10.15 : Introduction to Shell

```
### 이번 DataCamp 강의를 수강하면서 배운점
-
### 강의 내용중 궁금했던점
- 이런 것이 궁금했어요.
### 함께 이야기 나누며 얻은 것
- 음..🤔
```

파일 시스템(Filesystem)

파일과 디렉토리(또는 폴더)를 관리하는 시스템입니다. 각 파일과 디렉토리는 절대 경로(absolute path)로 식별되며, 이는 파일 시스템의 루트 디렉토리(/)에서 해당 파일이나 디렉토리까지 어떻게 도달할 수 있는지를 나타냅니다. 예를 들어, `/home/repl`은 `home` 디렉토리 안의 `repl` 디렉토리를 의미하고, `/home/repl/course.txt`는 그 디렉토리 안에 있는 `course.txt`라는 파일을 가리킵니다. `/`는 루트 디렉토리(root directory)를 의미합니다.

pwd 명령어

현재 파일 시스템에서 내가 어디에 위치해 있는지 확인하는 명령어로, "print working directory"의 약자입니다. 이 명령어를 실행하면 현재 작업 중인 디렉토리의 절대 경로를 출력해줍니다. Shell에서 기본적으로 명령어를 실행하거나 파일을 찾는 위치가 바로 이 작업 디렉토리입니다.

---

ls 명령어

`ls`는 현재 디렉토리의 내용을 나열해주는 명령어입니다. "listing"의 약자이며, 현재 작업 중인 디렉토리(즉, `pwd` 명령어로 표시된 디렉토리)의 파일 및 폴더 목록을 보여줍니다. 파일 이름을 추가로 입력하면 해당 파일을, 디렉토리 이름을 입력하면 그 디렉토리의 내용을 나열할 수 있습니다. 예를 들어, `ls /home/repl`은 `repl` 디렉토리의 내용을 보여줍니다.

응용 예시

디렉토리 `/home/repl/seasonal`에 있는 파일을 나열하려면, `ls /home/repl/seasonal` 명령어를 사용하면 됩니다. 이 디렉토리는 계절별로 날짜에 따른 치과 수술 정보가 들어 있는 파일들을 포함하고 있습니다.

절대 경로(Absolute Path)

절대 경로는 파일 시스템의 루트 디렉토리(/)부터 시작하여 특정 파일이나 디렉토리까지의 경로를 나타냅니다. 이는 마치 경도와 위도처럼, 어디에서 경로를 참조하든 항상 동일한 값을 갖습니다.

상대 경로(Relative Path)

상대 경로는 현재 위치를 기준으로 특정 파일이나 디렉토리까지의 경로를 나타냅니다. 예를 들어, "북쪽으로 20km"라고 하는 것과 같습니다. 현재 위치에 따라 달라지며, 상대적인 경로를 통해 이동할 때마다 그 값이 달라질 수 있습니다.

예시

- `/home/repl` 디렉토리에 있을 때, 상대 경로 `seasonal`은 절대 경로 `/home/repl/seasonal`과 동일한 디렉토리를 가리킵니다.
- `/home/repl/seasonal` 디렉토리에 있을 때, 상대 경로 `winter.csv`는 절대 경로 `/home/repl/seasonal/winter.csv`와 동일한 파일을 가리킵니다.

절대 경로와 상대 경로 구분

경로의 첫 번째 문자를 통해 Shell은 경로가 절대인지 상대인지 구분합니다.

- `/`로 시작하면 절대 경로입니다.
- `/`로 시작하지 않으면 상대 경로입니다.

---

cd 명령어

`cd`는 "change directory"의 약자로, 파일 브라우저에서 폴더를 더블 클릭해 이동하는 것처럼 터미널에서 디렉토리 간 이동을 가능하게 하는 명령어입니다.

예시

- `cd seasonal` 명령어를 입력하면, 현재 위치에서 `seasonal`이라는 디렉토리로 이동합니다. 이후 `pwd` 명령어를 입력하면, `/home/repl/seasonal`에 있다는 것을 확인할 수 있습니다.
- 디렉토리 `/home/repl/seasonal`에 있는 상태에서 `ls` 명령어를 입력하면, 해당 디렉토리의 파일과 폴더 목록을 보여줍니다.

홈 디렉토리로 돌아가기

`cd /home/repl` 명령어를 사용하여 홈 디렉토리로 다시 돌아갈 수 있습니다.

---

cp 명령어

`cp`는 "copy"의 약자로, 파일을 복사하거나 다른 디렉토리로 이동하는 명령어입니다.

파일 복사 예시

- `cp original.txt duplicate.txtoriginal.txt`라는 파일을 복사하여 `duplicate.txt`라는 이름으로 새롭게 만듭니다. 만약 이미 `duplicate.txt`라는 파일이 존재하면, 해당 파일은 덮어쓰기가 됩니다.

디렉토리로 파일 복사 예시

- `cp seasonal/autumn.csv seasonal/winter.csv backup`이 명령어는 `autumn.csv`와 `winter.csv` 파일을 `backup`이라는 디렉토리로 복사합니다.

---

mv 명령어

`mv`는 "move"의 약자로, 파일을 한 디렉토리에서 다른 디렉토리로 이동하는 명령어입니다. 이는 그래픽 파일 탐색기에서 파일을 드래그하여 이동하는 것과 유사합니다. `mv`는 `cp`와 동일한 방식으로 매개변수를 처리합니다.

파일 이동 예시

- `mv autumn.csv winter.csv ..`이 명령어는 현재 작업 디렉토리에 있는 `autumn.csv`와 `winter.csv` 파일을 상위 디렉토리(부모 디렉토리, `..`로 표시됨)로 이동시킵니다. 여기서 `..`은 항상 현재 위치에서 한 단계 위에 있는 디렉토리를 가리킵니다.

---

mv로 파일 이름 변경

`mv` 명령어는 파일을 이동하는 것뿐만 아니라, 파일의 이름을 변경하는 데에도 사용할 수 있습니다.

파일 이름 변경 예시

- `mv course.txt old-course.txt`이 명령어는 현재 작업 디렉토리에 있는 `course.txt` 파일의 이름을 `old-course.txt`로 변경합니다. 파일을 이동하지 않고 이름만 바꾸는 방식입니다. 이는 파일 탐색기와는 다른 방식이지만, 유용하게 사용할 수 있습니다.

주의 사항

`mv` 명령어도 `cp`와 마찬가지로, 기존 파일을 덮어쓸 수 있습니다. 만약 `old-course.txt`라는 파일이 이미 존재한다면, 위 명령어는 기존 파일을 `course.txt`의 내용으로 대체하게 됩니다.

---

rm 명령어

`rm`은 "remove"의 약자로, 파일을 삭제하는 명령어입니다. `cp`와 `mv`처럼 여러 파일을 한 번에 삭제할 수 있습니다.

파일 삭제 예시

- `rm thesis.txt backup/thesis-2017-08.txt`이 명령어는 `thesis.txt` 파일과 `backup` 디렉토리에 있는 `thesis-2017-08.txt` 파일을 모두 삭제합니다.

주의 사항

`rm` 명령어는 즉시 파일을 삭제하며, 그래픽 파일 탐색기와는 달리 휴지통(Trash Can)이 없습니다. 따라서 위 명령어를 실행하면 파일은 완전히 삭제되어 복구할 수 없으니, 사용 시 주의가 필요합니다.

디렉토리 이동 및 이름 변경: mv 명령어

`mv`는 파일뿐만 아니라 디렉토리도 동일하게 처리합니다. 예를 들어, 홈 디렉토리에서 `mv seasonal by-season` 명령어를 실행하면 `seasonal` 디렉토리의 이름을 `by-season`으로 변경합니다.

디렉토리 삭제: rm과 rmdir 명령어

- `rm` 명령어는 파일 삭제에 사용되지만, 디렉토리를 삭제하려고 하면 오류 메시지를 출력합니다. 이는 실수로 중요한 작업이 들어 있는 전체 디렉토리를 삭제하는 것을 방지하기 위한 것입니다.
- 디렉토리를 삭제하려면 `rmdir` 명령어를 사용해야 합니다. 하지만, `rmdir`은 안전하게 동작하기 위해 **빈 디렉토리**만 삭제할 수 있습니다. 따라서 디렉토리 내 파일을 먼저 삭제한 후에 디렉토리를 삭제해야 합니다.

숙련자용 옵션: -r 옵션

숙련된 사용자는 `rm` 명령어에 `-r`(recursive) 옵션을 사용하여 디렉토리와 그 안의 모든 파일을 한 번에 삭제할 수 있습니다. 이에 대한 자세한 내용은 명령어 옵션을 다룰 다음 장에서 다룹니다.

임시 파일 저장: /tmp 디렉토리

데이터 분석 중간에 생성되는 임시 파일들은 홈 디렉토리에 저장하기보다는 `/tmp` 디렉토리에 저장하는 것이 좋습니다. `/tmp` 디렉토리는 사람들이나 프로그램들이 잠깐 동안만 필요한 파일들을 저장하는 용도로 사용되는 공간입니다. 이 디렉토리는 루트 디렉토리 `/` 바로 아래에 위치해 있으며, 홈 디렉토리와는 별개입니다.

임시로 필요한 파일은 `/tmp`에 저장하면 관리하기 쉽고, 시스템이 자동으로 이곳의 파일을 일정 시간 후 삭제하기 때문에 파일 정리에 유용합니다.

---

cat 명령어

`cat`은 "concatenate"의 약자로, 파일의 내용을 화면에 출력하는 가장 간단한 방법 중 하나입니다. 여러 파일을 이어서 출력할 수 있기 때문에 "연결하다"는 의미를 담고 있습니다. 파일의 내용을 빠르게 확인할 때 유용합니다.

예시

- `cat agarwal.txt`이 명령어는 `agarwal.txt` 파일의 내용을 출력합니다. 예를 들어, 아래와 같이 파일에 저장된 직원 정보를 확인할 수 있습니다:

```
name: Agarwal, Jasmine
position: RCT2
```

```
start: 2017-04-01
benefits: full
```

---

less 명령어

`less`는 파일 내용을 한 번에 한 페이지씩 보여주는 명령어입니다. `cat`과 달리, 출력이 많을 경우 한 페이지씩 넘기며 볼 수 있어 대용량 파일을 다룰 때 더 편리합니다. `more`라는 명령어의 기능을 강화한 버전으로, Unix 세계에서는 `more`보다 강력한 기능을 가진다는 점에서 `less`라고 이름을 붙였습니다. 이 방식의 유머는 Unix 커뮤니티에서 종종 볼 수 있습니다.

사용법

- `less <파일명>` 명령어로 파일을 한 페이지씩 볼 수 있습니다.
- **spacebar**를 누르면 페이지를 아래로 넘기고, **q**를 누르면 프로그램을 종료합니다.

여러 파일 보기

- 여러 파일을 `less`로 열 경우, `:n`을 입력하면 다음 파일로 이동하고, `:p`를 입력하면 이전 파일로 돌아갑니다.
- **q**를 눌러 언제든지 종료할 수 있습니다.

추가 참고

연습 문제의 해답을 볼 때는 `less`로 페이지 넘김 기능을 끄는 추가 명령어가 붙을 수 있습니다. 이는 테스트를 더 효율적으로 하기 위한 조치입니다.

---

head 명령어

`head`는 파일의 처음 몇 줄을 출력하는 명령어입니다. 기본적으로 파일의 상위 10줄을 출력하지만, 출력할 줄 수를 지정할 수도 있습니다. CSV 파일과 같은 데이터 파일의 첫 부분을 빠르게 확인할 때 유용합니다.

예시

- `head seasonal/summer.csv`이 명령어는 `seasonal/summer.csv` 파일의 첫 10줄을 출력합니다. 출력된 데이터는 아래와 같습니다:

```
Date,Tooth
2017-01-11,canine
2017-01-18,wisdom
2017-01-21,bicuspid
2017-02-02,molar
2017-02-27,wisdom
2017-02-27,wisdom
```

```
2017-03-07,bicuspid
2017-03-15,wisdom
2017-03-20,canine
```

이처럼 `head` 명령어를 사용하면 파일의 상단 데이터를 빠르게 확인하여, 어떤 필드가 포함되어 있는지와 그 필드들의 값을 파악할 수 있습니다.

탭 자동 완성(Tab Completion)

쉘에서 제공하는 강력한 도구 중 하나는 **탭 자동 완성** 기능입니다. 파일 이름이나 경로의 일부를 입력한 후 탭 키를 누르면, 쉘이 가능한 한 경로를 자동으로 완성해 줍니다.

예시

- `sea`를 입력하고 탭 키를 누르면, 쉘은 디렉토리 이름을 `seasonal/`로 자동 완성합니다.
- 그 후 `a`를 입력하고 다시 탭을 누르면, 경로가 `seasonal/autumn.csv`로 자동 완성됩니다.

모호한 경로의 경우

경로가 모호한 경우, 예를 들어 `seasonal/s`를 입력하고 탭을 누르면 가능한 경로 목록이 나타납니다. 그 후 몇 글자를 더 입력해 경로를 구체화한 뒤 다시 탭을 누르면 나머지 경로가 자동 완성됩니다.

이 기능을 활용하면 파일 경로 입력이 더 빠르고 정확해집니다.

---

head 명령어의 플래그 사용

`head` 명령어는 기본적으로 파일의 첫 10줄을 출력하지만, 플래그(flag)를 사용하면 출력할 줄 수를 변경할 수 있습니다. 플래그는 명령어의 동작을 수정하는 추가 옵션을 제공하는데, `-n` 플래그는 출력할 줄 수를 지정하는 데 사용됩니다.

예시

- `head -n 3 seasonal/summer.csv`이 명령어는 `seasonal/summer.csv` 파일의 첫 3줄만 출력합니다.
- `head -n 100`이 명령어는 파일의 첫 100줄을 출력합니다. 파일에 100줄이 없으면, 가능한 만큼만 출력합니다.

플래그 명명 규칙

일반적으로 플래그 이름은 그 목적을 나타냅니다. 예를 들어, `-n`은 "number of lines(줄의 수)"를 의미합니다. 플래그는 반드시 한 글자일 필요는 없지만, `-`로 시작하는 한 글자 플래그가 많이 사용되는 관행입니다.

좋은 스타일

좋은 코딩 스타일로 간주되는 방법은 **모든 플래그를 파일 이름 앞에 배치하는 것**입니다. 이 코스에서도 이 규칙을 따르므로, 플래그는 파일 이름 앞에 놓는 것이 중요합니다.

---

ls -R 명령어: 디렉토리 내용의 재귀적 표시

`ls` 명령어에 `-R`(recursive) 플래그를 사용하면, 해당 디렉토리 아래에 있는 모든 파일과 디렉토리를 **재귀적으로** 표시합니다. 즉, 디렉토리 내의 서브디렉토리와 그 안의 파일들까지 모두 확인할 수 있습니다.

예시

- `ls -R` 명령어를 홈 디렉토리에서 실행하면, 아래와 같은 출력이 나타납니다:

```
backup          course.txt      people          seasonal

./backup:

./people:
agarwal.txt

./seasonal:
autumn.csv      spring.csv      summer.csv      winter.csv
```

이 출력은 다음과 같은 구조를 보여줍니다:

- 현재 디렉토리(`./`)의 파일과 디렉토리 목록을 먼저 표시합니다.
- 그 다음으로 각 서브디렉토리(`backup`, `people`, `seasonal`) 안에 있는 파일 및 디렉토리를 순차적으로 나열합니다.

이 방식으로, 디렉토리 구조가 깊이 중첩되어 있더라도 모든 파일과 폴더를 한 번에 볼 수 있습니다.

---

man 명령어: 매뉴얼 보기

`man`은 "manual"의 약자로, 명령어의 매뉴얼 페이지를 열어 명령어의 사용 방법과 옵션을 설명해줍니다. 예를 들어, `man head` 명령어는 `head` 명령어에 대한 상세한 설명을 제공합니다.

예시: man head

```
NAME
     head -- display first lines of a file
```

```
SYNOPSIS
     head [-n count | -c bytes] [file ...]

DESCRIPTION
     This filter displays the first count lines or bytes of
each of the specified files, or of the standard input if no
files are specified.  If count is omitted it defaults to 1
0.
```

- **NAME**: 명령어가 무엇을 하는지 간단히 설명합니다. 예를 들어, `head`는 파일의 첫 번째 줄을 출력하는 명령어입니다.
- **SYNOPSIS**: 명령어의 사용법과 옵션들을 나열합니다.
  - `[-n count | -c bytes]`: `n` 플래그로 출력할 줄 수를, `c` 플래그로 출력할 바이트 수를 지정할 수 있습니다.
  - `[file ...]`: 파일 이름을 하나 또는 여러 개 입력할 수 있음을 의미합니다.
- **DESCRIPTION**: 좀 더 상세한 설명을 제공합니다.

SEE ALSO 섹션과 탐색 방법

`man` 명령어는 `less`를 자동으로 호출하므로, 페이지를 넘기려면 **spacebar**를 누르고, 종료하려면

를 누릅니다.

문제점 및 대안

Unix 매뉴얼은 특정 명령어를 알고 있어야만 검색할 수 있습니다. 명령어에 대해 모를 경우:

- Stack Overflow에서 검색하거나
- DataCamp와 같은 학습 커뮤니티의 채널에서 질문하거나
- 이미 알고 있는 명령어의 매뉴얼 페이지의 **SEE ALSO** 섹션을 확인해 관련 명령어를 찾는 것이 유용합니다.

---

cut 명령어: 텍스트 파일에서 열 선택

`cut` 명령어는 텍스트 파일에서 특정 열을 선택할 수 있는 명령어입니다. 주로 `-f`(fields) 옵션과 `-d`(delimiter) 옵션을 사용하여 열을 선택하고, 열을 구분하는 구분자(쉼표, 탭 등)를 지정합니다.

예시

- `cut -f 2-5,8 -d , values.csv`이 명령어는 `values.csv` 파일에서 **2~5번 열과 8번 열**을 선택합니다. `d ,`는 쉼표(,)를 구분자로 사용하겠다는 의미입니다.

옵션 설명

- `f`: 선택할 열을 지정합니다. 예를 들어, `2-5`는 2번부터 5번까지의 열을, `8`은 8번 열을 의미합니다.
- `d`: 열을 구분하는 **구분자**를 지정합니다. CSV 파일에서는 주로 쉼표(,)를 사용하지만, 탭, 공백, 콜론 등도 구분자로 사용할 수 있습니다.

참고

파일에 사용된 구분자는 파일 형식에 따라 다를 수 있으므로, 구분자를 정확히 지정해야 원하는 열을 제대로 선택할 수 있습니다. 추가 옵션에 대한 내용은 `man cut` 명령어를 사용하여 탐색할 수 있습니다.

---

cut 명령어의 한계: 인용된 문자열 처리 문제

`cut` 명령어는 단순하게 열을 구분하기 때문에 **인용된 문자열(quoted strings)**을 제대로 처리하지 못하는 한계가 있습니다. 예를 들어, CSV 파일에 이름과 성이 쉼표로 구분된 경우, `cut` 명령어는 이 쉼표를 구분자로 인식해 잘못된 결과를 출력할 수 있습니다.

문제 상황

파일 내용이 아래와 같을 때:

```
Name,Age
"Johel,Ranjit",28
"Sharma,Rupinder",26
```

`cut -f 2 -d , everyone.csv` 명령어를 실행하면:

```
Age
Ranjit"
Rupinder"
```

이처럼 나이 열을 선택하려 했으나, 쉼표를 구분자로 오인하여 이름과 성 사이의 쉼표를 잘못 처리하고, 나이 대신 "Ranjit"과 "Rupinder"를 출력하게 됩니다.

대안

`cut` 명령어는 인용된 문자열을 처리하지 못하므로, CSV 파일에서 인용된 문자열을 제대로 다루고 싶다면 `awk` 또는 `csvkit` 같은 더 강력한 도구를 사용하는 것이 좋습니다. 이러한 도

구들은 구분자뿐만 아니라 인용된 문자열까지도 인식하여 정확하게 열을 처리할 수 있습니다.

명령어 재실행: 히스토리 및 편집 기능

쉘의 가장 큰 장점 중 하나는 **명령어를 쉽게 재실행**할 수 있다는 점입니다. 한 번 실행한 명령어를 다시 실행하거나 편집하여 반복적인 작업을 효율적으로 처리할 수 있습니다.

명령어 재실행 방법

- **위쪽 화살표**를 누르면 이전에 실행한 명령어들을 차례로 볼 수 있습니다. 왼쪽, 오른쪽 화살표 키와 **삭제 키**로 명령어를 수정한 후 **엔터 키**를 누르면 수정된 명령어가 실행됩니다.

history 명령어

- `history` 명령어는 최근에 실행한 명령어 목록을 출력합니다. 각 명령어 앞에 일련번호가 표시되므로, 특정 명령어를 다시 실행하고 싶을 때 유용합니다.
- 예를 들어, `!55`를 입력하면 **히스토리의 55번째 명령어**가 다시 실행됩니다.

명령어 이름으로 재실행

- 명령어 이름으로도 재실행이 가능합니다. `!head`를 입력하면 **가장 최근에 사용한** `head` **명령어**가 다시 실행되고, `!cut`은 **가장 최근에 사용한** `cut` **명령어**를 재실행합니다.

이러한 기능을 활용하면 반복적인 명령어 실행이 훨씬 간편해집니다.

---

grep 명령어: 특정 텍스트로 줄 선택

`grep`은 파일에서 특정 텍스트를 포함한 **줄을 선택**하는 명령어입니다. 간단한 형태로, `grep`은 텍스트와 하나 이상의 파일 이름을 인자로 받아, 해당 텍스트를 포함한 파일의 줄을 출력합니다.

예시

- `grep bicuspid seasonal/winter.csv`: `winter.csv` 파일에서 "bicuspid"를 포함한 줄을 출력합니다.

grep의 주요 플래그

- `c`: 일치하는 줄의 **개수**만 출력하고, 해당 줄 자체는 출력하지 않습니다.
- `h`: 여러 파일을 검색할 때, 파일 이름을 출력하지 않습니다.
- `i`: 대소문자를 **무시**하고 검색합니다. 예를 들어, "Regression"과 "regression"을 동일하게 처리합니다.
- `l`: 일치하는 줄이 있는 **파일 이름만** 출력하고, 해당 줄은 출력하지 않습니다.

- `n`: 일치하는 줄의 **줄 번호**를 함께 출력합니다.
- `v`: 일치하지 않는 줄만 출력합니다. 즉, 주어진 텍스트가 포함되지 않은 줄을 보여줍니다.

이러한 플래그를 사용하면 `grep`을 통해 더 정교한 검색과 필터링이 가능합니다.

---

paste 명령어: 데이터 파일 결합

`paste`는 여러 파일의 내용을 **결합**하는 데 사용되는 명령어입니다. `cut` 명령어가 파일에서 특정 열을 추출하는 데 사용되는 반면, `paste`는 여러 파일을 가로 방향으로 합쳐서 새로운 데이터를 만들 수 있습니다.

예시

- `paste file1.txt file2.txt`이 명령어는 `file1.txt`와 `file2.txt`의 내용을 **옆으로 나란히 결합**하여 한 줄씩 병합된 결과를 출력합니다. 즉, 각 파일의 첫 번째 줄이 합쳐지고, 두 번째 줄이 합쳐지는 방식입니다.

주요 사용 방식

- 기본적으로 `paste`는 탭(tab)으로 데이터를 구분하지만, `d` 옵션을 사용하여 구분자를 변경할 수 있습니다.
  - 예: `paste -d "," file1.txt file2.txt`쉼표(,)를 구분자로 사용하여 파일을 결합합니다.

`paste` 명령어는 데이터를 분리하는 `cut`과 상반된 기능을 수행하며, 데이터 파일을 손쉽게 결합하는 데 유용합니다.

출력 리디렉션: '>' 기호 사용

출력 리디렉션을 사용하면 명령어의 출력을 화면이 아닌 파일로 저장할 수 있습니다. `>` 기호는 명령어의 출력을 파일로 **리디렉션**하는 역할을 하며, 이는 모든 쉘 명령어에서 사용할 수 있습니다.

예시 1: 화면에 출력

- `head -n 5 seasonal/summer.csv`이 명령어는 `summer.csv` 파일의 첫 5줄을 화면에 출력합니다.

예시 2: 파일에 저장 (리디렉션)

- `head -n 5 seasonal/summer.csv > top.csv`이 명령어는 `summer.csv` 파일의 첫 5줄을 화면에 출력하지 않고, 대신 **새로운 파일**인 `top.csv`에 저장합니다. 따라서 화면에는 아무것도 나타나지 않습니다.

파일 내용 확인

- `cat top.csvtop.csv` 파일의 내용을 출력하여 저장된 데이터가 정확한지 확인할 수 있습니다.

리디렉션 사용 요령

- `>` 기호는 명령어 출력물을 **새로운 파일**로 저장하며, 기존 파일이 있으면 덮어씁니다.
- 리디렉션은 `head`, `grep`, `cut` 등 **출력을 생성하는 모든 쉘 명령어**에서 사용할 수 있습니다.

파일 중간 부분의 줄 선택: head와 tail의 조합

파일에서 특정 줄(예: 3~5줄)을 선택하려면, `head`와 `tail` 명령어를 조합하여 사용할 수 있습니다.

과정 설명

1. 먼저 `head` 명령어를 사용하여 파일의 **첫 5줄**을 추출합니다.
2. 그런 다음, `tail` 명령어를 사용하여 그 결과에서 **마지막 3줄**만 선택합니다.

예시

1. **첫 5줄을 선택**하여 파일에 저장:

   ```
   head -n 5 seasonal/winter.csv > top.csv
   ```

   이 명령어는 `winter.csv` 파일의 첫 5줄을 `top.csv` 파일에 저장합니다.

2. **마지막 3줄을 선택**하여 화면에 출력:

   ```
   tail -n 3 top.csv
   ```

   이 명령어는 `top.csv` 파일의 마지막 3줄을 출력합니다. 이는 원래 파일의 3~5번째 줄에 해당합니다.

이 방식은 파일의 중간 부분에서 원하는 줄을 추출하는 효과적인 방법이며, head와 tail 명령어의 조합을 통해 쉽게 구현할 수 있습니다.

파이프(pipe) 사용: 명령어 결합

쉘에서 **파이프(`|`)**는 여러 명령어를 결합하여 중간 파일을 생성하지 않고, **한 명령어의 출력을 다른 명령어의 입력으로** 사용할 수 있게 해줍니다. 이는 리디렉션보다 효율적이며, 여러 줄의 명령어를 한 줄로 축약할 수 있습니다.

문제점 해결

- **중간 파일 문제**: `top.csv`와 같은 임시 파일을 만들지 않음.
- **명령어 분산 문제**: 여러 명령어를 한 줄로 결합하여 사용.

예시: 파이프 사용

- 먼저, `head` 명령어로 첫 5줄을 선택:

```
head -n 5 seasonal/summer.csv
```

- 파이프(`|`)를 사용해 `head` 명령어의 출력을 **바로** `tail` 명령어로 전달:

```
head -n 5 seasonal/summer.csv | tail -n 3
```

작동 원리

- **파이프(`|`)**: `head -n 5` 명령어의 출력을 `tail -n 3`의 입력으로 사용합니다. 이때, 중간 파일 없이 두 명령어가 한 번에 실행됩니다.
- **결과**: `summer.csv` 파일에서 **3~5번째 줄**이 출력됩니다.

이 방식은 중간 파일을 생성하지 않으므로 작업이 깔끔하고, 명령어 히스토리도 간결하게 유지됩니다.

명령어 체이닝: 여러 명령어를 파이프로 연결

쉘에서는 여러 명령어를 **파이프(`|`)**를 사용해 연결하여, 한 번에 복잡한 작업을 수행할 수 있습니다. 각 명령어의 출력이 다음 명령어의 입력으로 전달되며, 이를 통해 데이터를 단계적으로 처리할 수 있습니다.

예시: 여러 명령어를 체이닝하여 실행

```
cut -d , -f 1 seasonal/spring.csv | grep -v Date | head -n 10
```

작업 순서

1. **첫 번째 열 선택 (`cut`)**

   `cut -d , -f 1 seasonal/spring.csv`: 쉼표(,)로 구분된 `spring.csv` 파일의 첫 번째 열을 선택합니다.

2. **헤더 줄 제거 (`grep -v`)**

`grep -v Date`: `Date`라는 단어가 포함된 줄(헤더 줄)을 제외하고 나머지 데이터를 출력합니다. `-v` 옵션은 일치하지 않는 줄을 출력하는 의미입니다.

3. **상위 10줄 출력 (`head`)**

   `head -n 10`: 위에서 필터링된 데이터 중 상위 10줄을 출력합니다.

작동 방식

- `cut` 명령어로 데이터를 처리하고, 그 결과를 **파이프**를 통해 `grep`으로 전달하여 헤더를 제거합니다.
- 마지막으로 `head` 명령어로 상위 10줄을 출력하여, 전체 데이터 중 필요한 부분만 확인할 수 있습니다.

이 방식은 데이터를 여러 단계로 필터링하거나 가공할 때 유용하며, 명령어들을 한 줄로 연결할 수 있어 효율적입니다.

---

wc 명령어: 파일 내 문자, 단어, 줄 수 세기

`wc`는 "word count"의 약자로, 파일의 **문자 수, 단어 수, 줄 수**를 계산하여 출력하는 명령어입니다. 기본적으로 이 세 가지 정보를 모두 출력하지만, 특정 정보만 보고 싶다면 플래그를 사용하여 제어할 수 있습니다.

주요 옵션

- `c`: **문자 수**만 출력합니다.
- `w`: **단어 수**만 출력합니다.
- `l`: **줄 수**만 출력합니다.

와일드카드 사용: 여러 파일을 효율적으로 처리

쉘에서 **와일드카드**는 파일이나 디렉토리 이름을 간단하게 지정하는 데 유용합니다. 가장 일반적으로 사용하는 와일드카드인 `*`는 **0개 이상의 문자**와 일치하는 모든 파일을 나타냅니다. 이를 사용하면 여러 파일을 처리할 때 파일 이름을 하나씩 나열하는 번거로움을 줄일 수 있습니다.

예시

- 특정 디렉토리의 **모든 파일**에 대해 첫 번째 열을 추출:

  ```
  cut -d , -f 1 seasonal/*
  ```

  `seasonal/` 디렉토리의 **모든 파일**에서 첫 번째 열을 선택합니다.

- 특정 확장자를 가진 **CSV 파일만** 대상으로 작업:

```
cut -d , -f 1 seasonal/*.csv
```

`seasonal/` 디렉토리 내의 **CSV 파일**만 선택하여 첫 번째 열을 출력합니다.

와일드카드의 장점

- **효율성**: 파일 이름을 일일이 입력할 필요 없이 `*`를 사용하여 여러 파일을 한 번에 처리할 수 있습니다.
- **오류 감소**: 파일을 누락하거나 중복하는 실수를 줄일 수 있습니다.

와일드카드는 파일을 다루는 작업에서 매우 강력한 도구로, 특히 많은 파일을 처리할 때 시간을 절약하고 오류를 방지할 수 있습니다.

추가적인 와일드카드: 보다 세밀한 파일 매칭

쉘에서는 `*` 외에도 **다양한 와일드카드**를 사용하여 파일을 더욱 세밀하게 선택할 수 있습니다. 이러한 와일드카드는 특정 패턴을 가진 파일을 찾을 때 유용합니다.

1. ? (하나의 문자와 일치)

`?`는 **하나의 문자**와 일치합니다.

- 예: `201?.txt`는 `2017.txt` 또는 `2018.txt`와 일치하지만, `2017-01.txt`와는 일치하지 않습니다.

2. [...] (대괄호 안의 문자와 일치)

대괄호 안에 있는 **하나의 문자**와 일치합니다.

- 예: `201[78].txt`는 `2017.txt`와 `2018.txt`와 일치하지만, `2016.txt`와는 일치하지 않습니다.

3. {...} (쉼표로 구분된 여러 패턴과 일치)

중괄호 안에 있는 **쉼표로 구분된 패턴들** 중 하나와 일치합니다.

- 예: `{*.txt,*.csv}`는 **.txt 또는 .csv**로 끝나는 모든 파일과 일치하며, `.pdf`로 끝나는 파일은 제외됩니다.

예시 사용

- **단일 문자 매칭**: `file?.txt`는 `file1.txt`, `file2.txt`와 일치하지만, `file10.txt`와는 일치하지 않습니다.
- **지정된 문자 매칭**: `201[78].txt`는 `2017.txt`와 `2018.txt`와 일치합니다.
- **여러 확장자 매칭**: `{*.txt,*.csv}`는 `.txt`나 `.csv`로 끝나는 파일만 선택합니다.

sort 명령어: 데이터 정렬

`sort`는 데이터를 **정렬**하는 명령어로, 기본적으로 **오름차순 알파벳 순서**로 데이터를 정렬합니다. 여러 가지 플래그를 사용하여 정렬 방식을 변경할 수 있으며, 파이프와 함께 사용하여 필터링 후 정렬하는 데 유용합니다.

주요 플래그

- `n`: **숫자** 기준으로 정렬합니다.
- `r`: **역순**으로 정렬합니다. (내림차순)
- `b`: **앞의 공백을 무시**하고 정렬합니다.
- `f`: **대소문자를 구분하지 않고** 정렬합니다. (case-insensitive)

예시

1. **알파벳 오름차순**으로 정렬 (기본값):

```
file.txt
```

2. **숫자 기준으로 정렬**:

```
sort -n numbers.tx
```

3. **숫자 역순으로 정렬**:

```
sort -nr numbers.txt
```

4. **공백 무시하고 정렬**:

```
sort -b names.txt
```

5. **대소문자 구분 없이 정렬**:

```
sort -f names.txt
```

파이프라인과 함께 사용

`grep`과 함께 사용하여 원하는 레코드를 먼저 필터링한 후 정렬할 수 있습니다.

- 예: 특정 패턴을 검색한 후, 그 결과를 숫자 기준으로 정렬:

```
grep "pattern" file.txt | sort -n
```

`sort` 명령어는 데이터를 원하는 방식으로 정렬할 수 있는 강력한 도구이며, 파이프라인을 통해 더욱 효율적으로 데이터를 처리할 수 있습니다.

---

uniq 명령어: 중복된 줄 제거

`uniq` 명령어는 **연속된 중복 줄**을 제거하는 데 사용됩니다. 하지만, **인접한 줄만** 중복 제거가 가능하며, 비연속적인 중복 줄은 제거되지 않습니다. 이는 메모리 효율성을 높이기 위해 설계된 방식입니다.

예시 1: 연속된 중복 줄 제거

파일이 다음과 같을 때:

```
017-07-03
2017-07-03
2017-08-03
2017-08-03
```

`uniq` 명령어를 사용하면:

```
2017-07-03
2017-08-03
```

중복된 연속된 줄이 제거됩니다.

예시 2: 비연속적인 중복 줄 제거 불가

파일이 다음과 같을 때:

```
2017-07-03
2017-08-03
2017-07-03
2017-08-03
```

`uniq` 명령어를 사용해도:

```
2017-07-03
2017-08-03
```

```
2017-07-03
2017-08-03
```

모든 줄이 출력됩니다. 왜냐하면 중복된 줄이 연속적이지 않기 때문입니다.

비연속 중복 제거 방법

중복된 비연속적인 줄을 제거하려면 먼저 파일을 정렬한 다음 `uniq`를 적용해야 합니다.

- 예: 파일을 정렬한 후 중복 제거:이렇게 하면 비연속적인 중복 줄도 제거됩니다.

  ```
  ort file.txt | uniq
  ```

`uniq` 명령어는 기본적으로 메모리 사용을 최소화하기 위해 **인접한 중복 줄**만 제거하도록 설계되었으며, 비연속 중복 줄을 제거할 경우에는 `sort`와 함께 사용해야 합니다.

출력 리디렉션의 위치: 파이프라인에서의 규칙

쉘에서 **출력 리디렉션(`>`)**은 반드시 파이프라인의 **끝에** 와야 합니다. 그렇지 않으면, 예상과 다르게 명령어의 흐름이 중단되거나 결과가 출력되지 않을 수 있습니다.

올바른 파이프라인 사용 예시

```
-d , -f 2 seasonal/*.csv | grep -v Tooth > teeth-only.txt
```

이 명령어는 다음을 수행합니다:

1. `cut` 명령어로 `seasonal/*.csv` 파일들의 두 번째 열을 선택.
2. 선택된 열에서 `grep -v Tooth`로 **"Tooth"**를 포함하지 않은 줄만 필터링.
3. 그 결과를 `teeth-only.txt` 파일에 저장.

잘못된 사용 예시: `>`를 파이프 중간에 사용

```
cut -d , -f 2 seasonal/*.csv > teeth-only.txt | grep -v Too
th
```

이 경우:

1. `cut`의 모든 출력이 `teeth-only.txt` 파일로 저장됩니다.
2. `grep`은 더 이상 입력이 없으므로 **기다리기만 하며** 실행되지 않습니다.

이유: `>` 기호는 **출력을 파일로 바로 리디렉션**하기 때문에, 파이프에 전달할 내용이 남지 않게 됩니다. 따라서 파이프라인에서 리디렉션은 **최종 출력에서만** 사용해야 합니다.

요약

- *리디렉션(`>`)**은 파이프라인에서 마지막에 위치해야 하며, 그 전에 명령어는 파이프로 연결해 데이터를 처리해야 합니다.
- 중간에 리디렉션을 사용하면, 후속 명령어가 처리할 데이터가 없어 문제가 발생합니다.

Ctrl + C: 실행 중인 프로그램 강제 종료

쉘에서 **Ctrl + C**(또는 **^C**)는 실행 중인 프로그램을 **강제 종료**하는 단축키입니다. 시간이 오래 걸리는 작업이나 잘못된 명령어로 인해 프로그램이 멈추거나 **작업을 중단**하고 싶을 때 사용할 수 있습니다.

사용 시나리오

1. **장시간 실행되는 작업**: 일부 명령어는 여러 분량의 데이터를 처리하거나 복잡한 계산을 포함해 시간이 오래 걸릴 수 있습니다. 만약 작업을 중단하고 싶다면 **Ctrl + C**를 눌러 프로그램을 종료할 수 있습니다.
2. **잘못된 파이프라인 및 리디렉션**: 파이프라인 중간에 `>`와 같은 잘못된 리디렉션을 넣어 프로그램이 **응답하지 않거나 대기 상태**가 될 수 있습니다. 이때도 **Ctrl + C**를 사용해 프로그램을 즉시 중단할 수 있습니다.

주의 사항

- **^C**는 대소문자를 구분하지 않고 `c`는 소문자로 사용됩니다.
- 강제 종료된 프로그램은 작업을 완전히 멈추고 더 이상 처리되지 않으며, 이 상태에서 다시 시작해야 합니다.

결론: `Ctrl + C`는 쉘에서 중단하고 싶은 작업을 빠르게 종료하는 방법으로, 긴 실행 시간을 가지는 작업이나 실수로 실행한 명령어로 인해 발생하는 문제를 해결하는 데 유용합니다.

환경 변수(Environment Variables)

쉘에서 **환경 변수**는 시스템 및 사용자 설정을 저장하는 변수들로, 프로그램이 실행되는 동안 사용할 수 있습니다. 환경 변수는 주로 **대문자**로 작성되며, 시스템 전체에서 항상 사용할 수 있는 값들을 제공합니다.

주요 환경 변수 예시

- **HOME**: 사용자의 홈 디렉토리 경로.
    - 예: `/home/repl`

- **PWD**: 현재 작업 디렉토리.
  - `pwd` 명령어의 출력과 동일한 값.
- **SHELL**: 현재 사용 중인 쉘 프로그램의 경로.
  - 예: `/bin/bash`
- **USER**: 현재 로그인된 사용자의 ID.
  - 예: `repl`

전체 환경 변수 목록 보기

- `set` 명령어를 사용하면 현재 설정된 **모든 환경 변수 목록**을 볼 수 있습니다. 이 목록은 시스템에 따라 상당히 길어질 수 있습니다.

환경 변수 사용

쉘에서 변수의 값을 참조하려면 `$`를 사용합니다. 예를 들어, `$HOME`은 사용자의 홈 디렉토리 경로를 참조합니다.

---

echo 명령어와 변수 출력

`echo` 명령어는 주어진 인자를 화면에 **출력**하는 간단한 명령어입니다. 텍스트나 변수의 값을 출력할 때 유용합니다.

예시: 텍스트 출력

```
echo hello DataCamp!
```

결과:

```
hello DataCamp!
```

변수 출력

변수의 값을 출력하려면 변수 이름 앞에 `$` **기호**를 붙여야 합니다. 그렇지 않으면 변수 이름 자체가 출력됩니다.

- **잘못된 방식**:

  ```
  echo USER
  ```

  결과:

```
USER
```

변수 이름이 출력됩니다.

- **올바른 방식**:

```
echo $USER
```

결과:

```
repl
```

변수 값인 `repl`이 출력됩니다.

변수 참조 규칙

쉘에서 변수를 참조하려면 **반드시** `$` **기호**를 사용해야 합니다. 이를 통해 쉘은 "파일 이름"과 "변수 값"을 구분할 수 있습니다.

예를 들어, 변수가 `X`라고 할 때, 변수의 값에 접근하려면 `$X`를 사용해야 합니다.

정리

- `echo`는 값을 출력하는 명령어.
- 변수를 출력할 때는 `$`**를 붙여** 변수의 값을 참조해야 함.

쉘 변수(Shell Variables)

쉘 변수는 프로그래밍 언어에서 사용하는 **로컬 변수**와 유사합니다. 쉘 변수를 만들기 위해서는 단순히 **변수 이름에 값을 할당**하면 됩니다. 쉘 변수는 현재 세션 내에서만 사용할 수 있습니다.

쉘 변수 생성

쉘 변수를 생성할 때는 `=` **기호** 앞뒤에 **공백을 넣지 않고** 변수를 할당합니다.

- 예:

```
training=seasonal/summer.csv
```

쉘 변수 값 확인

변수의 값을 출력하려면 `$` **기호**를 붙여서 변수 값을 참조할 수 있습니다.

- 예:결과:

```
echo $training
```

```
seasonal/summer.csv
```

주요 특징

- **쉘 변수는 현재 세션** 동안에만 유효하며, 쉘을 종료하면 변수가 사라집니다.
- 쉘 변수는 프로그래밍 언어의 **로컬 변수**처럼 특정 범위 내에서 사용됩니다.

정리

- 쉘 변수를 생성할 때는 **공백 없이** 값을 할당.
- 변수 값을 참조할 때는 `$` **기호**를 사용.

쉘 루프: for 루프 구조

쉘에서 `for` 루프는 **리스트의 항목들을 순차적으로 처리**하는 반복문입니다. 각 항목을 처리할 때마다 **변수**를 사용하여 현재 처리 중인 항목을 참조할 수 있습니다. 쉘에서 `for` 루프의 구조는 아래와 같습니다.

루프 구조

```
for 변수 in 리스트; do
    명령어
done
```

예시

```
for filetype in gif jpg png; do echo $filetype; done
```

이 명령어는 **gif**, **jpg**, **png**를 차례로 출력합니다.

결과:

```
gif
jpg
png
```

구조 설명

1. **for ... in ... ; do ... ; done**: 쉘에서 루프의 기본 구조입니다.
2. **리스트**: 루프에서 처리할 항목들의 리스트입니다. 여기서는 `gif`, `jpg`, `png`가 리스트에 해당합니다.
3. **변수**: `for` 루프에서 현재 처리 중인 항목을 **저장하는 변수**입니다. 예시에서는 `filetype`이라는 변수를 사용했습니다.
4. **본문 (body)**: 루프의 본문은 처리할 명령어들입니다. 본문 내에서 **`$filetype`*을 사용하여 현재 항목의 값을 참조합니다. 여기서는 `echo $filetype`이 본문입니다.
5. **세미콜론**: `;`는 `do` **이전**과 `done` **이전**에 위치합니다. 이는 쉘에서 한 줄에 명령어를 작성할 때 사용됩니다.

중요한 포인트

- 루프에서 변수의 값은 `$` **기호**를 사용해 참조해야 합니다.
- *세미콜론(`;`)**은 리스트와 `do` 사이, 본문과 `done` 사이에 위치합니다.

이 구조는 반복 작업을 자동화할 때 매우 유용하며, 리스트에 있는 항목들을 하나씩 처리할 수 있습니다.

와일드카드와 for 루프

쉘에서 **와일드카드(`*`)**를 사용하면 특정 패턴과 일치하는 파일들을 리스트로 처리할 수 있습니다. 이는 수동으로 파일 이름을 하나씩 입력하는 대신, **파일 패턴을 사용해 자동으로 파일을 선택**하는 방법입니다.

예시: 와일드카드 사용한 for 루프

```
for filename in seasonal/*.csv; do echo $filename; done
```

이 명령어는 `seasonal` 디렉토리 내의 **모든** `.csv` **파일**을 리스트로 간주하고, 그 파일 이름을 출력합니다.

결과:

```
seasonal/autumn.csv
seasonal/spring.csv
seasonal/summer.csv
seasonal/winter.csv
```

작동 원리

1. **와일드카드( )**: `seasonal/*.csv`는 `seasonal` 디렉토리 내에서 **모든** `.csv` **파일**을 의미합니다.
2. **쉘 확장**: 쉘은 루프를 실행하기 전에 `seasonal/*.csv`**를 네 개의 파일 이름**으로 확장합니다.
3. **루프 반복**: 각 파일 이름을 순서대로 `filename` **변수**에 할당하고, `echo $filename`을 통해 출력합니다.

와일드카드 사용의 장점

- **자동화**: 파일 이름을 일일이 입력할 필요 없이, 패턴에 맞는 파일을 모두 선택해 자동으로 처리할 수 있습니다.
- **유연성**: 파일 추가나 변경이 생기더라도 와일드카드를 사용한 루프는 자동으로 업데이트된 파일 목록을 처리합니다.

이 방식은 대량의 파일을 처리하거나 특정 형식을 가진 파일들만 선택해야 할 때 매우 유용합니다.

명령어는 다음과 같습니다:

1. `files=seasonal/*.csv`:

   이 명령어는 `files` 변수에 `seasonal` **디렉토리 내의 모든** `.csv` **파일들**을 저장합니다. 쉘은 와일드카드(`*`)를 사용하여 파일 리스트를 확장합니다.

2. `for f in $files; do echo $f; done`:

   이 루프는 `files` 변수에 저장된 **각 파일 이름**을 `f` 변수에 할당하고, 해당 파일 이름을 출력합니다.

출력될 라인의 수

`seasonal` 디렉토리 내에 있는 `.csv` 파일이 다음과 같다고 가정할 때:

- `seasonal/autumn.csv`
- `seasonal/spring.csv`
- `seasonal/summer.csv`
- `seasonal/winter.csv`

총 **4개의 파일**이 있으므로, 루프는 **4줄**의 출력을 생성하게 됩니다. 각 줄에는 파일 이름이 하나씩 출력됩니다.

쉘 변수 사용 시 흔한 실수: `$` 및 오타

1. $ 기호를 잊는 실수

쉘에서 변수를 참조할 때는 반드시 변수 이름 앞에 `$` **기호**를 붙여야 합니다. 그렇지 않으면 쉘은 **변수의 값**이 아니라 **변수 이름 자체**를 사용하게 됩니다.

예시:

```
datasets=seasonal/*.csv
echo datasets   # 잘못된 사용, "datasets"라는 단어가 출력됨
echo $datasets  # 올바른 사용, 변수의 값인 파일 목록이 출력됨
```

- `echo datasets`: **변수 이름**인 `datasets`가 그대로 출력됩니다.
- `echo $datasets`: **변수의 값**이 출력됩니다.

2. 변수 이름 오타

경험이 많은 사용자들이 자주 저지르는 실수는 **변수 이름을 오타**로 입력하는 경우입니다. 쉘에서 잘못된 변수 이름을 사용하면, 해당 변수가 정의되지 않았기 때문에 **아무것도 출력되지 않습니다**.

예시:

```
datasets=seasonal/*.csv
echo $datsets  # 오타로 인해 아무것도 출력되지 않음
```

여기서 `datasets`를 정의했지만, `echo $datsets`라고 오타가 발생하면 쉘은 변수를 찾지 못하고 **빈 문자열**을 출력합니다.

요약

- **$ 기호 사용**: 변수의 값을 참조하려면 반드시 `$`를 사용해야 합니다.
- **변수 이름 정확성**: 변수 이름에 오타가 있으면 쉘은 변수를 찾지 못하고 아무것도 출력하지 않으므로, 변수 이름을 정확히 입력해야 합니다.

파일 처리용 루프: 여러 파일에 대한 명령어 실행

루프의 실제 목적은 **여러 파일에 대해 작업을 수행**하는 것입니다. 아래 예시에서, 각 파일의 두 번째 줄을 출력하는 루프를 사용하고 있습니다:

예시:

```
for file in seasonal/*.csv; do head -n 2 $file | tail -n 1;
done
```

작동 원리:

1. **파일 리스트 생성**:

   `seasonal/*.csv`는 `seasonal` 디렉토리 내의 **모든** `.csv` **파일**을 대상으로 합니다.

2. **루프 구조**:

   각 파일을 변수 `file`에 할당하며, 루프가 돌 때마다 `file`에는 순서대로 각 파일의 경로가 저장됩니다.

3. **명령어 본문 (pipeline)**:

   - `head -n 2 $file`: 각 파일의 **첫 2줄**을 출력합니다.
   - `| tail -n 1`: 그 중에서 **두 번째 줄**만 선택해 출력합니다.

결과:

각 `seasonal/*.csv` 파일의 **두 번째 줄**이 차례대로 출력됩니다. 예를 들어, `seasonal/spring.csv` 파일의 두 번째 줄이 먼저 출력되고, 그다음 `seasonal/summer.csv` 등의 두 번째 줄이 출력됩니다.

요약:

- 루프의 본문은 **두 개의 명령어를 파이프로 연결**한 구조입니다.
- 각 파일에서 필요한 부분을 추출하여 처리할 수 있으며, 이 방식은 다수의 파일에 대해 같은 작업을 반복 수행할 때 매우 유용합니다.

파일 이름에 공백이 포함된 경우: 따옴표 사용

쉘에서 파일 이름에 **공백**이 포함된 경우, 쉘은 이를 **여러 개의 파일**이나 **인자로** 해석할 수 있습니다. 따라서 공백이 포함된 파일 이름을 사용할 때는 **따옴표**로 감싸서 **하나의 파일 이름**으로 인식하게 해야 합니다.

문제 상황:

```
mv July 2017.csv 2017 July data.csv
```

이 명령어는 쉘에서 **네 개의 파일**(`July`, `2017.csv`, `2017`, `July`)을 `data.csv`라는 디렉토리로 옮기려는 시도로 잘못 해석됩니다.

해결 방법: 따옴표로 파일 이름 묶기

공백이 있는 파일 이름은 `' '` 또는 **`"`**로 묶어 **단일 인자**로 처리되게 해야 합니다.

```
mv 'July 2017.csv' '2017 July data.csv'
```

작동 원리:

1. `'July 2017.csv'`: 쉘이 이 부분을 **단일 파일**로 인식.
2. `'2017 July data.csv'`: 새 이름도 **단일 파일 이름**으로 인식.

다른 따옴표 사용 예시:

- **큰따옴표 사용**:

  ```
  mv "July 2017.csv" "2017 July data.csv"
  ```

따옴표 없이 파일 이름에 공백이 있으면 쉘은 이를 여러 개의 파일이나 인자로 인식하므로, 공백을 포함한 파일 이름을 사용할 때는 **항상 따옴표**로 묶어 사용해야 합니다.

쉘 루프: 여러 명령어 실행

쉘에서 **루프 본문**에 **여러 명령어**를 포함시킬 수 있으며, 명령어를 구분하기 위해서는 **세미콜론(`;`)**을 사용해야 합니다. 루프 안에 여러 명령어를 넣으면, 각 반복에서 여러 작업을 수행할 수 있습니다.

예시:

```
for f in seasonal/*.csv; do echo $f; head -n 2 $f | tail -n
1; done
```

작동 원리:

1. **파일 이름 출력** (`echo $f`):

   각 파일(`f`)의 이름을 출력합니다.

2. **두 번째 줄 출력** (`head -n 2 $f | tail -n 1`):

   각 파일의 첫 2줄 중 **두 번째 줄**만 출력합니다.

결과:

```
seasonal/autumn.csv
2017-01-05,canine
seasonal/spring.csv
2017-01-25,wisdom
seasonal/summer.csv
2017-01-11,canine
seasonal/winter.csv
2017-01-03,bicuspid
```

설명:

- **세미콜론(`;`)**: 각 명령어를 구분하는 역할을 하며, 한 줄에 여러 명령어를 넣을 수 있게 해줍니다.
- **루프 본문**: `echo`와 `head`/`tail`을 사용하여 각 파일의 이름과 그 파일의 두 번째 줄을 출력하고 있습니다.

이 방식은 여러 작업을 순차적으로 수행하는 루프를 구성할 때 매우 유용합니다.

Nano 텍스트 에디터

Nano는 사용하기 쉬운 텍스트 에디터로, **쉘에서 파일을 편집**할 때 유용합니다. 다른 복잡한 에디터와 달리 **간단한 키 조합**을 통해 텍스트를 편집하고 저장할 수 있습니다.

Nano에서 파일 열기

```
nano filename
```

- **filename**이 존재하면 그 파일을 열고, 존재하지 않으면 새로 만듭니다.

Nano에서 자주 사용하는 키 조합

- **Ctrl + K**: **한 줄 삭제**.
- **Ctrl + U**: **삭제된 줄 복원**.
- **Ctrl + O**: **파일 저장**(Save).
  - 파일명을 확인한 후 **Enter**를 눌러 저장을 완료해야 합니다.
- **Ctrl + X**: **에디터 종료**(Exit).

사용 예시

1. **파일 열기**:

```
nano myfile.txt
```

   - `myfile.txt`라는 파일을 Nano에서 편집.
2. **내용 편집**:
   - 화살표 키로 이동하고, 백스페이스로 문자 삭제.
   - **Ctrl + K**로 줄을 삭제하고, **Ctrl + U**로 삭제된 줄을 복원.
3. **파일 저장**:
   - **Ctrl + O**를 눌러 파일 저장 후 **Enter**로 확인.
4. **에디터 종료**:
   - **Ctrl + X**로 Nano 에디터 종료.

Nano는 간단한 텍스트 편집 작업에 적합하며, 위의 단축키들만으로도 기본적인 편집과 파일 관리를 할 수 있습니다.

명령어 기록 저장: history와 리디렉션 사용

복잡한 분석을 수행할 때, 사용한 **명령어 기록**을 남기는 것이 중요합니다. 쉘에서는 이미 제공된 도구들을 이용해 명령어 기록을 파일로 저장할 수 있습니다.

방법:

1. `history` **명령어**로 모든 명령어 기록을 확인합니다.
2. *`tail -n`*으로 **최근의 명령어들**을 선택합니다.
3. *리디렉션(`>`)**을 사용해 결과를 파일로 저장합니다.

예시:

```
history | tail -n 10 > figure-5.history
```

작동 원리:

- `history`: 지금까지 실행된 모든 명령어를 출력합니다.
- `tail -n 10`: 최근 **10개**의 명령어만 선택합니다. 필요에 따라 다른 숫자를 지정할 수 있습니다.

- **리디렉션(`>`)**: 이 출력을 `figure-5.history` 파일로 저장합니다.

장점:

- **누락 방지**: 수동으로 명령어를 적을 필요 없이 정확한 기록을 남길 수 있습니다.
- **재현성**: 나중에 동일한 분석을 다시 실행하거나 공유할 때 명령어 기록이 유용합니다.
- **쉘의 핵심 철학: 간단한 도구들을 결합**하여 다양한 문제를 해결하는 쉘의 철학을 잘 보여줍니다.

이 방식은 분석 중 중요한 명령어들의 흐름을 기록하고, 나중에 **분석 과정을 재현**하거나 **공유**할 때 매우 유용합니다.

쉘 스크립트(Shell Script) 사용

쉘에서 실행하는 명령어는 **텍스트 파일**에 저장해 반복적으로 실행할 수 있습니다. 이 파일을 **쉘 스크립트**라고 부르며, 파일에 명령어를 저장해 한 번에 여러 명령어를 실행할 수 있습니다.

예시: 쉘 스크립트 작성 및 실행

1. **스크립트 파일 생성**

   먼저, 명령어를 포함한 `headers.sh` 파일을 생성합니다:

   ```
   head -n 1 seasonal/*.csv
   ```

   이 명령어는 `seasonal` 디렉토리 내 모든 `.csv` 파일에서 **첫 번째 줄**을 선택합니다.

2. **스크립트 실행**

   스크립트를 실행하려면 **bash**를 사용해 실행합니다:

   ```
   bash headers.sh
   ```

작동 원리:

- **스크립트 작성**: `headers.sh` 파일에 명령어를 입력하여 저장합니다.
- **스크립트 실행**: `bash headers.sh` 명령어를 입력하면, `bash`가 **파일에 포함된 모든 명령어를 순차적으로 실행**합니다.

장점:

- **반복 실행**: 매번 명령어를 입력할 필요 없이, 스크립트를 실행하기만 하면 동일한 작업을 반복할 수 있습니다.
- **간편한 관리**: 여러 명령어를 파일에 저장해 두고 필요할 때마다 쉽게 실행할 수 있습니다.

확장 가능성:

이 방식은 간단한 명령어뿐만 아니라 **복잡한 분석**이나 **다중 명령어** 처리에도 사용할 수 있습니다. 여러 명령어를 스크립트로 작성하면 대규모 작업도 자동화할 수 있습니다.

쉘 스크립트(Shell Script)와 파이프라인

쉘 스크립트는 **쉘 명령어를 담은 파일**로, 명령어들을 자동화할 수 있는 강력한 도구입니다. 스크립트에는 **파이프라인**도 포함할 수 있으며, 이를 통해 여러 명령어를 연결하여 복잡한 작업을 처리할 수 있습니다.

예시: 파이프라인이 포함된 스크립트

다음과 같은 내용을 `all-dates.sh` 파일에 저장합니다:

```
cut -d , -f 1 seasonal/*.csv | grep -v Date | sort | uniq
```

이 스크립트는 아래와 같은 작업을 수행합니다:

1. **cut**: `seasonal/*.csv` 파일들에서 **첫 번째 열(날짜)**을 선택합니다.
2. **grep -v Date**: **"Date"**라는 단어가 포함된 줄(헤더)을 제외합니다.
3. **sort**: 날짜들을 **정렬**합니다.
4. **uniq**: **중복된 날짜**를 제거하여 **고유한 날짜**만 남깁니다.

스크립트 실행 및 출력 리디렉션

```
bash all-dates.sh > dates.out
```

이 명령어는:

- **스크립트** `all-dates.sh`를 실행하여, **고유한 날짜**를 추출하고,
- **출력**을 `dates.out` **파일**에 저장합니다.

작동 원리:

- **스크립트 파일**: 스크립트에 명령어를 저장하여, 언제든지 반복적으로 실행할 수 있습니다.
- **파이프라인**: 여러 명령어를 조합하여 데이터를 필터링하고 정렬하며, 이를 스크립트로 자동화할 수 있습니다.
- **리디렉션(`>`)**: 스크립트의 결과를 파일로 저장할 수 있습니다.

장점:

- **자동화**: 복잡한 명령어 흐름을 스크립트로 작성해 언제든지 재사용할 수 있습니다.
- **유연성**: 파이프라인과 리디렉션을 사용해 데이터를 쉽게 처리하고 원하는 파일에 저장할 수 있습니다.

이 방식은 데이터 처리, 파일 관리 등 다양한 작업에서 효율적으로 사용할 수 있습니다.

쉘 스크립트에서 매개변수 사용: $@

쉘 스크립트에서 **`$@`**는 **스크립트에 전달된 모든 명령어 인자**를 의미합니다. 이를 통해 스크립트를 더욱 유연하게 작성할 수 있으며, 다양한 파일을 처리하는 스크립트를 만들 수 있습니다. 즉, **특정 파일이 아닌 원하는 파일들**을 인자로 전달하여 스크립트가 실행되도록 설정할 수 있습니다.

예시: 매개변수를 받는 스크립트 작성

1. **스크립트 작성**

   `unique-lines.sh`라는 파일에 아래와 같은 명령어를 작성합니다:

   ```
   sort $@ | uniq
   ```

2. **스크립트 실행: 한 파일 처리**

   한 개의 파일을 인자로 전달하여 실행:

   ```
   bash unique-lines.sh seasonal/summer.csv
   ```

   여기서 **`$@`**는 `seasonal/summer.csv`로 대체되며, 이 파일을 정렬하고 고유한 줄을 추출합니다.

3. **스크립트 실행: 여러 파일 처리**

   여러 파일을 인자로 전달하여 실행:

```
bash unique-lines.sh seasonal/summer.csv seasonal/autum
n.csv
```

- *`$@`*는 `seasonal/summer.csv`와 `seasonal/autumn.csv`로 대체되며, 두 파일의 내용을 정렬하고 고유한 줄을 추출합니다.

중요한 사항

- *`$@`*는 스크립트에 전달된 **모든 파일 또는 인자**를 나타냅니다. 이를 사용하면 파일 이름이 고정된 것이 아닌, 실행 시마다 다른 파일을 처리할 수 있습니다.

파일 저장 및 종료 (Nano 에디터 사용)

1. **파일 저장**:

   Ctrl + O를 눌러 **파일 저장**. 그 후 **Enter**로 파일명을 확인.

2. **에디터 종료**:

   Ctrl + X를 눌러 **Nano 에디터 종료**.

요약

- *`$@`*는 스크립트에 전달된 모든 인자를 처리하는 표현식입니다.
- 스크립트 실행 시마다 다른 파일을 인자로 전달하여 유연하게 작업할 수 있습니다.
- **Nano**에서 스크립트를 저장하고 종료하는 방법도 잊지 마세요!

쉘 스크립트에서 매개변수 참조: $1, $2 등

쉘은 `$1`, `$2` 등의 형식을 사용하여 **특정 위치의 명령어 인자**를 참조할 수 있습니다. 이를 통해 스크립트에서 **인자의 순서**에 따라 파일 이름이나 열 번호 같은 값을 처리할 수 있습니다.

예시: 특정 열을 선택하는 스크립트 작성

1. **스크립트 작성**

   `column.sh`라는 스크립트 파일을 생성하고, 아래와 같은 명령어를 작성합니다:

   ```
   cut -d , -f $2 $1
   ```

2. **스크립트 설명**

   - `$1`: 첫 번째 인자, 즉 **파일 이름**을 의미합니다.

- `$2`: 두 번째 인자, 즉 **선택할 열 번호**를 의미합니다.
- `cut` 명령어는 쉼표(`,`)로 구분된 CSV 파일에서 지정된 열을 추출합니다.

3. **스크립트 실행**

```
bash column.sh seasonal/autumn.csv 1
```

- `seasonal/autumn.csv`: 첫 번째 인자 **`$1`*로 파일 이름을 전달합니다.
- `1`: 두 번째 인자 **`$2`*로 **첫 번째 열**을 선택합니다.

4. **출력**

   이 명령어는 `seasonal/autumn.csv` 파일의 **첫 번째 열**을 출력합니다. 여기서 중요한 점은 **매개변수를 역순**으로 사용했다는 것입니다. 즉, **열 번호**가 **두 번째 인자**로, **파일 이름**이 **첫 번째 인자**로 전달됩니다.

요약

- `$1`, `$2`, ...: 명령어의 **특정 인자**를 참조할 때 사용.
- 이 방식은 명령어의 구조를 **더 직관적이고 유연하게** 만들 수 있습니다.
- 스크립트를 실행할 때, 사용자는 **파일 이름**과 **열 번호**를 인자로 전달하면 됩니다.

멀티라인 쉘 스크립트 작성

쉘 스크립트는 **여러 줄의 명령어**를 포함할 수 있습니다. 이를 통해 더 복잡한 작업을 처리할 수 있으며, 예를 들어 데이터 파일의 **최소 및 최대 레코드 수**를 구하는 스크립트를 작성할 수 있습니다.

예시: 데이터셋의 길이를 구하는 스크립트

1. **스크립트 작성**

   `record-range.sh`라는 스크립트를 생성하여, 아래와 같이 작성합니다:

```
echo "Finding the shortest and longest datasets..."
wc -l seasonal/*.csv | sort -n | head -n 1
wc -l seasonal/*.csv | sort -n | tail -n 1
```

2. **스크립트 설명**

- `wc -l seasonal/*.csv`: 각 CSV 파일의 **줄 수(레코드 수)**를 계산합니다.
- `sort -n`: 결과를 **숫자 순서로 정렬**합니다.

- `head -n 1`: 가장 작은 값을 출력하여 **가장 짧은 파일**을 찾습니다.
- `tail -n 1`: 가장 큰 값을 출력하여 **가장 긴 파일**을 찾습니다.

3. **스크립트 실행**

```
bash record-range.sh
```

파일 저장 및 종료 (Nano 에디터 사용)

- **복사 및 붙여넣기**:
  - **Ctrl + K**: 원하는 줄을 **잘라내기**.
  - **Ctrl + U**: 해당 줄을 **붙여넣기**. 두 번 누르면 두 개의 복사본이 붙여넣어집니다.
- **파일 저장**:

  Ctrl + O를 눌러 파일을 저장하고, **Enter**로 확인.
- **에디터 종료**:

  Ctrl + X로 Nano 에디터를 종료.

요약

- **멀티라인 쉘 스크립트**를 사용하면 여러 작업을 한 번에 처리할 수 있습니다.
- `wc -l`, `sort`, `head`, `tail` 등을 사용해 데이터 파일의 **레코드 수**를 확인하고 **가장 짧고 긴 파일**을 찾는 예시입니다.
- **Nano** 에디터에서 명령어를 편집할 때, **복사 및 붙여넣기** 기능을 활용해 쉽게 수정할 수 있습니다.

쉘 스크립트에서 루프 작성

쉘 스크립트는 **루프**를 포함할 수 있으며, 이는 여러 파일이나 데이터를 처리하는 데 매우 유용합니다. 루프는 한 줄에 **세미콜론(;)**을 사용해 명령어를 연결하거나, **여러 줄로 나누어** 더 가독성 있게 작성할 수 있습니다.

예시: 루프가 포함된 스크립트

```
# 각 파일의 첫 번째와 마지막 데이터를 출력하는 스크립트
for filename in $@
do
    head -n 2 $filename | tail -n 1
```

```
    tail -n 1 $filename
done
```

작동 원리:

1. `$@`: 스크립트에 전달된 모든 파일 이름을 처리합니다.
2. **루프 구조**:
   - 각 파일의 이름을 변수 `filename`에 할당.
   - `head -n 2 $filename | tail -n 1`: 파일의 첫 2줄 중 **두 번째 줄**을 출력합니다.
   - `tail -n 1 $filename`: 파일의 **마지막 줄**을 출력합니다.

주석(Comment):

- **주석은 `#`로 시작**하며, 해당 줄의 끝까지가 주석으로 간주됩니다. 주석은 스크립트의 목적과 동작을 설명하며, 나중에 코드를 이해하는 데 도움이 됩니다.

예시 스크립트 실행:

```
bash myscript.sh seasonal/spring.csv seasonal/summer.csv
```

이 명령어는 `spring.csv`와 `summer.csv` 파일에서 각각 **첫 번째 데이터와 마지막 데이터**를 출력합니다.

Nano 에디터에서 스크립트 작성 및 저장:

- **복사 및 붙여넣기**:
  - **Ctrl + K**: 원하는 줄을 **잘라내기**.
  - **Ctrl + U**: 해당 줄을 **붙여넣기**.
- **파일 저장**:

  Ctrl + O를 눌러 저장하고, **Enter**로 파일명을 확인.
- **에디터 종료**:

  Ctrl + X로 Nano 에디터를 종료.

요약:

- 쉘 스크립트에서 루프를 작성하여 **여러 파일을 반복적으로 처리**할 수 있습니다.
- *주석(`#`)**을 추가해 스크립트의 목적과 동작을 설명하는 것이 좋습니다.

- Nano 에디터에서 스크립트를 저장하고 나중에 사용할 수 있도록 작성합니다.

쉘 스크립트 및 명령어에서 파일명 위치 실수

쉘에서 **명령어에 파일명을 잘못된 위치에 넣는 실수**는 흔하게 발생합니다. 파일을 처리할 때는 명령어와 파이프라인의 흐름에 맞게 파일명을 정확히 지정해야 합니다.

예시 1: `tail` 명령어에서 파일명 누락

```
tail -n 3
```

이 명령어는 **파일명을 제공하지 않았기 때문에**, `tail`은 입력을 **키보드로부터** 대기합니다. 파일명을 명시해야 `tail`이 정상적으로 파일의 마지막 줄을 출력할 수 있습니다.

예시 2: 잘못된 파이프라인 사용

```
head -n 5 | tail -n 3 somefile.txt
```

이 명령어에서 발생하는 문제:

- `head -n 5`: 파일명을 받지 않았기 때문에, 입력을 **키보드로부터** 대기하게 됩니다.
- `tail -n 3 somefile.txt`: `somefile.txt` 파일의 마지막 3줄을 출력하지만, `head`는 입력 대기 상태로 멈춰 있습니다.

올바른 사용 예시:

1. **파일을 `head`에 전달**:

   ```
   head -n 5 somefile.txt | tail -n 3
   ```

   - `head -n 5 somefile.txt`: `somefile.txt`의 첫 5줄을 출력.
   - `tail -n 3`: 이 결과에서 마지막 3줄을 출력.

2. **`tail`에만 파일 전달**:

   ```
   tail -n 3 somefile.txt
   ```

   - 이 명령어는 `somefile.txt` 파일의 마지막 3줄을 출력합니다.

요약:

- 명령어에 **파일명을 올바른 위치**에 제공하는 것이 중요합니다. 그렇지 않으면 쉘 명령어가 **입력 대기 상태**로 멈추게 됩니다.
- **파이프라인**을 사용하여 파일을 처리할 때, 파일명은 명령어의 흐름에 맞게 지정해야 합니다.
