/local/

# 기본 명령어

## pushd/popd

$ push -n /usr/local/

-n옵션이 붙어 있으면 실제로 디렉터리 이동하지는 않음

내역을 확인하기 위해서는 다음과 같은 명령어로 확인 가능.

$ dirs -l -v

$ pushd +2

$ pushd -1. == $ cd -

# 기본 명령어

**pushd/popd**

popd명령어로 이전에 사용하였던 하나씩 이동하면서 스택(stack)에서 하나씩 제거한다.

$ pushd /usr/local

$ pushd /opt/

$ pushd /var/www

$ popd

# 기본 명령어

**set**

set명령어는 빌트인 명령어. 또한 이 명령어에 대해서 잘못 이해하는 경우가 많고 혹은 너무 복잡하게 설명하는 경우가 있다.

set은 현재 사용중인 쉘에 설정되어 있는 값을 화면에 변수 이름과 함께 출력해주며, 알파벳 순서대로 정렬해서 화면에 출력한다. 여기에 출력된 변수들은 전부 쉘에서 명시된 값이다.

실제로는 set명령어보다 declare명령어 사용을 더 권장한다. set명령어는 -옵션을 통해서 설정된 값 확인이 가능하다.

# 기본 명령어

**set**

-e: 프로그램이 종료가 되면 0 상태값을 반환한다. 하지만 이 방법을 스크립트에서 사용하는 경우 거의 대다수가 올바르지 않는 값을 반환한다. 그 이유는 -e옵션은 결과값 예상이 어렵기 때문에다. 그래서 대다수 스크립트는 set -e를 사용하지 않고 자체적으로 오류값 핸들링을 더 선호한다.

# 기본 명령어

**set**

```
#!/bin/bash
set -e
function hello()
{
  echo "I hate the Java language!!"
}
hello
```

# 기본 명령어

**set**

```
#!/bin/bash
function hello()
{
  set -e
  echo "I hate the Java language!!"
}
hello
```

# 기본 명령어

**set**

```
#!/bin/fish
set -e
function Error()
{
  echo "error occur at line $1"
}
trap 'error $LINENO' ERR
Eraaa
```

# 기본 명령어

**set**

-n: 명령어를 읽어 오지만 실행하지 않는다.

```
#!/bin/bash
set -n
os="linux"
echo $os
echo $unix
```

# 기본 명령어

**set**

-u: 변수가 선언이 되었지만, 값을 가지고 있지 않는 경우 경고 메시지 출력.

```
#!/bin/bash

set -u

os="linux"

echo $os

echo $unix
```

# 기본 명령어

**set**

-o: -u하고 비슷하지만 조금 더 자세하게 출력이 된다. 몇번째 줄에 문제가 있는 좀 더 자세히 메시지가 나온다.

```
#!/bin/bash

set -o nounset

os="linux"

echo $os

echo $unix
```

# 기본 명령어

**set**

-f: 글러빙 끄기

$ ls *

$ set -f

$ ls *

# 기본 명령어

**set**

-a: 변수를 외부에서도 사용할 수 있도록 한다.

$ set -a

$ number1=23

$ number2=34

# 기본 명령어

**set**

$ cat <<EOF> testfile.sh

#!/bin/bash

echo $number1 number2

EOF

$ bash testfile.sh

# 기본 명령어

**set**

-v: 읽어오는 값을 그대로 화면에 출력한다. 디버깅시 매우 유용하지만, -x와는 다른 정보를 출력한다.

# 기본 명령어

**set**

-x: 이 옵션은 각각 인자 값이 실행하는 걸 출력한다. 이 옵션을 실행하면 bash스크립트 디버깅에 도움이 많이 되며, 어떤 부분에서 bash프로그램이 입력을 받아서 확장이 되는지 확인이 가능하다.

# 기본 명령어

**set**

디버깅시 다음과 같은 방법으로 진행이 가능하다.

set -x 를 CLI에 같이 선언 혹은 매직뱅(MagicBang)에 다음과 같이 선언한다.

#!/bin/bash -x

또한 set의 종료 문자는 “--”으로 사용이 가능하며, 이는 위치 값(positional parameter)로 사용이 가능하다.

# 기본 명령어

**set**

```
#!/bin/bash
var = 5
while [ $var -gt 0 ] ; do
  var=$[ $var-1 ]
  echo $var
  sleep 2
done
```

# 기본 명령어

**set**

```
#!/bin/bash
set -x
var = 5
while [ $var -gt 0 ] ; do
  var=$[ $var-1 ]
  echo $var
  sleep 2
done
```

# 기본 명령어

**set**

혹은 다음처럼 실행하여도 된다.

bash -x <SCRIPT>

# 기본 명령어

**declare**

declare는 내장 명령어이다. 이를 통해서 변수, 배열 혹은 함수를 구현한다. 변수 구현을 위해서 자주 사용하지는 않지만, 쉘의 현재 상태 정보 및 다른 정보들 set를 통해서 훑어보기(overview)가 가능하다.

# 기본 명령어

**declare**

$ declare -p | head -5

declare -- BASH="/bin/sh"

declare -r BASHOPTS="checkwinsize:cmdhist:complete_fullquote:expand_aliases:extglob:extquote:force_fignore:histappend:interactive_comments:login_shell:progcomp:promptvars:sourcepath"

# 기본 명령어

**declare**

이 명령어를 사용할 때 -f옵션을 -p와 함께 사용하면 declare는 선언이 되어 있는 모든 함수에 대해서 출력한다.

# 기본 명령어

**test, [, [[**

test 내부 명령어는 두 개의 값을 비교할 때 보통 사용한다. 예를 들어서 다음과 같은 방법으로 사용이 가능하다.

$ test -e /etc/passwd && echo ‘Password file’

# 기본 명령어

**test, [, [[**

다음 명령어로 “[“이다. 이 명령어는 보통 사용시 type명령어와 같이 사용한다.

$ type -a [[

is a shell building [ is /usr/bin/[

# 기본 명령어

**test, [, [[**

하지만 test, [의 제일 큰 차이점은 test는 단일 명령어로 사용하며, 대괄호 “[“는 쌍으로 사용을 해야 한다. 그래서 이전에 사용하였던 test를 다음처럼 변경이 가능하다.

test -e /etc/passwd → [ -e /etc/passwd ]

위와 같이 변경이 가능하다.

# 기본 명령어

**test, [, [[**

하지만, 이 부분도 버전이 변경이 되면서 다음처럼 변경이 되었다.
이유는 간단하게 안전성 및 파싱(parsing)강화를 위해서 변경.

[ → [[

$ type [[

[[is a shell keyword

# 기본 명령어

**test, [, [[**

bash 스크립트 에서는 test를 많이 사용하지 않는다. 일반적으로 [ 혹은 [[ 키워드를 더 많이 사용하는데, 그냥 편의상 test 및 [ 비슷한 기능을 하며 유일한 명령어라고 생각하면 편하다.

# 기본 명령어

## 일반 시스템 명령어

$ ls

adir  bdir  cdir

$ ls -a

.  ..  .profile  .bash_profile  Downloads

# 기본 명령어

## 일반 시스템 명령어(ls)

```
$ ls -al
drwxrwxr-x.  4 tang tang   4096 May 24 12:19 kubectl-plugins
drwxrwxr-x. 10 tang tang    249 May 24 12:16 kubectx
drwxrwxr-x.  9 tang tang    251 May 24 12:15 kubelogin
drwxrwxr-x.  3 tang tang     80 May 24 12:26 kubergui
drwxrwxr-x. 17 tang tang   4096 May 24 12:23 kubernetes-yaml-templates
drwxr-xr-x.  3 tang tang     27 Jan 25 21:03 training
drwxrwxr-x.  3 tang tang    128 Jun  7 02:41 writing-ansible-modules-in-bash
drwxr-xr-x.  2 tang budimg    6 May  8 12:34 www
```

# 기본 명령어

**일반 시스템 명령어(ls)**

만약 ‘ls’명령어 없이 파일 이름 및 목록을 얻으려면 어떻게 해야할까?

$ ls -l

$ grep pattern -- *

# 기본 명령어

**일반 시스템 명령어(ls)**

이걸 좀 더 보기 좋게 표현하기 위해서 다음과 같이 시도한다.

```
$ find . -type f -exec grep pattern -- {} \;
```

```
$ for file in * ; do grep -F pattern -- “$file” ; done
```

# 기본 명령어

**일반 시스템 명령어(ls)**

이걸 좀 더 보기 좋게 표현하기 위해서 다음과 같이 시도한다.

```
$ find . -type f -exec grep pattern -- {} \;
```

```
$ for file in * ; do grep -F pattern -- “$file” ; done
```

# 기본 명령어

**일반 시스템 명령어(mv)**

mv 명령어는 디렉터리 및 파일을 이동하거나 혹은 이름을 바꾸는 명령어.

$ mv file path/to/directory

$ mv file1 file2

$ mv file1 file2 dir4 path/to/directory

$ mv testdir/ goodir/

# 기본 명령어

**일반 시스템 명령어(mv)**

대화형으로 사용을 원하는 경우 아래와 같은 명령어로 사용이 가능하다.

$ mv -i olduser/ newuser/

# 기본 명령어

**일반 시스템 명령어(mv)**

대화형으로 사용을 원하는 경우 아래와 같은 명령어로 사용이 가능하다.

$ mv -i olduser/ newuser/

# 기본 명령어

**일반 시스템 명령어(mv)**

```
#!/bin/bash
if [[ -e old_user ]] ; then
  printf ‘the old user exists.\n’
  mv old_user new_user
fi
mv new_user user
```

# 기본 명령어

**일반 시스템 명령어(cp)**
복사 명령어.

cp 명령어는 파일이나 혹은 파일을 복사시 사용한다. 이와 비슷한 명령어는 ln명령어가 있다.

```
$ cp doc1 doc2 doc3 dir/
$ ls
dir
$ cp dir/ back_dir/
```

# 기본 명령어

**일반 시스템 명령어(cp)**

$ cp -R

$ cp -r

$ cp -R back_dir/ rel_dir/

-r, -R의 차이점은 기본 옵션은 GNU에서는 -r를 사용한다. 하지만, 전통적인 유닉스에서는 -R옵션을 사용한다.

# 기본 명령어

**일반 시스템 명령어(rm/rmdir)**

rm, rmdir은 비슷한 동작을 하지만 약간의 차이가 있다.

$ rm testfile

$ rm -rf testdir/

$ rmdir testdir/

Directory not empty

# 기본 명령어

**일반 시스템 명령어(grep)**

grep 명령어는 특정 문자열을 표준 출력에서 검색 및 출력한다.

```
$ grep ‘ssh’ /etc/services
ssh             22/tcp                  # The Secure Shell (SSH) Protocol
ssh             22/udp                   # The Secure Shell (SSH) Protocol
x11-ssh-offset  6010/tcp                        # SSH X11 forwarding offset
ssh             22/sctp                 # SSH
sshell           614/tcp                 # SSLshell
sshell           614/udp                  #       SSLshell
```

# 기본 명령어

## 일반 시스템 명령어(grep)

```
$ grep '^ssh' /etc/services
ssh             22/tcp                  # The Secure Shell (SSH) Protocol
ssh             22/udp                  # The Secure Shell (SSH) Protocol
ssh             22/sctp                 # SSH
sshell          614/tcp                 # SSLshell
sshell          614/udp                 #       SSLshell
ssh-mgmt        17235/tcp               # SSH Tectia Manager
ssh-mgmt        17235/udp               # SSH Tectia Manager
```

# 기본 명령어

**일반 시스템 명령어(grep)**

```
$ grep -i 'ssh$' /etc/services
ssh             22/sctp                 # SSH
netconf-ssh     830/tcp                  # NETCONF over SSH
netconf-ssh     830/udp                  # NETCONF over SSH
sdo-ssh         3897/tcp                 # Simple Distributed Objects over SSH
sdo-ssh         3897/udp                 # Simple Distributed Objects over SSH
tl1-ssh         6252/tcp                 # TL1 over SSH
```

# 기본 명령어

## 일반 시스템 명령어(grep)

중복된 출력 결과를 다음과 같은 명령어로 정리가 가능하다.

$ uniq
$ sort

혹은 grep에서 -q옵션을 사용하여 조건에 맞는 문자열이 있는지 확인이 가능하다.

$ grep -q telnet /etc/service && echo ‘Matched’
$ echo $?
$ grep -q foobar /etc/service && echo ‘Matched’
$ echo $?

# 기본 명령어

## 일반 시스템 명령어(grep)

혹은 몇 개의 라인이 일치하는지 확인 하려면 다음과 같은 방법이 있다.

```
$ grep -c telnet /etc/service
5
$ grep telnet /etc/service |wc -l
```

복합적으로 문자열을 확인이 필요한 경우는 다음과 같은 명령어로 확인이 가능하다.

```
$ grep -e ssh -e telnet /etc/services
```

# 기본 명령어

**일반 시스템 명령어(grep)**

혹은 -F옵션을 통해서 문자열 검색이 가능하다.

$ grep -F ‘$’ ikea

만약 매치가 되지 않는 문자열만 출력하고 싶은 경우 -v문자열을 사용한다. ‘^#’옵션은 주석 문자인 #를 제외하고 화면에 출력.

$ grep -v ‘^#’ /etc/services

# 기본 명령어

**일반 시스템 명령어(grep)**

조금 더 복잡하게 특정 문자만 카운팅이 필요한 경우 다음과 같이 실행한다

$ grep -cFv -e bash -e ‘.’ /etc/shells

10

# 기본 명령어

**일반 시스템 명령어(cut)**

cut명령어는 말 그래도 짜르는 명령어. 특정 필드나 문자열을 짜르고 싶은 경우 cut명령어를 통해서 사용이 가능하다.

많이 사용하는 옵션은 보통 -d, -f이며, -d 는 필드 분리 문자, -f는 필드이다.

# 기본 명령어

## 일반 시스템 명령어(cut)

```
$ cut -d: -f1 /etc/passwd
root
bin
daemon
adm
lp
sync
shutdown
```

# 기본 명령어

**일반 시스템 명령어(cut)**

만약, 맨 끝 필드를 출력하고 싶은 경우 아래처럼 실행한다.

```
$ cut -d: -f6- /etc/passwd
/root:/bin/bash
/bin:/sbin/nologin
/sbin:/sbin/nologin
```

# 기본 명령어

**일반 시스템 명령어(wc)**

단어 혹은 줄 개수 그리고 바이트를 출력한다. 기본 값은 전부를 출력한다.

$ wc .bash_history

200  437 3373 .bash_history

위의 출력된 내용은 200줄, 437줄 그리고 총 3373바이트. 여기에서 특정값만 출력을 원하는 경우 -c, -w, -l같은 옵션을 사용해서 출력이 가능하다. -l옵션은 엄연히 따지고 보면 newline갯수만 출력한다.

# 기본 명령어

**일반 시스템 명령어(wc)**

-c 옵션 경우에는 바이트를 출력한다.

$ wc -c .bash_history

3373 .bash_history

# 기본 명령어

**일반 시스템 명령어(wc)**

-m 총 문자 개수를 카운팅 하기 위해서는 -m옵션을 사용한다.

$ wc -m .bash_history

3373 .bash_history

# 기본 명령어

## 일반 시스템 명령어(wc)

$ wc -m -c Korean

35 93 test

한국어를 UTF-8로 입력하면 영문과 다르게 바이트 및 문자가 카운트가 된다. 그 이유는 인코딩 및 2바이트 문자열로 인하여 개수와 크기가 다르게 출력이 된다.

# 기본 명령어

## 일반 시스템 명령어(du)

wc명령어로 파일의 크기를 알 수 있다.

$ wc -c /bin/bash

하지만 전체적인 크기에 대해서는 확인이 불가능 하기 때문에 du명령어를 사용하요 확인한다.

$ du -k /bin/bash
1124    /bin/bash

디렉터리 크기를 알고 싶은 경우 아래처럼 실행한다. 모든 결과는 POSIX 1kibibyte(1024)로 출력이 된다.

$ du -k /bin

# 기본 명령어

## 일반 시스템 명령어(find)

find 명령어는 리눅스에서 많이 사용하는 명령어 중 하나이다. find를 통해서 디렉터리 및 파일 검색이 가능하다. 기본적인 동작 방식은 다음과 같다.

```
$ find ~/
```

출력만 하고 싶은 경우 다음처럼 옵션을 지정한다.

```
$ find /usr/share  -print
```

# 기본 명령어

**일반 시스템 명령어(find)**

여러 위치에서 검색이 필요한 경우 다음처럼 사용이 가능하다.

$ find ~/docs ~/googles email.txt

출력된 내용을 정렬하기 위해서 다음과 같이 혼용해서 사용이 가능하다.

$ find ~/docs | sort

# 기본 명령어

**일반 시스템 명령어(find)**

여러 위치에서 검색이 필요한 경우 다음처럼 사용이 가능하다.

$ find ~/docs ~/googles email.txt

출력된 내용을 정렬하기 위해서 다음과 같이 혼용해서 사용이 가능하다.

$ find ~/docs | sort

# 기본 명령어

## 일반 시스템 명령어(find)

test라는 문자열을 가지고 있는 파일 및 디렉터리
$ find ~/docs -name ‘*test*’ -print

fish라는 문자열 가지고 있는 파일만 출력
$ find ~/docs -name ‘*fish*’ -type f -print

수정한지 3일이 지난 파일
$ find ~/docs -mtime +3 -print

5일전에 수정한 파일
$ find ~/docs -mtime -5 -print

# 기본 명령어

**일반 시스템 명령어(find)**

이름에 beef라는 문자열이 없는 것만 검색

$ find ~/docs ! -name ‘*beef*’

만약 특정 문자열에 대해서 무시를 하고 싶은 경우, 아래처럼 명령어를 실행한다.

$ find . -name ‘.git*’ -prune -o -print

# 기본 명령어

**일반 시스템 명령어(find)**

find를 사용하면, 각각 결과마다 명령어를 같이 실행이 가능하다.

$ find vim -type f -name ‘*.vim’ -exec grep -F search -- {} \;

-exec: 명령어 실행, 여기서는 ‘grep -F search --’ 명령어
{}: 이 부분에 find에 찾을 출력 결과물로 대체
\;: 명령어 종료 알림

# 기본 명령어

**일반 시스템 명령어(find)**

위의 명령어를 구분으로 나누면 다음과 같이 두개 구조로 나누어진다.

$ find vim -type f -name ‘*.vim’

vim/map.vim

vim/test.vim

$ grep -F search -- vim/map.vim

$ grep -F search -- vim/test.vim

# 기본 명령어

**일반 시스템 명령어(find)**

만약 find의 -exec에 더 많은 인자값이 필요한 경우 +기호를 사용하여 좀 더 많은 인자 값 옵션 사용이 가능하다.

$ find vim -type f -name ‘*.vim’ -exec grep -F search -- {} +

$ grep -F search -- vim/map.vim vim/test.vim

# 기본 명령어

**일반 시스템 명령어(xargs)**

find와 또한 많이 사용하는 명령어는 xargs이다. 이 명령어는 -exec와 비슷하지만 조금 다른 동작 구조를 가지고 있다.

$ find vim -type f -name ‘*.vim’ | xargs grep -F search --

위의 명령어는 find에서 실행한 결과를 표준 출력으로 오른쪽 xargs명령어에 전달하여 grep 명령어가 작업을 수행한다.

# 기본 명령어

**일반 시스템 명령어(xargs)**

만약, 출력되는 결과에 null-byte로 종료되는 경우, 올바르게 동작이 안될 수 있기 때문에 find나 xargs를 사용시에 다음처럼 사용을 권장한다.

find -print0 혹은 xargs -0

$ find vim -type f -name ‘*.vim’ -print0 | xargs -0 grep -F search --

# 기본 명령어

## 일반 시스템 명령어(sort, uniq)

sort, uniq도 쉘 프로그램밍 시 많이 사용하는 명령어 이다. sort는 출력된 결과를 알파벳 순서로 정렬 시 사용하는 문자이다.

$ sort /etc/shells

$ sort ~/file1 ~/file2

$ printf ‘%s\n’ ‘line 2’ ‘line 1’ | sort

혹은 특정 필드 조합을 통해서 특정 열만 추출이 가능하다.

$ sort -t: -k6,6 /etc/passwd

# 기본 명령어

**일반 시스템 명령어(sort, uniq)**

또한 같이 사용하는 옵션은 다음과 같다.(sort)

-n: 숫자가 알파벳 순서보다 먼저 출력이 된다.
-r: 역순정렬, 맨 마지막 결과가 맨 처음에 출력

# 기본 명령어

**일반 시스템 명령어(sort, uniq)**

$ cat ipaddress.txt

$ sort ipaddress.txt | uniq -c | sort -k1, 1nr

# 입력, 출력 그리고
# 재전달

# 기본 명령어

**Input, Output, Redirection**

$ printf “Hello World!\n”

Hello World!

$ printf “Hello World!\n” > myfile

$ cat myfile

# 기본 명령어

**Input, Output, Redirection**

$ printf ‘’ > empty.txt

$ wc -c empty.txt

0

디렉터리 위치로 재전달 혹은 출력하는 경우 다음처럼 한다.

$ cd

$ printf ‘Hello World!\n’ > ~/tmp/test.txt

$ printf ‘Hello World!\n’ > /tmp/test.txt

# 기본 명령어

**Input, Output, Redirection**

$ printf ‘Hello World!\n’ > “$HOME”/text.txt

$ printf ‘Hello, root! \n’ > ~root/tmp/text.txt

# 기본 명령어

## 덮어쓰기 금지

bash에서 덮어쓰기를 방지하고 싶은 경우 -C 옵션을 통해서 가능하다.

$ print ‘First Command’ > /tmp/test.txt

$ print ‘Second Command’ > /tmp/test.txt

$ cat test.txt

Second Command

# 기본 명령어

## 덮어쓰기 금지

$ set -C

그런 다음에 다음처럼 다시 시도한다.

$ printf ‘Third Command\n’ > /tmp/test.txt

# 기본 명령어

**덮어쓰기 금지**

덮어쓰기 금지 상태에서 강제로 덮어쓰기를 하려면 다음처럼 명령어를 실행한다.

```
$ set -C
$ printf ‘Thrid Command\n’ >| /tmp/test.txt
$ echo $?
```

# 기본 명령어

## 이어붙이기

기존 텍스트 내용에 추가를 하거나 혹은 이어붙이기를 하는 경우 다음처럼 명령어를 실행한다.

$ printf ‘First Command\n’ > /tmp/test_doc.txt

$ printf ‘Second Command\n’ >> /tmp/test_doc.txt

$ cat test_doc.txt

First Command

Second Command

# 기본 명령어

## 이어붙이기

```
#!/bin/bash
printf 'Starting script\n' >> log
printf 'Creating test directory\n' >> log
mkdir test || exit
printf 'Changing into test directory\n' >> log
cd test || exit
printf 'Writing current date\n' >> log
date > date || exit
```

# 기본 명령어

## 이어붙이기

mkdir test || exit

“||” 만약 해당 디렉터리가 이미 생성이 되어 있는 경우
exit명령어를 실행한다.

# 기본 명령어

## 퍼미션

여기에서는 chmod, chown에 대해서는 자세하게 다루지 않는다.
다만, 쉘 스크립트에서 소유권 및 퍼미션을 어떻게 핸들링 하는지 학습 한다.

## 미리 필요한 선수 지식

- chown
- chmod
- umask

# 기본 명령어

## 퍼미션

stat, stat명령어를 통해서 0~7사이의 숫자 8진수를 통해서 퍼미션 확인이 가능하다.

$ stat /tmp/test.txt

stat명령어를 통해서 퍼미션 확인을 원하는 경우 아래와 같은 명령어로 확인 가능하다.

$ stat -c %a /tmp/test.txt

# 기본 명령어

**일반 시스템 명령어(cut)**

여러 개 필드를 동시에 처리를 하려면 아래와 같이 하면 된다.

```
$ cut -d: -f1,6 /etc/passwd
root:/root
bin:/bin
daemon:/sbin
adm:/var/adm
lp:/var/spool/lpd
sync:/sbin
```

# 기본 명령어

**입출력 전환**

표준 입출력 오류는 각각 숫자를 가지고 있다. bash에서는 stdout이라고 부르며 이걸 “표준출력”이라고 말하며, stderr는 경우에는 표준 오류라고 부른다.

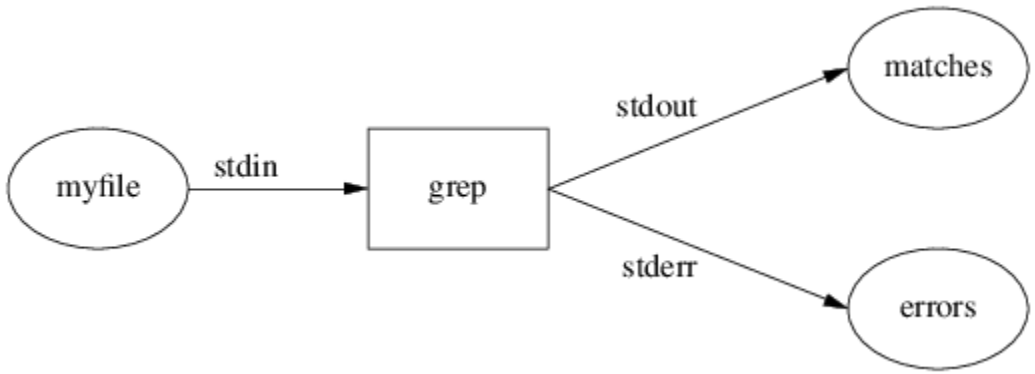
이 옵션은 매우 bash에서 스크립팅 시 매우 일반적으로 사용한다.

이걸 보통 파일 디스크립터(File Descriptor)라고 부른다.

디스크립터는 총 3개를 가지고 있다.

# 기본 명령어

## 입출력 전환



# 기본 명령어

**입출력 전환**

0: 표준 출력 및 오류에 대한 동시 출력, 일반적으로 사용하면 0번으로 기본동작

1: 표준출력, 오류가 아닌 부분에 대해서 화면에 출력한다.

2: 표준오류, 오류 부분에 대해서만 화면에 출력한다.

# 기본 명령어

**입출력 전환**

오류와 표준 출력을 같이 하는 경우 보통 다음처럼 사용한다.

$ grep pattern testfile.txt /thereisnowhere > matches 2> matches

아무튼, 위에 처럼 사용하는 경우 현재 오류 메시지만 matches파일에 기록이 남아 있다. 그래서 같이 출력을 하기 위해서는 &(ampersand)를 통해서 1에 대한 출력을 명시 해주어야 한다.

# 기본 명령어

## 입출력 전환

$ grep pattern testfile.txt /thereisnowhere > matches 2>&1

위와 같이 실행하면 위의 실행 결과는 표준 출력과 표준 오류를 동시에 파일에 저장한다.

주의할 부분은 동일한 작업을 수행하기 위해서는 >앞에다가 2>&1을 넣을 수 없습니다. 리디렉션은 왼쪽에서 오른쪽으로 항상 이루어 진다.

# 기본 명령어

## 입출력 전환

$ grep pattern testfile.txt /thereisnowhere > matches 2>&1

위와 같이 실행하면 위의 실행 결과는 표준 출력과 표준 오류를 동시에 파일에 저장한다.

주의할 부분은 동일한 작업을 수행하기 위해서는 >앞에다가 2>&1을 넣을 수 없습니다. 리디렉션은 왼쪽에서 오른쪽으로 항상 이루어 진다.

# 기본 명령어

## 블록 오류 및 처리

가끔 발생한 모든 오류를 화면에 출력할 필요는 없다. 그럴 때 사용하는 유용한 장치가 있다.

/dev/null

위의 장치로 오류 내용을 내보내면 된다. 전달하는 방법은 매우 간단하다.

$ grep pattern testfile /thereisnowhere > matches 2 > /dev/null

# 기본 명령어

**블록 오류 및 처리**

블록 오류를 처리하기 위해서 comm명령어를 사용하여 테스트 한다.

$ comm test1.txt test2.txt

$ comm test1.txt test2.txt 2> /dev/null

$ echo $?

# 기본 명령어

**여러 위치로 재 전달**

여러 파일로 내용을 재 전달 하기 위해서는 tee를 사용하여 다음과 같이 처리한다.

$ printf ‘Copy Copy rom rom\n’ | tee my_copy_rom.txt

Copy Copy rom rom

# 기본 명령어

**여러 위치로 재전달**

기존 명령어에 다음처럼 파일 이름만 더 추가한다.

$ printf ‘Copy Copy rom rom\n’ | tee myfile1 myfile2 myfile3

Copy Copy rom rom

$ ls

# 기본 명령어

**입력 재전달**

$ export LC_ALL=C

$ export LANG=C

$ grep pattern myfile1 myfile2 myfile3

$ sort -k1,1 myfile1 myfile2 myfile3

# 기본 명령어

## 입력 재전달

리눅스 명령어 중에 입력 재전달의 대표적인 명령어가 있다. 그건 바로 tr이다. tr명령어는 입력 받은 내용을 대문자 혹은 소문자로 다시 출력한다.

```
$ tr a-z A-Z
Hello, world!
HELLO, WORLD!
```

# 기본 명령어

## 입력 재전달

위의 명령어를 가지고 재전달 프로세스를 다루면 다음처럼 다룰 수 있다.

cat mysample | tr a-z A-Z

혹은 미리 작성된 파일을 대소문자로 변경하는 경우에는 다음과 같은 명령어로 실행이 가능하다.

$ tr a-z A-Z < mysample

# 기본 명령어

**입력 재전달**

재전달 오퍼레이터를 손쉽게 다음처럼 사용이 가능하다. 출력을 위한 cat같은 명령어를 사용할 필요가 없다.

$ < mysample tr a-z A-Z

혹은 아래처럼 복잡하게 입출력을 제어할 수 있다. 아래 명령어는 표준오류만 출력하는 예제이다.

$ tr a-z A-Z < mylines > mylines.capitalized 2> mylines.error

# 문자열 및 스트리밍

# 기본 명령어

## 긴 문자열 처리

```
#!/bin/bash
case $1 in
        -h | --help )
                printf '%s\n' foo command help'
-h, --help: show this help
-q, --quiet: run without diagnostics
-v, --verbose: add extra diagnostics
        exit 0
        ;;
esac
```

# 기본 명령어

## 긴 문자열 처리

```
#!/bin/bash
case $1 in
        -h | --help )
                cat << 'EOF'
-h, --help: show this help
-q, --quiet: run without diagnostics
-v, --verbose: add extra diagnostics
EOF
        exit 0
        ;;
esac
```

# 기본 명령어

## 긴 문자열 처리

```
#!/bin/bash
cat << EOF
Hello, $USER; you are running this program on $(hostname -s).
Your home directory is $HOME
EOF
```

# 기본 명령어

## 긴 문자열 처리

```
#!/bin/bash
cat <<- 'EOF'
        leading tabs
                will not be included in the output
EOF
```

# 기본 명령어

**Pipe**

```
#!/bin/bash

tr A-Z a-z > words.lowercase

sort words.lowercase > words.sorted

uniq -c words.sorted > words.frequency

sort -k1,1nr words.frequency
```

# 기본 명령어

**Pipe**

여러가지 명령어를 동시에 사용하기 위해서 쉘 스크립트 형식으로 명령어를 작성한다.

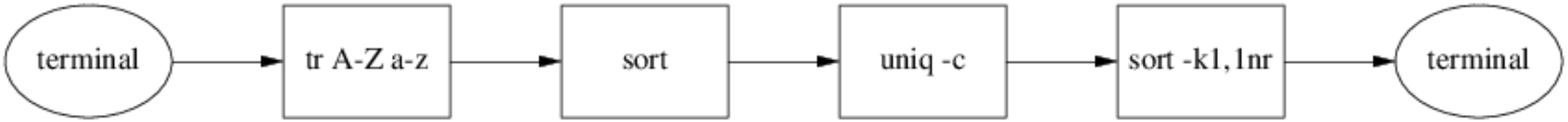
이를 동시에 처리하기 위해서 파이프를 사용한다. 예제로 tr명령어 기반으로 간단하게 구성을 하였다.

# 기본 명령어

**Pipe**

위와 같이 사용하는 경우 쉘 스크립트처럼 여러 명령어를 사용하기 때문에 다음처럼 파이프로 구성하여 동시에 처리 하도록 한다.

tr A-Z a-z | sort | uniq -c | sort -k1,1nr



# 기본 명령어

**Pipe**

$ tr A-Z a-z | sort | uniq -c | sort -k1,1nr

이것저것 입력 후 ctrl+d

sdlkjskljieurwjldklskdj
akljsnkcjnnjhdsiakjslia
38473904889orjksdjm,snbns
1 ($&$
    1 2948
    1 38473904889orjksdjm,snbns

# 기본 명령어

**Pipe**

이걸 쉘 스크립트로 변경하면 다음처럼 변경한다.

```
#!/bin/bash
tr A-Z a-z |
sort |
uniq -c |
sort -k1,1nr
```

# 기본 명령어

**Pipe**

여러 개의 텍스트 파일을 하나로 통합할 때 다음처럼 스트리밍 처리한다.

$ cat testfile1 testfile2 testfile3 > testfiles.combined

$ date > output

$ hostname >> output

$ cat output

# 기본 명령어

**Pipe**

그룹 명령어는 중괄호를 사용하여 처리한다. 아래처럼 실행하면 모든 출력이 대문자로 출력이 된다.

$ { date ; hostname ; } | tr a-z A-Z

# 기본 명령어

**Pipe**

$ { date ; printf ‘%s\n’ apple carrot banana ; } > testwords

$ { read date ; printf ‘%s\n’ “$date” ; sort ; } < testwords

# 기본 명령어

**stream editor**

$ cat /usr/share/doc/ls/LICENSE

$ sed '' /usr/share/doc/ls/LICENSE

$ sed '1,2d' /usr/share/doc/ls/LICENSE

$ sed '1,/CONFIG_SYSFS_DEPRECATED/d' /usr/share/doc/ls/LICENSE

$ sed 's/found/not_found/' /usr/share/doc/ls/LICENSE

# 기본 명령어

**AWK**

```
$ cat /etc/passwd

$ awk '{ print $1 }' /etc/passwd

$ awk '{ print $2, $3 }' /etc/passwd

$ awk 'NR > 1 { print $2, $3 }' /etc/passwd

$ awk 'NR > 1 { printf "$%.2f\n", $2 * $3 }' /etc/costs.txt
```

# 변수

# 변수 및 패턴

**변수**

쉘에서 변수는 다음과 같이 선언한다.

tangshell='This is mine shell'

tangshell를 변수 이름이 되며, 그 뒤에 홑 따옴표에 있는 내용은 쉘 변수에서 사용하는 자료(data)가 된다.  이걸 BASH에서는 string conten라고 한다.

# 변수 및 패턴

## 변수

bash는 C 스타일을 사용하기 때문에 변수 이름에 절대 빈 공간이 있으면 안된다. 보통 다음과 같이 많이 선언한다.

```
tangshell = 'fish'
tangvar='GNU'\''s Not Unix!'
tangprompt="$USER$HOST"
```

변수를 호출 할때 C언어와 마찬가지로 $붙여서 접근이 가능하며, 읽기전용(read only)상태이다.

# 변수 및 패턴

## 변수 목록

변수 목록 및 출력을 하기 위해서는 다음과 같은 방법을 사용해서 호출한다.

```
$ today="2022-06-26"
$ printf '%s\n' "$today"
2022-06-26

$ declare -p today
declare -- today='2022-06-26'
```

# 변수 및 패턴

**변수 목록**

declare에 표시되는 "--"는 특별한 속성이 아니라는 표시이다.

declare -p명령어를 실행하면 다음과 같은 내용을 볼 수 있다.

declare -- BASH="/bin/bash"
declare -- PS1="\\u@\\h:\\w\\$ "

# 변수 및 패턴

## 변수 네이밍

변수 네이밍에 대한 규칙은 따로 없지만 일반적으로 다음과 같은 규칙을 가진다.

- test_var
- TestVar
- testvar
- Testvar
- _testvar
- testVar
- TestVar_

# 변수 및 패턴

**변수 네이밍**

하지만 다음과 같은 이름은 허용하지 않는다.

- 1testVar
- test-var
- test.var
- test:var

# 변수 및 패턴

## 변수 네이밍

또한 변수 설정 시 기존 설정된 이름과 충돌을 주의해야 한다. 매우 자주 있는 케이스는 보통 다음과 같다.

```
#!/bin/bash
PATH=/home/tang/sbin:/home/tang/bin
grep -- teststring "$PATH"
```

결과는 다음과 같이 출력이 된다.

```
grep: command not found
```

# 변수 및 패턴

## 변수 네이밍

이유는 간단하다. $PATH는 시스템 변수이며, 기존에 설정된 경로 값에 덮어쓰기가 되어버린 상태다. 그래서 path라는 이름을 사용한다면 다음처럼 선언을 보통 한다.

```
#!/bin/bash
path=/home/tang/bin:/home/tang/sbin
grep -- teststring "$path"
```

# 변수 및 패턴

**변수 네이밍**

혹은 기존 $PATH를 사용하면서 내용을 추가하는 경우, 다음처럼 처리가 가능하다.

```
PATH="$PATH:/home/tang/bin/:/home/tang/sbin"
MYHOME=/home/tang
export MYHOME
```

# 변수 및 패턴

**변수 취소**
변수 초기화 하는 방법은 다음과 같다.

$ system_var =

그리고 초기화 된 내용을 확인 시 다음과 같은 방법으로 가능하다.

$ declare -p system
declare -- system=""

# 변수 및 패턴

## 변수 취소

POSIX 기준으로 변수에 값이 설정이 되어 있는지 확인 시, 다음과 같이 if문으로 확인한다.

```
#!/bin/bash
if [[ -n $test_system ]] ; then
  print 'the 'test_system' value is: %s\n' "$system"
fi
```

# 변수 및 패턴

**변수 취소**

변수를 완전히 제거하려면 unset명령어를 통해서 할 수 있다.

$ unset -v test_system

$ declare -p system

bash: declare: system: not found

# 변수 및 패턴

**환경변수**

쉘에서 제공하는 환경변수가 있다. 일반적인 변수들은 현재 쉘에서만 적용이 되며, 새로운 프로세서에서는 이전 쉘에서 추가적용 하였던 변수는 적용되지 않는다.

이를 우리는 forked process라고 부른다. 이 환경변수를 쉽게 확인하는 방법은 다음과 같은 방법으로 확인이 가능하다.

# 변수 및 패턴

## 환경변수

$ declare -p BASHPID

declare -ir BASHPID="327276"

$ shellusername='Tang'

$ declare -p shellusername

declare -- shellusername='Tang'

# 변수 및 패턴

**환경변수**

$ bash

$ declare -p BASHPID

declare -ir BASHPID= "331219"

$ declare -p shellusername

declare: shellusername: not found

# 변수 및 패턴

**환경변수**

다시 이전 환경으로 돌아가기 위해서는 exit명령어로 돌아가면 된다.

$ exit

$ declare -p BASHPID

$ declare -p shellusername

# 변수 및 패턴

## 환경변수

만약 다른 환경에도 값을 전달하고 싶은 경우 POSIX명령어 export로 선언한다.

$ USERNAME="choi"

$ export USERNAME

$ declare -p USERNAME

declare -x USERNAME="choi"

# 변수 및 패턴

## 환경변수

앞에서 이야기 하였지만, 변수 이름 설정 시 전부 대문자로 설정한다. 여하튼, expor로 설정하는 경우 declar명령어로 변수를 확인하면 다음처럼 변수 이름이 출력이 된다.

```
$ bash
$ declare -p USERNAME
declare -x USERNAME="choi"
$ exit
```

# 변수 및 패턴

## 환경변수

환경변수는 bash에서만 사용하는게 아니다. 모든 프로그램 및 환경에서 적용이 되며, 혹은 다른 프로그램 언어들, Python, PHP, Awk같은 명령어에서도 적용이 된다.

예를 들어서 sort, date, 혹은 ls같은 명령어는 몇몇 시스템 변수에 적용이 받는데 그중 하나가 LANG, LC_ALL, LC_COLLATE변수이다.

# 변수 및 패턴

## 환경변수

```
$ printf '%s\n' 'helloworld' 'HelloTang' > words
$ LC_COLLATE=en_US.UTF-8
$ export LC_COLLATE
$ sort words
helloworld
HelloTang
```

# 변수 및 패턴

**환경변수**

$ LC_COLLATE=C

$ sort words

HelloTang

helloworld

# 변수 및 패턴

**환경변수**

$ LC_COLLATE=C sort words

HelloTang

helloworld

# 변수 및 패턴

**변수확장**

쉘에서 변수를 확장하기 위해서 $1를 사용한다. 일반적으로 $0은 자신이 사용하는 명령어, $1부터는 확장 파라메타라고 한다.

```
$ yourname='tang'
$ printf '%s\n' "Hello, $yourname."
Hello, Tang.
```

# 변수 및 패턴

**변수확장**

변수 이름을 좀 더 명확하게 보기 위해서 중괄호로 표시를 할 수 있다.

```
$ yourname='tang'
$ printf '%s\n' "__Hello, ${yourname}__."
__Hello, Tang__.
```

# 변수 및 패턴

**변수확장**

역시 변수에 값이 선언이 되어 있지 않는 경우에는 다음처럼 출력이 된다.

$ printf '%s\n' "Hello, $notest."

Hello, .

# 변수 및 패턴

**변수확장**

만약에 선언이 되지 않는 변수에 대해서 오류 메시지를 출력하고 싶으면 다음처럼 선언한다.

$ set -u

$ printf '%s\n' "Hello, $notest."

bash: notest: unbound variable

# 변수 및 패턴

**변수확장**

변수을 외부에서 받아 오기 혹은 읽어오기를 할때 내장함수인 read를 사용해서 사용한다. 이때 같이 사용하는 옵션은 -r를 같이 사용해서 변수에 할당한다.

$ read -r yourname

$ declare -p yourname

declare -- yourname='tang'

# 변수 및 패턴

**명령어 결과를 변수로**

종종 사용하는 방법으로, 실행된 명령어의 결과를 변수에 저장한다. 이유는 여러가지가 있지만 일반적으로 재가공 혹은 출력 용도로 사용한다. 다음과 같은 예제를 본다.

```
$ whom=$(whoami)
$ printf '%s\n' "$whom"
tang
```

# 변수 및 패턴

**명령어 결과를 변수로**

두개 이상의 명령어를 동시에 사용해야 되는 경우 다음처럼 명령어를 구성한다.

```
$ yourpasswd="$(getent passwd $(whoami)")"
```

# 변수 및 패턴

## 기본값 설정

기본값 설정을 종종 필요하다. 예를 들어서 시스템에서 올바르게 기본값이 설정이 안되어 있으면 올바르게 동작이 안될 수 있다. 그래서 쉘에서는 "${default_var:-default}"같은 형식을 제공한다.

$ cd -- "${MYDATA_DIR:-/usr/local/data}"

혹은 다음과 같이 재선언이 가능하다.

$ cd -- "${MYDATA_DIR:=/usr/local/data}"

# 문자열

# 변수 및 패턴

## 자르기

```
$ ta='type:json'
$ print 'Type: %s\n' ${ta#type:}"
Type: json

$ ta='type:json'
$ printf 'Field name: %s\n' "${ta%:*}"
Field name: type
```

# 변수 및 패턴

## 자르기

```
$ text='*WARNING'
$ printf '%s\n' "${text#\*}"

$ printf '%s\n' '${text#'*'}"

$ path=/usr/local/test/bin/test
$ printf ' Removed the path as look like a file : %s\n' "${path##*/}"
Removed the path as look like a file: test
```

# 변수 및 패턴

## 문자열 자르기

문자열 처리 시, cut같은 명령어 사용이 가능하지만, 조금 더 편하게 ${var:<position>:<length>}사용이 가능하다.

```
$ title='== Password =='
$ printf '%s\n' "${title:3}"
Password ==
$ printf '%s\n' "${title:3:8}"
Password

$ alpha='abcdefghijklmn'
$ printf '%s\n' "${alpha: -3:2}"
ij
```

# 변수 및 패턴

## 길이측정

문자열 길이는 wc같은 명령어로 확인이 가능하다. 하지만, 쉘 빌트인을 사용하는 경우 좀 더 쉽게 가능하다.

```
$ alpha='abcdefghijklmn'
$ printf '%u\n' "${#alpha}"
11
```

# 변수 및 패턴

## 문자열 바꾸기

```
$ notice='I'\''ll do it today.'
$ printf '%s\n "${notice /today/tomorrow}'

$ notice='Yes, today. I'\''ll do it today.'
$ printf '%s\n' "${notice /today/tomorrow}"
Yes, tomorrow, I'll do it today.
$ printf '%s\n' "${notice //today/tomorrow}"
Yes, tomorrow, I'll do it tomorrow.
```

# 변수 및 패턴

## 문자열 바꾸기

```
$ notice='Yes, today. I'\''ll do it today.'
$ printf '%s\n' "${notice/today*/I\'ll do it soon.}"
Yes, I'll do it soon.
```

# 변수 및 패턴

## 대소문자 변경

```
$ text='helloWorld'
$ printf '%s\n' "${text^^}"
HELLOWORLD
$ printf '%s\n' "${text^}"
HelloWorld
$ printf '%s\n' "${text,}"
helloWorld
```

# 변수 및 패턴

## 외부 변수 합치기

```
#!/bin/bash
temp="${test:-foo}"
printf '%s\n' "${temp#.ext}"
```

# 쉘 산수

# 변수 및 패턴

## 쉘에서 산수하기(?)

```
$ tmp_a=3
$ tmp_b=2
$ printf 'The sum of the a + b is %u\n' $((tmp_a + tmp_b))"
The sum of the a + b is: 5
```

# 변수 및 패턴

**쉘에서 산수하기(?)**

```
$ diff=$((tmp_a - tmp_b))
$ printf 'The difference of the two numbers is: %u\n' "$diff"
The difference of the two number is: 1

$ printf '%u\n' "$(3**2))
$ printf '%u\n' "$((180/60))"
$ printf '%u\n' "$((2>1))"
$ printf '%u\n' "$((30>40))"
```

# 변수 및 패턴

## 소수점 자리 고정 혹은 유동적

```
$ printf '%.2f\n' "$((3/2))"
1.00

$ bc <<'EOF'
scale=2
3/2
EOF
1.50
$ awk 'BEGIN { printf "%.2f\n", 3/2 }'
1.50
```

# 글로브 혹은 글로빙

# 변수 및 패턴

## 글로브(*) 사용하기

글로브, 별표 혹은 눈이라고 말하는 문자이다. 쉘에서는 글로브 즉 별표를 사용하면 기본적으로 모든것(all of match)라고 표현한다. 매치가 되는 자원은 거의 모든 자원에 적용이 된다.

```
$ ls -a
.    ..    Documents  Pictures   Downloads

$ printf '%s\n' *
hello
world
Documents
Downloads
```

# 변수 및 패턴

## 글로브(*) 사용하기

```
$ printf '%s\n' ???uments
Documents

$ printf '%s\n' *[ts]
test1.txt
test2.txt
```

# 변수 및 패턴

## 글로브(*) 사용하기

$ printf '%s\n' ???uments

Documents

$ printf '%s\n' *[!ts]

jsues.txt

sjwue.txt

# 변수 및 패턴

## 글로브(*) 사용하기

```
$ cat -- c*
cat: 'c*' : No such file or directory

$ shopt -s nullglob
$ printf '%s\n' c*
$
```

'shopt'명령어는 bash의 빌트인 명령어이며, 내부 함수를 설정한다. 위의 명령어는 'nullglob'를 활성화 한다.

# 변수 및 패턴

## 글로브(*) 사용하기

```
$ shopt -s nocaseglob
$ printf '%s\n' A*
apple
Aux
```

# 변수 및 패턴

## 글로빙 확장하기

bash는 부족한 글로빙 기능을 확장하기 위해서 정규식(regular express)기반으로 고급기능(혹은 더 복잡하게) 사용이 가능하다.

?(pattern): 하나라도 패턴에 맞으면 출력
+(pattern): 최소 한 개로도 맞으면 출력
*(pattern): 숫자 하나라도 맞으면 출력
@(pattern): 명시한 조건만 맞으면 출력
!(pattern): 조건에 맞지 않는 것 만 출력

# 변수 및 패턴

## 글로빙 활용하기

예를 들어서 자주 사용하는 /var/log를 예를 들면 다음처럼 사용한다.

## /var/log에서 .gz로 끝나지 않는것만 출력

$ ls -l /var/log/!(*.log)

## haw혹은 mess단어가 맞으면 화면에 출력

$ ls -l /var/log/@(haw|mess)*

# 변수 및 패턴

## 글로빙 활용하기

$ ls -l /var/log/sys*

# 배열

# 변수 및 패턴

## 배열 사용하기

쉘에서 배열을 사용하기 위해서는 경계기호(delimiter)를 사용해서 변수와 배열을 구별한다.

$ devices=('iphone', 'android', 'winphone')

위의 배열은 매우 기본적인 배열 선언이다.

# 변수 및 패턴

## 배열 사용하기

```
$ printf '%s\n' "${devices[0]}"
iphone

$ printf '%s\n' "${divces[2]}"
winphone
```

# 변수 및 패턴

## 배열 사용하기

```
$ printf '%s\n' "${devices[-1]}
winphone

$ printf '%s\n' "${devices[@]}"
iphone
android
winphone
```

# 변수 및 패턴

## 배열 사용하기

```
$ printf '%s\n' "${devices[@]^^}"
IPHONE
ANDROID
WINPHONE
```

# 변수 및 패턴

## 배열 사용하기

```
$ homefiles=("$HOME"/*)
$ printf '%s\n' "${homefiles[@]}"
/home/tang/1.txt
/home/tang/2.txt
/home/tang/3.txt
/home/tang/4.txt
```

# 변수 및 패턴

## 배열 사용하기

bash4.0 이후에 새로 할당된 배열 할당이 있다. 파이썬의 사전(dictionary)처럼 기능을 제공한다.

```
$ declare -A colors
$ colors=([tomato]="red" [orange]="yellow" [carrot]="orange")
$ printf '%s\n' "${colors[orange]}"
yellow
```

# 루프 및 조건문

# 루프 및 조건문

## 조건문

쉘에서 조건문을 사용하기 위해서 다음과 같은 조건이 필요하다.

- if
- [, [[
- ((
- case
- for
- while

# 루프 및 조건문

## 조건문

if문을 사용하기 위해서는 다음과 같은 방법으로 구현이 가능하다.

```
$ grep -q bash /etc/shells && printf 'bash is a system shell\n"
```

먼저 중간에 있는 &&기호는 뒤에 두번째 명령어를 실행하는데, 이때 조건이 있다. -q bash 문자열이 /etc/shells에 있으면 true를 반환하면, 그 다음 명령어 printf를 실행한다.

만약, 해당 문자열이 없으면 false반환이 되기 때문에 뒤에 명령어가 실행이 되지 않는다.

# 루프 및 조건문

## 조건문

이걸 if문으로 변경하면 다음처럼 된다.

```
$ if grep -q bash /etc/shells ; then printf 'bash is a system shell\n' ; fi
```

좀 더 읽기 편하게 하기 위해서 다음처럼 구현이 가능하다.

```
if grep -q bash /etc/shells ; then
  printf 'bash is a system shell\n'
fi
```

# 루프 및 조건문

**조건문**

if 명령어는 반드시 fi로 종료를 해야 한다.

```
if cd /usr/local ; then
  printf >&2 'changed directory\n'
fi
```

# 루프 및 조건문

**조건문**

반대로도 사용이 가능하다.

```
if ! cd /usr/local/ ; then
  printf >&2 'Failed to changed directory\n'
  exit 1
fi
```

# 루프 및 조건문

## 조건문

if문에서 조건이 맞지 않았을 때 동작하는 else문법을 구현하기 위해서는 elif키워드를 사용한다.

```
if cd /usr/local/mysql ; then
  printf >&2 'Changed to primary directory'
elif cd /opt/mysql ; then
  printf >&2 'Changed to secondary directory'
else
  printf >&2 'Cound'\''t find a directory!'
  exit 1
fi
```

# 루프 및 조건문

## 조건문

POSIX기준으로 스크립팅 언어에서 test라는 도구를 사용할 수 있다.
그냥 test명령어만 사용하는 경우 다음처럼 사용하면 된다.

```
$ yourshell=bash
$ test "$yourshell" = 'sh' && printf 'DingDong!\n' echo -ne '\007\'
$ test "$yourshell" = 'bash' && printf 'DingDong!\n'  echo -ne '\007\'
```

# 루프 및 조건문

## 조건문

POSIX기준으로 스크립팅 언어에서 test라는 도구를 사용할 수 있다.
그냥 test명령어만 사용하는 경우 다음처럼 사용하면 된다.

```
$ yourshell=bash
$ test "$yourshell" = 'sh' && printf 'DingDong!\n' echo -ne '\007\'
$ test "$yourshell" = 'bash' && printf 'DingDong!\n'  echo -ne '\007\'
```

# 루프 및 조건문

## 조건문

**-e: file이라는 디렉터리나 파일이 있으면 참**

test -e 'file':

**-n: $vars라는 변수가 선언이 되어 있고 비어있 지 않는 경우.**

test -n "$vars"

# 루프 및 조건문

## 조건문

**-d: 디렉터리 이름이 dir인 경우**

test -d 'dir'

**-s: 파일이 존재하며 파일이 비어 있지 않는 경우**

test -s 'file'

# 루프 및 조건문

## 명령어 [

test명령어는 [명령어. [명령어는 두 개의 명령어가 있어야 올바르게 동작한다. []이 두개의 명령어는 쉘 스크립트에서 많이 사용하며, 또한 이것들은 그냥 명령어이다. 쉘 스크립트의 키워드와 다르다.

```
$ [ "$yourshell" = 'bash' ] && printf 'Correct!\n'
```

if문은 grep, cd명령어처럼 사용이 가능하다.

```
yourshell=bash
if [ "$myshell" = 'bash' ] ; then
  printf 'Correct!\n'
fi
```

# 루프 및 조건문

## 명령어 [[

이전 bash는 [를 사용하였지만, bash 4.0이후부터는 [[으로 변경되었다.
이 문법은 bash에서 특별한 문법이며 일반적인 빌트인 명령어가 아니다.

```
#!/bin/bash
yourshell=bash
if [[ $yourshell = 'bash' ]] ; then
  printf 'correct!\n'
fi
```

# 루프 및 조건문

**명령어 [[**

[, [[하고 제일 큰 차이점은 다음과 같다.

if [ "$myshell" = 'bash' ] ; then

if [[ $yourshell = 'bash' ]] ; then

[ 경우에는 변수를 쌍 따옴표로 감싸야 한다.

# 루프 및 조건문

**명령어 [[**

[[ $yourshell = "$yourshell" ]]

하지만, 기존 방식하고 혼동이 될 수 있기 때문에 그냥 이전에 사용하던 방식으로 사용해도 문제가 없다.

[[ "$yourshell" = "$yourshell" ]]

[방식과 비슷하지만, [[에서 크게 문제 없이 동작한다.

# 루프 및 조건문

**명령어 [[**

기존에 사용하였던 test, [처럼 [[에서도 조건문 사용이 가능하다.

[[ $yourshell = b* ]]

위와 같이 실행하는 경우 이전 앞에서 사용한 글러빙처럼 b로 시작하는 문자열을 찾는다.

# 루프 및 조건문

**명령어 [[**

혹은 다음 =~ 새로운 지시자(operator)를 사용해서 시작 문자를 지정할 수 있다.

[[ $yourshell =~ 'sh$' ]]

위의 명령어를 실행하면 마지막 문자가 sh로 끝나는 찾아서 화면에 출력한다.

# 루프 및 조건문

**산술조건(Arithmetic condition)**

POSIX기준으로 test, [명령어를 사용해서 산술 연산자 사용이 가능하다.

[ "$a" -eq "$b" ]: 같다
[ "$a" -ne "$b" ]: 같지 않다
[ "$a" -lt "$b" ]: 작거나
[ "$a" -le "$b" ]: 작거나 혹은 같거나
[ "$a" -gt "$b" ]: 크거나
[ "$a" -ge "$b" ]: 크거나 혹은 같거나

# 루프 및 조건문

**산술조건(Arithmetic condition)**

바이트, 키로바이트 출력을 위한 조건문을 적으면 아래와 같이 1,000으로 나누어서 비교하는 조건문을 아래와 같이 일반적으로 작성한다.

```
if  [ "$((bytes / 1000))" -gt "$kbytes" ] ; then

fi
```

# 루프 및 조건문

**산술조건(Arithmetic condition)**

하지만 bash에서 산술을 위한 특별한 문법이 있다. $대신 ((으로 표현이 가능하다.

```
if ((bytes / 1000 > kbytes )) ; then


fi
```

# 루프 및 조건문

**산술조건(Arithmetic condition)**

모든 산술조건이 올바르게 맞으면 보통 다음처럼 값을 반환한다.

(( 2 > 0)) : True, exit 0

((a = 1)): True, exit 0

(( 0 >3 )): False, exit 1

(( 0 )): False, exit 1

((a = 0 )): False, exit 1

# 루프 및 조건문

**산술조건(Arithmetic condition)**

모든 산술조건이 올바르게 맞으면 보통 다음처럼 값을 반환한다.

(( 2 > 0)) : True, exit 0

((a = 1)): True, exit 0

(( 0 >3 )): False, exit 1

(( 0 )): False, exit 1

((a = 0 )): False, exit 1

# 스크립트 펑션(function)
# 및 별칭(Alias)

# 함수

**Function**

함수는 명령어를 기능별로 묶을 때 사용하는 가능. C언어의 "void function(){} return 0"과 비슷한 기능.

함수는 일반적으로 여러가지 명령어를 사용하여 하나의 세트로 만든 다음에 그 기반으로 동작 순서(routine)를 구성한다.

# 함수

**Function**

함수는 다음과 같은 구조로 생성한다.

```
function HelloWorldSkt(){
  echo "Hello SKT"
}
```

호출은 다음과 같은 명령어로 함수를 호출한다.

```
HelloWorldSkt
```

# 함수

**Function**

생성한 함수를 확인 하고 싶은 경우, 다음명령어로 확인이 가능하다.

$ declare -F <function>

$ delclare -F HelloWorldSkt

# 함수

**Function**

함수 생성 후 함수를 제거(unset)하고 싶은 경우 다음과 같은 명령어로 함수 선언을 해제한다.

$ unset <function>

$ unset HelloWorldSkt

# 함수

## Function

함수에서 변수 선언은 매우 단순한다.

```
var1=1
var2=2
function change(){
  echo internal function
  echo "Variable 1 is: $var1"
  echo "Variable 2 is: $var2"
  local var1=5
```

# 함수

**Function**

```
var2=5
  echo
  echo "After change internal function
   echo Variable 1 is locally $var1
   echo Variable 2 is gobally $var2
}
```

# 함수

**Function Arguments**

bash에서 함수 인자값은 다음처럼 다룬다.

**$:** 예약된 쉘 함수 인자 값. 보통 $0번은 명령어(스크립트)실행 위치를 표시

**$1, $2, etc:** $9번까지는 일반적으로 받는 변수 이름

**$#:** 현재 가지고 있는 인자 값 출력

# 함수

**Function Arguments**

bash에서 함수 인자값은 다음처럼 다룬다.

**$@, $*:** 둘 다 기능은 비슷하다. 차이점은 리스트이냐 혹은 문자열이냐 차이다.

**$@:** 문자열을 리스트 형태로 변환한다. 스페이스 기반으로 리스트 배열을 구성한다.

**$*:** 리스트를 하나의 문자열로 만든다. 스페이스 기반으로 분리한다. $1, $2

# 함수

## 종료코드

bash에서 사용하는 리턴 코드는 보통 다음과 같다.

**1:** 일반적인 오류 코드.

**2:** 빌트인 키워드 올바르게 찾지 못함, 예를 들어서 올바르지 않는 함수.

**126:** 명령어 실행이 되지 못함. 예를 들어서 /dev/null를 명령어로 접근.

**127:** 명령어를 찾을 수 없음. $PATH나 혹은 오타(typo)

**128:** 올바르지 않는 리턴 값. 예를 들어서 exit 3.14159.

**128+n:** 오류 코드. 예를 들어서 프로그램 종료 시 kill -9한 경우 128+9.

**130:** 스크립트 종류. 예를 들어서 ctrl+c 128코드에 +2.

**255*:** 프로그램이 종료가 되었으나, 범위에 포함이 되지 않음.

# 함수

**alias**

alias는 긴 명령어 혹은 명령어 + 옵션을 짧게 줄여서 사용하기 위해서 사용한다.

설정 되어있는 alias를 확인하기 위해서 다음 명령어로 확인이 가능하다.

$ alias

$ alias -p

# 함수

**alias**

alias설정은 다음처럼 한다.

$ alias move='mv -I'

$ alias Documents='/tmp/documents/'

# 함수

**alias**

해제하기 위해서는 다음 명령어로 alias해제가 가능하다.

$ unalias -a

$ unaias move

# 조건문 / 루프

# 연습문제

# 연습문제

**Lab1**

```
#!/bin/bash

for CreateUser in helix jay syndy jeff
do
   adduser $CreateUser 2> /dev/null
done

jeff=$(grep ^jeff /etc/passwd | awk -F: '{ print $1 }')
jay=$(grep ^jay /etc/passwd | awk -F: '{ print $1 }')
```

# 연습문제

## Lab1

- helix
- jay
- syndy
- jeff

1. 위의 모든 사용자는 비밀번호가 설정이 안되어야 한다.
2. 사용자를 다음 명령어로 생성 후 올바르게 생성이 되었는지 확인한다.
3. 생성된 사용자 중에서 jeff사용자가 있으면 "Welcome to Jeff!!"라는 메세지를 콘솔에 출력한다.
4. 생성된 사용자 중에서 jay사용자가 있으면 "Hell low jay!! :X"라는 메세지를 콘솔에 출력한다.
5. 사용자 서비스 http가 /etc/services에 있는지 확인한다.
6. 사용자 서비스 ssh가 /etc/services에 있는지 확인한다.
7. 4, 5에 결과는 shell 콘솔로 메세지 출력한다.

- 예상 출력 결과

# 연습문제

**Lab1**

**예상 출력 결과**

Welcome to Jeff!!

Hell low jay!! :x

http: http 80/tcp

ssh: ssh 22/tcp

# 연습문제

## Lab1

```
if [[ $jeff == jeff ]]
then
echo "Welcome to jeff!!"
fi

if [ $jay == jay ]
then
echo "Welcome to jay!!"
fi

echo $(grep ^http /etc/services | head -1 | awk '{ print $1, $2 }')
echo $(grep ^ssh /etc/services | head -1 | awk '{ print $1, $2 }')
```

# 연습문제

## Lab2

location=/var/www/html

file_name="index.html"

**힌트**

- 한번 스크립트 실행 후, yum remove httpd -y
- 디렉터리만 검사! "/var/www/html"
- !이것은 부정 조건
- 1: 표준출력, 2: 표준오류
- systemctl is-active 0: true, 3: false
- rpm -qa 실행시 내용 안보이게
- 추가된 if문장은
- -f: 일반파일, -e 파일, -d 디렉터리

# 연습문제

## Lab2

```
function menu(){
    clear
    echo
    echo -e "\t\t\tHTTP MENU\n"
    echo -e "\t1. Control Httpd Service"
    echo -e "\t2. Create to the \"index.html\""
    echo -e "\t3. Check the httpd.service"
    echo -e "\t0. Exit Program\n\n"
    echo -e "\t\Enter Number: "
    read -n 1 option
}
```

# 연습문제

## Lab2 Hint

```
function menu(){
    clear
    echo
    echo -e "\t\t\tHTTP MENU\n"
    echo -e "\t1. Control Httpd Service"
    echo -e "\t2. Create to the \"index.html\""
    echo -e "\t3. Check the httpd.service"
    echo -e "\t0. Exit Program\n\n"
    echo -e "\t\Enter Number: "
    read -n 1 option
}
```

# 앤서블 모듈

문제가 있는 소스코드는
제외하였습니다.

# 앤서블 모듈

**설명**

앤서블 모듈은 파이썬 언어 기반으로 작성 권장.

하지만, 대 다수가 많이 사용하는 쉘 언어 기반으로도 사용이 가능.

이 목차에서는 짧지만 어떻게 쉘 스크립트 기반으로 작성하는지
그리고 집적 만들어서 테스트 해본다.

# 앤서블 모듈

**설명**

여러 언어들을 사용하여 모듈을 만들 수 있지만, 간단한 지식을 통해서 빠르게 만들 수 있는 언어는 아마 쉘 스크립트가 유일하다.

Perl, Python, Go Lang, PHP같은 언어로 만들려고 하는 경우 다시 언어에 대해서 기본적인 학습이 필요하기 때문에 그러한 과정을 최대한 줄이고 간결하게 사용할 수 있도록 리눅스에서 많이 사용하는 bash script기반으로 작성한다.

# 앤서블 모듈

**설명**

앤서블 모듈은 다음과 같은 인자 값 핸들링 한다.

bashmodule:

  dest: /tmp/hello

  state: present

위와 같은 구조를 가지고 있다.

# 앤서블 모듈

**설명**

이 변수를 쉘에서는 다음과 같은 방식으로 핸들링 한다.

$ source $1

$ echo $1 $2

# 앤서블 모듈

## 출력

앤서블 모듈은 다음과 규칙이 적용이 된다.

**changed** 성공적으로 작업이 완료가 된 경우 출력되는 메시지. 올바르게 수행이 되면 **true**, 수행이 되지 않으면 false으로 반환한다.

**failed**: 작업 수행이 실패한 경우, 해당 작업에 대해서 실패하면 true를 반환한다.

**msg**: 성공/오류/변경 같은 작업이 문제없이 수행이 되는 경우 메시지를 반환 합니다.

# 앤서블 모듈

**출력**

위의 기능은 JSON기반으로 동작하면 메시지는 다음과 같은 형식으로 전달 합니다.

```
echo ‘{“changed”: true, “msg”: “Updated some thing”}’
```

반드시 모든 데이터는 JSON형태, 그리고 중괄호 안에서 홑 따옴표 및 쌍 따옴표로 전달이 되어야 한다. 다만, 안쪽에 있는 데이터 형식은 딕셔너리 형태로 반드시 쌍 따옴표로 구별이 되어 있어야 한다.

# 앤서블 모듈

## 출력 시 주의사항

하지만, 쉘 스크립트에서는 특수문자 핸들링 하기 위해서는 다음과 같이 두 가지 해결 방법이 있다.

```
echo “{\“changed\”: true, \“msg\”: \“Updated some thing\”}”
```

혹은 다음과 같이 bash에서 printf를 사용하여 출력이 가능하다.

```
printf ‘{“changed”: true, “msg”: “%s”}’ “$msg”
```

쉘에서 json형태로 출력 시 위의 같은 형태로 구현만 해주면 된다.

# 앤서블 모듈

## 출력 시 주의사항

하지만, bash또한 JSON형태로 출력이 불가능하기 때문에 파이썬 같은 도구를 사용해서 출력 도움을 받아야 한다.

```
msg="{\"changed\": true, \"msg\": \"$msg\"}"
echo $(echo “$msg” | python -c ‘import json,sys; print
json.dumps(sys.stdin.read())’)
```

쉘에서 실행하면 다음과 같은 결과가 출력이 된다.

# 앤서블 모듈

**출력**

간단하게 정리하면, echo, printf 둘 중 아무거나 사용하여도 됨.
올바르게 json 형태로 출력하면 된다.

# 앤서블 모듈

## 간단한 문법

앤서블 모듈 작성시 꼭 필요한 기본 함수
명시된 함수는 없으나 모듈에서 사용하는 옵션 그대로 함수를 생성 해주어야 함.

예를 들어서 다음과 같은 옵션이 있으면 다음처럼 함수를 만들어야 함.

```
backup:
  src: /usr/share/docs
  dest: /tmp/docs.tar
  label: 20220202_backup_ansible
```

# 앤서블 모듈

## case문 예외처리

case:문법을 통해서 예외 처리가 필요함. 아래는 모듈에서 사용하는 옵션 핸들링 하는 부분.

```
case $state in
    present)
        create_file
        ;;
    absent)
        delete_file
        ;;
    upper)
        convert_to_upper
        ;;
```

# 앤서블 모듈

## case문 예외처리

case:문법을 통해서 예외 처리가 필요함. 아래는 모듈에서 사용하는 옵션 핸들링 하는 부분.

```
lower)
        convert_to_lower
        ;;
    *)
        printf '{"failed": true, "msg": "invalid state: %s"}' "$state"
        exit 1
        ;;
esac
printf '{"changed": %s, "msg": "%s", "contents": %s}' "$changed" "$msg"
"$contents"
exit 0
```

# 앤서블 모듈

## 코드 구성

```
#!/bin/bash

function create_file
    {
    if [ -f "$dest" ]; then
        changed="false"
        msg="file already exists"
    else
        echo 'Hello, "world!"' >> $dest
        changed="true"
        msg="file created"
    fi
    contents=$(cat "$dest" 2>&1 | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    }
```

# 앤서블 모듈

## 코드 구성

```
function delete_file
    {
    if [ -f "$dest" ]; then
        changed="true"
        msg="file deleted"
        contents=$(cat "$dest" 2>&1 | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
        output=$(rm -f $dest 2>&1 | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
        if [ $? -ne 0 ]; then
            printf '{"failed": true, "msg": "error deleting file", "output": %s}' "$output"
            exit 1
        fi
    else
        changed="false"
        msg="file not present"
        contents='""'
    fi
    }
```

# 앤서블 모듈

## 코드 구성

```
function convert_to_upper
    {
    if [ ! -f "$dest" ]; then
        create_file
        msg="$msg, "
    fi
    current=$(cat $dest)
    new=$(echo "$current" | tr '[:lower:]' '[:upper:]')
    if [ "$current" = "$new" ]; then
        changed="false"
        msg="${msg}file not changed"
        contents=$(printf "$current" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    else

        changed="true"
        msg="${msg}file converted to upper case"
        contents=$(printf "$new" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    fi
    }
```

# 앤서블 모듈

## 코드 구성

```
function convert_to_lower
    {
    if [ ! -f "$dest" ]; then
        create_file
        msg="$msg, "
    fi
    contents=$(ls -l "$dest" 2>&1 | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    current=$(cat $dest)
    new=$(echo "$current" | tr '[:upper:]' '[:lower:]')
    if [ "$current" = "$new" ]; then
        changed="false"
        msg="${msg}file not changed"
        contents=$(printf "$current" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    else
        echo "$new" > $dest
        changed="true"
        msg="${msg}file converted to lower case"
        contents=$(printf "$new" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    fi
    }
```

# 앤서블 모듈

## 코드 구성

```
source $1

if [ -z "$dest" ]; then
    printf '{"failed": true, "msg": "missing required arguments: dest"}'
    exit 1
fi
if [ -z "$state" ]; then
    printf '{"failed": true, "msg": "missing required arguments: state"}'
    exit 1
fi

changed="false"
msg=""
contents=""
```

# 앤서블 모듈

## 코드 구성

```
case $state in
    present)
        create_file
        ;;
    absent)
        delete_file
        ;;
    upper)
        convert_to_upper
        ;;
    lower)
        convert_to_lower
        ;;
    *)
        printf '{"failed": true, "msg": "invalid state: %s"}' "$state"
        exit 1
        ;;
esac
printf '{"changed": %s, "msg": "%s", "contents": %s}' "$changed" "$msg" "$contents"
exit 0
```

# 앤서블 모듈

**모듈 테스트**

테스트 모듈에서 사용할 데이터를 임으로 작성한다.

```
$ cat <<EOF> test.txt
hello world tang and choi
EOF
```

# 앤서블 모듈

**모듈 테스트**

모듈 테스트 하기 위해서는 다음과 같은 명령어로 확인이 가능하다.

```
$ ansible localhost, -m bashmod -a ‘dest=hello.txt state=upper’
```

# 앤서블 모듈

**모듈 테스트**

추가 기능인 제거를 실행한다.

```
$ ansible localhost, -m bashmod -a ‘dest=hello.txt
state=absent’
```

# 앤서블 모듈

**모듈 테스트**

bash에서 테스트는 다음과 같이 한다.

```
$ echo ‘dest=test.txt state=present’ > args
$ bash bashmod args
```

# 앤서블 모듈

**모듈 테스트**

디버깅 하기 위해서는 bash의 -x옵션을 통해서 디버깅이 가능하다.

```
$ echo ‘dest=test.txt state=present’ > args
```

```
$ bash -x bashmod args
```