챗GPT 유닉스 쉘

이광춘 신종화

2023년 06월 16일

목차

서	서문					
Ι	들c	거가며	3			
1	환경	설정	5			
	1.1	챗GPT 연동	5			
		1.1.1 가상환경 설정	6			
		1.1.2 API KEY 얻기	6			
		1.1.3 API KEY 저장	7			
		1.1.4 헬로월드	7			
		1.1.5 OpenAI 모형	10			
	1.2	윈도우 쉘 환경	11			
		1.2.1 WSL	12			
		1.2.2 ShellGPT	13			
	1.3	실습 데이터 준비	18			
2 OpenAI 들어가며		nAI 들어가며	21			
	2.1	텍스트 완성	21			
	2.2	키워드 추출	22			
	2.3	텍스트 요약	25			
	2.4	여론조사 할일 생성	27			
II	쉘	프로그래밍	31			
3	쉨 <i>(</i> S	Shell) 소개	33			
	-	배경	33			

	3.2	쉘(Shell)	35
	3.3	어떻게 생겼을까?	35
	3.4	ls 와 플래그 의미 파악	37
	3.5	어려운가요?	38
	3.6	유연성과 자동화	38
	3.7	사례:문제정의	38
4	파일	과 폴더 넘나들기	41
	4.1	도움말 얻기	44
		4.1.1help 플래그	
		4.1.2 man 명령어	46
	4.2	cd 디렉토리 변경	48
	4.3	상대/절대 경로	51
	4.4	쉘 명령 일반구문	54
	4.5	사례:파일 구성	56
5	파일	<u> </u> 과 디렉토리 작업	61
	5.1	· · · · · · · · · · · · · · · · · · ·	
		해답과 설명	
	5.2	다수 파일과 폴더 작업	73
6	파이	프와 필터	79
	6.1	·— · c · 명령어에서 출력 캡처하기	
	6.2	출력 필터링	
	6.3	다른 명령에 출력 전달하기	
		여러 명령 결합하기	
	6.5	함께 작동하도록 설계된 도구	
	6.6	사례:파일 확인하기	92
7	루끄	E(Loops)	97
•	, _	7.0.1 프롬프트 따라가기	
	7.1	파일, 디렉토리, 변수 등 이름에 공백	
		7.1.1 역사(history)를 아는 사람은 반복할 수 있다	
	7.2	사례: 많은 파일 처리하기	

8	쉘 스크립트	115
	8.1 사례:스크립트 생성하기	. 126
9	파일, 텍스트, 폴더 찾기	129
	9.0.1 작은 아낙네(Little Women)	. 135
II	GPT 데이터 과학	145
10	CLI 데이터 과학	147
	10.1 R/파이썬	. 147
	R 코드	. 148
	파이썬 코드	. 148
	10.2 쉘 스크립트	. 149
	10.3 윈도우 파일 변환	. 149
	10.4 파일 살펴보기	. 150
	10.5 파일 구조	. 151
	10.6 기술통계량	. 153
	10.7 대용량 데이터	. 155
	10.7.1 압축파일 풀기	. 155
	10.7.2 표본추출 전략 수립	. 156
	10.7.3 표본추출 툴설치	. 156
	10.7.4 1% 표본 추출	. 157
11	챗GPT 데이터 과학	159
	11.1 실습 데이터	. 159
	11.2 펭귄종별 개체수	. 160
	11.3 대용량 데이터	. 165
	11.3.1 오픈 데이터셋	. 165
참	고문헌	171

서문

한국 R 사용자회는 기존 수년동안 블로그 형태로 저작된 데이터 과학 관련 저작물을 2022년 쿼토(Quarto) 전자책 기능에 도입하여 디지털 글쓰기의 한 사례로 다양한 전자책을 공개했다. 1 데이터 과학 글쓰기를 확장한 디지털 글쓰기와 출판은 전혀 다른 개념이라 특히 한국어 특징을 반영한 출판 도구가 전무한 상황에서 한국 R 사용자회가 주도되어 bitPublish 패키지를 개발하게 되었다. 이를 바탕으로 한국 R 사용자회 출판사를 통해 첫번째 챗GPT 시대 데이터 과학 자동화 필수도구인 유닉스 쉘(Unix Shell)을 소개하는 "챗GPT 유닉스 쉘" 출판을 하게 되었다.

학습목표

bitPublish가 제공하는 기능을 이해하고, bitPublish을 이용해서 원하는 책을 저작할 수 있다.

¹https://r2bit.com/book.html

제I편

들어가며



제 1 장

환경설정

학습목표

챗GPT와 유닉스 쉘에서는 파일 시스템과 쉘의 기초를 소개한다. 컴퓨터에 파일을 저장해 본 적이 있고 '파일'이라는 단어와 '디렉토리' 또는 '폴더'(의미 동일한두 가지 단어)를 알고 있다면 준비가 된 것이다.

"챗GPT와 유닉스 쉘(Unix Shell)"은 전세계적으로 수천번의 워크샵과 국내에서도 수년 동안 소프트웨어 카펜트리 워크샵 경험을 정식 교재를 통해서 한글화하여 출판을 준비하면서 최근에 큰 반향을 일으키고 있는 챗GPT 관련 내용도 담아내어 한번 학습하여 평생토록 활용할 수 있는 것에 방점을 두고 있다.[2]

1.1 챗GPT 연동

OpenAI GPT 제품은 파이썬이 주 개발언어로 되어 있어 파이썬을 기준으로 챗GPT 개발을 진행한다. 일반적인 API를 활용한 파이썬 개발작업흐름은 먼저, 가상환경을 구축하고, API KEY를 발급받고 해당 openai 패키지를 설치하고 헬로월드를 찍고 본격적인 AI제품 개발을 진행하는 방식으로 추진한다.



1.1.1 가상환경 설정

다양한 가상환경이 있어 필요한 패키지를 사용하여 파이썬 가상환경을 구축한다. 파이썬 3.3 버전부터 내장된 venv, 많이 사용되는 virtualenvwrapper, virtualenv 등이 유명하다. 본인 취향에 맞는 가상환경을 특정하여 업무에 사용한다. 다음은 venv를 사용해서 가상 개발환경을 구축하는 것을 예시로 보여주고 있다.

```
## 디렉토리 생성 및 프로젝트 디렉토리 이동
mkdir myproject
cd myproject

## 가상환경 생성
python -m venv myenv

## 가상환경 활성화
myenv\Scripts\activate # 윈도우즈
source myenv/bin/activate # 리눅스/맥

## 가상환경 비활성화
deactivate
```

1.1.2 API KEY 얻기

가상환경을 구축한 다음 OpenAI에서 제공하는 공식 API에 접근할 수 있는 API 키를 생성하는 것이다. https://openai.com/api/ ¹ 로 이동하여 계정을 만듭니다.

안내에 따라 계정을 생성한 다음 https://platform.openai.com/account/api-keys ² 로 이동 하여 API 키를 생성한다.

API 키는 조직에 속해야 하며, 조직을 생성하라는 메시지가 표시되는 경우 조직명을 입력한다. 하나의 조직에 속한 경우 조직 ID(Organization ID)를 별도 생성할 필요는 없다. OpenAI 계정을 통해서는 생성한 API KEY는 다시 볼 수 없기 때문에 생성한 비밀 키를 안전하고 접근하기 쉬운 곳에 저장한다.

¹https://openai.com/api/

²https://platform.openai.com/account/api-keys

1.1.3 API KEY 저장

API KEY를 환경변수로 지정하여 호출하는 방식도 있고, 작업 프로젝트 디렉토리에 로 컬 파일에 저장하여 사용하는 방식도 있다. 먼저 윈도우에서 시스템으로 들어가서 환경 변수로 지정하면 해당 변수(OPENAI_API_KEY)를 다양한 프로그램에서 호출하여 사용할 수 있다.



다른 방식은 .env와 같은 파일을 프로젝트 디렉토리 아래 숨긴 파일에 지정하여 사용하 는 방식이다. 이런 경우 .gitignore 파일에 버전제어 대상에서 제외시켜 두는 것을 필히 기억한다.



1.1.4 헬로월드

OpenAI API KEY도 준비가 되었으면 헬로월드 프로그램을 작성해보자. 개발자가 하나의 조직에 속한 경우, API KEY를 운영체제 환경변수로 지정한 경우 다음과 같이 시스템 환 경에서 OPENAI_API_KEY 키를 가져와서 OpenAI에서 제공하는 모델목록을 확인할 수 있다.

```
import os
import openai
```

```
openai.api_key = os.getenv("OPENAI_API_KEY")
# API 호출 및 모델 목록 출력
models = openai.Model.list()
print(models['data'][0])
#> {
    "created": 1677532384,
#>
    "id": "whisper-1",
    "object": "model",
#>
    "owned_by": "openai-internal",
#>
    "parent": null,
#>
     "permission": [
#>
#>
         "allow_create_engine": false,
#>
        "allow_fine_tuning": false,
#>
         "allow_logprobs": true,
#>
        "allow_sampling": true,
#>
        "allow_search_indices": false,
#>
        "allow view": true,
#>
#>
        "created": 1683912666,
        "group": null,
#>
        "id": "modelperm-KlsZlfft3Gma8pI6A8rTnyjs",
#>
#>
        "is_blocking": false,
         "object": "model_permission",
#>
        "organization": "*"
#>
     }
#>
#> ],
#> "root": "whisper-1"
#> }
```

다른 방식은 로컬 파일에 API KEY와 ORG ID 를 저장하고 이를 불러와서 개발에 사용하는 방식이다.

```
import os
import openai
# .env 파일에서 API_KEY 와 ORG_ID 을 읽어온다.
with open(".env") as lines:
  for line in lines:
    key, value = line.strip().split("=")
    os.environ[key] = value
# api_key와 organization 지정
openai.api_key = os.environ.get("API_KEY")
openai.organization = os.environ.get("ORG_ID")
# API 호출 및 모델 목록 출력
gpt_models = openai.Model.list()
print(gpt_models['data'][0])
#> {
    "created": 1677532384,
    "id": "whisper-1",
#>
#>
    "object": "model",
    "owned_by": "openai-internal",
#>
    "parent": null,
#>
#>
    "permission": [
#>
         "allow_create_engine": false,
#>
         "allow_fine_tuning": false,
#>
         "allow_logprobs": true,
#>
         "allow_sampling": true,
#>
         "allow_search_indices": false,
#>
        "allow_view": true,
#>
#>
         "created": 1683912666,
```

```
#>
        "group": null,
        "id": "modelperm-KlsZlfft3Gma8pI6A8rTnyjs",
#>
        "is_blocking": false,
#>
        "object": "model_permission",
#>
        "organization": "*"
#>
     }
#>
#>
   ],
#> "root": "whisper-1"
#> }
```

1.1.5 OpenAl 모형

OpenAI에서 제공하는 다양한 모델을 확인할 수 있다. system이 소유한 GPT 모형을 살펴 보자.

created id object owned_by

프로그램 코드(code) 관련된 GPT 모형도 확인할 수 있다.

```
gpt_models_tbl %>%
 filter(str_detect(id, "code")) %>%
 gt::gt() %>%
 gtExtras::gt_theme_nytimes()
```

created	id	object	owned_by
19111	babbage-code-search-code	model	openai-dev
19111	babbage-code-search-text	model	openai-dev
19111	code-search-babbage-text-001	model	openai-dev
19111	code-search-babbage-code-001	model	openai-dev
19111	ada-code-search-code	model	openai-dev
19111	code-search-ada-text-001	model	openai-dev
19111	ada-code-search-text	model	openai-dev
19111	code-search-ada-code-001	model	openai-dev
19096	code-davinci-edit-001	model	openai

1.2 윈도우 쉘 환경

윈도우 운영체제에서 유닉스 쉘(Shell)을 사용하는 방법은 여러가지가 있다. 윈도우 운 영체제에 내장된 명령 프롬프트(cmd.exe)는 OS/2, 윈도우 임베디드, 윈도우 NT 5.0 이상 기반 시스템의 명령어 인터프리터로 친숙하고 최근에 파워쉘(PowerShell)은 마이크로소 프트가 개발한, 확장 가능한 명령 줄 인터페이스 셸 및 스크립트 언어를 특징으로 하는 명령어 인터프리터다. 하지만, 이런 cmd, PowerShell은 윈도우 운영체제에서만 동작된다 는 한계가 있다. 맥과 리눅스 특히, 클라우드를 지배하고 있는 유닉스 쉘을 윈도우에서 도 무리없이 사용하게 된다면 모든 운영체제에서 공통작업을 수행하는데 큰 도움이 될 것이다. 이를 위해 WSL(Windows SubSystem for Linux)를 통해 다양한 리눅스 운영체제를 설치하는 경우와 Git Bash를 설치하여 유닉스 쉘을 윈도우에서도 활용이 가능하다.

1.2.1 WSL

WSL(Linux용 Windows 서브시스템)을 통해 우분투(Ubuntu), 데비안(Debian)를 포함한 다양한 리눅스 배포판을 설치할 수 있다. 여기서 데비안를 설치하는 경우를 살펴보자. 기본적인 작업흐름은 다른 리눅스 배포판 설치와 동일한데 먼저 WSL을 설치하고 이어서데비안 리눅스 배포판을 설치한다.

윈도우즈 PowerShell을 통해 WSL(Linux용 Windows 하위 시스템)을 설치한다.

\$ wsl --install

상기 명령을 실행하면 시스템이 WSL과 Linux 배포판(기본값은 우분투)을 설치하고 PC 를 재시작한다.

'가상 머신 플랫폼' 선택적 구성 요소를 활성화시킨다. 새 Linux 배포판을 설치하기 전에 '가상 머신 플랫폼' 구성 요소가 활성화되어 있는지 확인해야 하고, 관리자 권한으로 PowerShell을 다시 열고 실행한다.

PC를 다시 시작하여 WSL 설치 및 WSL 2로 업데이트를 완료하고, WSL 2를 기본 버전으로 설정한다.

\$ wsl --set-default-version 2

wsl --install 디폴트로 우분투 리눅스 배포판이 설치되지만 -d Debian 인자를 넣게 되면 데비안 리눅스 배포판을 설치할 수 있다.

\$ wsl --install -d Debian

정상으로 WSL 데비안 배포판이 설치되었는지 확인한다.

```
$ cat /etc/os-release
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
NAME="Debian GNU/Linux"
VERSION_ID="11"
VERSION="11 (bullseye)"
VERSION_CODENAME=bullseye
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

1.2.2 ShellGPT

OpenAI의 ChatGPT(GPT-3.5)는 콘텐츠 생성에 주된 방점이 있지만 다양한 프로그래밍 코드도 작성함은 물론 유닉스 쉘 프로그램도 작성하여 CLI 생산성을 높이는데 사용될수 있다. ChatGPT 기능을 활용하여 쉘 명령, 코드 스니펫, 주석, 문서 등을 생성할 수 있다. 즉, 데이터 과학자를 비롯한 개발자가 기존에 업무를 수행하던 방식이 전혀 다르게 된다. 즉, 책, 매뉴얼, 비밀노트(Cheat Sheet), 인터넷 북마크, 구글링 같은 검색없이 바로 터미널에서 바로 정확한 답변을 얻어 귀중한 시간과 노력을 절약할 수 있다. 예를 들어 앞서 프로젝트를 할 때 해당 유닉스 쉘 명령어가 기억나지 않는다고 하면 ShellGPT를 사용하여해당 작업을 신속하게 수행할 수 있다

ShellGPT를 사용하기 위한 설정은 다음과 같다. ShellGPT가 파이썬으로 개발되어 WSL 데비안 혹은 우분투로 설치한 경우 다음 순서대로 설치를 해야한다.

cURL은 다양한 통신 프로토콜을 이용하여 데이터를 전송하기 위한 라이브러리와 명령 줄 도구를 제공하는 컴퓨터 소프트웨어로 쉘에서 설치하는데 파이썬 설치파일을 전송 받는데 필요하다.

\$ sudo apt install curl

파이썬을 설치하기 위해서 get-py.py 파일이 필요한데 설치에 앞서 python3-distutils 파일을 설치하다.

```
$ sudo apt-get install python3-distutils
```

파이썬 설치파일을 다운로드하고 파이썬을 설치하고 python3 명령어를 통해 버전을 확 인한다.

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ sudo python3 get-pip.py
$ python3 --version
Python 3.9.2
```

Python 3.x는 Python 2.x가 설치된 시스템에서 python3로 설치되는 경우가 많으며, python 명령은 이전 버전(Python 2.x)에 링크되어 있는 경우가 많고 리눅스 배포판에서 일반적이다. Python 3.x에서 python 명령을 사용하려면 별칭(alias)을 만들어야 한다.

nano 편집기를 열고 alias python=python3을 추가하고 저장한다.

```
$ nano ~/.bashrc
```

alias python=python3

다시 쉘에서 변경사항을 적용시키면 python 명령어를 실행하게 되면 python3 가 대신 실행되는 것을 확인할 수 있다.

```
$ source ~/.bashrc
$ python --version
Python 3.9.2
```

ShellGPT를 설치한다.

```
$ sudo pip install shell-gpt==0.9.2
```

sgpt를 처음 실행하게 되면 OpenAI API Key를 입력해야 한다. OpenAI에서 API Keys를 생성하여 붙여넣기 하면 ~/.config/shell_gpt/.sgptrc 파일에 저장되어 ShellGPT를 사용하게 된다.

\$ sgpt

Please enter your OpenAI API key:

cat 명령어로 OPENAI API KEY가 정상등록된 것을 확인하고 헬로월드를 보내보자. sgpt 다 음에 자연어로 지시사항을 보내면 gpt-3.5-turbo 엔진을 통해 자연어를 해석하고 이를 결과로 반환해준다.

```
$ cat ~/.config/shell_gpt/.sgptrc
CHAT_CACHE_PATH=/tmp/chat_cache
CACHE PATH=/tmp/cache
CHAT_CACHE_LENGTH=100
CACHE LENGTH=100
REQUEST_TIMEOUT=60
DEFAULT_MODEL=gpt-3.5-turbo
OPENAI_API_HOST=https://api.openai.com
DEFAULT_COLOR=magenta
ROLE_STORAGE_PATH=/home/xxxxxxxx/.config/shell_gpt/roles
SYSTEM ROLES=false
DEFAULT_EXECUTE_SHELL_CMD=false
$ sgpt "대한민국의 수도는 어디야?"
서울입니다.
```

sqpt 다음에 "현재 폴더 및 하위 폴더에 있는 모든 txt 파일을 보여주세요." 명령어를 보내 면 이를 쉘 명령어로 변환한 후 유닉스 쉘 명령어를 제시하여 준다.

```
$ sqpt "현재 폴더 및 하위 폴더에 있는 모든 txt 파일을 보여주세요."
현재 폴더 및 하위 폴더에 있는 모든 txt 파일을 보여드리겠습니다. 아래 명령어를
↔ 입력해주세요.
find . -name "*.txt"
tidyverse@dl:/mnt/d/tcs/gpt-shell/shell-lesson-data$ find . -name "*.txt"
```

```
./exercise-data/numbers.txt
./exercise-data/writing/haiku.txt
./exercise-data/writing/LittleWomen.txt
./north-pacific-gyre/NENE01729A.txt
./north-pacific-gyre/NENE01729B.txt
./north-pacific-gyre/NENE01736A.txt
./north-pacific-gyre/NENE01751A.txt
./north-pacific-gyre/NENE01751B.txt
./north-pacific-gyre/NENE01812A.txt
./north-pacific-gyre/NENE01843A.txt
./north-pacific-gyre/NENE01843B.txt
./north-pacific-gyre/NENE01971Z.txt
./north-pacific-gyre/NENE01978A.txt
./north-pacific-gyre/NENE01978B.txt
./north-pacific-gyre/NENE02018B.txt
./north-pacific-gyre/NENE02040A.txt
./north-pacific-gyre/NENE02040B.txt
./north-pacific-gyre/NENE02040Z.txt
./north-pacific-gyre/NENE02043A.txt
./north-pacific-gyre/NENE02043B.txt
```

한단계 더 들어가 -s 매개변수를 제공하면 실행([E]xecute) 명령어 기술([D]escribe) 혹은 중단([A]bort)을 지정할 수 있다.

```
$ sgpt -s "현재 폴더 및 하위 폴더에 있는 모든 txt 파일을 보여주세요."
find . -name "*.txt"

[E]xecute, [D]escribe, [A]bort: D

Searches for all files with the extension ".txt" in the current directory and
  its subdirectories.

[E]xecute, [D]escribe, [A]bort: E
  ./exercise-data/numbers.txt
  ./exercise-data/writing/haiku.txt
```

```
./exercise-data/writing/LittleWomen.txt
./north-pacific-gyre/NENE01729A.txt
./north-pacific-gyre/NENE01729B.txt
./north-pacific-gyre/NENE01736A.txt
./north-pacific-gyre/NENE01751A.txt
./north-pacific-gyre/NENE01751B.txt
./north-pacific-gyre/NENE01812A.txt
./north-pacific-gyre/NENE01843A.txt
./north-pacific-gyre/NENE01843B.txt
./north-pacific-gyre/NENE01971Z.txt
./north-pacific-gyre/NENE01978A.txt
./north-pacific-gyre/NENE01978B.txt
./north-pacific-gyre/NENE02018B.txt
./north-pacific-gyre/NENE02040A.txt
./north-pacific-gyre/NENE02040B.txt
./north-pacific-gyre/NENE02040Z.txt
./north-pacific-gyre/NENE02043A.txt
./north-pacific-gyre/NENE02043B.txt
```

사용방법은 Git Bash를 설치한 후 터미널을 열구 sgpt --shell 혹은 sgpt -s 다음에 자연어를 넣게 되면 해당되는 쉘 명령어를 알려준다. 이를 실행하게 되면 유닉스 쉘을 이용하여 해당 자동화 작업에 생산성을 높일 수 있다. 얼마전까지만해도 한국어는 지원하지 않아 한글로 작성한 다음 번역기를 사용하여 영어로 입력해야 하고 결과를 얻게 되면 이를 실행하는 방식이 활용되었지만 지금은 유닉스 쉘 개념을 이해하여 한국어로 말을 풀어주게 되면 구현이 가능하다.

```
NENE01736A.txt* NENE01812A.txt* NENE01971Z.txt*
goostats.sh*
 S NENE02018B.txt* NENE02040Z.txt*
NENE01729A.txt* NENE01751A.txt* NENE01843A.txt* NENE01978A.txt*
 → NENE02040A.txt* NENE02043A.txt*
$ sgpt -s '현재 디렉터리 콘텐츠를 나열하고 각 파일 이름 끝에 파일 유형을 나타
↔ 내는 특수 문자를 표시합니다.'
1s -F
[E]xecute, [D]escribe, [A]bort: E
               NENE01729B.txt* NENE01751B.txt* NENE01843B.txt*
goodiff.sh*
NENE01978B.txt* NENE02040B.txt* NENE02043B.txt*
goostats.sh*
              NENE01736A.txt* NENE01812A.txt* NENE01971Z.txt*

→ NENE02018B.txt* NENE02040Z.txt*

NENE01729A.txt* NENE01751A.txt* NENE01843A.txt* NENE01978A.txt*
 → NENE02040A.txt* NENE02043A.txt*
```

1.3 실습 데이터 준비

실습데이터는 소프트웨어 카펜트리 유닉스 쉘 학습 페이지에서 직접 다운로드 받을 수 있다. 3 WSL 우분투 혹은 데비안 쉘을 실행시켜 cd 명령어로 다음 디렉토리로 이동한다.

\$ cd /mnt/c/users/사용자명/Desktop

curl 명령어로 shell-lesson-data.zip를 다운로드 받고 저장 파일명도 -o 옵션을 주고 동 일한 shell-lesson-data.zip 파일명으로 바탕화면(Desktop)에 저장한다.

```
$ curl https://swcarpentry.github.io/shell-novice/data/shell-lesson-data.zip
   -o shell-lesson-data.zip
```

unzip 명령어로 shell-lesson-data.zip 파일 압축을 바탕화면(Desktop) 디렉토리에 -d . 선택옵션을 주어 작업을 실행한다.

 $^{^3} https://swcarpentry.github.io/shell-novice/data/shell-lesson-data.zip \\$

\$ unzip shell-lesson-data.zip -d .

tree 명령어로 디렉토리 구조만 -d 옵션을 주어 압축을 푼 shell-lesson-data 폴더 아래 구조를 확인한다.

\$ tree -d shell-lesson-data shell-lesson-data — exercise-data — alkanes ├─ animal-counts — creatures └─ writing □ north-pacific-gyre 6 directories

i -bash: tree: command not found

tree 명령어를 실행할 수 없는 경우 다음 명령어로 패키지를 설치하여 문제를 해결 한다.

\$ sudo apt-get install tree

1.1 연습문제

- 1. 실습 데이터를 준비하고 압축을 해제하는데 사용된 명령어는 무엇일까요?
 - (1) cd
 - (2) curl
 - (3) unzip
 - (4) 모두 사용됨

- 2. 만약 'tree' 명령어를 실행할 수 없다면, 어떤 명령어를 사용하여 문제를 해결해야 합 니까?
- 3. 챗GPT 연동 과정에서 가상환경 구축에 사용되는 파이썬 모듈은 무엇일까요?
 - (1) virtualenv
 - (2) venv
 - (3) virtualenvwrapper
 - (4) 모두 맞음
- 4. OpenAI API Key를 저장하는 방식 중에서 어떤 파일에 지정하여 사용하면 버전 제어 대상에서 제외시켜야 하는 파일명은 무엇일까요?
 - (1) .env
 - (2) .gitignore
 - (3) .apikey
 - (4) apikey.txt
- 5. API Kev를 환경변수로 지정하여 사용하려면 윈도우에서 어디로 접근해야 하는가?
 - (1) 시스템 환경 변수 설정
 - (2) API Key 관리
 - (3) 프로젝트 설정
 - (4) 환경변수.txt 파일 생성 및 저장
- 6. 소프트웨어 카펜트리 유닉스 쉘에 사용되는 실습데이터를 'curl' 명령어로 다운로드 받아 압축을 풀어 'curl' 명령어로 저장을 한 후에 압축을 풀고 전체 디렉토리 구조를 일별합니다. 실습데이터는 소프트웨어 카펜트리 유닉스 쉘 학습 페이지에서 직접 다운로드받을 수 있다. ⁴ 'shell-lesson-data.zip' 압축을 풀고 폴더명은 shell-data 으로 지정합니다. 마지막으로, shell-data 아래 전체 디렉토리 구조만 나무형식으로 파일은 제외하고 보여주세요.

 $^{^4} https://swcarpentry.github.io/shell-novice/data/shell-lesson-data.zip \\$

제 2 장

OpenAI 들어가며

학습목표

본서에서는 주로 작업 자동화 관련 챗GPT를 다루지만 OpenAI를 비롯한 다양한 거대언어모형(LLM) 기반 AI를 활용하여 다양한 자연어 처리 관련 업무를 수행할 수 있다. OpenAI 환경설정을 통해 유닉스 쉘 작업자동화 외에 챗GPT를 활용할 수 있는 업무에 대해서 간략히 살펴본다.

2.1 텍스트 완성

GPT를 사용하여 다양한 작업을 수행할 수 있지만 가장 기본적인 작업은 글쓰기다. GPT가 생성형 AI로 해당 텍스트를 주어지면 나머지 텍스트를 해당 최대 토큰 크기(max_tokens) 길이만큼 텍스트를 생성해준다.

```
import os
import openai

openai.api_key = os.getenv("OPENAI_API_KEY")

complete_next = openai.Completion.create(
    model="text-davinci-003",
```

```
prompt="나의 살던 고향은",
max_tokens=7,
temperature=0)

complete_next['choices'][0]['text']
'\n\n나의'
```

토큰 크기를 100으로 지정하면 제법 긴 텍스트를 출력한다. 영어 토큰에 최적화되어 있는 관계로 한글의 경우 토큰 낭비(?)가 심한 것으로 보인다. 고로 비용이 제법 나가는 점은 한국어로 작업을 할 때 고려해야만 된다.

```
complete_next_100 = openai.Completion.create(
    model="text-davinci-003",
    prompt="나의 살던 고향은",
    max_tokens=100,
    temperature=0)

complete_next_100['choices'][0]['text']

'\n\n나의 고향은 전라남도 여수시입니다. 여수는 전라남도의 동부에 위치한 해안
    도시로'
```

2.2 키워드 추출

조금더 흥미로운 주제로 해당 문서를 제시하고 관련 텍스트의 주요 키워드를 추출해보자. Attention Is All You Need 논문[1]은 AI 분야에서 획기적인 논문으로 평가받있지만 별도 키워드는 제시되고 있지 않아 논문 초록을 앞에 제시하고 Keywords:를 뒤에 두고 논문의 주요 키워드를 추출하게 한다.

prompt_keywords = "The dominant sequence transduction models are based on

- complex recurrent or convolutional neural networks that include an encoder
- and a decoder. The best performing models also connect the encoder and
- △ decoder through an attention mechanism. We propose a new simple network
- □ architecture, the Transformer, based solely on attention mechanisms,
- □ dispensing with recurrence and convolutions entirely. Experiments on two
- → machine translation tasks show these models to be superior in quality
- 4 while being more parallelizable and requiring significantly less time to
- train. Our model achieves 28.4 BLEU on the WMT 2014 Englishto-German
- translation task, improving over the existing best results, including
- ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation
- 🗸 task, our model establishes a new single-model state-of-the-art BLEU score
- $_{ iny -}$ of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the
- ← training costs of the best models from the literature. We show that the
- Transformer generalizes well to other tasks by applying it successfully to
- English constituency parsing both with large and limited training
- data.\n\n keywords:"

```
keywords = openai.Completion.create(
  model="text-davinci-003",
  prompt=prompt_keywords,
  temperature = 0.5,
  max_tokens = 50)
```

max_tokens을 50으로 제한하여 temperature = 0.5로 너무 창의적이지 않게 키워드 추출 작업을 지시한 경우 다음과 같은 결과를 어덱 된다.

'\nSequence transduction, neural networks, attention mechanisms, machine translation, parsing'

이번에는 한글 논문초록에서 키워드를 추출해보자. 2020년 출간된 논문[3]의 한글 초록에서 제시된 키워드와 OpenAI GPT가 제시하고 있는 키워드와 비교해보자.

• 논문 소스코드: 바로가기

• PDF 출판 논문: 다운로드

prompt_keywords = "알파고가 2016년 바둑 인간 챔피언 이세돌 9단을 현격한 기량차

- 4 이로 격파하면서 인공지능에 대한 관심이 급격히 증가하였다. 그와 동시에 기계
- 4 가 인간의 일자리 잠식을 가속화하면서 막연한 불안감이 삽시간에 전파되었다.
- 4 기계와의 일자리 경쟁은 컴퓨터의 출현이전부터 시작되었지만 인간만의 고유한
- 4 영역으로 알고 있던 인지, 창작 등 다양한 분야에서 오히려 인간보다 더 우수한
- 4 성능과 저렴한 가격 경쟁력을 보여주면서 기존 인간의 일자리가 기계에 대체되는
- 4 것이 가시권에 들었다. 이번 문헌조사와 실증 데이터 분석을 통해서 기계가 인
- 4 간의 일자리를 대체하는 자동화의 본질에 대해서 살펴보고, 인간과 기계의 업무
- □ 분장을 통해 더 생산성을 높일 수 있는 방안을 제시하고자 한다.\n\n 키워드:"

```
keywords = openai.Completion.create(
  model="text-davinci-003",
  prompt=prompt_keywords,
  temperature = 0.5,
  max_tokens = 100)
```

keywords['choices'][0]['text']

'인공지능, 자동화, 인간과 기계의 업무 분장, 생산성 \n\n이 문헌조사는 인공지능 이 인'

GPT가 생성한 키워드를 논문저자가 추출한 키워드와 비교하면 다소 차이가 있지만 그래도 상위 3개 키워드는 높은 일치도를 보이고 있다.

OpenAI GPT 키워드	논문저자 추출		
- - 인공지능	- 자동화		
- 자동화	- 데이터 과학		
- 인간과 기계의 업무 분장	- 인공지능		
- 생산성이 문헌조사는 인공지능이 인'	- 일자리		
	- 기계와 사람의 업무분장		

GPT-4는 더 높은 성능을 보여주고 있다. https://chat.openai.com/chat?model=gpt-4에 해당 텍스트를 던져주면 다음과 같이 키워드를 추출하고 요약을 해준다.



2.3 텍스트 요약

Attention Is All You Need 논문 초록은 https://platform.openai.com/tokenizer 계산기를 통해 230개 토큰 1,138 문자로 작성된 것이 확인된다. 영어 기준 다음과 같은 맥락을 이해하고 이를 대략 20% 수준 50 토큰으로 줄여보자.

- 100 토큰은 대략 75 단어
- 평균 단어는 대략 5 문자로 구성
- 100 토큰은 375개 문자

```
prompt_keywords = "The dominant sequence transduction models are based on

    complex recurrent or convolutional neural networks that include an encoder

   and a decoder. The best performing models also connect the encoder and
   decoder through an attention mechanism. We propose a new simple network
   architecture, the Transformer, based solely on attention mechanisms,
   dispensing with recurrence and convolutions entirely. Experiments on two
   machine translation tasks show these models to be superior in quality
   while being more parallelizable and requiring significantly less time to
   train. Our model achieves 28.4 BLEU on the WMT 2014 Englishto-German
   translation task, improving over the existing best results, including
   ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation
   task, our model establishes a new single-model state-of-the-art BLEU score
   of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the
  training costs of the best models from the literature. We show that the
   Transformer generalizes well to other tasks by applying it successfully to
   English constituency parsing both with large and limited training
   data.\n\n summary:"
keywords = openai.Completion.create(
 model="text-davinci-003",
 prompt=prompt_keywords,
 temperature = 0.5,
 max tokens = 50)
keywords['choices'][0]['text']
```

상기 논문 초록을 50개 토큰으로 요약하면 다음과 같이 48개 토큰, 277 문자수로 요약해 준다.

'\n\nThe Transformer is a new neural network architecture based solely on

- 4 attention mechanisms, which is shown to outperform existing models on two
- ← machine translation tasks. It is more parallelizable and requires
- □ significantly less time to train than existing models, achieving a B'

2.4 여론조사 할일 생성

TO-DO 리스트를 제작하는 것은 해당 작업을 절차적으로 구분지어 수행할 수 있게 되어 해당 작업의 성공가능성을 높이고 생산성도 높일 수 있다. 여론조사를 한사람이 수행하 는 경우는 거의 없지만 일반적으로 여론조사에서 수행할 일에 대해서 지시명령어를 작 성하여 결과를 살펴보자.

```
todo_list = openai.Completion.create(
 model="text-davinci-003",
 prompt="여론조사를 위해서 해야될 일을 작성하세요\n\n1.",
 temperature=0.3,
 max tokens = 1000,
 top_p = 0.1,
 frequency_penalty=0,
 presence_penalty=0.5,
 stop=["6."]
)
todo_text = todo_list['choices'][0]['text']
```

- ' 여론조사 대상자를 선정하고, 여론조사 대상자의 수를 결정합니다.\n\n2. 여론조
- 사 대상자들에게 여론조사 질문지를 배포합니다.\n\n3. 여론조사 대상자들에게
- 4 여론조사 응답을 요청합니다.\n\n4. 여론조사 응답을 수집하고, 분석합니다.
- ↔ \n\n5. 여론조사 결과를 보고서로 작성합니다.'

```
library(reticulate)
cat(glue::glue("1. {py$todo_text}"))
```

작업수행결과를 가독성 좋게 정리하면 다음과 같다.

- 1. 여론조사 대상자를 선정하고, 여론조사 대상자의 수를 결정합니다.
- 2. 여론조사 대상자들에게 여론조사 질문지를 배포합니다.
- 3. 여론조사 대상자들에게 여론조사 응답을 요청합니다.
- 4. 여론조사 응답을 수집하고, 분석합니다.
- 5. 여론조사 결과를 보고서로 작성합니다.

지금까지 OpenAI API를 사용하여 텍스트 자동생성기능을 활용하여 키워드 추출, 문서 요약, 작업목록 생성과 같은 업무를 통해 가능성을 살펴봤다. 이제 데이터 과학업무 생산성의 주요한 도구인 유닉스 쉘(Unix Shell)을 챗GPT로 또 다른 데이터 과학의 세계로 나아가자.

2.1 연습문제

- 1. 챗GPT를 사용하면 어떤 작업을 수행할 수 있을까요?
 - (1) 강화 학습 모델 훈련
 - (2) 음성 인식 시스템 구축
 - (3) 대화형 챗봇 개발
 - (4) 이미지 분류 알고리즘 개발
- 2. 다음 중 GPT-4를 사용하여 자연어 처리에 어떻게 활용할 수 있는지 가장 적합하지 않은 예는 무엇일까요?
 - (1) 감성 분석
 - (2) 기계 번역
 - (3) 음성을 텍스트로 변환
 - (4) 문장 생성
- 3. 챗GPT를 이용하여 불가능한 자연어 처리 작업은 무엇일까요?
 - (1) 텍스트 요약

- (2) 텍스트 기반의 질의 응답
- (3) 전문가 시스템을 위한 지식 표현
- (4) 실시간 비디오 분석
- 4. 챗GPT가 할 수 없는 자연어 처리 작업은 무엇일까요?
 - (1) 문장 교정
 - (2) 개체명 인식
 - (3) 독해 이해
 - (4) 특정 개인의 미래 행동 예측
- 5. 다음 중 챗GPT를 활용한 챗봇 개발이 아닌 작업은 무엇일까요?
 - (1) 고객 지원 서비스 개발
 - (2) 대화형 게임 스크립트 작성
 - (3) 자동 완성 기능 개발
 - (4) 이미지 인식 시스템 개발

제 II 편

쉘 프로그래밍



제 3 장

쉘(Shell) 소개

학습목표

명령 쉘(Command Shell)이란 무엇이며 왜 사용할까요? 라는 물음에 쉘이 키보드, 화면, 운영 체제 및 사용자 프로그램과 어떻게 연관되어 있는지 이해하고, 그래픽 인터페이스(GUI) 대신 명령줄 인터페이스(CLI)를 사용해야 하는 경우와그 이유를 학습한다. 더 나아가 챗GPT를 통해 새롭게 부상하고 있는 언어 사용자 인터페이스(NUI)도 적절히 활용하게 된다.

유닉스 쉘(Unix Shell)은 대부분의 컴퓨터 사용자가 살아온 것보다 오래 동안 존재했다. 오래동안 생존한 이유는 사용자로 하여금 단지 키보드 몇번 쳐서 복잡한 작업을 수행할 수 있게 하는 강력한 도구이기 때문이다. 좀더 중요하게는 기존의 프로그램을 새로운 방식으로 조합해서 반복적인 작업을 자동화함으로써, 동일한 작업을 반복적으로 하지 않게 만든다. 쉘 사용은 폭넓게 다양하고 강력한 도구와 컴퓨팅 자원(슈퍼컴퓨터와 "고성능 컴퓨팅(High Performance Computing, HPC)"이 포함)을 사용하는 근본이 된다.

3.1 배경

상위 수준에서 컴퓨터는 네가지 일을 수행한다:

- 프로그램 실행
- 데이터 저장
- 컴퓨터간 상호 의사소통
- 사람과 상호작용

마지막 작업을 뇌-컴퓨터 연결, 음성 인터페이스를 포함한 다양한 많은 방식으로 수행하고 있지만 아직은 초보적인 수준이어서, 대부분은 WIMP((Window) 윈도우, (Icon)아이콘, (Mouse)마우스, (Pointer)포인터)를 사용한다. 1980년대까지 이러한 기술은 보편적이지 않았지만, 기술의 뿌리는 1960년대 Doug Engelbart의 작업에 있고, "The Mother of All Demos"로 불리는 것에서 볼 수 있다.

조금 더 멀리 거슬러 올라가면, 초기 컴퓨터와 상호작용하는 유일한 방법은 와이어로 다시 연결하는 것이다. 하지만, 중간에 1950년에서 1980년 사이 대부분의 사람들이 라인 프린터(line printer)를 사용했다. 이런 장치는 표준 키보드에 있는 문자, 숫자, 특수부호의 입력과 출력만 허용해서, 프로그래밍 언어와 인터페이스는 이러한 제약사항에서 설계됐다.

여전히 전통적인 화면, 마우수, 터치패드, 키보드를 사용하지만 터치 인터페이스와 음성 인터페이스가 보편화되고 있다.

이런 종류의 인터페이스를 지금 대부분의 사람들이 사용하는 그래픽 사용자 인터페이스(GUI, graphical user interface)과 구별하기 위해서 명령-라인 인터페이스 (CLI, command-line interface)라고 한다. CLI의 핵심은 읽기-평가-출력(REPL,readevaluate-print loop)이다: 사용자가 명령어를 타이핑하고 엔터(enter)/반환(return)키를 입력하면, 컴퓨터가 읽고, 실행하고, 결과를 출력한다. 그러고 나면, 사용자는 다른 명령를 타이핑하는 것을 로그 오프해서 시스템을 빠져 나갈때까지 계속한다.

GUI는 WIMP((Window) 윈도우, (Icon)아이콘, (Mouse)마우스, (Pointer)포인터)로 구성되는데 배우기 쉽고, 단순 작업에 대해서는 환상적이다. "클릭"하게 되면 명령이 "내가 원하는 작업을 수행해"라고 손쉽게 컴퓨터에 통역된다. 하지만, 이런 마술은 단순한 작업을 수행하고, 정확하게 이러한 유형의 작업을 수행할 수 있는 프로그램에 불과하다.

만약 복잡하고, 특정 목적에 부합되는 훨씬 묵직한 작업을 컴퓨터에 내리고자 한다고 해서, 난해하거나 어렵거나할 필요는 없고, 단지 명령 어휘가 필요하고 이를 사용하는데 필요한 단순한 문법만 필요로 한다.

쉘이 이런 기능을 제공한다 - 단순한 언어로 이를 사용하는데 **명령-라인 인터페이스**가 필요하다. 명령라인 인터페이스의 심장은 **읽기-평가-출력(REPL,read-evaluate-print loop)**이다. REPL로 불리는 이유는 쉘에 명령어를 타이핑하고 Return를 치게되면 컴퓨터 가 명령어를 읽어들이고 나서, 평가(혹은 실행)하고 출력결과를 화면에 뿌린다. 또 다른 명령어를 입력할 때까지 대기하는 루푸를 반복하게 되서 그렇다.

상기 묘사가 마치 사용자가 직접 명령어를 컴퓨터에 보내고, 컴퓨터는 사용자에게 직접

적으로 출력을 보내는 것처럼 들린다. 사실 중간에 명령 쉘(command shell)로 불리는 프로그램이 있다. 사용자가 타이핑하는 것은 쉘로 간다. 쉘은 무슨 명령어를 수행할지 파악해서 컴퓨터에게 수행하도록 지시한다. 쉘을 조개 껍데기(shell)로 불리는데 이유는 운영체제를 감싸서, 복잡성 일부를 숨겨서 운영체제와 더 단순하게 상호작용하게 만든다.

3.2 쉘(Shell)

쉘(Shell)은 다른 것과 마찬가지로 프로그램이다. 조금 특별한 것은 자신이 연산을 수행하기 보다 다른 프로그램을 실행한다는 것이다. 가장 보편적인 유닉스 쉘(Unix Shell)은 Bash(Bourne Again SHell)다. Stephen Bourne이 작성한 쉘에서 나와서 그렇게 불리우고 — 프로그래머 사이에 재치로 통한다. Bash는 대부분의 유닉스 컴퓨터에 기본으로 장착되는 쉘이고, 윈도우용으로 유닉스스런 도구로 제공되는 패키지 대부분에도 적용된다.

Bash나 다른 쉘을 사용하는 것이 마우스를 사용하는 것보다 프로그래밍 작성하는 느낌이 난다. 명령어는 간략해서 (흔히 단지 2~3자리 문자다), 명령어는 자주 암호스럽고, 출력은 그래프같이 시각적인 것보다 텍스트줄로 쭉 뿌려진다. 다른 한편으로, 쉘을 사용하여 좀더 강력한 방식으로 현존하는 도구를 단지 키보드 입력값 몇개를 조합해서 대용량의 데이터를 자동적으로 처리할 수 있는 파이프라인을 구축할 수 있게 한다. 추가로, 명령 라인은 종종 멀리 떨어진 컴퓨터 혹은 슈퍼컴퓨터와 상호작용하는 가장 쉬운 방법이다. 고성능 컴퓨팅 시스템에 포함된 다양한 특화된 도구와 자원을 실행하는데 쉘과 친숙성이 거의 필연적이다. 클러스트 컴퓨팅과 클라우드 컴퓨팅이 과학 데이터 클런칭 (scientific data cruching)이 점점 대중화됨에 따라 원격 컴퓨터를 구동하는 것이 필수적인 기술이 되어가고 있다. 여기서 다뤄지는 명령-라인 기술에 기반해서 광범위한 과학적 질문과 컴퓨터적 도전과제를 처리할 수 있다.

3.3 어떻게 생겼을까?

전형적인 쉘 윈도우는 다음과 같다:

```
bash-3.2$
bash-3.2$ ls -F /
Applications/ System/
Library/ Users/
```

Network/

Volumes/

bash-3.2\$

첫번째 줄은 프롬프트(prompt)만 보여주고 있고, 쉘이 입력준비가 되었다는 것을 나타 낸다. 프롬프트로 다른 텍스트를 지정할 수도 있다. 가장 중요한 것: 명령어를 타이핑할 때, 프롬프트를 *타이핑하지* 말고, 인식되거나 수행할 수 있는 명령어만 타이핑한다.

예제 두번째 줄에서 타이핑한 ls -F / 부분이 전형적인 구조를 보여주고 있다: 명령 어(command), 플래그(flags) (선택옵션(options) 혹은 스위치(switches)) 그리고 인자 (argument). 플래그는 대쉬(-) 혹은 더블 대쉬(--)로 시작하는데 명령어의 행동에 변화를 준다.

인자는 명령어에 작업할 대상을 일러준다(예를 들어, 파일명과 디렉토리). 종종 플래그를 매개변수(parameter)라고도 부른다. 명령어를 플래그 한개 이상, 인자도 한개 이상 사용하기도 한다: 하지만, 명령어가 항상 인자 혹은 플래그를 요구하지는 않는다.

상기 예제의 두번째 줄에서, 명령어는 ls, 플래그는 -F, 인자는 /이 된다. 각각은 공백으로 뚜렸하게 구분된다: 만약 ls 와 -F 사이 공백을 빼먹게 되면 쉘은 ls-F 명령어를 찾게 되는데, 존재하지 않는 명령어다. 또한, 대문자도 문제가 될 수 있다: LS 명령어와 ls 명령어는 다르다.

다음으로 명령어가 생성한 출력결과를 살펴보자. 이번 경우에 / 폴더에 위치한 파일 목록을 출력하고 있다. 금일 해당 출력결과가 무엇을 의미하는지 다룰 예정이다. 맥OS를 사용하시는 참석자분들은 이번 출력결과를 이미 인지하고 있을지도 모른다.

마지막으로, 쉘은 프롬프트를 출력하고 다음 명령어가 타이핑되도록 대기모드로 바뀐다.

이번 학습예제에서 프롬프트가 \$이 된다. 명령어를 PS1='\$' 타이핑하게 되면 동일하게 프롬프트를 맞출 수 있다. 하지만, 본인 취향에 맞추어 프롬프트를 둘 수도 있다 - 흔히 프롬프트에 사용자명과 디렉토리 현재 위치정보를 포함하기도 하다.

쉘 윈도우를 열고, ls -F / 명령어를 직접 타이핑한다.(공백과 대문자가 중요함으로 잊지 말자.)원하는 경우 프롬프트도 변경해도 좋다.

3.4 ls 와 플래그 의미 파악

모든 쉘 명령어는 컴퓨터 어딘가에 저장된 프로그램으로, 쉘은 명령어를 검색해서 찾을 장소를 목록으로 이미 가지고 있다. (명령목록은 PATH로 불리는 변수(variable)에 기록되 어 있지만, 이 개념을 나중에 다룰 것이라 현재로서는 그다지 중요하지는 않다.) 명령어, 플래그, 인자가 공백으로 구분된다는 점을 다시 상기하자.

REPL(읽기-평가-출력(read-evaluate-print) 루프)를 좀더 살펴보자. "평가(evaluate)" 단계 는 두가지 부분으로 구성됨에 주목한다:

- 1. 타이핑한 것을 읽어들인다(이번 예제에서 ls -F /) 쉘은 공백을 사용해서 명령어로 입력된 것을 명령어, 플래그, 인자로 쪼갠다.
- 2. 평가(Evaluate):
 - a. ls 라는 프로그램을 찾는다.
 - b. 찾은 프로그램을 실행하고 프로그램이 인식하고 해석한 플래그와 인자를 전 달하다.
- 3. 프로그램 실행 결과를 출력한다.

그리고 나서, 프롬프트를 출력하고 또다른 명령어를 입력받도록 대기한다.

i Command not found 오류

쉘이 타이핑한 명령어 이름을 갖는 프로그램을 찾을 수 없는 경우, 다음과 같은 오 류 메시지가 출력된다:

\$ ls-F

-bash: ls-F: command not found

일반적으로 명령어를 잘못 타이핑했다는 의미가 된다 - 이 경우, ls 와 -F 사이 공 백을 빼먹어서 그렇다. 즉, ls -F와 같이 명령을 전달하면 의도한 바가 기계에 정확 히 전달된다.

3.5 어려운가요?

GUI와 비교하여 컴퓨터와 상호작용하는데 있어 어려운 모형이고 학습하는데 노력과 시간이 다소 소요된다. GUI는 선택지를 보여주고, 사용자가 선택지중에서 선택하는 하는 것이다. 명령라인 인터페이스(CLI)로 선택지가 명령어와 패러미터의 조합으로 표현된다. 사용자에게 제시되는 것이 아니라서 새로운 언어의 어휘를 학습하듯이 일부 학습이 필요하다. 명령어의 일부만 배우게 되면 정말 도움이 많이 되고, 핵심적인 명령어를 다뤄보자.

3.6 유연성과 자동화

쉘문법(Grammar of Shell)은 기존 도구를 조합해서 강력한 파이프라인을 구축하도록 해서 방대한 데이터를 자동화하여 다룰 수 있다. 명령 순서는 스크립트(script)로 작성하여 작업흐름의 재현가능성을 향상시켜서 쉽게 반복이 가능하도록 한다.

추가로, 명령 라인은 종종 멀리 떨어진 컴퓨터 혹은 슈퍼컴퓨터와 상호작용하는 가장 쉬운 방법이다. 고성능 컴퓨팅 시스템에 포함된 다양한 특화된 도구와 자원을 실행하는데 쉘과 친숙성이 거의 필연적이다. 클러스트 컴퓨팅과 클라우드 컴퓨팅이 과학 데이터 클런칭(scientific data cruching)이 점점 대중화됨에 따라 원격 컴퓨터를 구동하는 것이 필수적인 기술이 되어가고 있다. 여기서 다뤄지는 명령-라인 기술에 기반해서 광범위한 과학적 질문과 컴퓨터적 도전과제를 처리할 수 있다.

3.7 사례: 문제정의

해양 생물학자 넬 니모(Nell Nemo) 박사가 방금전 6개월간 북태평양 소용돌이꼴 조사를 마치고 방금 귀환했다. 태평양 거대 쓰레기 지대에서 젤리같은 해양생물을 표본주출했다. 총 합쳐서 1,520개 시료가 있고 다음 작업이 필요하다:

- 서로 다른 300개 단백질의 상대적인 함유량을 측정하는 분석기계로 시료를 시험한다.
 다. 한 시료에 대한 컴퓨터 출력결과는 각 단백질에 대해 한 줄 파일형식으로 표현된다.
- 2. goostat으로 명명된 그녀의 지도교수가 작성한 프로그램을 사용하여 각 단백질에 대한 통계량을 계산한다.
- 3. 다른 대학원 학생중 한명이 작성한 goodiff로 명명된 프로그램을 사용해서, 각 단

백질에 대한 통계량과 다른 단백질에 대해 상응하는 통계량을 비교한다.

4. 결과를 작성한다. 그녀의 지도교수는 이달 말까지 이 작업을 정말로 마무리해서, 논 문이 다음번 *Aquatic Goo Letters* 저널 특별판에 게재되기를 희망한다.

각 시료를 분석장비가 처리하는데 약 반시간 정도 소요된다. 좋은 소식은 각 시료를 준비하는데는 단지 2분만 소요된다. 연구실에 병렬로 사용할 수 있는 분석장비 8대가 있어서, 이 단계는 약 2주정도만 소요될 것이다.

나쁜 소식은 goostat, goodiff를 수작업으로 실행한다면, 파일이름 입력하고 "OK" 버튼을 45,150번 눌려야 된다는 사실이다 (goostat 300회 더하기 goodiff $\frac{300\times299}{2}$). 매번 30초씩 가정하면 2주 이상 소요될 것이다. 논문 마감일을 놓칠 수도 있지만, 이 모든 명령어를 올바르게 입력할 가능성은 거의 0에 가깝다.

다음 수업 몇개는 대신에 그녀가 무엇을 해야되는지 탐색한다. 좀더 구체적으로, 처리하는 파이프라인 중간에 반복되는 작업을 자동화하는데 쉘 명령어(command shell)를 어떻게 사용하는지 설명해서, 논문을 쓰는 동안에 컴퓨터가 하루에 24시간 작업한다. 덤으로 중간 처리작업 파이프라인을 완성하면, 더 많은 데이터를 얻을 때마다 다시 재사용할 수있게 된다.

추가적으로 일단 처리 파이프라인을 구축하면 더 많은 데이터를 수집할 때마다 재사용의 강력함을 경험할 수 있다. 파이프라인 구축 작업을 수행하기 위해 넬 박사는 다음과 같은 방법을 숙지해야 한다.

- 파일/디렉토리로 이동
- 파일/디렉토리 생성
- 파일 길이 확인
- 명령어를 조합하여 연결하기
- 파일셋 불러오기
- 파일별 반복작업
- 파이프라인을 담고 있는 쉘 스크립트(Shell Script) 실행

3.1 연습문제

- 1. 유닉스 쉘(Unix Shell)이 오랫동안 생존한 이유는 무엇인가요?
 - (1) 사용자로 하여금 키보드 몇 번 쳐서 복잡한 작업을 수행할 수 있게 만들기 때문

- (2) 고성능 컴퓨팅(High Performance Computing, HPC)를 사용할 수 있게 해주기 때문
- (3) 프로그램을 새로운 방식으로 조합하여 반복적인 작업을 자동화할 수 있게 해주기 때문
- (4) 상기 모두
- 2. 명령-라인 인터페이스(CLI)의 핵심 기능은 무엇인가요?
 - (1) 읽기-평가-출력(REPL,read-evaluate-print loop)
 - (2) WIMP((Window) 윈도우, (Icon)아이콘, (Mouse)마우스, (Pointer)포인터)
 - (3) GUI (Graphical User Interface)
 - (4) 컴퓨터와의 상호작용
- 3. 다음 중 ls 명령어에 대한 올바른 설명은 무엇인가요?
 - (1) ls는 리스트의 약자로, 디렉토리의 내용을 보여주는 명령어입니다.
 - (2) ls는 항상 현재 디렉토리의 내용만 보여줍니다.
 - (3) ls 명령어는 파일이나 디렉토리를 삭제하는 데 사용됩니다.
 - (4) ls 명령어는 실행할 수 있는 프로그램을 찾는 데 사용됩니다.
- 4. 쉘에 명령어를 입력하고 실행할 때의 순서를 올바르게 배열한 것은 무엇인가요?
 - (1) 결과 출력 -> 프로그램 찾기 -> 타이핑한 것을 읽기 -> 프로그램 실행
 - (2) 타이핑한 것을 읽기 -> 프로그램 찾기 -> 프로그램 실행 -> 결과 출력
 - (3) 프로그램 찾기 -> 프로그램 실행 -> 타이핑한 것을 읽기 -> 결과 출력
 - (4) 프로그램 실행 -> 결과 출력 -> 타이핑한 것을 읽기 -> 프로그램 찾기

제 4 장

파일과 폴더 넘나들기

학습목표

파일과 디렉토리(폴더)의 유사성과 차이점을 확인하고, 절대경로와 상대경로를 자유로이 변환시키고, 탭 자동완성과 명령어 선택옵션 인자를 달리한 작업 자동 화를 익힌다.

파일과 디렉토리 관리를 담당하고 있는 운영체제 부분을 **파일 시스템(file system)**이라고 한다. 파일 시스템은 데이터를 정보를 담고 있는 파일과 파일 혹은 다른 디렉토리를 담고 있는 디렉토리(혹은 "폴더"")로 조직화한다.

파일과 디렉토리를 생성, 검사, 이름 바꾸기, 삭제하는데 명령어 몇개가 자주 사용된다. 명령어를 살펴보기 위해, 쉘 윈도우를 연다:

먼저, pwd 명령어를 사용해서 위치를 찾아낸다; pwd는 "print working directory"를 의미한다. 디렉토리는 장소(place) 같다 - 쉘을 사용할 때마다 정확하게 한 장소에 위치하게 되는데, 이를 **현재 작업 디렉토리(current working directory)**라고 부른다. 명령어 대부분은 현재 작업 디렉토리에 파일을 읽고 쓰는 작업을 "이곳(here)"에 수행한다. 그래서 명령어를 실행하기 전에 현재 위치가 어디인지 파악하는 것이 중요하다. pwd 명령어를 숳애하게 되면 현재 위치를 다음과 같이 보여주게 된다:

\$ pwd

/Users/nelle

다음에서, 컴퓨터의 응답은 /Users/nelle으로 넬(Nelle)의 **홈 디렉토리(home directory)** 다:

⚠ 홈 디렉토리(Home Directory) 변종

홈 디렉토리 경로는 운영체제마다 다르게 보인다. 리눅스에서 /home/nelle 처럼 보이고, 윈도우에서는 C:\Documents and Settings\nelle, C:\Users\nelle와 유사하게 보인다. (윈도우 버젼마다 다소 차이가 있을 수 있음에 주목한다.) 다음 예제부터, 맥OS 출력결과를 기본설정으로 사용할 것이다; 리눅스와 윈도우 출력결과에 다소 차이가 날 수 있지만, 전반적으로 유사하다.

"홈 디렉토리(home directory)"를 이해하기 위해서, 파일 시스템이 전체적으로 어떻게 구성되었는지 살펴보자. 최상단에 다른 모든 것을 담고 있는 **루트 디렉토리(root directory)**가 있다. 슬래쉬 / 문자로 나타내고, /users/nelle에서 맨 앞에 슬래쉬이기도 하다.

Nelle 과학자 컴퓨터의 파일시스템을 사례로 살펴보자. 시연을 통해서 유사한 방식으로 (하지만 정확하게 동일하지는 않지만) 본인 컴퓨터 파일시스템을 탐색하는 명령어를 학습하게 된다.

넬 과학자 컴퓨터의 파일 시스템은 그림 4.1과 같이 도식적으로 표현할 수 있다:

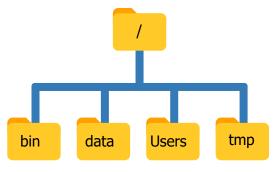


그림 4.1: "파일 시스템"

최상단에 다른 모든 것을 담고 있는 **루트 디렉토리(root directory)**가 있다. 슬래쉬 / 문자로 나타내고, /users/nelle에서 맨 앞에 슬래쉬이기도 하다.

홈 디렉토리 안쪽에 몇가지 다른 디렉토리가 있다: bin(몇몇 내장 프로그램이 저장된 디렉토리), data(여러가지 데이터 파일이 저장된 디렉토리), Users (사용자의 개인 디렉토리) 저장된 디렉토리), tmp(장기간 저장될 필요가 없는 임시 파일을 위한 디렉토리), 등등:

현재 작업 디렉토리 /Users/nelle는 /Users 내부에 저장되어 있다는 것을 알고 있는데,

이유는 /Users가 이름 처음 부분이기 때문에 알 수 있다. 마찬가지로 /Users는 루트 디렉 토리 내부에 저장되어 있다는 것을 알 수 있는데, 이름이 /으로 시작되기 때문이다.

♦ 슬래쉬(Slashes)

슬래쉬 / 문자는 두가지 의미가 있는 것에 주목한다. 파일 혹은 디렉토리 이름 앞에 나타날 때, 루트 디렉토리를 지칭하게 되고, 이름 *가운데* 나타날 때, 단순히 구분자 역할을 수행한다.

/Users 그림 4.2에서 Nelle 과학자 컴퓨터 계정과, 랩실 동료 미이라(Mummy)와 늑대인간 (Wolfman) 디렉토리를 볼 수 있다.

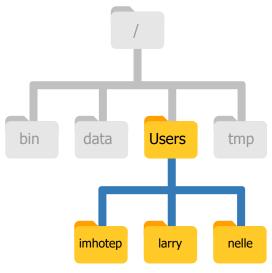


그림 4.2: "홈 디렉토리"

미이라(Mummy) 파일은 /Users/imhotep 디렉토리에 저장되어 있고, 늑대인가(Wolfman) 의 파일은 /Users/larry 디렉토리에 저장되어 있고 /Users/nelle 디렉토리에 nelle의 정보가 저장되어 있는데, 이것이 왜 nelle이 디렉토리 이름의 마지막 부분인 이유다. 일반적으로 명령 프롬프트를 열게 되면, 처음 시작하는 곳이 본인 계정 홈 디렉토리가 된다.

본인 파일시스템에 담긴 내용물을 파악하는데 사용하는 명령어를 학습해 보자. (Nelle의 홈 디렉토리에 무엇이 있는지 ls 명령어를 실행해서 살펴보자.) ls는 "목록보기(listing)"를 나타낸다:

\$ ls

Applications Documents Library Music Public Desktop Downloads Movies Pictures

(다시 한번, 본인 컴퓨터 운영체제와 파일시스템을 취향에 따라 바꿨는지에 따라 출력결과는 다소 다를 수 있다.)

ls는 알파벳 순서로 깔끔하게 열로 정렬하여 현재 디렉토리에 있는 파일과 디렉토리 이름을 출력한다. 플래그(flag) -F(스위치(switch) 혹은 옵션(option)으로도 불린다)를 추가하여 출력을 좀더 이해하기 좋게 출력괄를 생성할 수도 있다. ls으로 하여금 디렉토리이름 뒤에 /을 추가하게 일러준다: 끝에 붙은 /은 디렉토리라는 것을 지칭한다. 설정에따라 달라지도록 파일이냐 디렉토리냐에 따라 다른 색상을 입힐 수도 있다. 앞선 학습에서 ls -F 명령어를 사용한 것을 상기한다.

\$ ls -F

Applications/ Documents/ Library/ Music/ Public/
Desktop/ Downloads/ Movies/ Pictures/

4.1 도움말 얻기

ls 명령어에 딸린 **플래그**가 많다. 일반적으로 명령어와 수반되는 플래그 사용법을 파악하는 방식이 두개 있다:

1. --help 플래그를 명령어에 다음과 같이 전달하는 방법:

\$ ls --help

2. man 명령어로 다음과 같이 매뉴얼을 읽는 방법:

\$ man ls

본인 컴퓨터 환경에 따라 상기 방법 중 하나만 동작(man 혹은 --help)할 수도 있다. 아래에서 두가지 방법 모두 살펴보자.

4.1.1 --help 플래그

배쉬 내부에서 동작하도록 작성된 배쉬 명령어와 프로그램은 --help 플래그를 지원해서 명령어 혹은 프로그램을 사용하는 방식에 대한 더 많은 정보를 볼 수 있게 해 준다.

```
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
Mandatory arguments to long options are mandatory for short options too.
  -a, --all
                            do not ignore entries starting with .
 -A, --almost-all
                            do not list implied . and ..
                           with -l, print the author of each file
     --author
 -b, --escape
                             print C-style escapes for nongraphic characters
                           scale sizes by SIZE before printing them; e.g.,
     --block-size=SIZE
                               '--block-size=M' prints sizes in units of
                               1,048,576 bytes; see SIZE format below
 -B, --ignore-backups
                             do not list implied entries ending with \sim
                             with -lt: sort by, and show, ctime (time of last
  -c
                               modification of file status information);
                               with -l: show ctime and sort by name;
                               otherwise: sort by ctime, newest first
... 중략
  -X
                             sort alphabetically by entry extension
 -Z, --context
                             print any security context of each file
                             list one file per line. Avoid '\n' with -q or -b
  -1
                display this help and exit
      --help
      --version output version information and exit
```

```
The SIZE argument is an integer and optional unit (example: 10K is 10*1024).

Units are K,M,G,T,P,E,Z,Y (powers of 1024) or KB,MB,... (powers of 1000).

Using color to distinguish file types is disabled both by default and with --color=never. With --color=auto, ls emits color codes only when standard output is connected to a terminal. The LS_COLORS environment variable can change the settings. Use the dircolors command to set it.

Exit status:

0  if OK,

1  if minor problems (e.g., cannot access subdirectory),

2  if serious trouble (e.g., cannot access command-line argument).

GNU coreutils online help: <a href="http://www.gnu.org/software/coreutils/">http://www.gnu.org/software/coreutils/</a>>

or available locally via: info '(coreutils) ls invocation'
```

♦ 지원되지 않는 명령-라인 선택옵션

지원되지 않는 선택옵션(플래그)를 사용하게 되면, ls를 비롯한 다른 프로그램은 다음과 같은 오류 메시지를 일반적으로 출력하게 된다:

```
$ ls -j
ls: invalid option -- 'j'
Try 'ls --help' for more information.
```

4.1.2 man 명령어

ls에 대해 배울 수 있는 다른 방식은 다음 명령어를 타이핑하는 것이다.

\$ man ls

상기 명령어를 실행하게 되면 ls 명령어와 선택 옵션에 대해 기술된 페이지로 탈바꿈하

게 된다. 만약 운이 좋은 경우 상용법에 대한 예제도 포함되어 있다.

man 페이지를 살펴보는 방법은 행단위로 이동하는데 ↑, ↓을 사용하거나 전체 페이지 단 위로 건너뛰거나 아래 페이지로 이동할 경우 B, Spacebar을 사용한다. man 페이지에서 단 어나 문자를 찾는 경우 / 다음에 검색할 문자 혹은 단어를 타이핑하면 된다.

man 페이지에서 빠져 나오고자 종료(quit)하고자 한다면 Q을 누른다.

♦ 주의

웹상의 매뉴얼 페이지

물론 명령어에 대한 도움말에 접근하는 세번째 방식이 있다: 웹브라우저를 통해서 인터넷을 검색하는 것이다. 인터넷 검색을 이용할 때, 검색쿼리에 unix man page 문 구를 포함할 경우 연관된 정보를 찾는데 도움이 될 수 있다.

GNU도 GNU 핵심 유틸리티(core GNU utilities)이 포함된 매뉴얼을 제공하고 있는데 이번 학습에 소개된 많은 명령어를 망라하고 있다.

♦ 더많은 ls 플래그 탐색

-1, -h 플래그를 붙여 1s 명령어를 수행하게 되면 출력결과는 어떻게 나올까? 출력결과의 일부는 이번 학습에서 다루지 않는 속성(property)에 대한 것으로 파일 권한과 파일 소유에 대한 것이다. 그럼에도 불구하고 나머지는 유용할 것이다.

ls와 사용되는 -l 플래그는 long을 축약한 것으로 파일/디렉토리 명칭 뿐만 아니라 파일 크기, 최종 변경 시간 같은 부가정보가 출력된다. -h 플래그는 "human readable" 사람이 읽기 편한 형태로 파일크기를 지정 한다. 예를 들어, 5369 대신에 5.3K이 화면에 출력된다.

♦ 재귀적으로 시간순으로 목록 출력

ls -R 명령어는 디렉토리에 담긴 내용을 재귀적으로 화면에 출력한다; 즉, 각 단계 별로 하위 디렉토리, 하위-하위 디렉토리 내용을 확면에 출력한다. ls -t 명령어는 마지막 변경된 시점순으로 가장 최근에 변경된 파일 혹은 디렉토리를 화면에 정렬해서 출력한다. ls -r 명령어는 어떤 순서로 화면엘 출력할까?

힌트: ls -l 명령어를 사용해서 시간도장(timestamp)을 볼 수 있도록 전체 목록을 화면에 출력한다.

각 디렉토리의 파일/디렉토리가 가장 마지막 시간 변경순으로 정렬되

어 출력된다.

여기서 홈 디렉토리가 **하위 디렉토리(sub-directories)**가 포함된것을 알 수 있다. 슬래쉬(/)가 붙지 않는 명칭을 갖는 것은 것은 평범한 **파일(file)**이다. ls 와 -F 사이에 공백이 있는 것에 주목한다: 공백이 없다면 쉘은 존재하지 않는 ls-F 명령어를 실행시키려 한다고 간주한다.

ls 명령어를 사용해서 다른 디렉토리에 들어 있는 파일과 디렉토리를 살펴볼 수 있다. ls -F Desktop 명령어를 실행해서 바탕화면 Desktop 디렉토리에 담긴 것을 살펴보자. 즉, ls 명령어는 -F 플래그, 그리고 인자(argument) Desktop으로 구성된다. Desktop 인자는 ls로 하여금 현재 작업 디렉토리가 아닌 바탕화면 디렉토리 내용을 출력하도록 지정하는 역할을 수행한다:

\$ ls -F Desktop
shell-lesson-data/

작업한 출력결과는 웹사이트에서 다운로드 받아 압축을 풀어 작업하여 생성한 shell-lesson-data 디렉토리와 본인 바탕화면에 저장된 모든 파일과 하위디렉토리가 출력되어 야 한다.

4.2 cd 디렉토리 변경

지금 확인했듯이, 배쉬 쉘은 파일을 계층적 파일 시스템으로 구성한다는 아이디어에 강력히 의존하고 있다. 이런 방식으로 계층적으로 파일과 디렉토리를 구조화하게 되면 본인 작업을 추적하는데 도움이 된다: 책상위에 출력한 논문 수백개를 쌓아놓은 것는 것이가능하듯이, 홈 디렉토리에 파일 수백개를 저장하는 것도 가능하다. 하지만, 이런 접근법은 자멸하는 전략이나 마찬가지다.

shell-lesson-data 디렉토리가 바탕화면(Desktop)에 위치하는 것을 확인했으니, 다음 두 가지를 수행할 수 있다.

먼저, shell-lesson-data 디렉토리에 담긴 것을 살펴보자; 디렉토리 이름에 ls를 전달해서 앞서 확인된 동일한 전략을 사용하자:

\$ ls -F Desktop/shell-lesson-data

creatures/ molecules/ notes.txt solar.pdf data/ north-pacific-gyre/ pizza.cfg writing/

둘째로, 다른 디렉토리로 위치를 실제로 바꿀 수 있다. 그렇게 하면 더이상 홈 디렉토리에 있지는 않게 된다.

작업 디렉토리를 변경하기 위해서 cd 다음에 디렉토리 이름을 사용한다. cd는 "change directory"의 두문어다. 하지만 약간 오해의 소지가 있다: 명령어 자체가 디렉토리를 변경하지는 않고, 단지 사용자가 어느 디렉토리에 있는지에 대한 쉘의 생각만 바꾼다.

앞서 확인한 exercise-data 디렉토리로 이동해 보자. 다음 명령어를 쭉 이어서 실행하게 되면 목적지에 도달할 수 있다:

- \$ cd Desktop
- \$ cd shell-lesson-data
- \$ cd exercise-data

상기 명령어는 홈 디렉토리에 바탕화면(Desktop) 디렉토리로 이동하고 나서, shell-lesson-data 디렉토리로 이동하고 나서, exercise-data 디렉토리에 이동하게 된다. cd 명령어는 아무것도 출력하지는 않지만, pwd 명령어를 실행하게 되면 /Users/nelle/Desktop/shell-lesson-data/exercise-data 위치한 것을 확인하게 된다. 인자 없이 ls 명령어를 실행하게 되면, /Users/nelle/Desktop/shell-lesson-data/data 디렉토리 파일과 디렉토리를 출력하게 되는데 이유는 지금 있는 위치이기 때문이다:

\$ pwd

/Users/nelle/Desktop/shell-lesson-data/exercise-data

\$ ls -F

animal-counts/ creatures/ numbers.txt alkanes/ writing/

이제 디렉토리 나무를 타서 아래로 내려가는 방법을 익혔다. 하지만 어떻게 하면 위로 올라갈 수 있을까? 다음 명령어를 시도해보자:

\$ cd shell-lesson-data

-bash: cd: shell-lesson-data: No such file or directory

하지만, 오류 발생! 이유가 뭘까?

지금까지 방법으로 cd 명령어는 현재 디렉토리 내부에 하위 디렉토리만 볼 수 있다. 현재 디렉토리에서 상위 디렉토리를 볼 수 있는 다른 방법이 있다; 가장 단순한 것부터 시작해보자.

쉘에서 한단계 위 디렉토리로 이동할 수 있는 단축키가 존재하는데 다음과 같이 생겼다:

\$ cd ..

..은 특별한 디렉토리명인데 "현재 디렉토리를 포함하는 디렉토리", 좀더 간결하게 표현하면 현재 디렉토리의 부모를 의미한다. 물론, cd .. 명령어를 실행하고 나서 pwd을 실행하게 되면 /Users/nelle/Desktop/shell-lesson-data로 되돌아 간다:

\$ pwd

/Users/nelle/Desktop/shell-lesson-data

단순히 ls 명령어를 실행하게 되면 특수 디렉토리 ..이 화면에 출력되지는 않는다... 디렉토리를 출력하려면 ls 명령어와 -a 플래그를 사용한다:

\$ ls -F -a

```
./ .bash_profile data/ north-pacific-gyre/ pizza.cfg thesis/
../ creatures/ molecules/ notes.txt solar.pdf writing/
```

-a은 "show all"의 축약으로 모두 보여주기를 의미한다; ls로 하여금 ..와 같은 .로 시작하는 파일과 디렉토리명도 화면에 출력하게 강제한다. (/Users/nelle 디렉토리에 위치한다면, /Users 디렉토리를 지칭) .도 또다른 특별한 디렉토리로, "현재 작업 디렉토리 (current working directory)"를 의미한다. 중복되어 불필요해 보일 수 있지만, 곧 .에 대한사용법을 학습할 것이다.

대부분의 명령라인 도구에서 플래그 다수룰 조합해서 플래그 사이 공백없이 단일 -로 사용함에 주목한다: ls -F -a은 ls -Fa와 동일하다.

♦ 다른 숨은 파일들

숨은 .., . 디렉토리에 더해서, .bash_profile 파일도 봤을 것이다. .bash_profile 파일에는 쉘 환경설정 정보가 담겨져 있다. .으로 시작하는 다른 파일과 디렉토리를 봤을 수도 있다. 이런 파일은 본인 컴퓨터의 다른 프로그램에서 환경설정을 하기 위해서 사용되는 파일과 디렉토리라고 보면 된다. . 접두어를 사용해서 ls 명령어를 사용할 때 이러한 환경설정 파일들이 터미널을 난잡하게 만드는 것을 방지하는 기능을 수행한다.

♦ 직교(Orthogonality)

특수 이름 .과 ..는 1s에만 속하는 것이 아니고; 모든 프로그램에서 같은 방식으로 해석된다. 예를 들어, /Users/nelle/data 디렉토리에 있을 때, 1s .. 명령어는 /Users/nelle의 목록을 보여줄 것이다. 어떻게 조합되든 상관없이 동일한 의미를 가지게 될 때, 프로그래머는 이를 **직교(orthogonal)**한다고 부른다. 직교 시스템은 사람들이 훨씬 배우기 쉬운데, 이유는 기억하고 추적할 특수 사례와 예외가 더 적기 때문이다.

4.3 상대/절대 경로

컴퓨터에 파일시스템을 돌아다니는데 기본 명령어는 pwd, ls, cd을 들 수 있다. 지금까지 사용했던 했던 방식을 벗어난 사례를 살펴보자. 프롬프트에서 cd 명령어를 디렉토리를 특정하지 않고 실행시키면 어떻게 될까?

\$ cd

상기 명령어 실행 결과를 어떻게 확인할 수 있을까? pwd 명령어가 정답을 제시하다!

\$ pwd

/Users/nelle

어떤 플래그도 없는 cd 명령어는 홈디렉토리로 이동시킨다. 파일시스템에서 방향을 잃었을 경우 큰 도움이 된다.

data 디렉토리로 되돌아가자. 앞서 명령어 세개를 동원했지만 한방에 해당 디렉토리를

명세해서 바로 이동할 수 있다.

\$ cd Desktop/shell-lesson-data/data

pwd 와 ls -F 명령어를 실행해서 올바른 자리로 돌아왔는지 확인하자. data 디렉토리에서 한단계 위로 올라가려고 하면 cd .. 명령어를 사용했다. 현재 디렉토리 위치에 관계없이 특정 디렉토리로 이동할 수 있는 다른 방식도 있다.

지금까지 디렉토리명을 명세할 때 **상대경로(relative paths)**를 사용했다. ls 혹은 cd와 같은 명령어와 상대 경로를 사용할 때는 시스템이 파일시스템의 루트 위치(/)에서 차근차근 찾기보다 해당 위치를 현재 위치를 찾아 명령을 실행시킨다.

하지만, / 슬래쉬로 표현되는 루트 디렉토리에서 전체 경로를 추가한 **절대경로(absolute path)**로 명세하는 것도 가능하다. / 슬래쉬는 컴퓨터가 루트 디렉토리에서 경로를 탐색하도록 지시한다. 따라서, 명령어를 실행할 때 현재 디렉토리 위치에 관계없이 정확한 특정 디렉토리를 항상 명세하게 된다.

절대경로를 사용하면 파일 시스템에 어느 위치에서든 있던 관계없이 data-shell 디렉토리로 이동할 수 있다. 절대경로를 찾기 쉬운 방법은 pwd 명령어를 사용해서 필요한 디렉토리 정보를 추출하고 이를 활용해서 data-shell 디렉토리로 이동한다.

\$ pwd

/Users/nelle/Desktop/shell-lesson-data/data

\$ cd /Users/nelle/Desktop/shell-lesson-data

pwd와 ls -F 명령어를 실행하게 되면 원하던 디렉토로리 제대로 이동되었는지 확인이 가능하다.

예제 4.1 (절대 경로 vs. 상대 경로). 동일 작업을 수행하는 수많은 방법을 연습문제를 통해 확인해보자. /home/amanda/data/ 디렉토리에서 시작할 때, Amanda가 홈디렉토리인 /home/amanda로 돌아가도록 사용할 수 있는 명령어를 아래에서 선택하시요.

- 1. cd.
- 2. cd /
- 3. cd /home/amanda

- 4. cd ../..
- 5. cd
- 6. cd home
- 7. cd /data/..
- 8. cd
- 9. cd..

♦ 단축(Shortcuts) 두개 더

쉘을 ~ (틸드) 문자를 경로의 시작으로 해석해서 "현재 사용자 홈 디렉토리"를 의미 하게 된다. 예를 들어, Nelle의 홈 디렉토리가 /Users/nelle이라면, ~/data은 /Users/ nelle/data와 동치가 된다. 경로명에 첫 문자로 있을 때만 이것이 동작한다: here/ there/~/elsewhere이 here/there/Users/nelle/elsewhere이 되는 것은 아니다. 따라 서, cd ~을 홈 디렉토리로 변경하는데 사용한다.

또 다룩 단축은 대쉬(-) 문자다. cd는 - 문자를 지금 있는 이전 디렉토리로 변역한 다. 이 방법이 전체 경로를 기억하고 있다가 타이핑하는 것보다 더 빠르다. 이를 통 해 디렉토리를 앞뒤로 *매우* 효율적으로 이동하게 된다. cd .. 와 cd - 명령어 사이 차이점은 전자(cd ..)는 위로, 후자(cd -)는 아래로 이동하게 위치를 바꾸는 역할을 수행한다. TV 리모컨의 이전 채널 기능으로 생각하면 편하다.

실습해봅시다! ~/Desktop/shell-lesson-data로 이동한다 (이미 이 위치에 있을 것이 다).

\$ cd ~/Desktop/shell-lesson-data

그런 다음 cd 명령어로 exercise-data/creatures 디렉터리로 이동한다.

cd exercise-data/creatures

cd - 을 실행하면 ~/Desktop/shell-lesson-data로 돌아간 것을 볼 수 있습니다. cd -를 다시 실행하면 ~/Desktop/shell-lesson-data/exercise-data/creatures로 되돌아 간다.

4.4 쉘 명령 일반구문

지금까지 명령어, 선택옵션 및 명령인자를 살펴보았지만 몇 가지 용어를 공식화하는 것이 필요하다. 아래 쉘 명령어를 일반적인 쉘 명령어 사례로 간주하고 구성요소별로 분해하면 다음과 같다:

ls -F /

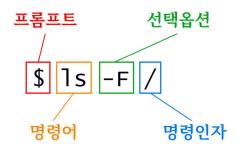


그림 4.3: 전형적인 쉘 프롬프트

그림 4.3 에서 제시된 쉘 명령어는 1s 명령어, 선택옵션 -F, 명령인자 /로 구성되어 있다. 이미 대시(-) 하나 또는 대시 두개(--)가 명령의 동작을 변경한다. 명령인자는 명령에 작업할 대상(예: 파일 및 디렉터리)을 지정해준다. 선택옵션과 명령인자를 매개변수라고 부른다. 쉘 명령은 둘 이상의 선택옵션과 둘 이상의 명령인자를 사용하여 실행할 수도 있지만, 쉘 명령어 작성에 항상 선택옵션과 명령인자가 필요한 것은 아니다.

선택옵션을 스위치(switches) 또는 플래그(flag)라고 부르는 경우도 있다. 특히 명령인 자가 없는 선택옵션을 스위치나 플래그로 부르지만, 선택옵션이라는 용어를 통일해서 사용한다.

쉘 명령어 각 부분은 공백으로 구분된다. ls와 -F 사이에 공백을 생략하는 경우 공백을 생략하면 쉘은 ls-F라는 명령을 찾지만 이런 명령어는 존재하지 않는다. 쉘 명령어에서 대소문자 구분도 중요하다. 예를 들어 ls -s는 파일 및 디렉터리의 크기를 이름과 함께 표시하지반, ls -S는 아래와 같이 파일과 디렉터리를 크기별로 정렬한다:

cd shell-lesson-data
ls -s exercise-data

total 28

4 animal-counts 4 creatures 12 numbers.txt 4 proteins 4 writing

ls -s가 반환하는 크기는 블록 단위로 운영 체제마다 다르게 정의되어 있으므로 예제와 동일한 수치를 얻지 못할 수도 있다. 참고로 윈도우 10 운영체제에서 작성되었다.

ls -S는 파일과 디렉토리를 크기별로 정렬한다.

```
$ ls -S exercise-data
animal-counts creatures proteins writing numbers.txt
```

이 모든 것을 종합하면 위의 ls -F / 명령은 루트 디렉터리 /에 있는 파일 및 디렉터리 목록을 출력한다. 참고로 윈도우 10 환경에 WSL 우분투 리눅스를 환경 사례다. 사용자의 운영체제에 따라 달리 나올 수도 있다.

```
$ ls -F /
```

Applications/ System/
Library/ Users/
Network/ Volumes/

예제 4.2 (상대경로 해결). 만약 pwd 명령어를 쳤을 때, 화면에 /Users/thing이 출력된다면, ls -F ../backup은 무엇을 출력할까요?

- 1. ../backup: No such file or directory
- 2. 2012-12-01 2013-01-08 2013-01-27
- 3. 2012-12-01/2013-01-08/2013-01-27/
- 4. original/pnas_final/pnas_sub/

예제 4.3 (ls 독해 능력). 그림 4.4 (도전과제 질문에 사용되는 파일 시스템)에 나온 디렉 토리 구조를 상정한다. 만약 pwd 명령어를 쳤을 때 화면에 /Users/backup이 출력되고, -r 인자는 ls 명령어가 역순으로 화면에 출력하게 한다면, 어떤 명령어가 다음을 화면에 출력할까요?

- 1. ls pwd
- 2. ls -r -F
- 3. ls -r -F /Users/backup
- 4. 위 2혹은 3, 하지만, 1은 아님.

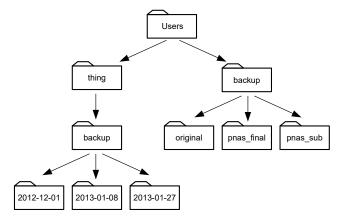


그림 4.4: 연습문제 제시된 파일 시스템

4.5 사례: 파일 구성

파일과 디렉토리에 대해서 알았으니, Nelle은 단백질 분석기가 생성하는 파일을 구성할 준비를 마쳤다. 우선 north-pacific-gyre 디렉토리를 생성해서 데이터가 어디에서 왔는 지를 상기하도록 한다. 2012-07-03 디렉토리를 생성해서 시료 처리를 시작한 날짜를 명기했다. Nelle은 conference-paper와 revised-results같은 이름을 사용하곤 했다. 하지만, 몇년이 지난 후에 이해하기 어렵다는 것을 발견했다. (마지막 지푸라기는 revised-results-3 디렉토리를 본인이 생성했다는 것을 발견했을 때였다.)

♦ 출력결과 정렬

Nelle은 월과 일에 0을 앞에 붙여 디렉토리를 "년-월-일(year-month-day)" 방식으로 이름지었다. 왜냐하면 쉘이 알파벳 순으로 파일과 디렉토리 이름을 화면에 출력하기 때문이다. 만약 월이름을 사용한다면, 12월(December)이 7월(July) 앞에 위치할 것이다: 만약 앞에 0을 붙이지 않으면 11월이 7월 앞에 올 것이다.

각각의 물리적 시료는 "NENE01729A"처럼 10자리 중복되지 않는 ID로 연구실 관례에 따라 표식을 붙였다. 시료의 장소, 시간, 깊이, 그리고 다른 특징을 기록하기 위해서 수집 기록에 사용된 것과 동일하다. 그래서 이를 각 파일 이름으로 사용하기로 결정했다. 분석

기 출력값이 텍스트 형식이기 때문에 NENE01729A.txt, NENE01812A.txt, … 같이 확장자를 붙였다. 총 1,520개 파일 모두 동일한 디렉토리에 저장되었다.

이제 data-shell 현재 작업 디렉토리에서 Nelle은 다음 명령어를 사용해서, 무슨 파일이 있는지 확인할 수 있다:

\$ ls north-pacific-gyre/2012-07-03/

엄청나게 많은 타이핑이지만 **탭 자동완성(tab completion)**을 통해 쉘에게 많은 일을 시 킬 수도 있다. 만약 다음과 같이 타이핑하고:

\$ ls nor

그리고 나서 탭(키보드에 탭 키)을 누르면, 자동으로 쉘이 디렉토리 이름을 자동완성 시 켜준다:

\$ ls north-pacific-gyre/

탭 Tab 을 다시 누르면, Bash가 명령문에 2012-07-03/을 추가하는데, 왜냐하면 유일하게 가능한 자동완성조건이기 때문이다. 한번더 탭을 누려면 아무것도 수행하지 않는다. 왜 냐하면 1520가지 경우의 수가 있기 때문이다; 탭을 두번 누르면 모든 파일 목록을 가져 온다. 이것을 탭 자동완성(tab completion)이라고 부르고, 앞으로도 다른 많은 툴에서도 많이 볼 것이다.

그런 다음 Nelle이 G를 누른 다음 Tab 키를 다시 누르면 'g'로 시작하는 모든 파일이 처음 세 글자인 'goo'를 공유하기 때문에 쉘이 자동으로 'goo'를 추가한다.

\$ ls north-pacific-gyre/goo

모든 파일을 보려면 Tab 키를 두 번 더 누르면 된다.

```
$ ls north-pacific-gyre/goo
goodiff.sh goostats.sh
```

4.1 연습문제

- 1. 현재 'projects'라는 디렉토리에 있고, 파일 크기를 사람이 읽기 쉬운 형식으로 긴 목록 형식으로 내용을 표시하려고 합니다. 이를 수행하는 명령어를 작성하십시오.
 - (1) ls -lh
 - (2) ls -l -h
 - (3) ls -hl
 - (4) ls -h -l
 - (5) 위의 모든 선택사항
- 2. 파일로 가득 찬 디렉토리가 있고, 마지막 변경 시간에 따라 디렉토리의 내용을 나열하려고 합니다. 가장 최근에 변경된 파일이 마지막에 표시되도록 해야 합니다. 어떤 명령 어를 사용해야 할까요?
 - (1) ls -rt
 - (2) ls -tr
 - (3) ls -r -t
 - (4) ls -t -r
 - (5) 위의 모든 선택사항
- 3. 현재 /home/user/projects 디렉토리에 있다고 가정합니다. 현재 디렉토리를 /home/user/projects/subdir로 변경하려고 합니다. 이를 수행하는 명령어는 무엇인 가요?
 - (1) cd subdir
 - (2) cd /home/user/projects/subdir
 - (3) cd /projects/subdir
 - (4) cd./subdir
 - (5) 위의 모든 선택사항
- 4. 현재 /home/user/projects/subdir 디렉토리에 있다고 가정합니다. 이전에 있던 디렉토리 /home/user/projects로 돌아가려고 합니다. 어떤 명령어를 사용해야 할까요?
 - (1) cd ..
 - (2) cd -
 - (3) cd /projects

- (4) cd /home/user/projects
- (5) cd ../..
- 5. 'ls -l' 명령어와 'ls -lh' 명령어의 차이점은 무엇인가요?
- 6. 'cd ..' 명령어와 'cd -' 명령어의 차이점은 무엇인가요?

제 5 장

파일과 디렉토리 작업

이제는 어떻게 파일과 디렉토리를 살펴보는지 알게 되었지만, 우선, 어떻게 파일과 디렉 토리를 생성할 수 있을까요? 바탕화면(Desktop) data-shell 디렉토리로 돌아가서 ls -F 명령어를 사용하여 무엇을 담고 있는지 살펴봅시다:

\$ pwd

/Users/nelle/Desktop/data-shell

\$ ls -F

creatures/ data/ molecules/ north-pacific-gyre/ notes.txt pizza.cfg
 solar.pdf writing/

명령어 mkdir thesis을 사용하여 새 디렉토리 thesis를 생성합시다 (출력되는 것은 아무 것도 없습니다.):

\$ mkdir thesis

이름에서 유추를 할 수도, 하지 못할 수도 있지만, mkdir은 "make directory(디렉토리 생성하기)"를 의미한다. thesis는 상대 경로여서(즉, 앞에 슬래쉬가 없음), 새로운 디렉토리는 현재 작업 디렉토리 아래 만들어진다:

\$ ls -F

creatures/ data/ molecules/ north-pacific-gyre/ notes.txt pizza.cfg
 solar.pdf thesis/ writing/

🌢 주의

동일한 작업을 수행하는 두가지 방법

쉘을 사용해서 디렉토리를 생성하는 것이나 파일 탐색기를 사용하는 것과 별반 차이가 없다. 운영체제 그래픽 파일 탐색기를 사용해서 현재 디렉토리를 열게 되면, thesis 디렉토리가 마찬가지로 나타난다. 파일과 상호작용하는 두가지 다른 방식이 존재하지만, 파일과 디렉토리는 동일하다.

♦ 파일과 디렉토리를 위한 좋은 명칭

명령라인으로 작업할 때, 복잡하고 어려운 파일과 디렉토리는 삶을 질을 현격히 저하시킨다. 다음에 파일 명칭에 대한 유용한 팁이 몇개 있다.

- 1. 공백(whitespaces)을 사용하지 마라 공백은 이름을 의미있게 할 수도 있지만, 공백이 명령라인 인터페이스에서 인자를 구별하는데 사용되기에, 파일과 디렉토리 명에서는 피하는 것이 상책이다. 공백 대신에 혹은 _ 문자를 사용한다.
- 2. 대쉬(-)로 명칭을 시작하지 않는다. 명령어가 -으로 시작되는 명칭을 선택옵 션으로 처리하기 때문이다.
- 3. 명칭에 문자, 숫자, . (마침표), (대쉬) and _ (밑줄)을 고수한다. 명령라인 인터 페이스에서 다른 많은 문자는 특별한 의미를 갖는다. 학습을 진행하면서 이들 중 일부를 배울 것이다. 일부 특수 문자는 명령어가 기대했던 대로 동작하지 못하게 하거나, 심한 경우 데이터 유실을 야기할 수도 있다.

공백을 포함하거나 알파벳이 아닌 문자를 갖는 파일명이나 디렉토리명을 굳이 지정할 필요가 있다면, 인용부호("")로 파일명이나 디렉토리명을 감싸야 한다.

thesis 디렉토리를 방금 생성했기에 내부에는 아무것도 없다:

\$ ls -F thesis

cd 명령어를 사용하여 thesis로 작업 디렉토리를 변경하자. Nano 텍스트 편집기를 실행

해서 draft.txt 파일을 생성하자:

- \$ cd thesis
- \$ nano draft.txt

♦ 주의

어떤 편집기가 좋을까요?

"nano가 텍스트 편집기다"라고 말할 때, 정말 "텍스트"만 의미한다. 즉, 일반 문자 데이터만 작업할 수 있고, 표, 이미지, 혹은 다른 형태의 인간 친화적 미디어는 작업할수 없다. nano를 워크샵에서 사용하는데 이유는 거의 누구나 훈련없이 사용할수있기 때문이다. 하지만, 실제 작업에는 좀더 강력한 편집기 사용을 추천한다. 유닉스 시스템 계열(맥 OS X, 리눅스)에서 많은 프로그래머는 Emacs 혹은 Vim을 사용하거나, (둘다 완전히 비직관적이만, 심지어 유닉스 표준이기도 하다) 혹은 그래픽편집기로 Gedit를 사용한다. 윈도우에서는 Notepad++를 사용하는 것도 좋다. 윈도우에는 메모장(notepad)이라고 불리는 자체 내장 편집기도 있는데 nano 편집기와마찬가지로 명령라인에서 바로 불리 실행될 수 있다.

어떤 편집기를 사용하는, 파일을 검색하고 저장하는 것을 알 필요가 있다. 쉘에서 편집기를 시작하면, (아마도) 현재 작업 디렉토리가 디폴트 시작 위치가 된다. 컴퓨터 시작 메뉴에서 시작한다면, 대신에 바탕화면(Desktop) 혹은 문서 디렉토리에 파일을 저장하고 싶을지도 모른다. "다른 이름으로 저장하기(Save As…)"로 다른 디렉토리로 이동하여 작업 디렉토리를 변경하여 파일을 저장할 수도 있다.

텍스트 몇 줄을 타이핑하고, 컨트롤+O (Control-O, Ctrl 혹은 콘트롤 키보드를 누르면서 0를 누름)를 눌러서 데이터를 디스크에 쓰면 저장된다: (저장하고자 하는 파일명을 입력하도록 독촉받게 되면 draft.txt 기본디폴트로 설정된 것을 받아들이고 엔터키를 친다.)파일이 저자되며 커트록+Y (Ctrl-Y Control-Y)를 사용하여 펴짓기를 끌내고 쉐로 돈아

파일이 저장되면, 컨트롤+X (Ctrl-X, Control-X)를 사용하여 편집기를 끝내고 쉘로 돌아 간다.

♦ 주의

Control, Ctrl, ^ Key

컨트롤 키를 줄여서 "Ctrl" 키라고도 부른다. 컨트롤 키를 기술하는 몇가지 방식이 있다. 예를 들어, "컨트롤 키를 누룬다", "컨트롤 키를 누르면서 X 키를 친다"라는 표 GNU nano 2.0.6 File: draft.txt Modified

It's not "publish or perish" any more, it's "share and thrive".

^G Get Help **^X** Exit

^0 WriteOut **^J** Justify

Read File Y Prev Page Cut Text Cur Pos Where Is Next Page UnCut Text To Spell

그림 5.1: Nano in Action

현은 다음 중 하나로 기술된다:

- Control-X
- Control+X
- Ctrl-X
- Ctrl+X
- ^X
- C-x

nano 편집기에서 화면 하단에 'G Get Help 'O WriteOut을 볼 수 있다. Control-G를 눌러 도움말을 얻고, Control-0를 눌러 파일을 저장한다는 의미를 갖는다.

nano는 화면에 어떤 출력도 뿌려주지 않고 끝내지만, ls 명령어를 사용하여 draft.txt 파 일이 생성된 것을 확인할 수 있다:

\$ ls

draft.txt

예제 5.1 (파일을 생성하는 다른 방법). nano 편집기를 사용해서 텍스트 파일을 생성하는 방법을 살펴봤다. 홈 디렉토리에서 다음 t쉘 명령어를 실행해 보자:

- 1. touch 명령어는 어떤 작업을 수행하는가? GUI 파일 탐색기를 사용해서 본인 홈 디 렉토리를 살펴보게 되면, 파일이 생성된 것이 보이는가?
- 2. ls -l 명령어를 사용해서 파일을 살펴보자. my_file.txt 파일은 얼마나 큰가?
- 3. 이런 방식으로 파일을 언제 생성하면 좋을까?

\$ cd

홈 디렉토리로 이동하기

\$ touch my_file.txt

♀ 실행결과 및 해석*

- 1. touch 명령어가 홈 디렉토리에 'my_file.txt' 파일을 새로 생성시킨다. 터미널로 현재 홈 디렉토리에 있는 경우, ls 를 타이핑하게 되면 새로 생성된 파일을 확인할 수 있다. GUI 파일 탐색기로도 'my_file.txt' 파일을 볼 수 있다.
- 2. 'ls -l' 명령어로 파일을 조사하게 되면, 'my_file.txt' 파일크기가 0kb 임에 주목한다. 다른 말로 표현하면, 데이터가 아무 것도 없다는 의미가 된다. 텍스트 편집기로 'my_file.txt' 파일을 열게 되면, 텅 비어 있다.
- 3. 일부 프로그램은 그 자체로 출력 파일을 생성하지 않지만, 빈 파일이 이미 생성되어 있는 것을 요구조건으로 하는 경우가 있다. 프로그램이 실행되면, 출력 결과를 채울 수 있는 파일이 존재하는지 검색한다. 이런 프로그램에게 touch 명령어는 빈 텍스트 파일을 효율적으로 생성할 수 있는 메커니즘을 제공한다는 점에서 유용하다.

shell-lesson-data 디렉토리로 돌아가서, 생성한 초안을 제거해서 thesis 디렉토리를 깔 끔하게 정리하자:

\$ cd thesis

\$ rm draft.txt

상기 명령어는 파일을 제거한다(rm은 "remove"를 줄인 것이다.) ls 명령어를 다시 실행하게 되면, 출력결과는 아무 것도 없게 되는데 파일이 사라진 것을 확인시켜준다:

\$ ls

▲ 주의

삭제는 영원하다

유닉스에는 삭제된 파일을 복구할 수 있는 휴지통이 없다. (하지만, 유닉스에 기반한 대부분의 그래픽 인터페이스는 휴지통 기능이 있다) 파일을 삭제하면 파일시스

템의 관리대상에서 빠져서 디스트 저장공간이 다시 재사용되게 한다. 삭제된 파일을 찾아 되살리는 도구가 존재하지만, 어느 상황에서나 동작한다는 보장은 없다. 왜 냐하면 파일이 저장되었던 공간을 컴퓨터가 바로 재사용할지 모르기 때문이다.

파일을 다시 생성하고 나서, cd ..를 사용하여 /Users/nelle/Desktop/data-shell 상위 디렉토리로 이동해보자:

\$ pwd

/Users/nelle/Desktop/data-shell/thesis

\$ nano draft.txt

\$ ls

draft.txt

\$ cd ..

rm thesis을 사용하여 전체 thesis 디렉토리를 제거하려고 하면 오류 메시지가 생긴다:

\$ rm thesis

rm: cannot remove `thesis': Is a directory

rm 명령어는 파일에만 동작하고 디렉토리에는 동작하지 않기 때문에 오류가 발생한다. thesis 디렉토리를 제거하려면, draft.txt 파일도 삭제해야 한다. rm 명령어에 재귀 (recursive) 선택옵션을 사용해서 삭제 작업을 수행할 수 있다:

\$ rm -r thesis

♦ 주의

rm 안전하게 사용하기

rm -i thesis/quotations.txt 타이핑하면 무슨 일이 일어날까? rm 명령어를 사용할 때 왜 이러한 보호장치가 필요할까?

\$ rm: remove regular file 'thesis/quotations.txt'?

-i 선택옵션은 삭제하기 전에 삭제를 확인하게 해준다. 유닉스 쉘에는 휴지통이 없어서, 삭제되는 모든 파일은 영원히 사라진다. -i 플래그를 사용하게 되면, 삭제를 원하는 파일만 삭제되는지 점검할 수 있는 기회를 갖게된다.

♦ 주의

큰 힘에는 큰 책임이 따른다(With Great Power Comes Great Responsibility)

디렉토리에 먼저 파일을 제거하고, 그리고 나서 디렉토리를 제거하는 방식은 지루하고 시간이 많이 걸린다. 대신에 -r 옵션을 가진 rm 명령어를 사용할 수 있다. -r 플래그 옵션은 "recursive(재귀적)"을 나타낸다.

\$ rm -r thesis

디렉토리에 모든 것을 삭제하고 나서 디렉토리 자체도 삭제한다. 만약 디렉토리가하위 디렉토리를 가지고 있다면, rm -r은 하위 디렉토리에도 같은 작업을 반복한다. 매우 편리하지만, 부주위하게 사용되면 피해가 엄청날 수 있다.

디렉톨리 파일을 재귀적으로 제거하는 것은 매우 위험할 수 있다. 삭제되는 것에 염려가 된다면, rm 명령어에 -i 인터랙티브 플래그를 추가해서 삭제단계마다 확인을 하고 삭제하는 것도 가능하다.

```
$ rm -r -i thesis
rm: descend into directory 'thesis'? y
rm: remove regular file 'thesis/draft.txt'? y
rm: remove directory 'thesis'? y
```

상기 명령어는 thesis 디렉토리 내부 모든 것을 삭제하고 나서 thesis 디렉토리도 삭제하는데 삭제단계별로 확인 절차를 거친다.

다시 한번 디렉토리와 파일을 생성하자. 이번에는 thesis/draft.txt 파일경로로 바로 nano를 실행함을 주목하자. 이전에는 thesis디렉토리로 가서 draft.txt이름으로 nano를 실행했다.

\$ pwd

/Users/nelle/Desktop/data-shell

- \$ mkdir thesis
- \$ nano thesis/draft.txt
- \$ ls thesis

draft.txt

5.1 파일과 폴더 이동

draft.txt가 특별한 정보를 제공하는 이름이 아니어서 mv를 사용하여 파일 이름을 변경하자.mv는 "move"의 줄임말이다:

\$ mv thesis/draft.txt thesis/quotes.txt

첫번째 매개변수는 mv 명령어에게 이동하려는 대상을, 두번째 매개변수는 어디로 이동 되는지를 나타낸다. 이번 경우에는 thesis/draft.txt 파일을 thesis/quotes.txt으로 이동한다. 이렇게 파일을 이동하는 것이 파일 이름을 바꾸는 것과 동일한 효과를 가진다. 아니나 다를까, ls 명령어를 사용하여 확인하면 thesis 디렉토리에는 이제 quotes.txt 파일만 있음을 확인할 수 있다:

\$ ls thesis
quotes.txt

목표 파일명을 명세할 때 주의를 기울일 필요가 있다. 왜냐하면, mv 명령어는 동일 명칭을 갖는 어떤 기존 파일도 아주 조용히 덮어 써버리는 재주가 있어 데이터 유실에 이르게 된다. 부가적인 옵션 플래그, mv -i (즉 mv --interactive)를 사용해서 덮어쓰기 전에 사용자가 확인하도록 mv 명령어를 활용할 수도 있다.

일관성을 갖고 있어서, mv는 디렉토리에도 동작한다 — 별도 mvdir 명령어는 없다.

quotes.txt 파일을 현재 작업 디렉토리로 이동합시다. mv를 다시 사용한다. 하지만 이번에는 두번째 매개변수로 디렉토리 이름을 사용해서 파일이름을 바꾸지 않고, 새로운 장

소에 놓는다. (이것이 왜 명령어가 "move(이동)"으로 불리는 이유다.) 이번 경우에 사용되는 디렉토리 이름은 앞에서 언급한 특수 디렉토리 이름. 이다.

```
$ mv thesis/quotes.txt .
```

과거에 있던 디렉토리에서 파일을 현재 작업 디렉토리로 옮긴 효과가 나타난다. ls 명령 어가 thesis 디렉토리가 비였음을 보여준다:

\$ ls thesis

더 나아가, ls 명령어를 인자로 파일 이름 혹은 디렉토리 이름과 함께 사용하면, 그 해당 파일 혹은 디렉토리만 화면에 보여준다. 이렇게 사용하면, quotes.txt 파일이 현재 작업 디렉토리에 있음을 볼 수 있다:

```
$ ls quotes.txt
quotes.txt
```

예제 5.2 (현재 폴더로 이동하기). 다음 명령어를 실행한 후에, 정훈이는 sucrose.dat, maltose.dat 파일을 잘못된 폴더에 넣은 것을 인지하게 되었다:

```
$ ls -F
analyzed/ raw/
$ ls -F analyzed
fructose.dat glucose.dat maltose.dat sucrose.dat
$ cd raw/
```

해당 파일을 현재 디렉토리(즉, 현재 사용자가 위치한 폴더)로 이동시키도록 아래 빈칸을 채우시오:

```
$ mv ___/sucrose.dat ___/maltose.dat ___
```

🥊 해답과 설명

\$ mv ../analyzed/sucrose.dat ../analyzed/maltose.dat .

.. 디렉토리는 부모 디렉토리(즉, 현재 디렉토리에서 상위 디렉토리를 지칭). 디렉 토리는 현재 디렉토리를 지칭함을 상기한다.

cp 명령어는 mv 명령어와 거의 동일하게 동작한다. 차이점은 이동하는 대신에 복사한다 는 점이다. 인자로 경로를 두개 갖는 ls 명령어로 제대로 작업을 했는지 확인할 수 있다. 대부분의 유닉스 명령어와 마찬가지로, ls 명령어로 한번 경로 다수를 전달할 수도 있다:

```
$ cp quotes.txt thesis/quotations.txt
$ ls quotes.txt thesis/quotations.txt
quotes.txt thesis/quotations.txt
```

복사를 제대로 수행했는지 증명하기 위해서, 현재 작업 디렉토리에 있는 quotes.txt 파 일을 삭제하고 나서, 다시 동일한 ls 명령어를 실행한다.

```
$ rm quotes.txt
$ ls quotes.txt thesis/quotations.txt
ls: cannot access quotes.txt: No such file or directory
thesis/quotations.txt
```

이번에는 현재 디렉토리에서 quotes.txt 파일은 찾을 수 없지만, 삭제하지 않은 thesis 폴 더의 복사본은 찾아서 보여준다.

▲ 주의

파일명이 뭐가 중요해?

Nelle의 파일 이름이 "무엇.무엇"으로 된 것을 알아챘을 것이다. 이번 학습에서, 항 상 .txt 확장자를 사용했다. 이것은 단지 관례다: 파일 이름을 mythesis 혹은 원하 는 무엇이든지 작명할 수 있다. 하지만, 대부분의 사람들은 두 부분으로 구분된 이 름을 사용하여 사람이나 프로그램이 다른 유형의 파일임을 구분하도록 돕는다. 이 름에 나온 두번째 부분을 파일 확장자(filename extension)라고 부르고, 파일에 어

떤 유형의 데이터가 담고 있는지 나타낸다. .txt 확장자는 텍스트 파일임을, .pdf는 PDF 문서임을, .cfg 확장자는 어떤 프로그램에 대한 구성정보를 담고 있는 형상관 리 파일임을 내고, .png 확장자는 PNG 이미지 등등을 나타낸다.

단지 관습이기는 하지만 중요하다. 파일은 바이트(byte) 정보를 담고 있다: PDF 문 서, 이미지, 등에 대해서 규칙에 따라 바이트를 해석하는 것은 사람과 작성된 프로 그램에 맡겨졌다.

whale.mp3처럼 고래 PNG 이미지 이름을 갖는 파일을 고래 노래의 음성파일로 변환 하는 마술은 없다. 설사 누군가 두번 클릭할 때, 운영체제가 음악 재생기로 열어 실 행할 수는 있지만 동작은 되지 않을 것이다.

예제 5.3 (파일 이름 바꾸기). 데이터를 분석하는데 필요한 통계 검정 목록을 담고 있는 .txt 파일을 현재 디렉토리에 생성했다고 가정하자; 파일명은 statstics.txt. 파일을 생 성하고 저장한 후에 곰곰히 생각해 보니 파일명 철자가 틀린 것을 알게 되었다! 틀린 철 자를 바로잡고자 하는데, 다음 중 어떤 명령어를 사용해야 하는가?

- 1. cp statstics.txt statistics.txt
- 2. mv statstics.txt statistics.txt
- mv statstics.txt .
- 4. cp statstics.txt .

🥊 해답과 설명

- 1. No. 철자오류가 수정된 파일이 생성되지만, 철자가 틀린 파일도 디렉토리에 여전히 존재하기 때문에 삭제작업이 필요하다.
- 2. Yes, 이 명령어를 통해서 파일명을 고칠 수 있다.
- 3. No, 마침표(.)는 파일을 이동할 디렉토리를 나타내지 새로운 파일명을 제시하 고 있지는 않고 있다; 동일한 파일명은 생성될 수 없다.
- 4. No, 마침표(.)는 파일을 복사할 디렉토리를 나타내지 새로운 파일명을 제시하 고 있지는 않고 있다; 동일한 파일명은 생성될 수 없다.

예제 5.4 (이동과 복사). 아래 보여진 일련의 명령문에 뒤에 ls명령어의 출력값은 무엇일 까요?

\$ pwd

/Users/jamie/data

\$ ls

proteins.dat

- \$ mkdir recombine
- \$ mv proteins.dat recombine/
- \$ cp recombine/proteins.dat ../proteins-saved.dat
- \$ ls
- 1. proteins-saved.dat recombine
- 2. recombine
- 3. proteins.dat recombine
- 4. proteins-saved.dat

해답과 설명

/Users/jamie/data 디렉토리에서 출발해서, recombine 이름의 디렉토리를 새로 생성한다. 두번째 행은 proteins.dat 파일을 새로 만든 폴더 recombine으로 이동(mv) 시킨다. 세번째 행은 방금전에 이동한 파일에 대한 사본을 생성시킨다. 여기서 조금 까다로운 점은 파일이 복사되는 디렉토리다. . . 이 의미하는 바가 "한단계 위로 이동"하라는 의미라서, 복사되는 파일은 이제 /Users/jamie 디렉토리에 위치하게 됨을 상기한다. . . 이 의미하는 바는 복사되는 파일 위치에 대한 것이 아니라 현재 작업 디렉토리에 대한 것으로 해석됨에 유의한다. 그래서, 그래서, ls 명령어를 사용해서 보여지게 되는 것은 (/Users/jamie/data에 있기 때문에) recombine 폴더가 된다.

- 1. No, 상기 해설을 참조한다. proteins-saved.dat 데이터는 /Users/jamie 폴더에 위치한다.
- 2. Yes
- 3. No, 상기 해설을 참조한다. proteins.dat 데이터는 /Users/jamie/data/recombine 폴더에 위치한다.
- 4. No, 상기 해설을 참조한다. proteins-saved.dat 데이터는 /Users/jamie 폴더에 위치

한다.

5.2 다수 파일과 폴더 작업

한 번에 여러 파일을 복사하거나 이동해야 하는 경우가 종종 있다. 파일명을 목록으로 제 공하거나 와일드카드를 사용하여 패턴을 지정하면 된다. **와일드카드(Wildcards)**는 Unix 파일 시스템을 탐색할 때 특정한 패턴이나 일련의 문자를 대체하거나 대표하는 데 사용 되는 특수 문자로 '*','?','%' 등이 대표적이다.

예제 5.5 (다수 파일을 복사하기). 이번 연습문제에서는 data-shell/data 디렉토리에서 명령어를 테스트한다. 아래 예제에서, 파일명 다수와 디렉토리명이 주어졌을 때 cp 명령 어는 어떤 작업을 수행하는가?

- \$ mkdir backup
- \$ cp amino-acids.txt animals.txt backup/

아래 예제에서, 3개 혹은 그 이상의 파일명이 주어졌을 때 cp 명령어는 어떤 작업을 수행 하는가?

\$ ls -F

amino-acids.txt animals.txt backup/ elements/ morse.txt pdb/ planets.txt ⇔ salmon.txt sunspot.txt

\$ cp amino-acids.txt animals.txt morse.txt

● 해답과 설명

하나이상 파일명 다음에 디렉토리명이 주어지게 되면(즉, 목적지 디렉토리는 마지 막 인자에 위치해야 한다.), cp 명령어는 파일을 해당 디렉토리에 복사한다. 연달아 파일명이 세게 주어지면, cp 명령어는 오류를 던지는데 이유는 마지막 인자 로 디렉토리를 기대했기 때문이다.

cp: target 'morse.txt' is not a directory

🌢 주의

와일드 카드(Wildcards)

*는 **와일드카드(wildcard)**다. 와일드카드는 0 혹은 그 이상의 문자와 매칭되서, *.pdb은 ethane.pdb, propane.pdb 등등에 매칭한다. 반면에, p*.pdb은 propane.pdb와 pentane.pdb만 매칭하는데, 맨 앞에 'p'로 시작되는 파일명만 일치하기만 하면 되기 때문이다.

?도 또한 와일드카드지만 단지 단일 문자만 매칭한다. 이것이 의미하는 바는 p?.pdb 은 pi.pdb

혹은 p5.pdb을 매칭하지만 (molecules 디렉토리에 두 파일이 있다면), propane.pdb은 매칭하지 않는다. 한번에 원하는 수만큼 와일드카드를 사용할 수 있다. 예를 들어, p*.p?*는 'p'로 시작하고'.'과 'p', 그리고 최소 한자의 이상의 문자로 끝나는 임의의 문자열을 매칭한다고 표현할 수 있는데 '?'이 한 문자를 매칭해야하고 마지막'*'은 끝에 임의의 문자숫자와 매칭할 수 있기 때문이다. 그래서 p*.p?*은 preferred.practice 과 심지어 p.pi도 매칭한다(첫번째'*'은 어떤 문자도 매칭할 수가 없음). 하지만 quality.practice은 매칭할 수 없는데 이유는 'p'로 시작하지 않고, preferred.p도 매칭할 수 없는데 'p' 다음에 최소 하나의 문자가 필요한데 없기 때문이다.

쉘이 와일드카드를 봤을 때, 요청된 명령문을 시작하기 전에 와일드카드를 확장하여 매칭할 파일 이름 목록을 생성한다. 예외로, 와일드카드 표현식이 어떤 파일과도 매칭되지 않게되면, 배수는 명령어에 인자로 표현식을 있는 그대로 전달한다. 예를들어, molecules 디렉토리(.pdb 확장자로 끝나는 파일만 모여있다.)에 ls *.pdf을 타이핑하게 되면, *.pdf으로 불리는 파일이 없다고 오류 메시지를 출력한다. 하지만, 일반적으로 wc과 ls 명령어는 와일드카드 표현식과 매칭되는 파일명 목록을 보게되고 와일드카드 자체가 아니다. 다른 프로그램은 아니지만, 쉘은 와일드카드를 확장한 것을 다룬다는 점에서 직교 설계(orthogonal design)의 또 다른 사례로 볼 수 있다.

예제 5.6 (와일드카드 추가 문제). 정훈이는 미세조정(calibration), 원본 데이터(dataset), 데이터 설명 데이터를 디렉토리에 보관하고 있다:

2015-10-23-calibration.txt 2015-10-23-dataset1.txt

2015-10-23-dataset2.txt

```
2015-10-23-dataset_overview.txt
2015-10-26-calibration.txt
2015-10-26-dataset1.txt
2015-10-26-dataset2.txt
2015-10-26-dataset_overview.txt
2015-11-23-calibration.txt
2015-11-23-dataset1.txt
2015-11-23-dataset2.txt
2015-11-23-dataset2.txt
```

또 다른 견학여행을 떠나기 전에, 정훈이는 데이터를 백업하고 일부 데이터를 랩실 동료 기민에게 보내고자 한다. 정훈이는 백업과 전송 작업을 위해서 다음 명령어를 사용한다:

```
$ cp *dataset* /backup/datasets
$ cp ____calibration___ /backup/calibration
$ cp 2015-___ ~/send_to_bob/all_november_files/
$ cp ___ ~/send_to_bob/all_datasets_created_on_a_23rd/
```

정훈이가 빈칸을 채우도록 도움을 주세요.

```
● 해답과 설명

$ cp *calibration.txt /backup/calibration

$ cp 2015-11-* ~/send_to_bob/all_november_files/

$ cp *-23-dataset* ~send_to_bob/all_datasets_created_on_a_23rd/
```

에제 5.7 (디렉토리와 파일 조직화). 정훈이가 프로젝트 작업을 하고 있는데, 작업 파일이 그다지 잘 조직적으로 정리되어 있지 않음을 알게 되었다:

```
$ ls -F
analyzed/ fructose.dat raw/ sucrose.dat
```

fructose.dat 와 sucrose.dat 파일은 자료분석 결과 산출된 출력결과를 담고 있다. 이번 학습에서 배운 어떤 명령어를 실행해야, 아래 명령어를 실행했을 때 다음에 보여지는 출

력을 생성할까요?

\$ ls -F
analyzed/ raw/

\$ ls analyzed

fructose.dat sucrose.dat

🥊 해답과 설명

\$ mv *.dat analyzed

정훈이는 analyzed 디렉토리에 fructose.dat, sucrose.dat 파일을 이동시킬 필요가 있다. 쉘에서 현재 디렉토리에서 *.dat 와일드카드가 .dat 확장자를 갖는 모든 파일을 매칭한다. mv 명령어가 .dat 확장자를 갖는 파일을 analyzed 디렉토리로 이동시킨다.

예제 5.8 (폴더 구조를 복사하지만, 파일을 복사하지 말자.). 새로운 실험을 시작해 보자. 데이터 파일 없이 이전 실험에게 만들었던 파일 구조만 복제하자. 그렇게 하면 새로운 데이터를 쉽게 추가할 수 있게 된다. '2016-05-18-data' 디렉토리에 data 폴더로 raw와 processed가 있는데, 각자 데이터 파일이 담겨있다.

목적은 2016-05-18-data 폴더를 2016-05-20-data 폴더로 복사하는 것인데 복사된 폴더에는 모든 데이터 파일을 제거해야 된다. 다음 명령어 집합 중 어떤 명령어 집합이 상기 목적을 달성할까요? 다른 명령어 집합은 무슨 작업을 수행하는 것일가?

```
$ cp -r 2016-05-18-data/ 2016-05-20-data/
```

- \$ rm 2016-05-20-data/raw/*
- \$ rm 2016-05-20-data/processed/*
- \$ rm 2016-05-20-data/raw/*
- \$ rm 2016-05-20-data/processed/*
- \$ cp -r 2016-05-18-data/ 2016-5-20-data/

\$ cp -r 2016-05-18-data/ 2016-05-20-data/

\$ rm -r -i 2016-05-20-data/

● 해답과 설명

첫번째 명령어들이 해당 목적을 달성한다. 먼저 재귀적으로 데이터 폴더를 복사한 다. 그리고 나서 rm 명령어 두번 사용해서 복사한 디렉토리의 모든 파일을 제거한 다. 쉘은 * 와일드카드로 매칭되는 모든 파일과 하위디렉토리를 확장하도록 한다. 두번째 명령어들은 순서가 잘못되었다: 복사하지 않는 파일을 샂게하고 나서 재귀 복사 명령어로 디렉토리를 복사했다.

세번째 명령어도 목적을 달성하는데, 시간이 다소 소요된다: 첫번째 명령어가 디렉 토리를 재귀적으로 복사하지만, 두번째 명령어는 인터랙티브하게 각 파일과 디렉 토리에 대한 확인하는 과정을 거쳐 삭제를 하게 되어 시간이 추가로 소요된다.

5.1 연습문제

- 1. 다음 중 쉘에서 디렉토리를 생성하는 명령어는 무엇인가요?
 - (1) cd thesis
 - (2) ls -F
 - (3) mkdir thesis
 - (4) pwd
 - (5) rm thesis
- 2. 쉘에서 'mkdir thesis' 명령어를 실행하면 어떤 결과를 기대할 수 있나요?
 - (1) 현재 디렉토리에서 thesis라는 이름의 파일을 생성합니다.
 - (2) 현재 디렉토리에서 thesis라는 이름의 디렉토리를 생성합니다.
 - (3) thesis라는 이름의 파일을 삭제합니다.
 - (4) thesis 디렉토리로 이동합니다.
 - (5) thesis라는 이름의 파일을 엽니다.
- 3. 다음 중 파일이나 디렉토리의 이름으로 적절하지 않은 것은 무엇인가요?
 - (1) data_analysis

- (2) data-analysis
- (3) data analysis
- (4) data.analysis
- (5) data_analysis_1
- 4. 다음 중 파일이나 디렉토리의 이름으로 사용하면 좋지 않은 문자는 무엇인가요?
 - (1)_(밑줄)
 - (2) (대쉬)
 - (3).(마침표)
 - (4)(공백)
 - (5) 숫자
- 5. 명령라인으로 작업할 때 파일과 디렉토리 이름을 지을 때 고려해야 할 사항들은 무엇 인가요?

제 6 장

파이프와 필터

몇가지 기초 유닉스 명령어를 배웠기 때문에, 마침내 쉘의 가장 강령한 기능을 살펴볼수 있게 되었다: 새로운 방식으로 기존에 존재하던 프로그램을 쉽게 조합해 낼 수 있게 한다. 간단한 유기분자 설명을 하는 6개 파일을 담고 있는 molecules(분자)라는 디렉토리에서 시작한다. .pdb 파일 확장자는 단백질 데이터 은행 (Protein Data Bank) 형식으로, 분자의 각 원자 형식과 위치를 표시하는 간단한 텍스트 형식으로 되어 있다.

\$ ls molecules

```
cubane.pdb ethane.pdb methane.pdb
octane.pdb pentane.pdb propane.pdb
```

명령어 cd로 해당 디렉토리로 가서 wc *.pdb 명령어를 실행한다. wc 명령어는 "word count" 의 축약어로 파일의 라인 수, 단어수, 문자수를 개수한다. (왼쪽에서 오른쪽 순서로)

*.pdb에서 *은 0혹은 더 많이 일치하는 문자를 매칭한다. 그래서 쉘은 *.pdb을 통해 .pdb 전체 리스트 목록을 반환한다:

```
$ cd molecules
$ wc *.pdb

20 156 1158 cubane.pdb
```

```
      12
      84
      622
      ethane.pdb

      9
      57
      422
      methane.pdb

      30
      246
      1828
      octane.pdb

      21
      165
      1226
      pentane.pdb

      15
      111
      825
      propane.pdb

      107
      819
      6081
      total
```

wc 대신에 wc -1을 실행하면, 출력결과는 파일마다 행수만을 보여준다:

```
$ wc -l *.pdb

20 cubane.pdb
12 ethane.pdb
9 methane.pdb
30 octane.pdb
21 pentane.pdb
15 propane.pdb
107 total
```

단어 숫자만을 얻기 위해서 -w, 문자 숫자만을 얻기 위해서 -c을 사용할 수 있다.

6.1 명령어에서 출력 캡처하기

파일 중에서 어느 파일이 가장 짧을까요? 단지 6개의 파일이 있기 때문에 질문에 답하기는 쉬울 것이다. 하지만 만약에 6000 파일이 있다면 어떨까요? 해결에 이르는 첫번째 단계로 다음 명령을 실행한다:

```
$ wc -l *.pdb > lengths.txt
```

> 기호는 쉘로 하여금 화면에 처리 결과를 뿌리는 대신에 파일로 **방향변경(redirect)**하게 한다. 만약 파일이 존재하지 않으면 파일을 생성하고 파일이 존재하면 파일에 내용을 덮어쓰기 한다. 조용하게 덮어쓰기 하기 때문에 자료가 유실될 수 있어서 주의가 요구된다. (이것이 왜 화면에 출력결과가 없는 이유다. wc가 출력하는 모든 것은 lengths.txt 파

일에 대신 들어간다.) ls lengths.txt 을 통해 파일이 존재하는 것을 확인한다:

\$ ls lengths.txt

lengths.txt

cat lengths.txt을 사용해서 화면으로 lengths.txt의 내용을 보낼 수 있다. cat은 "concatenate"를 줄인 것이고 하나씩 하나씩 파일의 내용을 출력한다.이번 사례에는 단지 파일이 하나만 있어서, cat 명령어는 단지 한 파일이 담고 있는 내용만 보여준다:

\$ cat lengths.txt

- 20 cubane.pdb
- 12 ethane.pdb
- 9 methane.pdb
- 30 octane.pdb
- 21 pentane.pdb
- 15 propane.pdb
- 107 total

♦ 주의

페이지 단위 출력결과 살펴보기

이번 학습에서 편리성과 일관성을 위해서 cat 명령어를 계속 사용한다. 하지만, 파일 전체를 화면에 쭉 뿌린다는 면에서 단점이 있다. 실무적으로 less 명령어가 더유용한데 \$ less lengths.txt와 같이 사용한다. 파일을 화면 단위로 출력한다. 아래로 내려가려면 스페이스바를 누르고, 뒤로 돌아가려면 b를 누르면 되고, 빠져 나가려면 q를 누른다.

6.2 출력 필터링

이제 sort 명령어를 사용해서 파일 내용을 정렬합니다.

예제 6.1 (sort -n 명령어는 어떤 작업을 수행할까?). 다음 파일 행을 포함하고 있는 파일

에 sort 명령어를 실행하면:

출력결과는 다음과 같다:

동일한 입력에 대해서 sort -n을 실행하면, 대신에 다음 결과를 얻게 된다:

인수 -n이 왜 이런 효과를 가지는지 설명하세요.

♥ 해답과 설명

-n 플래그는 알파벳 정렬이 아닌, 숫자 정렬하도록 명세한다.

-n 플래그를 사용해서 알파벳 대신에 숫자 방식으로 정렬할 것을 지정할 수 있다. 이 명령어는 파일 자체를 변경하지 않고 대신에 정렬된 결과를 화면으로 보낸다:

```
$ sort -n lengths.txt
```

- 9 methane.pdb
- 12 ethane.pdb
- 15 propane.pdb
- 20 cubane.pdb
- 21 pentane.pdb
- 30 octane.pdb
- 107 total

> lengths.txt을 사용해서 wc 실행결과를 lengths.txt에 넣었듯이, 명령문 다음에 > sorted-lengths.txt을 넣음으로서, 임시 파일이름인 sorted-lengths.txt에 정렬된 목록 정보를 담을 수 있다. 이것을 실행한 다음에, 또 다른 head 명령어를 실행해서 sorted-lengths.txt 에서 첫 몇 행을 뽑아낼 수 있다:

```
$ sort -n lengths.txt > sorted-lengths.txt
$ head -n 1 sorted-lengths.txt

9 methane.pdb
```

head에 -n 1 매개변수를 사용해서 파일의 첫번째 행만이 필요하다고 지정한다. -n 20은처음 20개 행만을 지정한다. sorted-lengths.txt이 가장 작은 것에서부터 큰 것으로 정렬된 파일 길이 정보를 담고 있어서, head의 출력 결과는 가장 짧은 행을 가진 파일이 되어야만 된다.

예제 6.2 (동일한 파일에 방향변경하기). 명령어 출력결과를 방향변경하는데 동일한 파일에 보내는 것은 매우 나쁜 아이디어다. 예를 들어:

```
$ sort -n lengths.txt > lengths.txt
```

위와 같이 작업하게 되면 틀린 결과를 얻을 수 있을 뿐만 아니라 경우에 따라서는 lengths.txt 파일 내용을 잃어버릴 수도 있다.

>>은 무엇을 의미하는가?

> 사용법을 살펴봤지만, 유사한 연산자로 >>도 있는데 다소 다른 방식으로 동작한다. 문 자열을 출력하는 echo 명령어를 사용해서, 두 연산자 차이를 밝혀내는데 아래 명령어를 테스트 한다:

```
$ echo hello > testfile01.txt
```

```
$ echo hello >> testfile02.txt
```

힌트: 각 명령문을 연속해서 두번 실행하고 나서, 출력결과로 나온 파일을 면밀히 조사한다.

🥊 해답과 설명

> 연산자를 갖는 첫번째 예제에서 문자열 "hello"는 testfile01.txt 파일에 저장된다. 하지만, 매번 명령어를 실행할 때마다 파일에 덮어쓰기를 한다.

두번째 예제에서 >> 연산자도 마찬가지로 "hello"를 파일에 저장(이 경우 testfile02.txt)하는 것을 알 수 있다. We see from the second example that the >> operator also writes "hello" to a file 하지만, 파일이 이미 존재하는 경우(즉, 두번째 명령어를 실행하게 되면) 파일에 문자열을 덧붙인다.

에제 6.3 (데이터 덧붙이기). head 명령어는 이미 만나봤다. 파일 시작하는 몇줄을 화면에 출력하는 역할을 수행한다. tail 명령어도 유사하지만, 반대로 파일 마지막 몇줄을 화면에 출력하는 역할을 수행한다. data-shell/data/animals.txt 파일을 생각해 보자. 다음 명령어를 실행하게 되면 animalsUpd.txt 파일에 저장될 내용이 어떤 것일지 아래에서 정답을 고르세요:

```
$ head -n 3 animals.txt > animalsUpd.txt
```

- \$ tail -n 2 animals.txt >> animalsUpd.txt
- 1. animals.txt 파일 첫 3줄.
- 2. animals.txt 파일 마지막 2줄.
- 3. animals.txt 파일의 첫 3줄과 마지막 2줄.
- 4. animals.txt 파일의 두번째 세번째 줄.

€ 해답과 설명

정답은 3.1번이 정답이 되려면, head 명령어만 실행한다. 2번이 정답이 되려면, tail 명령어만 실행한다. 4번이 정답이 되려면, head -3 animals.txt ¦ tail -2 >> animalsUpd.txt 명령어를 실행해서 head 출력결과를 파이프에 넣어 tail -2를 실행 해야 한다.

6.3 다른 명령에 출력 전달하기

이것이 혼란스럽다면, 좋은 친구네요: wc, sort, head 명령어 각각이 무엇을 수행하는지 이 해해도, 중간에 산출되는 파일에 무슨 일이 진행되고 있는지 따라가기는 쉽지 않다. sort 와 head을 함께 실행해서 이해하기 훨씬 쉽게 만들 수 있다:

```
$ sort -n lengths.txt | head -n 1
```

9 methane.pdb

두 명령문 사이의 수직 막대를 **파이프(pipe)**라고 부른다. 수직막대는 쉘에게 왼편 명령 문의 출력결과를 오른쪽 명령문의 입력값으로 사용된다는 뜻을 전달한다. 컴퓨터는 필 요하면 임시 파일을 생성하거나, 한 프로그램에서 주기억장치의 다른 프로그램으로 데 이터를 복사하거나, 혹은 완전히 다른 작업을 수행할 수도 있다; 사용자는 알 필요도 없 고 관심을 가질 이유도 없다.

어떤 것도 파이프를 연속적으로 사슬로 엮어 사용하는 것을 막을 수는 없다. 즉, 예를 들 어 또 다른 파이프를 사용해서 wc의 출력결과를 sort에 바로 보내고 나서, 다시 처리 결 과를 head에 보낸다. wc 출력결과를 sort로 보내는데 파이프를 사용했다:

```
$ wc -l *.pdb | sort -n
   9 methane.pdb
 12 ethane.pdb
 15 propane.pdb
  20 cubane.pdb
```

```
21 pentane.pdb
30 octane.pdb
107 total
```

6.4 여러 명령 결합하기

또 다른 파이프를 사용해서 wc의 출력결과를 sort에 바로 보내고 나서, 다시 처리 결과를 head로 보내게 되면 전체 파이프라인은 다음과 같이 된다:

```
$ wc -l *.pdb | sort -n | head -n 1
9 methane.pdb
```

이것이 정확하게 수학자가 log(3x) 같은 중첩함수를 사용하는 것과 같다. "log(3x)은 x에 3을 곱하고 로그를 취하는 것과 같다." 이번 경우는, *.pdb의 행수를 세어서 정렬해서 첫부분만 계산하는 것이 된다.

예제 6.4 (명령문을 파이프로 연결하기). 현재 작업 디렉토리에, 최소 행수를 갖는 파일을 세개 찾고자 한다. 아래 열거된 어떤 명령어 중 어떤 것이 원하는 파일 3개를 찾아줄까?

```
    wc -l * > sort -n > head -n 3
    wc -l * | sort -n | head -n 1-3
    wc -l * | head -n 3 | sort -n
    wc -l * | sort -n | head -n 3
```

###{.callout-tip} ### 해답과 설명

해답은 4. 파이프 문자 ¦을 사용해서 이 프로세스 표준출력을 다른 프로세스 표준입력으로 넣어준다. > 기호는 표준입력을 파일로 방향변경할 때 사용한다. data-shell/molecules 디렉토리에서도 시도해 보라!

:::::

파이프를 생성할 때 뒤에서 실질적으로 일어나는 일은 다음과 같다. 컴퓨터가 한 프로그램(어떤 프로그램도 동일)을 실행할 때 프로그램에 대한 소프트웨어와 현재 상태 정보를 담기 위해서 주기억장치 메모리에 **프로세스(process)**를 생성한다. 모든 프로세스는

표준 입력(standard input)이라는 입력 채널을 가지고 있다. (여기서 이름이 너무 기억하기 좋아서 놀랄지도 모른다. 하지만 걱정하지 마세요. 대부분의 유닉스 프로그래머는 "stdin"이라고 부른다). 또한 모든 프로세스는 표준 출력(standard output)(혹은 "stdout")이라고 불리는 기본디폴트 출력 채널도 있다. 이 채널이 일반적으로 오류 혹은 진단 메시지 용도로 사용되어서 터미널로 오류 메시지를 받으면서도 그 와중에 프로그램 출력 값이 또다른 프로그램에 파이프되어 들어가는 것이 가능하게 한다.

쉘은 실질적으로 또다른 프로그램이다. 정상적인 상황에서 사용자가 키보드로 무엇을 타이핑하는 모든 것은 표준 입력으로 쉘에 보내지고, 표준 출력에서 만들어지는 무엇이든지 화면에 출력된다. 쉘에게 프로그램을 실행하게 할때, 새로운 프로게스를 생성하고, 임시로 키보드에 타이핑하는 무엇이든지 그 프로세스의 표준 입력으로 보내지고, 프로세스는 표준 출력을 무엇이든 화면에 전송한다.

wc -1 *.pdb > lengths을 실행할 때 여기서 일어나는 것을 설명하면 다음과 같다. wc 프로 그램을 실행할 새로운 프로세스를 생성하라고 쉘이 컴퓨터에 지시한다. 파일이름을 인자로 제공했기 때문에 표준입력 대신 wc는 인자에서 입력값을 읽어온다. >을 사용해서 출력값을 파일로 방향변경 했기했기 때문에, 쉘은 프로세스의 표준 출력결과를 파일에 연결한다.

wc -l *.pdb | sort -n을 실행한다면, 쉘은 프로세스 두개를 생성한다. (파이프 프로세스 각각에 대해서 하나씩) 그래서 wc과 sort은 동시에 실행된다. wc의 표준출력은 직접적으로 sort의 표준 입력으로 들어간다; >같은 방향변경이 없기 때문에 sort의 출력은 화면으로 나가게 된다. wc -l *.pdb | sort -n | head -1을 실행하면, 파일에서 wc에서 sort로, sort에서 head을 통해 화면으로 나가게 되는 데이터 흐름을 가진 프로세스 3개가 있게된다.

6.5 함께 작동하도록 설계된 도구

이 간단한 아이디어가 왜 유닉스가 그토록 성공적이었는지를 보여준다. 다른 많은 작업을 수행하는 거대한 프로그램을 생성하는 대신에, 유닉스 프로그래머는 각자가 한가지 작업만을 잘 수행하는 간단한 도구를 많이 생성하는데 집중하고, 서로간에 유기적으로 잘 작동하게 만든다. 이러한 프로그래밍 모델을 파이프와 필터(pipes and filters)라고 부른다; 파이프는 이미 살펴봤고, 필터(filter)는 wc, sort같은 프로그램으로 입력 스트림을 출력 스트림으로 변환하는 것이다. 거의 모든 표준 유닉스 도구는 이런 방식으로 동작한다: 별도로 언급되지 않는다면, 표준 입력에서 읽고, 읽은 것을 가지고 무언가를 수행하

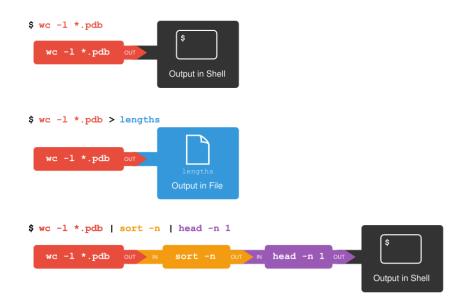


그림 6.1: 방향변경과 파이프

고 표준출력에 쓴다.

중요한 점은 표준입력에서 텍스트 행을 읽고, 표준 출력에 텍스트 행을 쓰는 임의 프로그램은 이런 방식으로 동작하는 모든 다른 프로그램과 조합될 수 있다는 것이다. 여러분도 여러분이 작성한 프로그램을 이러한 방식으로 작성할 수 있어야 하고 *작성해야 한다.* 그래서 여러분과 다른 사람들이 이러한 프로그램을 파이프에 넣어서 생태계 전체 힘을 배가할 수 있다.

♦ 주의

입력 방향변경

프로그램의 출력 결과 방향변경을 위해서 >을 사용하는 것과 마찬가지로, <을 사용해서 입력을 되돌릴 수도 있다. 즉, 표준입력 대신에 파일로부터 읽어 들일 수 있다. 예를 들어, wc ammonia.pdb 와 같이 작성하는 대신에, wc < ammonia.pdb 작성할 수 있다. 첫째 사례는, wc는 무슨 파일을 여는지를 명령 라인의 매개변수에서 얻는다. 두번째 사례는, wc에 명령 라인 매개변수가 없다. 그래서 표준 입력에서 읽지만, 쉘에게 ammonia.pdb의 내용을 wc에 표준 입력으로 보내라고 했다.

:::{#exr-stdin} ### < 기호이 의미하는 것은 무엇인가?

(다운로드 예제 데이터를 갖고 있는 최상위) data-shell 디렉토리로 작업 디렉토리를 변

경한다. 다음 두 명령어 차이는 무엇인가?

```
$ wc -l notes.txt
$ wc -l < notes.txt</pre>
```

🥊 해답과 설명

<기호는 입력을 방향변경을 해서 명령어로 전달한다.

상기 예제 모두에서, 쉘은 입력에서 wc 명령어를 통해 행수를 반환한다. 첫번째 예 제에서, 입력은 notes.txt 파일이고, 파일명이 wc 명령어로부터 출력으로 주어지게 된다. 두번째 예제로부터, notes.txt 파일 내용이 표준입력으로 방향변경을 통해 보내지게 된다. 이것은 마치 프롬프트에서 파일 콘텐츠를 타이핑하는 것과 같다. 따라서, 파일명이 출력에 주어지지 않는다 - 단지 행번호만 주어진다. 다음과 같이 타이핑해보자:

```
$ wc -l
this
is
a test
Ctrl-D # Ctrl-D를 타이핑하게 되면 쉘이 입력을 마무리한 것을 알게 전달하는 역할을 한다.
```

예제 6.5 (uniq가 왜 인접한 중복 행만을 단지 제거한다고 생각합니까?). 명령문 uniq는 입력으로부터 인접한 중복된 행을 제거한다. 예를 들어, salmon.txt 파일에 다음이 포함 되었다면,

coho
coho
steelhead
coho
steelhead
steelhead

shell-lesson-data/data 디렉토리의 uniq salmon.txt 명령문 실행은 다음을 출력한다. coho

steelhead

```
coho
steelhead
```

uniq가 왜 인접한 중복 행만을 단지 제거한다고 생각합니까? (힌트: 매우 큰 파일을 생각해보세요.) 모든 중복된 행을 제거하기 위해, 파이프로 다른 어떤 명령어를 조합할 수 있을까요?

```
♥ 해답과 설명
$ sort salmon.txt ¦ uniq
```

예제 6.6 (파이프 독해능력). shell-lesson-data/data 폴더에 animals.txt로 불리는 파일은 다음 데이터를 포함한다

```
2012-11-05, deer

2012-11-05, rabbit

2012-11-05, raccoon

2012-11-06, rabbit

2012-11-06, deer

2012-11-06, fox

2012-11-07, rabbit

2012-11-07, bear
```

다음 아래 파이프라인에 각 파이프를 통과하고, 마지막 방향변경을 마친 텍스트는 무엇이 될까요?

```
$ cat animals.txt | head -n 5 | tail -n 3 | sort -r > final.txt
```

힌트: 명령어를 한번에 하나씩 작성해서 파이프라인을 구축한 뒤에 이해한 것이 맞는지 시험한다.

● 해답과 설명

해답 head 명령어는 animals.txt 파일에서 첫 5 행을 추출한다. 그리고 나서, tail 명 령어로 이전 5 행에서 마지막 3 행을 추출된다. sort -r 명령어는 역순으로 정렬을 시키게 된다. 마지막으로 출력결과는 final.txt 파일에 방향변경하여 화면이 아닌 파일로 보내진다. 파일에 저장된 내용은 cat final.txt 명령어를 실행하면 확인이 가능하다. 파일에는 다음 내용이 저장되어야 한다:

```
2012-11-06, rabbit
2012-11-06, deer
2012-11-05, raccoon
```

예제 6.7 (파이프 구성하기). 이전 연습문제에 사용된 animals.txt 파일을 가지고 다음 명 령어를 실행한다:

```
$ cut -d , -f 2 animals.txt
```

콤마를 구분자로 각 행을 쪼개려고 하면 -d 플래그를 사용하고, -f 플래그는 각행의 두번 째 필드를 지정하게 되서 출력결과는 다음과 같다:

```
deer
rabbit
raccoon
rabbit
deer
fox
rabbit
bear
```

파일에 담겨 있는 동물이 무엇인지를 알아내려면, 다른 어떤 명령어가 파이프라인에 추 가되어야 하나요?(동물 이름에 어떠한 중복도 없어야 합니다.)

🥊 해답과 설명

```
$ cut -d , -f 2 animals.txt | sort | uniq
```

예제 6.8 (파이프 선택?). animals.txt 파일은 아래 형식으로 586줄로 구성되어 있다:

```
2012-11-05, deer
2012-11-05, rabbit
2012-11-05, raccoon
2012-11-06, rabbit
```

data-shell/data/ 현재 디렉토리로 가정하고, 다음 중 어떤 명령어가 동물 종류별로 전체 출현 빈도수를 나타내는 표를 작성하는데 사용하면 좋을까요?

```
    grep {deer, rabbit, raccoon, deer, fox, bear} animals.txt | wc -l
    sort animals.txt | uniq -c
    sort -t, -k2,2 animals.txt | uniq -c
    cut -d, -f 2 animals.txt | uniq -c
    cut -d, -f 2 animals.txt | sort | uniq -c
    cut -d, -f 2 animals.txt | sort | uniq -c | wc -l
```

€ 해답과 설명

정답은 5.

정답을 이해하는데 어려움이 있으면, (data-shell/data 디렉토리에 위치한 것을 확인한 후) 명령어 전체를 실행하거나, 파이프라인 일부를 실행해 본다.

6.6 사례: 파일 확인하기

앞에서 설명한 것처럼 Nelle은 분석기를 통해 시료를 시험해서 17개 파일을 northpacific-gyre/2012-07-03 디렉토리에 생성했다. 빠르게 건전성 확인하기 위해, 홈디렉토 리에서 시작해서, 다음과 같이 타이핑한다:

```
$ cd north-pacific-gyre/2012-07-03
$ wc -l *.txt
```

결과는 다음과 같은 18 행이 출력된다:

```
300 NENE01729A.txt
300 NENE01729B.txt
300 NENE01736A.txt
300 NENE01751A.txt
300 NENE01751B.txt
300 NENE01812A.txt
. . . . . .
```

이번에는 다음과 같이 타이핑한다:

```
$ wc -l *.txt | sort -n | head -n 5
240 NENE02018B.txt
300 NENE01729A.txt
300 NENE01729B.txt
300 NENE01736A.txt
300 NENE01751A.txt
```

이런, 파일중에 하나가 다른 것보다 60행이 짧다. 다시 돌아가서 확인하면, 월요일 아침 8:00 시각에 분석을 수행한 것을 알고 있다 — 아마도 누군가 주말에 기계를 사용했고, 다 시 재설정하는 것을 깜빡 잊었을 것이다. 시료를 다시 시험하기 전에 파일중에 너무 큰 데이터가 있는지를 확인한다:

```
$ wc -l *.txt | sort -n | tail -n 5
300 NENE02040B.txt
300 NENE02040Z.txt
```

300 NENE02043A.txt 300 NENE02043B.txt 5040 total

숫자는 예뻐 보인다 — 하지만 끝에서 세번째 줄에 'Z'는 무엇일까? 모든 시료는 'A' 혹은 'B'로 표시되어야 한다. 시험실 관례로 'Z'는 결측치가 있는 시료를 표식하기 위해 사용된다. 더 많은 결측 시료를 찾기 위해, 다음과 같이 타이핑한다:

\$ ls *Z.txt

NENE01971Z.txt NENE02040Z.txt

노트북의 로그 이력을 확인할 때, 상기 샘플 각각에 대해 깊이(depth) 정보에 대해서 기록된 것이 없었다. 다른 방법으로 정보를 더 수집하기에는 너무 늦어서, 분석에서 두 파일을 제외하기로 했다. rm 명령어를 사용하여 삭제할 수 있지만, 향후에 깊이(depth)정보가 관련없는 다른 분석을 실시할 수도 있다. 그래서 와일드 카드 표현식 *[AB]. txt을 사용하여 파일을 조심해서 선택하기로 한다. 언제나 그렇듯이, '*'는 임의 숫자의 문자를 매칭한다. [AB] 표현식은 'A'혹은 'B'를 매칭해서 Nelle이 가지고 있는 유효한 데이터 파일모두를 매칭한다.

예제 6.9 (불필요한 파일 제거하기). 저장공간을 절약하고자 중간 처리된 데이터 파일을 삭제하고 원본 파일과 처리 스크립트만 보관했으면 한다고 가정하자.

원본 파일은 .dat으로 끝나고, 처리된 파일은 .txt으로 끝난다. 다음 중 어떤 명령어가 처리과정에서 생긴 중간 모든 파일을 삭제하게 하는가?

- 1. rm ?.txt
- 2. rm *.txt
- 3. rm * .txt
- 4. rm *.*

♀ 정답과 설명

- 1. 한문자 .txt 파일을 제거한다.
- 2. 정답

- 3. 기호로 인해 현재 디렉토리 모든 파일과 디렉토리를 매칭시킨다. 그래서 *기호로 매칭되는 모든 것과 추가로 .txt 파일도 삭제한다.
- 4. . 기호는 임의 확장자를 갖는 모든 파일을 매칭시킨다. 따라서 . 기호는 모든 파일을 삭제한다.

예제 6.10 (와일드카드 표현식(Wildcard Expressions)). 와일드카드 표현식은 매우 복잡할 수 있지만, 종종 다소 장황할 수 있는 비용을 지불하고 간단한 구문만 사용해서 작성하 기도 한다.

data-shell/north-pacific-qyre/2012-07-03 디렉토리를 생각해 보자: *「AB].txt 와일드카 드 표현식은 A.txt 혹은 B.txt으로 끝나는 모든 파일을 매칭시킨다. 이 와일드카드 표현 식을 잊었다고 상상해보자:

- 1. [] 구문을 사용하지 않는 기본 와일드드카드 표현식으로 동일하게 파일을 매칭할 수 있을까? 힌트: 표현식이 하나 이상 필요할 수도 있다.
- 2. [] 구문을 사용하지 않고 작성한 표현식은 동일한 파일을 매칭한다. 두 출력결과의 작은 차이점은 무엇인가?
- 3. 최초 와잌드카드 표현식은 오류가 나지 않는데 어떤 상황에서 본인 표현식은 오류 메시지를 출력하는가?

● 해답과 설명

\$ ls *A.txt

\$ ls *B.txt

- 2. 새로운 명령어에서 나온 출력결과는 명령어가 두개라 구분된다.
- 3. A.txt로 끝나는 파일이 없거나 B.txt로 끝나는 파일이 없는 경우 그렇다.

6.1 연습문제

- 1. sort -n 명령어는 어떤 기능을 수행하나요?
 - (1) 파일의 내용을 알파벳 순으로 정렬한다.

- (2) 파일의 내용을 무작위로 재정렬한다.
- (3) 파일의 내용을 숫자 순으로 정렬한다.
- (4) 파일의 내용을 역순으로 정렬한다.
- (5) 파일의 내용을 대소문자를 구분하여 정렬한다.
- 2. head -n 1 명령어가 수행하는 작업은 무엇인가요?
 - (1) 파일의 마지막 행을 출력한다.
 - (2) 파일의 첫 번째 행을 출력한다.
 - (3) 파일의 첫 번째 열을 출력한다.
 - (4) 파일의 첫 1KB를 출력한다.
 - (5) 파일의 첫 1MB를 출력한다.
- 3. >와 » 연산자의 차이점은 무엇인가요?
 - (1) >는 파일에 문자열을 추가하고, »는 파일에 문자열을 덮어쓴다.
 - (2) >는 파일에 문자열을 덮어쓰고, »는 파일에 문자열을 추가한다.
 - (3) >는 파일을 삭제하고, »는 파일을 생성한다.
 - (4) >는 파일을 생성하고, »는 파일을 삭제한다.
 - (5) >와 »사이에는 별다른 차이가 없다.
- 4. sort -n 명령어가 숫자와 알파벳이 섞인 파일에 적용될 경우 어떻게 동작하는지 설명해보세요. 그리고 이를 보완하거나 다른 결과를 얻기 위해 사용할 수 있는 다른 명령어나 옵션은 무엇인지 제시하세요.
- 5. 리눅스의 입출력 리다이렉션에 대해 설명하세요. 특히 >, », <, | 등의 기호들이 어떤 의미를 가지고 어떻게 사용되는지에 대해 자세히 쓰세요. 또한, 이러한 리다이렉션 기능이유용한 상황의 예를 들어주세요.

제 7 장

루프(Loops)

반복적으로 명령어를 실행하게 함으로써 자동화를 통해서 루프는 생산성 향상에 핵심이 된다. 와일드카드와 탭 자동완성과 유사하게, 루프를 사용하면 타이핑 상당량(타이핑 실수)을 줄일 수 있다. 와일드카드와 탭 자동완성은 타이핑을 (타이핑 실수를)줄이는 두가지 방법이다. 또다른 것은 쉘이 반복해서 특정 작업을 수행하게 하는 것이다. basilisk.dat, unicorn.dat 등으로 이름 붙여진 게놈 데이터 파일이 수백개 있다고 가정하자. 이번 예제에서, 단지 두개 예제 파일만 있는 creatures 디렉토리를 사용할 것이지만 동일한 원칙은 훨씬 더 많은 파일에 즉시 적용될 수 있다. 디렉토리에 있는 파일을 변경하고 싶지만, 원본 파일을 original-basilisk.dat와 original-unicorn.dat으로 이름을 변경해서 저장한다. 하지만 다음 명령어를 사용할 수 없다:

\$ cp *.dat original-*.dat

왜냐하면 상기 두 파일 경우에 전개가 다음과 같이 될 것이기 때문이다:

\$ cp basilisk.dat unicorn.dat original-*.dat

상기 명령어는 파일을 백업하지 않고 대신에 오류가 발생된다:

cp: target `original-*.dat' is not a directory

cp 명령어는 입력값 두개 이상을 받을 때 이런 문제가 발생한다. 이런 상황이 발생할 때,

마지막 입력값을 디렉토리로 예상해서 모든 파일을 해당 디렉토리로 넘긴다. creatures 디렉토리에는 original-*.dat 라고 이름 붙은 하위 디렉토리가 없기 때문에, 오류가 생기다.

대신에, 리스트에서 한번에 연산작업을 하나씩 수행하는 **루프(loop)**를 사용할 수 있다. 교대로 각 파일에 대해 첫 3줄을 화면에 출력하는 단순한 예제가 다음에 나와 있다:

\$ for filename in basilisk.dat unicorn.dat

> do

> head -n 3 \$filename # 루프 내부에 들여쓰기는 가독성에 도움을 준다.

> done

COMMON NAME: basilisk

CLASSIFICATION: basiliscus vulgaris

UPDATED: 1745-05-02 COMMON NAME: unicorn

CLASSIFICATION: equus monoceros

UPDATED: 1738-11-24

♦ for 루프 내부에 코드 들여쓰기

for 루프 내부의 코드를 들여쓰는 것이 일반적인 관행이다. 들여쓰는 유일한 목적은 코드를 더 읽기 쉽게 하는 것 밖에 없다 - for 루프를 실행하는데는 꼭 필요하지는 않다.

쉘이 키워드 for를 보게 되면, 쉘은 리스트에 있는 각각에 대해 명령문 하나(혹은 명령문 집합)을 반복할 것이라는 것을 알게 된다. 루프를 반복할 때마다(iteration 이라고도 한다), 현재 작업하고 있는 파일 이름은 filename으로 불리는 변수(variable)에 할당된다. 리스트의 다음 원소로 넘어가기 전에 루프 내부 명령어가 실행된다. 루프 내부에서, 변수 이름 앞에 \$ 기호를 붙여 변수 값을 얻는다: \$ 기호는 쉘 해석기가 변수명을 텍스트나 외부 명령어가 아닌 변수로 처리해서 값을 해당 위치에 치환하도록 지시한다.

이번 경우에 리스트는 파일이름이 두개다: basilisk.dat, unicorn.dat. 매번 루프가 돌 때마다 파일명을 filename 변수에 할당하고 head 명령어를 실행시킨다.즉,루프가 첫번째돌 때 \$filename은 basilisk.dat이 된다.쉘 해석기는 basilisk.dat 파일에 head 명령어를

실행시켜서 basilisk.dat 파일의 첫 3줄을 화면에 출력시킨다.

두번째 반복에서, \$filename은 unicorn.dat이 된다. 이번에는 쉘이 head 명령어를 unicorn.dat 파일에 적용시켜 unicorn.dat 파일 첫 3줄을 화면에 출력시킨다. 리스트에 워소가 두개라서, 쉘은 for 루프를 빠져나온다.

변수명을 분명히 구분하는데, 중괄호 내부에 변수명을 넣어서 변수로 사용하는 것도 가능하다: \$filename 은 \${filename}와 동치지만, \${file}name와는 다르다. 이 표기법을 다른 사람 프로그램에서 찾아볼 수 있다.

예제 7.1 (루프 내부의 변수). 이번 예제는 data-shell/molecules 디렉토리를 가정한다. ls 명령어를 던지면 출력결과는 다음과 같다:

```
cubane.pdb ethane.pdb methane.pdb octane.pdb pentane.pdb propane.pdb
```

다음 코드의 출력결과는 어떻게 나오는가?

```
$ for datafile in *.pdb
> do
> ls *.pdb
> done
```

이제 다음 코드의 출력결과는 무엇인가?

```
$ for datafile in *.pdb
> do
> ls $datafile
> done
```

왜 상기 두 루프 실행결과는 다를까?

♥ 해답과 설명

첫번째 코드 블록은 루프를 돌릴 때마다 동일한 출력결과를 출력한다. 배쉬는 루프 몸통 내부 와일드카드 *.pdb을 확장해서 .pdb로 끝나는 모든 파일을 매칭시킨다. 확장된 루프는 다음과 같이 생겼다:

```
$ for datafile in cubane.pdb ethane.pdb methane.pdb octane.pdb

→ pentane.pdb propane.pdb

  > do
    ls cubane.pdb ethane.pdb methane.pdb octane.pdb pentane.pdb

→ propane.pdb

  > done
  cubane.pdb ethane.pdb methane.pdb octane.pdb pentane.pdb propane.pdb
  cubane.pdb ethane.pdb methane.pdb octane.pdb pentane.pdb propane.pdb
두번째 코드 블록은 루프를 돌 때마다 다른 파일을 출력한다. datafile 파일 변수값
이 $datafile을 통해 평가되고 ls 명령어를 사용해서 파일 목록을 출력하게 된다.
  cubane.pdb
  ethane.pdb
  methane.pdb
```

7.0.1 프롬프트 따라가기

octane.pdb pentane.pdb propane.pdb

루프안에서 타이핑을 할 때, 쉘 프롬프트가 \$에서 >으로 바뀐다. 두번째 프롬프트는, >, 온전한 명령문 타이핑이 끝마치지 않았음을 상기시키려고 다르게 표기된다. 세미콜론; 을 사용해서 두 명령어로 구성된 문장을 단일 명령줄로 단순화한다.

♦ 주의

동일한 기호, 하지만 다른 의미

쉘 프롬프트로 > 기호가 사용되는 것을 확인했지만, > 기호는 출력결과를 방향변경 (redirect) 하는데도 사용된다. 유사하게 \$ 기호를 쉘 프롬프트로 사용했지만, 앞에서 살펴봤듯이, 쉘로 하여금 변수값을 추출하는데도 사용된다.

쉘이 > 혹은 \$ 기호를 출력하게 되면, 사용자가 뭔가 타이핑하길 기대하고 있다는 것으로 해당 기호는 프롬프트를 의미한다.

사용자 본인이 > 혹은 \$ 기호를 타이핑하게 되면, 출력결과를 방향변경하거나 변수 값을 끄집어내는 지시를 쉘에 전달하게 된다.

data-shell/creatures 디렉토리의 예제로 돌아가자. 사람 코드를 읽는 독자에게 목적을 좀더 명확히 하기 위해서 루프의 변수명을 filename로 했다. 쉘 자체는 변수명이 어떻게 작명되든지 문제삼지 않는다. 만약 루프를 다음과 같이 작성하거나:

- \$ for x in basilisk.dat unicorn.dat
- > do
- > head -n 3 \$x
- > done

혹은:

- \$ for temperature in basilisk.dat unicorn.dat
- > do
- > head -n 3 \$temperature
- > done

둘다 정확하게 동일하게 동작한다. 이렇게는 절대 하지 마세요. 사람이 프로그램을 이해할 수 있을 때만 프로그램이 유용하기 때문에, (x같은) 의미없는 이름이나, (temperature 같은) 오해를 줄 수 있는 이름은 오해를 불러일으켜서 독자가 생각하기에 당연히 프로그램이 수행해야 할 작업을 프로그램이 수행하지 못하게 할 가능성을 높인다.

예제 7.2 (파일 집합 제한걸기). data-shell/molecules 디렉토리에서 다음 루프를 실행하게 되면 출력결과는 어떻게 될까?

\$ for filename in c*

- > do
- > ls \$filename
- > done
- 1. 어떤 파일도 출력되지 않는다.
- 2. 모든 파일이 출력된다.
- 3. cubane.pdb, octane.pdb, pentane.pdb 파일만 출력된다.
- 4. cubane.pdb 파일만 출력된다.

♥ 해답과 설명

정답은 4. 와일드카드 * 문자는 0 혹은 그 이상 문자를 매칭하게 된다. 따라서, 문자 c로 시작하는 문자 다음에 0 혹은 그 이상 문자를 갖는 모든 파일이 매칭된다.

대신에 다음 명령어를 사용하면 출력결과는 어떻게 달라지나?

- \$ for filename in *c*
- > do
- > ls \$filename
- > done
- 1. 동일한 파일이 출력된다.
- 2. 이번에는 모든 파일이 출력된다.
- 3. 이번에는 어떤 파일도 출력되지 않는다.
- 4. cubane.pdb 와 octane.pdb 파일이 출력된다.
- 5. octane.pdb 파일만 출력된다.

♥ 해답과 설명

정답은 4. 와일드카드 * 문자는 0 혹은 그 이상 문자를 매칭하게 된다. 따라서, c 앞에 0 혹은 그 이상 문자가 올 수 있고, c 문자 다음에 0 혹은 그 이상 문자가 모두 매칭된다.

shell-lesson-data/creatures 디렉토리에서 예제를 계속해서 학습해보자. 다음에 좀더 복

잡한 루프가 있다:

```
$ for filename in *.dat
> do
> echo $filename
> head -n 100 $filename | tail -n 20
> done
```

쉘이 *.dat을 전개해서 쉘이 처리할 파일 리스트를 생성한다. 그리고 나서 루프 몸통 (loop body) 부분이 파일 각각에 대해 명령어 두개를 실행한다. 첫 명령어 echo는 명령라인 매개변수를 표준 출력으로 화면에 뿌려준다. 예를 들어:

```
$ echo hello there
```

상기 명령은 다음과 같이 출력된다:

```
hello there
```

이 사례에서, 쉘이 파일 이름으로 \$filename을 전개했기 때문에, echo \$filename은 단지 파일 이름만 화면에 출력한다. 다음과 같이 작성할 수 없다는 것에 주의한다:

```
$ for filename in *.dat
> do
> $filename
> head -n 100 $filename | tail -n 20
> done
```

왜냐하면, \$filename이 basilisk.dat으로 전개될 때 루프 처음에 쉘이 프로그램으로 인식한 basilisk.dat를 실행하려고 하기 때문이다.마지막으로, head와 tail 조합은 어떤 파일이 처리되든 81-100줄만 선택해서 화면에 뿌려준다.(파일이 적어도 100줄로 되었음을 가정)

7.1 파일, 디렉토리, 변수 등 이름에 공백

공백(whitespace)을 사용해서 루프를 돌릴 때 리스트의 각 원소를 구별했다. 리스트 원소 중 일부가 공백을 갖는 경우, 해당 원소를 인용부호로 감싸서 사용해야 된다. 데이터 파일이 다음과 같은 이름으로 되었다고 가정하자:

```
red dragon.dat
purple unicorn.dat
```

다음을 사용하여 파일을 처리하려고 한다면:

```
$ for filename in "red dragon.dat" "purple unicorn.dat"
> do
> head -n 100 "$filename" | tail -n 3
> done
```

파일명에 공백(혹은 다른 특수 문자)를 회피하는 것이 더 단순하다. 상기 파일은 존재하지 않는다. 그래서 상기 코드를 실행하게 되면, head 명령어는 파일을 찾을 수가 없어서 예상되는 파일명을 보여주는 오류 메시지가 반환된다:

```
head: cannot open 'red dragon.dat' for reading: No such file or directory head: cannot open 'purple unicorn.dat' for reading: No such file or directory
```

상기 루프 내부 \$filename 파일명 주위 인용부호를 제거하고 공백 효과를 살펴보자. creatures 디렉토리에서 코드를 실행시키게 되면 unicorn.dat 파일에 대한 결과를 루프 명령어 실행 결과를 얻게 됨에 주목한다:

```
head: cannot open 'red' for reading: No such file or directory
head: cannot open 'dragon.dat' for reading: No such file or directory
head: cannot open 'purple' for reading: No such file or directory
CGGTACCGAA
AAGGGTCGCG
```

CAAGTGTTCC

원래 파일 복사문제로 되돌아가서, 다음 루프를 사용해서 문제를 해결해 보자:

```
$ for filename in *.dat
> do
> cp $filename original-$filename
> done
```

상기 루프는 cp 명령문을 각 파일이름에 대해 실행한다. 처음에 \$filename이 basilisk.dat 로 전개될 때, 쉘은 다음을 실행한다:

```
cp basilisk.dat original-basilisk.dat
```

두번째에는 명령문은 다음과 같다:

```
cp unicorn.dat original-unicorn.dat
```

cp 명령어는 아무런 출력결과도 만들어내지 않기 때문에, 루프가 제대로 돌아가는지 확 인하기 어렵다. echo로 명령문 앞에 위치시킴으로써, 명령문 각각이 제대로 동작되고 있 는 확인하는 것이 가능하다. 다음 도표를 통해서 스크립트가 동작할 때 어떤 작업이 수 행하고 있는지 상술하고 있다. 또한 echo 명령어를 사려깊이 사용하는 것이 어떻게 훌륭 한 디버깅 기술이 되는지도 보여주고 있다.

7.1.1 역사(history)를 아는 사람은 반복할 수 있다.

앞선 작업을 반복하는 또다른 방법은 history 명령어를 사용하는 것이다. 실행된 마지 막 수백개 명령어 리스트를 얻고 나서, 이들 명령어 중 하나를 반복실행하기 위해서! 123("123"은 명령 숫자로 교체된다.)을 사용한다. 예를 들어 Nelle이 다음과 같이 타이핑 하다면:

```
$ history | tail -n 5
 456 ls -l NENEO*.txt
```

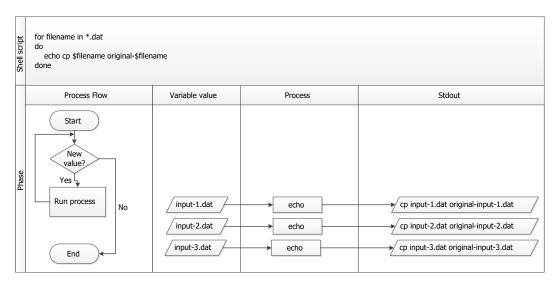


그림 7.1: For Loop in Action

- 457 rm stats-NENE01729B.txt.txt
- 458 bash goostats NENE01729B.txt stats-NENE01729B.txt
- 459 ls -1 NENEO*.txt
- 460 history

그리고 나서, 단순히 !458을 타이핑함으로써, NENE01729B. txt 파일에 goostats을 다시 실 행할 수 있게 된다.

♦ 다른 이력(history) 명령어

- 이력(history)에 접근하는 단축 명령어가 다수 존재한다.
 - Ctrl-R 단축키는 "reverse-i-search" 이력 검색모드로 입력한 텍스트와 매칭되 는 가장 최슨 명령어를 이력에서 찾아서 제시한다. Ctrl-R 단축키를 한번 혹 은 그 이상 누르게 되면 그 이전 매칭을 검색해 준다.
 - !! 명령어는 바로 직전 명령어를 불러온다. (키보드 윗화살표를 사용하는 것 보다 더 편리할수도 편리하지 않을 수도 있다.)
 - !\$ 명령어는 마지막 명령문의 마지막 단어를 불러온다. 기대했던 것보다 훨씬 유용할 수 있다: bash goostats NENE01729B.txt stats-NENE01729B.txt 명령문을 실행한 후에 less !\$을 타이핑하게 되면 stats-NENE01729B.txt 파일을 찾아준 다. 키보드 위화살표를 눌러 명령라인을 편집하는 것보다 훨씬 빠르다.

예제 7.3 (루프 내부에서 파일에 저장하기 - 1부). data-shell/molecules 디렉토리에 있다고 가정하자. 다음 루프의 효과는 무엇인가?

```
$ for alkanes in *.pdb
> do
> echo $alkanes
> cat $alkanes > alkanes.pdb
> done
```

- 1. fructose.dat, glucose.dat, sucrose.dat을 출력하고, sucrose.dat에서 나온 텍스트를 xylose.dat에 저장된다.
- 2. fructose.dat, glucose.dat, sucrose.dat을 출력하고, 모든 파일 3개에서 나온 텍스트를 합쳐 xylose.dat에 저장된다.
- 3. fructose.dat, glucose.dat, sucrose.dat, xylose.dat을 출력하고, sucrose.dat에서 나 온 텍스트를 xylose.dat에 저장된다.
- 4. 위 어느 것도 아니다.

€ 해답과 해설

1. 순차적으로 각 파일의 텍스트가 alkanes.pdb 파일에 기록된다. 하지만, 루프가 매번 반복될 때마다 파일에 덮어쓰기가 수행되어서 마지막 alkanes.pdb 파일 텍스트만 alkanes.pdb 파일에 기록된다.

예제 7.4 (루프 내부에서 파일에 저장하기 - 2부). 이번에도 data-shell/molecules 디렉토리에 있다고 가정하고, 다음 루프 실행 출력결과는 무엇일까?

```
$ for datafile in *.pdb
> do
> cat $datafile >> all.pdb
> done
```

- 1. cubane.pdb, ethane.pdb, methane.pdb, octane.pdb, pentane.pdb 파일에 나온 모든 모든 텍스트가 하나로 붙여져서 all.pdb 파일에 저장된다.
- 2. ethane.pdb 파일에 나온 텍스트만 all.pdb 파일에 저장된다.
- 3. cubane.pdb, ethane.pdb, methane.pdb, octane.pdb, pentane.pdb, propane.pdb 파일에서

나온 모든 텍스트가 하나로 풀여져서 all.pdb 파일에 저장된다.

4. cubane.pdb, ethane.pdb, methane.pdb, octane.pdb, pentane.pdb, propane.pdb 파일에서 나온 모든 텍스트가 화면에 출력되고 all.pdb 파일에 저장된다.

● 해답과 설명

정답은 3. 명령어 실행 출력결과를 방향변경하여 덮었는 것이 아니라 >> 기호는 파일에 덧붙인다. cat 명령어에서 나온 출력결과가 파일로 방향변경되어 어떤 출력결과도 화면에 출력되지는 않는다.

에제 7.5 (시운전(Dry Run)). 루프는 한번에 많은 작업을 수행하는 방식이다 — 만약 잘 못된 것이 있다면, 한번에 실수를 대단히 많이 범하게 된다. 루프가 수행하는 작업을 점 검하는 한 방법이 실제로 루프를 돌리는 대신에 echo 명령어를 사용하는 것이다. 실제로 명령어를 실행하지 않고, 다음 루프가 실행할 명령어를 머릿속으로 미리보고자 한다고 가정한다:

```
$ for file in *.pdb
> do
> analyze $file > analyzed-$file
> done
```

아래 두 루프 사이에 차이는 무엇이고, 어느 것을 시운전으로 실행하고 싶은가?

```
# Version 1
$ for file in *.pdb
> do
> echo analyze $file > analyzed-$file
> done

# Version 2
$ for file in *.pdb
> do
> echo "analyze $file > analyzed-$file"
```

> done

€ 해답과 설명

두번째 버젼을 실행하면 좋을 것이다. 달러 기호로 접두명을 주었기 때문에 루프 변수를 확장해서 인용부호로 감싼 모든 것을 화면에 출력한다.

첫번째 버전은 echo analyze \$file 명령을 수행해서 analyzed-\$file 파일로 출력결과를 방향변경하여 저장시킨다. 따라서 파일이 쭉 자동생성된다:analyzed-cubane.pdb, analyzed-ethane.pdb ···

두가지 버젼을 직접 실행해보고 출력결과를 살펴보자! analyzed-*.pdb 파일을 열어서 파일에 기록된 내용도 살펴보다.

::::: {#exr-nested-loop} ### 중첩루프(Nested Loops)

다른 화합물과 다른 온도를 갖는 조합을 해서, 각 반응율 상수를 측정하는 실험을 조직하도록 이에 상응하는 디렉토리 구조를 갖추고자 한다. 다음 코드 실행결과는 어떻게 될까?

- \$ for species in cubane ethane methane
- > do
- > for temperature in 25 30 37 40
- > do
- > mkdir \$species-\$temperature
- > done
- > done

● 해답과 설명

중첩 루프(루프 내부에 루프가 포함됨)를 생성하게 된다. 외부 루프에 각 화학물이, 내부 루프(중첩된 루프)에 온도 조건을 반복하게 되서, 화학물과 온도를 조합한 새 로운 디렉토리가 쭉 생성된다.

직접 코드를 실행해서 어떤 디렉토리가 생성되는지 확인한다!

7.2 사례: 많은 파일 처리하기

넬(Nelle)은 이제 goostats 프로그램(논문 지도교수가 작성한 쉘 스크립트)을 사용해서 데이터 파일을 처리할 준비가 되었다. goostats 프로그램은 표본추출 단백질 파일에서 통계량을 산출하는데 인자를 두개 받는다:

- 1. 입력파일 (원본 데이터를 포함)
- 2. 출력파일 (산출된 통계량을 저장)

아직 쉘을 어떻게 사용하는지 학습단계에 있기 때문에, 단계별로 요구되는 명령어를 차근히 작성하기로 마음먹었다. 첫번째 단계는 적합한 파일을 선택했는지를 확인하는 것이다 — 'Z'가 아닌 'A' 혹은 'B'로 파일이름이 끝나는 것이 적합한 파일이라는 것을 명심한다. 홈 디렉토리에서 시작해서, 박사과정 Nelle이 다음과 같이 타이핑한다:

```
$ cd north-pacific-gyre/2012-07-03
$ for datafile in NENE*[AB].txt
> do
> echo $datafile
> done

NENE01729A.txt
NENE01729B.txt
NENE01736A.txt
...
NENE02043A.txt
NENE02043B.txt
```

다음 단계는 goostats 분석 프로그램이 생성할 파일이름을 무엇으로 할지 결정하는 것이다. "stats"을 각 입력 파일에 접두어로 붙이는 것이 간단해 보여서, 루프를 변경해서 작업을 수행하도록 한다:

```
$ for datafile in NENE*[AB].txt
> do
> echo $datafile stats-$datafile
```

> done

```
NENE01729A.txt stats-NENE01729A.txt
NENE01729B.txt stats-NENE01729B.txt
NENE01736A.txt stats-NENE01736A.txt
NENE02043A.txt stats-NENE02043A.txt
NENE02043B.txt stats-NENE02043B.txt
```

goostats을 아직 실행하지는 않았지만, 이제 확신할 수 있는 것은 올바른 파일을 선택해 서, 올바른 출력 파일이름을 생성할 수 있다는 점이다.

명령어를 반복적으로 타이핑하는 것은 귀찮은 일이지만, 더 걱정이 되는 것은 Nelle이 타 이핑 실수를 하는 것이다. 그래서 루프를 다시 입력하는 대신에 위쪽 화살표를 누른다. 위쪽 화살표에 반응해서 컴퓨터 쉘은 한줄에 전체 루프를 다시 보여준다. (스크립트 각 부분이 구분되는데 세미콜론이 사용됨):

```
$ for datafile in NENE*[AB].txt; do echo $datafile stats-$datafile; done
```

왼쪽 화살표 키를 사용해서, Nelle은 echo 명령어를 bash goostats으로 변경하고 백업한 다:

```
$ for datafile in NENE*[AB].txt; do bash goostats $datafile stats-$datafile;

→ done
```

엔터키를 누를 때, 쉘은 수정된 명령어를 실행한다. 하지만, 어떤 것도 일어나지 않는 것 처럼 보인다 — 출력이 아무것도 없다. 잠시뒤에 Nelle은 작성한 스크립트가 화면에 아 무것도 출력하지 않아서, 실행되고 있는지, 얼마나 빨리 실행되는지에 대한 정보가 없다 는 것을 깨닫는다. 컨트롤+C(Control-C)를 눌러서 작업을 종료하고, 반복할 명령문을 위 쪽 화살표로 선택하고, 편집해서 다음과 같이 작성한다:

```
$ for datafile in NENE*[AB].txt; do echo $datafile; bash goostats $datafile

    stats-$datafile; done
```

시작과 끝

쉘에 ^A, 콘트롤+A(Control-A, Ctrl-a)를 타이핑해서 해당 라인 처음으로 가고, ^E(Ctrl-e, Control-E)를 쳐서 라인의 끝으로 이동한다.

이번에 프로그램을 실행하면, 매 5초간격으로 한줄을 출력한다:

NENE01729A.txt

NENE01729B.txt

NENE01736A.txt

. . .

1518 곱하기 5초를 60으로 나누면, 작성한 스크립트를 실행하는데 약 2시간 정도 소요된 다고 볼 수 있다. 마지막 점검으로, 또다른 터미널 윈도우를 열어서, north-pacific-gyre/2012-07-03 디렉토리로 가서, cat stats-NENE01729B. txt을 사용해서 출력파일 중 하나를 면밀히 조사한다. 출력결과가 좋아보인다. 그래서 커피를 마시고 그동안 밀린 논문을 읽기로 하다.

7.1 연습문제

- 1. 다음 중 쉘 스크립트에서 for 루프의 일반적인 구조에 대한 설명으로 가장 올바른 것은 무엇인가요?
 - (1) for 키워드 다음에는 루프가 실행될 명령어가 따라옵니다.
 - (2) 리스트의 각 아이템에 대해 루프를 실행하려면 for 키워드를 사용해야 합니다.
 - (3) 루프 내에서는 \$기호를 이용해 변수의 이름을 호출할 수 있습니다.
 - (4) done 키워드를 사용하면 루프가 종료되고, 이후 코드가 실행됩니다.
 - (5) 모든 위의 내용이 올바릅니다.
- 2. 다음 중 변수 이름에 대한 설명으로 가장 올바른 것은 무엇인가요?
 - (1) 변수 이름은 \$filename과 \$filename의 두 가지 형태로 사용할 수 있습니다.
 - (2) 변수 이름은 코드를 작성하는 사람과 읽는 사람이 이해할 수 있는 의미 있는 이름을 가져야 합니다.
 - (3) 쉘은 변수 이름을 식별하지 못하므로 변수 이름은 프로그램의 기능에 영향을 주지 않습니다.

- (4) 변수 이름은 항상 temperature와 같은 구체적인 이름이어야 합니다.
- (5)(1)와(2)의 내용이 모두 올바릅니다
- 3. 다음 중 쉘 스크립트에서 for문을 사용하여 파일에서 특정 정보를 추출하는 코드를 가장 잘 설명한 것은 무엇인가요?
 - (1) for filename in basilisk.dat minotaur.dat unicorn.dat; do echo \$filename; head -n 2 \$filename | tail -n 1; done
 - (2) for filename in basilisk.dat minotaur.dat unicorn.dat; do tail -n 2 \$filename | head -n 1; done
 - (3) for file in basilisk.dat minotaur.dat unicorn.dat; do echo \$file; tail -n 2 \$file | head -n 1; done
 - (4) for file in basilisk.dat minotaur.dat unicorn.dat; do echo \$file; head -n 2 \$file | tail -n 1; done
 - (5) for filename in basilisk.dat minotaur.dat unicor
- 4. 다음 중 쉘 스크립트에서 변수 이름을 정할 때 고려해야 할 사항을 가장 잘 설명한 것은 무엇인가요?
 - (1) 변수 이름은 반드시 x나 y와 같은 한 글자로 이루어져야 합니다.
 - (2) 변수 이름은 반드시 해당 코드를 작성한 사람이 이해할 수 있도록 해야 합니다.
 - (3) 변수 이름은 반드시 해당 코드를 읽는 사람이 이해할 수 있도록 해야 합니다.
 - (4) 변수 이름은 항상 filename 또는 file로 지정해야 합니다.
 - (5) 변수 이름은 반드시 해당 코드를 작성한 사람과 해당 코드를 읽는 사람 모두가 이 해할 수 있도록 해야 합니다.
- 5. 쉘 스크립트에서 0부터 9까지의 숫자를 출력하는 for loop을 작성하세요.
- 6. basilisk.dat, minotaur.dat, unicorn.dat 세 파일에 대하여, 각 파일의 두 번째 줄을 출력하는 쉘 스크립트의 for loop을 작성하세요.

제 8 장

쉘 스크립트

마침내 쉘을 그토록 강력한 프로그래밍 환경으로 탈바꾼할 준비가 되었다. 자주 반복적으로 사용되는 명령어들을 파일에 저장시키고 나서, 단 하나의 명령어를 타이핑함으써 나중에 이 모든 연산 작업작업을 다시 재실행할 수 있다. 역사적 이유로 파일에 저장된 명령어 꾸러미를 통상 쉘 스크립트(shell script)라고 부르지만, 실수로 그렇게 부르는 것은 아니다: 실제로 작은 프로그램이다.

molecules/ 디렉토리로 돌아가서 middle.sh 파일에 다음 행을 추가하게 되면 쉘스크립트 가 된다:

- \$ cd molecules
- \$ nano middle.sh

nano middle.sh 명령어는 middle.sh 파일을 텍스트 편집기 "nano"로 열게 한다. (편집기 프로그램은 쉘 내부에서 실행된다.) middle.sh 파일이 존재하지 않는 경우, middle.sh 파일을 생성시킨다. 텍스트 편집기를 사용해서 직접 파일을 편집한다 - 단순히 다음 행을 삽입시킨다:

```
head -n 15 octane.pdb | tail -n 5
```

앞서 작성한 파이프에 변형이다: octane.pdb 파일에서 11-15 행을 선택한다. 기억할 것은 명령어로서 실행하지 않고: 명령어를 파일에 적어 넣는다는 것이다.

그리고 나서 나노 편집기에서 Ctrl-0를 눌러 파일을 저장하고, 나노 편집기에서 Ctrl-X를 눌러 텍스트 편집기를 빠져나온다. molecules 디렉토리에 middle.sh 파일이 포함되어 있는지 확인한다.

Once we have saved the file, we can ask the shell to execute the commands it contains. Our shell is called bash, so we run the following command:

파일을 저장하면, 쉘로 하여금 파일에 담긴 명령어를 실행하도록 한다. 지금 쉘은 bash라서, 다음과 같이 다음 명령어를 실행시킨다:

\$ bash middle.sh

ATOM	9	Н	1	-4.502	0.681	0.785	1.00	0.00
MOTA	10	Н	1	-5.254	-0.243	-0.537	1.00	0.00
MOTA	11	Н	1	-4.357	1.252	-0.895	1.00	0.00
ATOM	12	Н	1	-3.009	-0.741	-1.467	1.00	0.00
ATOM	13	Н	1	-3.172	-1.337	0.206	1.00	0.00

아니나 다를까, 스크립트의 출력은 정확하게 파이프라인을 직접적으로 실행한 것과 동 일하다.

♦ 텍스트 vs. 텍스트가 아닌 것 아무거나

종종 마이크로소프트 워드 혹은 리브르오피스 Writer 프로그램을 "텍스트 편집기"라고 부른다. 하지만, 프로그래밍을 할때 조금더 주의를 기울일 필요가 있다. 기본 디폴트로, 마이크로소프트 워드는 .docx 파일을 사용해서 텍스트를 저장할 뿐만 아니라, 글꼴, 제목, 등등의 서식 정보도 함께 저장한다. 이런 추가 정보는 문자로 저장되지 않아서, head 같은 도구에게는 무의미하다: head 같은 도구는 입력 파일에 문자, 숫자, 표준 컴퓨터 키보드 특수문자만이 포함되어 있는 것을 예상한다. 따라서, 프로그램을 편집할 때, 일반 텍스트 편집기를 사용하거나, 혹은 일반 텍스트로 파일을 저장하도록 주의한다.

만약 임의 파일의 행을 선택하고자 한다면 어떨까요? 파일명을 바꾸기 위해서 매번 middle.sh을 편집할 수 있지만, 단순히 명령어를 다시 타이핑하는 것보다 아마 시간이 더 걸릴 것이다. 대신에 middle.sh을 편집해서 좀더 다양한 기능을 제공하도록 만들어보자:

\$ nano middle.sh

나노 편집기로 octane.pdb을 \$1으로 불리는 특수 변수로 변경하자:

```
head -n 15 "$1" | tail -n 5
```

쉘 스크립트 내부에서, \$1은 "명령라인의 첫 파일 이름(혹은 다른 인자)"을 의미한다. 이 제 스크립트를 다음과 같이 바꿔 실행해 보자:

\$ bash middle.sh octane.pdb

ATOM	9	Н	1	-4.502	0.681	0.785	1.00	0.00
ATOM	10	Н	1	-5.254	-0.243	-0.537	1.00	0.00
ATOM	11	Н	1	-4.357	1.252	-0.895	1.00	0.00
ATOM	12	Н	1	-3.009	-0.741	-1.467	1.00	0.00
ATOM	13	Н	1	-3.172	-1.337	0.206	1.00	0.00

혹은 다음과 같이 다른 파일에 대해 스크립트 프로그램을 실행해 보자:

\$ bash middle.sh pentane.pdb

```
1
                           1.324 0.350 <del>-1.332</del> 1.00 0.00
MOTA
        9 H
MOTA
        10 H
                      1
                            1.271 1.378 0.122 1.00 0.00
       11 H
                           -0.074 -0.384 1.288 1.00 0.00
MOTA
                     1
                     1
                           -0.048 -1.362 -0.205 1.00 0.00
MOTA
        12 H
MOTA
        13 H
                            -1.183
                                   0.500 - 1.412 1.00 0.00
```

♦ 인자 주위를 이중 인용부호로 감싸기

파일명에 공백이 포함된 경우 루프 변수 내부에 이중 인용부호로 감싼 것과 동일한 사유로, 파일명에 공백이 포함된 경우 이중 인용부호로 \$1을 감싼다.

하지만, 매번 줄 범위를 조정할 때마다 여전히 middle.sh 파일을 편집할 필요가 있다. 이

문제를 특수 변수 \$2 와 \$3 을 사용해서 고쳐보자: head, tail 명령어에 해당 줄수를 출력하도록 인자로 넘긴다.

\$ nano middle.sh

```
head -n "$2" "$1" | tail -n "$3"
```

이제 다음을 실행시킨다:

\$ bash middle.sh pentane.pdb 15 5

```
MOTA
       9 H
            1 1.324 0.350 -1.332 1.00 0.00
                       1.271 1.378 0.122 1.00 0.00
MOTA
      10 H
                 1
                       -0.074 -0.384 1.288 1.00 0.00
MOTA
     11 H
                 1
                 1
                       -0.048 -1.362 -0.205 1.00 0.00
MOTA
      12 H
MOTA
      13 H
                 1
                       -1.183 0.500 -1.412 1.00 0.00
```

명령문의 인자를 변경함으로써 스크립트 동작을 바꿀 수 있게 된다:

\$ bash middle.sh pentane.pdb 20 5

```
1 -1.259 1.420 0.112 1.00 0.00
MOTA
      14 H
MOTA
      15 H
                  1
                        -2.608 -0.407 1.130 1.00 0.00
                       -2.540 -1.303 -0.404 1.00 0.00
MOTA
                  1
     16 H
MOTA
      17 H
                  1
                       -3.393 0.254 -0.321 1.00 0.00
TER
      18
                   1
```

제대로 동작하지만, middle.sh 쉘스크립트를 읽는 다른 사람은 잠시 시간을 들여, 스크립트가 무엇을 수행하는지 알아내야 할지 모른다. 스크립트를 상단에 **주석(comments)**을 추가해서 좀더 낫게 만들 수 있다:

\$ nano middle.sh

```
# Select lines from the middle of a file.
# Usage: bash middle.sh filename end_line num_lines
head -n "$2" "$1" | tail -n "$3"
```

주석은 #문자로 시작하고 해당 행 끝까지 주석으로 처리된다. 컴퓨터는 주석을 무시하지만, 사람들이(미래의 본인 자신도 포함) 스크립트를 이해하고 사용하는데 정말 귀중한 존재다. 유일한 단점은 스크립트를 변경할 때마다, 주석이 여전히 유효한지 확인해야 된다는 점이다: 잘못된 방향으로 독자를 오도하게 만드는 설명은 아무것도 없는 것보다 더나쁘다.

만약 많은 파일을 단 하나 파이프라인으로 처리하고자 한다면 어떨까? 예를 들어, .pdb 파일을 길이 순으로 정렬하려면, 다음과 같이 타이핑한다:

wc -l은 파일에 행갯수를 출력하고(wc는 'word count'로 -l플래그를 추가하면 'count lines' 의미가 됨을 상기한다), sort -n은 숫자순으로 파일의 행갯수를 정렬한다. 파일에 담을 수 있지만, 현재 디렉토리에 .pdb 파일만을 정렬한다. 다른 유형의 파일에 대한 정렬된 목록을 얻으려고 한다면, 스크립트에 이 모든 파일명을 얻는 방법이 필요하다. \$1, \$2 등 등은 사용할 수 없는데, 이유는 얼마나 많은 파일이 있는지를 예단할 수 없기 때문이다. 대신에, 특수 변수 \$@을 사용한다. \$@은 "쉘 스크립트 모든 명령-라인 인자"를 의미한다. 공백을 포함한 매개변수를 처리하려면 이중 인용부호로 \$@을 감싸두어야 된다. ("\$@"은 "\$1" "\$2" … 와 동치다). 예제가 다음에 있다:

\$ nano sorted.sh

```
# Sort filenames by their length.
# Usage: bash sorted.sh one_or_more_filenames
wc -l "$@" | sort -n
```

실행방법과 실행결과는 다음과 같다.

```
$ bash sorted.sh *.pdb ../creatures/*.dat

9 methane.pdb
12 ethane.pdb
15 propane.pdb
20 cubane.pdb
21 pentane.pdb
30 octane.pdb
163 ../creatures/basilisk.dat
163 ../creatures/unicorn.dat
```

예제 8.1 (유일무이한 개체 목록으로 나열). 정훈이는 데이터 파일 수백개를 갖고 있는데, 각각은 다음과 같은 형식을 가지고 있다:

```
2013-11-05,deer,5

2013-11-05,rabbit,22

2013-11-05,raccoon,7

2013-11-06,rabbit,19

2013-11-06,deer,2

2013-11-06,fox,1

2013-11-07,rabbit,18

2013-11-07,bear,1
```

data-shell/data/animal-counts/animals.txt 파일을 대상으로 예제를 작성한다. 임의 파일이름을 명령-라인 인자로 갖는 species.sh 이름의 쉘 스크립트를 작성하라. cut, sort, uniq를 사용해서 각각의 파일별로 나오는 유일무이한 개체에 대한 목록을 화면에 출력하세요.

🥊 해답과 설명

```
# csv 파일에 유일무이한 개체를 찾는 스크립트로 개체는 두번째 데이터 필드
4 가 된다.
```

스크립트는 명령라인 인자로 모든 파일명을 인자로 받는다.

모든 파일에 대해 루프를 돌려 반복한다.

```
for file in $@

do

echo "Unique species in $file:"
# 개체명을 추출한다.

cut -d , -f 2 $file | sort | uniq

done
```

◊ 왜 쉘 스크립트가 어떤 작업도 수행하지 않을까?

스크립트가 아주 많은 파일을 처리하고 했지만, 어떠한 파일 이름도 부여하지 않는다면 무슨 일이 발생할까? 예를 들어, 만약 다음과 같이 타이핑한다면 어떻게 될까요?:

```
$ bash sorted.sh
```

하지만 *.dat (혹은 다른 어떤 것)를 타이핑하지 않는다면 어떨까요? 이 경우 \$@은 아무 것도 전개하지 않아서, 스크립트 내부의 파이프라인은 사실상 다음과 같다:

```
$ wc -1 | sort -n
```

어떠한 파일이름도 주지 않아서, wc은 표준 입력을 처리하려 한다고 가정한다. 그래서, 단지 앉아서 사용자가 인터랙티브하게 어떤 데이터를 전달해주길 대기하고만 있게 된다. 하지만, 밖에서 보면 사용자에게 보이는 것은 스크립트가 거기 앉아서 정지한 것처럼 보인다: 스크립트가 아무 일도 수행하지 않는 것처럼 보인다.

유용한 무언가를 수행하는 일련의 명령어를 방금 실행했다고 가정하자 — 예를 들어, 논

문에 사용될 그래프를 스크립트가 생성. 필요하면 나중에 그래프를 다시 생성할 필요가 있어서, 파일에 명령어를 저장하고자 한다. 명령문을 다시 타이핑(그리고 잠재적으로 잘 못 타이핑할 수도 있다)하는 대신에, 다음과 같이 할 수도 있다:

```
$ history | tail -n 5 > redo-figure-3.sh
```

redo-figure-3.sh 파일은 이제 다음을 담고 있다:

```
297 bash goostats NENE01729B.txt stats-NENE01729B.txt

298 bash goodiff stats-NENE01729B.txt /data/validated/01729.txt >

01729-differences.txt

299 cut -d ',' -f 2-3 01729-differences.txt > 01729-time-series.txt

300 ygraph --format scatter --color bw --borders none 01729-time-series.txt

figure-3.png

301 history | tail -n 5 > redo-figure-3.sh
```

명령어의 일련 번호를 제거하고, history 명령어를 포함한 마지막 행을 지우는 작업을 편집기에서 한동안 작업한 후에, 그림을 어떻게 생성시켰는지에 관한 정말 정확한 기록을 갖게 되었다.

예제 8.2 (왜 명령어를 실행하기 전에 history에 명령어를 기록할까?). 다음 명령어를 실행시키게 되면:

```
$ history | tail -n 5 > recent.sh
```

파일에 마지막 명령어는 history 명령 그자체다; 즉 쉘이 실제로 명령어를 실행하기 전에 명령 로그에 먼저 history를 추가했다. 실제로 *항상* 쉘은 명령어를 실행시키기 전에 로그에 명령어를 기록한다. 왜 이런 동작을 쉘이 한다고 생각하는가?

♥ 해답과 설명

만약 명령어가 죽던가 멈추게 되면, 어떤 명령어에서 문제가 발생했는지 파악하는 것이 유용할 수 있다. 명령어가 실행된 후에 기록하게 되면, 크래쉬(crash)가 발생된 마지막 명령어에 대한 기록이 없게 된다.

실무에서, 대부분의 사람들은 쉘 프롬프트에서 몇번 명령어를 실행해서 올바르게 수행되는지를 확인한 다음, 재사용을 위해 파일에 저장한다. 이런 유형의 작업은 데이터와 작업흐름(workflow)에서 발견한 것을 history를 호출해서 재사용할 수 있게 하고, 출력을 깔끔하게 하기 위해 약간의 편집을 하고 나서, 쉘 스크립트로 저장하는 흐름을 탄다.

예제 8.3 (쉘 스크립트의 변수). molecules 디렉토리에서, 다음 명령어를 포함하는 script.sh 라는 쉘스크립트가 있다고 가정한다:

```
head -n $2 $1
tail -n $3 $1
```

molecules 디렉토리에서 다음 명령어를 타이핑한다:

```
bash script.sh '*.pdb' 1 1
```

다음 출력물 결과 중 어떤 결과가 나올 것으로 예상하나요? 1. molecules 디렉토리에 있는 *.pdb 확장자를 갖는 각 파일의 첫번줄과 마지막줄 사이 모든 줄을 출력. 2. molecules 디렉토리에 있는 *.pdb 확장자를 갖는 각 파일의 첫번줄과 마지막 줄을 출력. 3. molecules 디렉토리에 있는 각 파일의 첫번째와 마지막 줄을 출력. 4. *.pdb 를 감싸는 인용부호로 오류가 발생.

🥊 해답과 설명

정답은 2.

특수 변수 \$1, \$2, \$3은 스크립트에 명령라인 인수를 나타낸다. 따라서 실행되는 명령어는 다음과 같다:

```
$ head -n 1 cubane.pdb ethane.pdb octane.pdb pentane.pdb propane.pdb
$ tail -n 1 cubane.pdb ethane.pdb octane.pdb pentane.pdb propane.pdb
```

인용부호로 감싸져서 쉘이 '*.pdb'을 명령라인에서 확장하지 않는다. 이를 테면, 스 크립트의 첫번째 인자는 '*.pdb'으로 전달되어 스크립트 내부에서 확장되어 head 와 tail 명령어를 실행시키게 된다.

예제 8.4. 주어진 확장자 내에서 가장 긴 파일을 찾아낸다 인자로 디렉토리 이름과 파일

이름 확장자를 갖는 longest.sh이름의 쉘 스크립트를 작성해서, 그 디렉토리에서 해당 확장자를 가지는 파일 중에 가장 긴 줄을 가진 파일이름을 화면에 출력하세요. 예를 들어, 다음은

```
$ bash longest.sh /tmp/data pdb
```

/tmp/data 디렉토리에 .pdb 확장자를 가진 파일 중에 가장 긴 줄을 가진 파일이름을 화면에 출력한다.

```
    ● 해답과 해설
    # 쉘 스크립트는 다음 두 인자를 갖는다:
    # 1. 디렉토리명
    # 2. 파일 확장자
    # 해당 디렉토리에서 파일 확장자와 매칭되는 가장 길이가 긴 파일명을 출력한
    4. 다.
    WC -l $1/*.$2 ¦ sort -n ¦ tail -n 2 ¦ head -n 1
```

예제 8.5 (스크립트 독해 능력). 이번 문제에 대해, 다시 한번 data-shell/molecules 디렉토리에 있다고 가정한다. 지금까지 생성한 파일에 추가해서 디렉토리에는 .pdb 파일이 많다. 만약 다음 행을 담고 있는 스크립트로 bash example.sh *.dat을 실행할 때, example.sh 이름의 스크립트가 무엇을 수행하는지 설명하세요:

```
# 스크립트 1
echo *.*

# 스크립트 2
for filename in $1 $2 $3
do
    cat $filename
done
```

스크립트 3 echo **\$@.pdb**

🥊 해답과 설명

스크립트 1은 파일명에 구두점(.)이 포함된 모든 파일을 출력한다. 스크립트 2 는 파일 확장자가 매칭되는 첫 3 파일의 내용을 화면에 출력시킨다. 쉘이 인자를 example.sh 스크립트에 전달하기 전에 와일드카드를 확장시킨다.

스크립트 3은 .pdb로 끝나는 스크립트의 모든 인자(즉, 모든 .pdb 파일)를 화면에 출력시킨다.

cubane.pdb ethane.pdb methane.pdb octane.pdb pentane.pdb propane.pdb.pdb

예제 8.6 (스크립트 디버깅). Nelle 컴퓨터 north-pacific-gyre/2012-07-03 디렉토리의 doerrors.sh 파일에 다음과 같은 스크립트가 저장되었다고 가정하자.

```
# Calculate stats for data files.
for datafile in "$@"
do
    echo $datfile
    bash goostats $datafile stats-$datafile
done
```

다음을 실행하게 되면:

```
$ bash do-errors.sh NENE*[AB].txt
```

출력결과는 아무 것도 없다. 원인을 파악하고자 -x 선택옵션을 사용해서 스크립트를 재실행시킨다:

```
bash -x do-errors.sh NENE*[AB].txt
```

보여지는 출력결과는 무엇인가? 몇번째 행에서 오류가 발생했는가?

♥ 해답과 설명

-x 플래그를 사용하면 디버그 모드에서 bash를 실행시키게 된다. 각 명령어를 행단 위로 실행시키고 출력결과를 보여주는데, 오류를 특정하는데 도움이 된다. 이번 예 제에서 echo 명령어는 아무 것도 출력하지 않는 것을 볼 수 있다. 루프 변수명의 철자가 잘못 타이핑 되어 있다. datfile 변수가 존재하지 않아서 빈 문자열이 반환되었다.

8.1 사례: 스크립트 생성하기

Nelle의 지도교수는 모든 분석결과가 재현가능해야 된다는 고집을 갖고 있다. 모든 분석 단계를 담아내는 가장 쉬운 방법은 스크립트에 있다. 편집기를 열어서 다음과 같이 작성 한다:

```
# 데이터 파일별로 통계량 계산.

for datafile in "$@"

do

    echo $datafile
    bash goostats $datafile stats-$datafile

done
```

do-stats.sh 이름으로된 파일에 저장해서, 다음과 같이 타이핑해서 첫번째 단계 분석을 다시 실행할 수 있게 되었다:

```
$ bash do-stats.sh NENE*[AB].txt
```

또한 다음과 같이도 할 수 있다:

```
$ bash do-stats.sh NENE*[AB].txt | wc -l
```

그렇게 해서 출력은 처리된 파일 이름이 아니라 처리된 파일의 숫자만 출력된다.

Nelle의 스크립트에서 주목할 한가지는 스크립트를 실행하는 사람이 무슨 파일을 처리 할지를 결정하게 하는 것이다. 스크립트를 다음과 같이 작성할 수 있다:

```
# Site A, Site B 데이터 파일에 대한 통계량 계산
for datafile in NENE*[AB].txt
do
    echo $datafile
    bash goostats $datafile stats-$datafile
done
```

장점은 이 스크립트는 항상 올바른 파일만을 선택한다: 'Z'파일을 제거했는지 기억할 필요가 없다. 단점은 항상 이 파일만을 선택한다는 것이다 — 모든 파일('Z'를 포함하는 파일), 혹은 남극 동료가 생성한 'G', 'H' 파일에 대해서 스크립트를 편집하지 않고는 실행할수 없다. 좀더 모험적이라면, 스크립트를 변경해서 명령-라인 매개변수를 검증해서 만약어떠한 매개변수도 제공되지 않았다면 NENE*[AB]. txt을 사용하도록 바꿀수도 있다. 물론, 이런 접근법은 유연성과 복잡성 사이에 서로 대립되는 요소 사이의 균형, 즉 트레이드오프(trade-off)를 야기한다.

8.1 연습문제

- 1. 쉘 스크립트에 대한 다음 설명 중 가장 정확한 것은 무엇인가요?
 - (1) 쉘 스크립트는 특정한 파일집합에 대해 명령어나 일련의 명령어를 실행하는 것을 저장하는 것입니다.
 - (2) 쉘 스크립트는 항상 .docx 파일 형식으로 저장해야 합니다.
 - (3) 쉘 스크립트는 오직 한 번만 실행할 수 있습니다.
 - (4) 쉘 스크립트는 텍스트 파일이 아닙니다.
 - (5) 쉘 스크립트는 항상 특정 파일에 작동합니다.
- 2. 쉘 스크립트 내에서 \$1은 무엇을 나타내는가요?
 - (1) 스크립트의 첫 번째 줄
 - (2) 스크립트의 첫 번째 문자
 - (3) 명령 줄에서 첫 번째 파일명이나 다른 인수
 - (4) 스크립트에서 첫 번째로 실행되는 명령
 - (5) 스크립트의 첫 번째 에러 메시지.

- 3. 쉘 스크립트에서 \$1, \$2, \$3 등의 표현은 무엇을 의미하나요? 이러한 표현들은 어떻게 사용되나요?
- 4. 쉘 스크립트에서 인수에 대해 이중 따옴표를 사용하는 이유는 무엇인가요?
- 5. 쉘 스크립트 middle.sh가 다음과 같을 때, '\$ bash middle.sh pentane.pdb 15 5'라는 명령 어를 실행하면 어떤 결과를 기대할 수 있을까요?
 - (1) pentane.pdb 파일의 첫 15줄을 출력합니다.
 - (2) pentane.pdb 파일의 첫 5줄을 출력합니다.
 - (3) pentane.pdb 파일의 15번째 줄부터 5줄을 출력합니다.
 - (4) pentane.pdb 파일의 15번째 줄부터 마지막까지 출력합니다.
 - (5) pentane.pdb 파일의 15번째 줄부터 5줄 뒤까지 출력합니다.

#!/bin/bash

- # middle.sh: 파일의 특정 범위의 줄들을 출력하는 스크립트
- # 사용방법: bash middle.sh filename end_line num_lines
- # \$1은 첫 번째 인자 (파일 이름)
- # \$2는 두 번째 인자 (끝 줄 번호)
- # \$3는 세 번째 인자 (출력할 줄의 수)

head -n "\$2" "\$1" | tail -n "\$3"

제 9 장

파일, 텍스트, 폴더 찾기

"구글(Google)"을 "검색"을 의미하는 동사로 많은 분들이 사용하는 것처럼 유닉스 프로그래머는 "grep"을 동일하게 사용한다. grep은 "global/regular expression/print(전역/정규 표현식/출력)"의 축약어로 초기 유닉스 편집기에서 일반적인 일련의 연산작업을 뜻한다. 매우 유용한 명령-라인 프로그램 이름이기도 하다.

grep은 패턴과 매칭되는 파일의 행을 찾아 화면에 뿌려준다. 예제 파일로, *Salon* 잡지 1988 년 경쟁부문에서 하이쿠(haiku, 일본의 전통 단시) 3개를 담고 있는 파일을 사용례로 활용할 것이다. 이 예제 파일을 갖는 "writing" 하위 디렉토리에서 작업을 할 것이다:

- \$ cd
- \$ cd Desktop/data-shell/writing
- \$ cat haiku.txt

The Tao that is seen
Is not the true Tao, until
You bring fresh toner.

With searching comes loss and the presence of absence: "My Thesis" not found.

Yesterday it worked
Today it is not working
Software is like that.

♦ 영원히 혹은 5년

원본 하이쿠에 링크를 걸지 않았는데 이유는 Salon 사이트에 더 이상 보이는 것 같지 않아서다. Jeff Rothenberg가 말했듯이, "디지털 정보는 어느 것이 먼저 오든 영원한 영속성을 가지거나 혹은 5년이다." 운이 좋은 경우 인기 콘텐트는 종종 백업된다.

단어 "not"을 포함하는 행을 찾아 봅시다:

\$ grep not haiku.txt

Is not the true Tao, until
"My Thesis" not found
Today it is not working

여기서 not이 찾고자 하는 패턴이다. grep 명령어는 파일을 뒤져 지정된 패턴과 매칭되는 것을 찾아낸다. 명령어를 사용하려면 grep을 타이핑하고 나서, 찾고자 하는 패턴을 지정하고 나서 검색하고자 하는 파일명(혹은 파일 다수)를 지정하면 된다.

출력값으로 "not"을 포함하는 파일에 행이 3개 있다.

다른 패턴을 시도해 보자. 이번에는 "The"이다.

\$ grep The haiku.txt
The Tao that is seen
"My Thesis" not found.

이번에는 문자 "The"를 포함한 행이 두줄 출력되었다. 하지만, 더 큰 단어 안에 포함된 단어("Thesis")도 함께 출력된다.

grep 명령어에 -w 옵션을 주면, 단어 경계로 매칭을 제한해서, "day" 단어만을 가진 행만이 화면에 출력된다.

매칭을 "The" 단어 자체만 포함하는 행만 매칭시키려면, grep명령어에 -w 옵션을 주게 되면, 단어 경계로 매칭을 제한시킨다.

\$ grep -w The haiku.txt

The Tao that is seen

"단어 경계"는 행의 시작과 끝이 포함됨에 주의한다. 그래서 공백으로 감싼 단어는 해당 사항이 없게 된다. 때때로, 단어 하나가 아닌, 문구를 찾고자 하는 경우도 있다. 인용부호 내부에 문구를 넣어 grep으로 작업하는 것이 편하다.

\$ grep -w "is not" haiku.txt

Today it is not working

지금까지 단일 단어 주위를 인용부호로 감쌀 필요가 없다는 것을 알고 있다. 하지만, 단어 다수를 검색할 때 인용부호를 사용하는 것이 유용하다. 이렇게 하면, 검색어(term) 혹은 검색 문구(phrase)와 검색 대상이 되는 파일 사이를 더 쉽게 구별하는데 도움을 준다. 나머지 예제에서는 인용부호를 사용한다.

또다른 유용한 옵션은 -n으로, 매칭되는 행에 번호를 붙여 출력한다:

\$ grep -n "it" haiku.txt

5:With searching comes loss

9:Yesterday it worked

10:Today it is not working

상기에서 5, 9, 10번째 행이 문자 "it"를 포함함을 확인할 수 있다.

다른 유닉스 명령어와 마찬자기로 옵션(즉, 플래그)을 조합할 수 있다. 단어 "the"를 포함하는 행을 찾아보자. "the"를 포함하는 행을 찾는 -w 옵션과 매칭되는 행에 번호를 붙이

는 -n을 조합할 수 있다:

```
$ grep -n -w "the" haiku.txt
2:Is not the true Tao, until
6:and the presence of absence:
```

이제 -i 옵션을 사용해서 대소분자 구분없이 매칭한다:

```
$ grep -n -w -i "the" haiku.txt

1:The Tao that is seen
2:Is not the true Tao, until
6:and the presence of absence:
```

이제, -v 옵션을 사용해서 뒤집어서 역으로 매칭을 한다. 즉, 단어 "the"를 포함하지 않는 행을 출력결과로 한다.

```
$ grep -n -w -v "the" haiku.txt

1:The Tao that is seen
3:You bring fresh toner.
4:
5:With searching comes loss
7:"My Thesis" not found.
8:
9:Yesterday it worked
10:Today it is not working
11:Software is like that.
```

grep 명령어는 옵션이 많다. grep 명령어에 대한 도움을 찾으려면, 다음 명령어를 타이핑 한다:

```
$ grep --help
Usage: grep [OPTION]... PATTERN [FILE]...
Search for PATTERN in each FILE or standard input.
PATTERN is, by default, a basic regular expression (BRE).
Example: grep -i 'hello world' menu.h main.c
Regexp selection and interpretation:
 -E, --extended-regexp
                          PATTERN is an extended regular expression (ERE)
 -F, --fixed-strings PATTERN is a set of newline-separated fixed strings
 -G, --basic-regexp
                          PATTERN is a basic regular expression (BRE)
 -P, --perl-regexp
                          PATTERN is a Perl regular expression
  -e, --regexp=PATTERN
                         use PATTERN for matching
 -f, --file=FILE
                          obtain PATTERN from FILE
 -i, --ignore-case
                          ignore case distinctions
  -w, --word-regexp
                          force PATTERN to match only whole words
  -x, --line-regexp
                          force PATTERN to match only whole lines
                          a data line ends in 0 byte, not newline
  -z, --null-data
Miscellaneous:
```

예제 9.1 (grep 사용). 다음중 어떤 명령어가 다음 결과를 만들어낼까요?

grep "of" haiku.txt grep -E "of" haiku.txt

and the presence of absence:

grep -w "of" haiku.txt
 grep -i "of" haiku.txt

🥊 해답과 설명

정답은 3번. -w 플래그는 온전한 단어만 매칭되는 것을 찾기 때문이다.

♦ 와일드카드(Wildcards)

grep의 진정한 힘은 옵션에서 나오지 않고; 패턴에 와일드카드를 포함할 수 있다는 사실에서 나온다. (기술적 명칭은 **정규 표현식(regular expressions)**이고, "grep" 명령어의 "re"가 정규표현식을 나타낸다.) 정규 표현식은 복잡하기도 하지만 강력하기도 하다. 복잡한 검색을 하고자 한다면, 소프트웨어 카펜트리 웹사이트에서 수업내용을 볼 수 있다. 맛보기로, 다음과 같이 두번째 위치에 'o'를 포함한 행을 찾을수 있다:

```
$ grep -E '^.o' haiku.txt

You bring fresh toner.

Today it is not working

Software is like that.
```

-E 플래그를 사용해서 인용부호 안에 패턴을 넣어서 쉘이 해석하는 것을 방지한다. (예를 들어, 패턴에 '*'이 포함된다면, grep을 실행되기 전에 쉘이 먼저 전개하려 할 것이다.) 패턴에서'^'은 행의 시작에 매칭을 고정시키는 역할을 한다.''은 한 문자만 매칭하고(쉘의 '?'과 마찬가지로), 'o'는 실제 영문 'o'와 매칭된다.

예제 9.2 (개체(species) 추적하기). 정훈이는 한 디렉토리에 수백개 데이터 파일이 있는데, 형태는 다음과 같다:

```
2013-11-05,deer,5
2013-11-05,rabbit,22
2013-11-05,raccoon,7
2013-11-06,rabbit,19
2013-11-06,deer,2
```

명령라인에서 첫번째 인자로 개체(species), 두번째 인자로 디렉토리를 인자로 받는 쉘스

크립트를 작성하고자 한다. 스크립트는 일자별로 관측된 개체수를 담아 species.txt 라는 파일로 저장하면 된다.

```
2013-11-05,22
2013-11-06,19
```

스크립트를 작성하는데 다음에 나온 명령어를 적절한 순서로 파이프에 연결시키면 된다:

```
cut -d : -f 2
>
!
grep -w $1 -r $2
!
$1.txt
cut -d , -f 1,3
```

힌트: man grep 명령어를 사용해서 디렉토리에서 재귀적으로 텍스트를 grep하는지 찾아 본다. man cut 명령어를 사용해서 한줄에 필드 하나 이상을 선택하는 방법을 살펴본다. data-shell/data/animal-counts/animals.txt 파일이 예제 파일로 제공되고 있다:

9.0.1 작은 아낙네(Little Women)

Louisa May Alcott가 지은 작은 아낙네(Little Women)를 친구과 함께 읽고 논쟁중이다. 책에는 Jo, Meg, Beth, Amy 네자매가 나온다. 친구가 Jo가 가장 많이 언급되었다고 생각한다. 하

지만, 나는 Amy라고 확신한다. 운좋게도, 소설의 전체 텍스트를 담고 있는 LittleWomen.txt 파일이 있다(data-shell/writing/data/LittleWomen.txt). for 루프를 사용해서, 네자매 각 각이 얼마나 언급되었는지 횟수를 개수할 수 있을까?

힌트: 한가지 해결책은 grep, wc, ¦ 명령어를 동원하는 것이지만, 다른 해결책으로 grep 옵션을 활용하는 것도 있다. There is often more than one way to solve a programming task, so a particular solution is usually chosen based on a combination of yielding the correct result, elegance, readability, and speed. 프로그래밍 문제를 푸는 방식은 한가지 이상 존재한다. 따라서, 올바른 결과를 도출해야 하고, 우아하고(elegance), 가독성이 좋고(readability), 속도를 다 함께 고려하여 선택한다.

🥊 해답과 설명

```
for sis in Jo Meg Beth Amy
do
    echo $sis:
grep -ow $sis LittleWomen.txt | wc -l
done
```

또다른 해법으로, 다소 떨어지는 해답은 다음과 같다:

```
for sis in Jo Meg Beth Amy

do

echo $sis:

grep -ocw $sis LittleWomen.txt

done
```

이 해답이 다소 뒤떨어지는 이유는 grep -c는 매칭되는 행 숫자만 출력하기 때문이다. 행마다 매칭되는 것이 하나 이상 되는 경우, 이 방법으로 매칭되는 전체 갯수는 낮아질 수 있기 때문이다.

grep이 파일의 행을 찾는 반면에, find 명령어는 파일 자체를 검색한다. 다시, find 명령어는 정말 옵션이 많다; 가장 간단한 것이 어떻게 동작하는지 시연하기 위해, 다음과 같은 디렉토리 구조를 사용할 것이다.

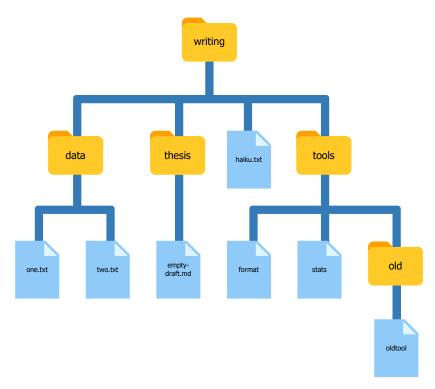


그림 9.1: Find 찾기 예제 파일 구조

Nelle의 writing 디렉토리는 haiku.txt로 불리는 파일 하나와, 하위 디렉토리 4개를 포함한다. thesis 디렉토리는 슬프게고 아무것도 담겨있지 않는 빈 파일 empty-draft.md만 있고, data 디렉토리는 LittleWomen.txt, one.txt과 two.txt 총 파일 3개를 포함하고, tools 디렉토리는 format과 stats 프로그램을 포함하고, oldtool 파일을 담고 있는 old 하위 디렉토리로 구성되어 있다.

첫 명령어로, find .을 실행하자.

\$ find .

.
./data
./data/one.txt
./data/LittleWomen.txt
./data/two.txt
./tools

```
./tools/format
./tools/old
./tools/old/oldtool
./tools/stats
./haiku.txt
./thesis
./thesis/empty-draft.md
```

항상 그렇듯이, . 자체가 의미하는 바는 현재 작업 디렉토리로, 검색을 시작하는 디렉토리가 된다. find 출력결과로 현재 작업 디렉토리 아래 있는 모든 파일, **그리고** 디렉토리명이 나온다. 출력결과가 쓸모없어 보이지만, find 명령어에 선택옵션이 많아서 출력결과를 필터할 수 있다. 이번 학습에서는 그중 일부만 다뤄볼 것이다.

첫번째 선택옵션은 -type d로 "디렉토리인 것들"을 의미한다. 아니나 다를까, find의 출력에는 (.을 포함해서) 디렉토리 5개가 나온다.

```
$ find . -type d

./
./data
./thesis
./tools
./tools/old
```

find 명령어가 찾는 객체가 특별한 순서를 갖고 출력되는 것이 아님에 주목한다. -type d에서 -type f로 옵션을 변경하면, 대신에 모든 파일 목록이 나온다:

```
$ find . -type f

./haiku.txt
./tools/stats
./tools/old/oldtool
./tools/format
```

```
./thesis/empty-draft.md
./data/one.txt
./data/LittleWomen.txt
./data/two.txt
```

이제 이름으로 매칭을 하자:

```
$ find . -name *.txt
./haiku.txt
```

모든 텍스트 파일을 찾기를 기대하지만, 단지 ./haiku.txt만을 화면에 출력한다. 문제는 명령문을 실행하기 전에, *같은 와일드카드 문자를 쉘이 전개하는 것이다. 현재 디렉토리에서 *.txt을 전개하면 haiku.txt이 되기 때문에, 실제 실행하는 명령어는 다음과 같다:

```
$ find . -name haiku.txt
```

find 명령어는 사용자가 요청한 것만 수행한다; 사용자는 방금전에 잘못된 것을 요청했다.

사용자가 원하는 것을 얻기 위해서, grep을 가지고 작업했던 것을 수행하자. 단일 인용부호에 *.txt을 넣어서 쉘이 와일드카드 *을 전개하지 못하게 한다. 이런 방식으로, find 명령어는 확장된 파일명 haiku.txt이 아닌, 실제로 *.txt 패턴을 얻는다:

```
$ find . -name '*.txt'

./data/one.txt
./data/LittleWomen.txt
./data/two.txt
./haiku.txt
```

♦ 목록(Listing) vs. 찾기(Finding)

올바른 옵션이 주어진 상태에서, ls와 find 명령어를 사용해서 비슷한 작업을 수행하도록 만들 수 있다. 하지만, 정상 상태에서 ls는 가능한 모든 것을 목록으로 출력하는 반면에, find는 어떤 특성을 가진 것을 검색하고 보여준다는 점에서 차이가난다.

앞에서 언급했듯이, 명령-라인(command-line)의 힘은 도구를 조합하는데 있다. 파이프로 어떻게 조합하는지를 살펴봤고; 또 다른 기술을 살펴보자. 방금 보았듯이, find . -name '*.txt' 명령어는 현재 디렉토리 및 하위 디렉토리에 있는 모든 텍스트 파일 목록을 보여준다. 어떻게 하면 wc -1 명령어와 조합해서 모든 파일의 행을 개수할 수 있을까?

가장 간단한 방법은 \$() 내부에 find 명령어를 위치시키는 것이다:

```
$ wc -l $(find . -name '*.txt')

11 ./haiku.txt
300 ./data/two.txt
21022 ./data/LittleWomen.txt
70 ./data/one.txt
21403 total
```

쉘이 상기 명령어를 실행할 때,처음 수행하는 것은 \$() 내부를 무엇이든 실행시키는 것이다. 그리고 나서 \$() 표현식을 명령어의 출력 결과로 대체한다. find의 출력 결과가 파일 이름 4개,즉,./data/one.txt,./data/LittleWomen.txt,./data/two.txt,./haiku.txt라서, 쉘은 다음과 같이 명령문을 구성하게 된다:

```
$ wc -l ./data/one.txt ./data/LittleWomen.txt ./data/two.txt ./haiku.txt
```

상기 명령문이 사용자가 원하는 것이다. 이러한 확장이 *과 ? 같은 와일드카드로 확장할 때, 정확하게 쉘이 수행하는 것이다. 하지만 자신의 "와일드카드"로 사용자가 원하는 임의 명령어를 사용해보자.

find와 grep을 함께 사용하는 것이 일반적이다. 먼저 find가 패턴을 매칭하는 파일을 찾고; 둘째로 grep이 또 다른 패턴과 매칭되는 파일 내부 행을 찾는다. 예제로 다음에 현재

부모 디렉토리에서 모든 .pdb 파일에 "FE" 문자열을 검색해서, 철(FE) 원자를 포함하는 PDB파일을 찾을 찾을 수 있다:

예제 9.3 (매칭후 빼내기). grep 명령어의 -v 옵션은 패턴 매칭을 반전시킨다. 패턴과 매칭하지 않는 행만 출력시킨다. 다음 명령어 중에서 어느 것이 /data 폴더에 s.txt로 끝나는 (예로, animals.txt 혹은 planets.txt), 하지만 net 단어는 포함하지 않게 모든 파일을 찾아낼까요? 정답을 생각해냈다면, data-shell 디렉토리에서 다음 명령어를 시도해본다.

- 1. find data -name '*s.txt' | grep -v net
- 2. find data -name *s.txt | grep -v net
- 3. grep -v "temp" \$(find data -name '*s.txt')
- 4. 위에 해당하지 않습니다.

€ 해답과 설명

정답은 1. 매칭 표현식을 인용부호로 감싸서 쉘이 전개하는 것을 방지시킨 상태로 find 명령어에 전개시킨다.

2번은 틀렸는데, 이유는 쉘이 find 명령어에 와일드카드를 전달하는 대신에 *s.txt 을 전개하기 때문이다.

3번은 틀렸는데, 이유는 파일명을 찾는 대신에 "temp"와 매칭되지 않는 행을 갖는 파일을 검색하기 때문이다.

♦ 바이너리 파일(Binary File)

텍스트 파일에 존재하는 것을 찾는 것에만 배타적으로 집중했다. 데이터가 만약 이미지로, 데이터베이스로, 혹은 다른 형식으로 저장되어 있다면 어떨까? 한가지 선택사항은 grep 같은 툴을 확장해서 텍스트가 아닌 형식도 다루게 한다. 이 접근법은 발생하지도 않았고, 아마도 그러지 않을 것이다. 왜냐하면 지원할 형식이 너무나도 많은 존재하기 때문이다.

두번째 선택지는 데이터를 텍스트로 변환하거나, 데이터에서 텍스트같은 비트를 추출하는 것이다. 아마도 가장 흔한 접근법이 (정보를 추출하기 위해서) 각 데이터 형식마다 도구 하나만 개발하면 되기 때문이다. 한편으로, 이 접근법은 간단한 것을 쉽게 할 수 있게 한다. 부정적인 면으로 보면, 복잡한 것은 일반적으로 불가능하다. 예를 들어, grep을 이리 저리 사용해서 이미지 파일에서 X와 Y 크기를 추출하는 프로그램을 작성하기는 쉽다. 하지만, 공식을 담고 있는 엑셀 같은 스프레드쉬트 셀에서 값을 찾아내는 것을 어떻게 작성할까? 세번째 선택지는 쉘과 텍스트 처리가모두 한계를 가지고 있다는 것을 인지하고, 대신에 (R 혹은 파이썬 같은) 프로그램언어를 사용하는 것이다. 이러한 시점이 왔을 때 쉘에서 너무 고생하지 마세요: R 혹은 파이썬을 포함한 많은 프로그래밍 언어가 많은 아이디어를 여기에서 가져왔다. 모방은 또한 칭찬의 가장 충심어린 형태이기도 하다.

유닉스 쉘은 지금 사용하는 대부분의 사람보다 나이가 많다. 그토록 오랫동안 생존한 이유는 지금까지 만들어진 가장 생산성이 높은 프로그래밍 환경 중 하나 혹은 아마도 가장 생산성 높은 프로그래밍 환경이기 때문이다. 구문이 암호스러울 수도 있지만, 숙달한 사람은 다양한 명령어를 대화하듯이 실험하고 나서, 본인 작업을 자동화하는데 학습한 것을 사용한다. 그래픽 사용자 인터페이스(GUI)가 처음에는 더 좋을 수 있지만, 여전히 쉘이 최강이다.

화이트헤드(Alfred North Whitehead) 박사가 1911년 썼듯이 "문명은 생각없이 수행할 수 있는 중요한 작업의 수를 확장함으써 발전한다. (Civilization advances by extending the number of important operations which we can perform without thinking about them.)"

예제 9.4 (find 파이프라인 독해 능력). 다음 쉘 스크립트에 대해서 어떤 작업을 수행하는지 짧은 설명문을 작성하세요.

\$ wc -l \$(find . -name '*.dat') | sort -n}

● 해답과 설명

- 1. 현재 디렉토리에서 .dat 확장자를 갖는 모든 파일을 찾아내시오.
- 2. 파일 각각이 담고 있는 행을 개수한다.
- 3. 앞선 단계에서 나온 출력결과를 숫자로 인식해서 정렬시킨다.

예제 9.5 (다른 특성을 갖는 파일 찾아내기). find 명령어에 "test"로 알려진 다른 기준을 제시해서 특정 속성을 갖는 파일을 지정할 수 있다. 예를 들어, 파일 생성시간, 파일 크기, 파일권한, 파일소유를 확인한다. man find 명령어를 사용해서 이를 살펴보고 나서, 지난

24시간 이내 ahmed 사용자가 변경시킨 모든 파일을 찾는 명령어를 작성한다.

- 힌트: -type, -mtime, -user 플래그 세개를 모두 사용해야 한다.
- 힌트: -mtime 값을 음수를 지정해야 된다 왜일까?

● 히트

Nelle의 홈이 작업 디렉토리라고 가정하고, 다음 명령어를 타이핑한다:

\$ find ./ -type f -mtime -1 -user ahmed

9.1 연습문제

- 1. 유닉스 쉘에서 grep 명령어는 어떤 기능을 합니다?
 - (1) 파일과 디렉토리를 찾습니다.
 - (2) 파일에서 패턴이 일치하는 라인을 선택합니다.
 - (3) 파일의 속성을 변경합니다.
 - (4) 서버의 특정 포트를 열거나 닫습니다.
 - (5) 파일의 내용을 출력합니다.
- 2. 유닉스 쉘에서 grep -n "it" haiku.txt 명령을 사용하면 어떤 결과가 출력됩니까?
 - (1) 파일에서 'it' 단어가 있는 행의 번호를 출력합니다.
 - (2) 파일에서 'it' 단어가 있는 행을 모두 제거합니다.
 - (3) 파일에서 'it' 단어가 있는 행을 대소문자를 구분하지 않고 찾습니다.
 - (4) 파일에서 'it' 단어가 있는 행만을 출력합니다.
 - (5) 파일에서 'it' 단어가 없는 행만을 출력합니다.
- 3. find 명령어는 어떤 기능을 하는가?
 - (1) 특정 패턴이 일치하는 라인을 파일에서 선택합니다.
 - (2) 특정 속성이 일치하는 패턴의 파일을 찾습니다.
 - (3) 파일의 속성을 변경합니다.
 - (4) 서버의 특정 포트를 열거나 닫습니다.
 - (5) 파일의 내용을 출력합니다.

- 4. grep 명령어를 사용하여 'the'라는 단어가 포함된 라인을 대소문자 구분 없이 찾으려면 어떤 명령어를 입력해야 합니까?
- 5. grep -w "is not" haiku.txt 명령어를 사용하면 어떤 결과가 출력되는지 설명해 주세요.

제 III 편 GPT 데이터 과학



제 10 장

CLI 데이터 과학

dir.create() 함수로 data 디렉토리를 생성하고 download.file() 함수로 penguins.csv 파일을 웹사이트에서 가져와서 "data/penguins.csv" 위치에 저장시킨다.

데이터가 준비되면 펭귄 데이터 .csv 정형데이터에서 범주형 펭귄 종(species) 칼럼을 추출하여 빈도수를 계산해보자.

```
dir.create("data")

download.file("https://gist.githubusercontent.com/slopp/
    ce3b90b9168f2f921784de84fa445651/raw/
    4ecf3041f0ed4913e7c230758733948bc561f434/penguins.csv", destfile = "data/
    penguins.csv")
```

10.1 R/파이썬

전통적으로 데이터 사이언스에서 R 혹은 파이썬을 사용해서 빈도수를 구할 수 있다. tidyverse 문법에 익숙하면 직관적으로 파이프를 연결하여 의식의 흐름에 맞춰 직관적인 코드를 작성한다. 코드가 길지 않아 파이썬 pandas를 사용해서 동일한 결과를 얻을 수 있다. 즉, R 혹은 파이썬으로 동작하는 코드를 작성하게 되면 이를 챗GPT에 코드 변환지시명령어로 해당 코드를 생성할 수 있게 된다.

R 코드

파이썬 코드

```
import pandas as pd

penguins = pd.read_csv("data/penguins.csv")

species_count = penguins["species"].value_counts().reset_index()
species_count.columns = ["species", "n"]
species_count = species_count.sort_values(by="n", ascending=False)

print(species_count)

#> species n

#> 0 Adelie 152

#> 1 Gentoo 124

#> 2 Chinstrap 68
```

10.2 쉘 스크립트

R 혹은 파이썬 언어를 설치하고 연관된 패키지를 설치하고 통합개발환경(VS코드, RStudio, 파이참 등)도 준비한 다음 R/파이썬 코드를 작성하여 펭귄 종별 개체수를 구하는 것도 가능하지만 기본적으로 운영체제에서 제공되는 기능을 사용하여 동일한 결과를 얻을 수 있다는 점에서 명령라인 데이터 과학은 큰 장점이 있다.

이를 위해 csvkit을 설치한다.

```
$ pip install csvkit
```

다음 챗GPT에 상기 코드를 쉘 스크립트(unix code)로 작성요청을 하게 되면 난이도가 높지 않고 직관적인 쉘 스크립트를 작성해준다. data/penguins.csv 파일에서 species에 해당되는 칼럼을 찾아낸 다음 정렬하고 앞서 학습한 uniq 명령어를 사용하여 빈도수를 계산하고 보기 좋은 형태로 정렬하게 만들면 된다. 작성한 code/count_species.sh 쉘 스크립트는 다음과 같다.

수행결과 다음과 같은 결과를 얻게 된다.

```
bash code/count_species.sh
Adelie,152
Gentoo,124
Chinstrap,68
```

10.3 윈도우 파일 변환

윈도우 환경에서 저장한 텍스트 파일을 우분투와 같은 환경에서 배쉬(.sh)확장자로 실행할 경우 다음과 같은 오류가 발생된다.

```
line 2: $'\r': command not found
```

이런 오류를 해결하기 위해서 dos2unix 유틸리티를 설치한 후 유닉스 환경에 실행가능한 파일 형태로 변환 시킨 후 작업을 수행한다.

```
$ pip install dos2unix
```

dos2unix 유틸리티을 설치한 후 작성한 쉘 스크립트 파일이 윈도우에서 작성되었다면 리눅스에서 실행할 수 있도록 dos2unix 명령어를 사용해서 변환작업을 수행한다.

```
$ dos2unix code/count_species.sh
dos2unix: converting file code/count_species.sh to Unix format...
$ bash code/count_species.sh
```

10.4 파일 살펴보기

CLI 데이터 분석을 위해서 분석 대상이 되는 파일을 살펴볼 필요가 있다. 이를 위해서 cat 명령어를 통해 분석 대상 파일 data 디렉토리 아래 penguins.csv 파일을 열어본다. 하지만 너무 길어 한정된 화면에서 파이프 ¦ 로 연결하여 첫 5줄만 head 명령어로 확인한다.

csvkit 패키지를 설치해서 csvlook 명령어로 좀더 가시성을 높혀 확인도 가능하다. 파일 마지막부분을 확인하고자 한다면 tail -n 5와 같이 살펴보면 된다.

10.5 파일 구조

.csv 파일을 확인했다면 다음 단계로 자료형을 파악한다. 이를 위해서 csvkit에 내장된 csvsql 명령어를 통해서 변수가 범주형인지 숫자형인지 파악한다.

penguins.csv 파일에서 변수의 숫자와 관측점의 숫자를 파악하는 것은 탐색적 데이터 분석을 시작하기 전에 필수적으로 수행해야하는 작업이다. 다양한 방법이 있지만 다음 명령어로 행수와 칼럼수를 계산할 수 있다.

• csvcut -n 명령어는 penguins.csv 파일 변수명만 추출한다. 그다음 wc -l 명령어는 변수명의 행수를 세어 변수갯수를 파악한다.

• csvstat --count 명령어를 통해 행의 갯수도 산출할 수 있다.

즉, penguins.csv 데이터셋은 344 × 9 크기를 갖는 정형데이터다.

```
$ csvcut -n data/penguins.csv | wc -l
9
$ csvstat --count data/penguins.csv
344
```

다음 단계로 각 변수별로 유일무이한 값(unique)의 갯수를 파악하는 것도 데이터셋을 이해하고 후속 분석 방향을 잡는데 큰 도움이 된다. 예를 들어, 펭귄 종수(species)가 3개로 나타나는데 이는 파머 관측소에서 관측한 펭귄 종수와 일치하고 island 섬도 마찬가지다. 하지만 sex 암수 범주는 3으로 뭔가 잘못된 것을 직관적으로 파악할 수 있다.

```
$ csvstat data/penguins.csv --unique
1. rowid: 344
2. species: 3
3. island: 3
4. bill_length_mm: 165
5. bill_depth_mm: 81
6. flipper_length_mm: 56
7. body_mass_g: 95
8. sex: 3
9. year: 3
```

결측값(missing value)은 간단한 기술통계를 비롯하여 기계학습 알고리즘이나 시각화 등 후속 업무에 큰 장애가 되기 때문에 조기에 이를 파악하여 제거하든가 아니면 적절한 값으로 추정하여 치환하여 온전한 데이터셋(complete dataset)으로 만들어야 된다. csvstat --nulls를 통해 변수에 결측값 유무를 파악할 수 있다.

```
$ csvstat data/penguins.csv --nulls
1. rowid: False
2. species: False
```

3. island: False

4. bill_length_mm: True 5. bill_depth_mm: True

6. flipper_length_mm: True

7. body_mass_g: True

8. sex: True 9. year: False

10.6 기술통계량

csvstat 명령어를 통해서 각 칼럼별로 요약통계량을 상세히 뽑아주고 있다.

\$ csvstat data/penguins.csv | head -n 27

1. "rowid"

Number Type of data: Contains null values: False Unique values: 344 Smallest value: 1 Largest value: 344 Sum: 59340 Mean: 172.5 Median: 172.5 StDev: 99.448 Most common values: 1 (1x) 2 (1x)

3 (1x) 4 (1x) 5 (1x)

2. "species"

Type of data: Text
Contains null values: False
Unique values: 3

Longest value: 9 characters
Most common values: Adelie (152x)

Gentoo (124x)
Chinstrap (68x)

만약, 특정 변수를 선택하여 기술통계량을 뽑아보고자 한다면, csvstat 명령어에 -c 5, sex 와 같이 칼럼 번호와 칼럼명을 지정하면 된다.

\$ csvstat data/penguins.csv -c 5,sex

5. "bill_depth_mm"

Type of data: Number

Contains null values: True (excluded from calculations)

Unique values: 81 Smallest value: 13.1 Largest value: 21.5 Sum: 5865.7 Mean: 17.151 Median: 17.3 StDev: 1.975 Most common values: 17 (12x)

> 18.6 (10x) 17.9 (10x) 18.5 (10x) 15 (10x)

8. "sex"

Type of data: Text

Contains null values: True (excluded from calculations)

Unique values: 3

Longest value: 6 characters
Most common values: male (168x)

female (165x)

None (11x)

Row count: 344

10.7 대용량 데이터

수 GB가 넘어가는 대용량 데이터를 데이터 분석에 앞서 적절한 크기로 줄이는 것이 필요하다. 특히, 빠른 데이터 분석을 위해서 대용량 데이터에서 정보를 잃지 않으면서 일부를 표본추출하여 적절한 크기로 줄인 후에 이를 R/파이썬 등 전문 데이터 분석 언어로 코딩을 하는 것도 유닉스 쉘과 좋은 성과를 내는 전략 중 하나가 된다.

10.7.1 압축파일 풀기

데이터분석을 위해 정형 데이터가 아닌 비정형 데이터를 마추치게 되면 통상 압축화일 형태로 전달이 된다. 기본적인 소용량 압축파일은 쉽게 풀리나 대용량 압축파일(GB가 넘어감)은 전용 압축/압축해제 프로그램을 사용한다.

맥 기준으로 설명하면 p7zip 전용 프로그램을 통해 대용량 압축 파일을 푼다. 설치방법은 brew install p7zip을 터미널에서 실행한다. 7z x 압축파일명 명령어를 입력하면 압축파일이 풀려 원본 파일이 나타난다.

```
$ brew install p7zip
```

\$ 7z x data_2016-10-05.zip

\$ ls -al

total 11236792

drwxr-xr-x+ 59 stat..... staff 2006 10 12 18:35 . drwxr-xr-x 5 root admin 170 1 15 2016 ...

```
-rwxrwxrwx 1 stat.... staff 1169766972 10 7 20:21 data_2016-10-05.zip
-rw-r--r-- 1 stat.... staff 4578470987 10 5 23:18 players_result.txt
```

data_2016-10-05.zip 1.1 GB 압축파일을 풀게되면 4.5 GB 텍스트 파일로 생성된 것이 확인 되다.

10.7.2 표본추출 전략 수립

표본추출을 위한 작업을 위해서 먼저 전략을 잘 수립하여야 한다. 전체 파일에 대한 1% 임의추출을 목표로 삼고 표집하는 경우를 상정한다.

```
$ wc -l players_result.txt
174163238 players_result.txt
```

wc -l 명령어는 해당 파일에 행이 얼마나 되는지 알아내는 명령어다. 이를 통해서 1.7억 줄이 있는 것이 확인된다. 이를 바탕으로 1% 임의추출할 경우 약 170만줄을 임의추출하면 된다.

10.7.3 표본추출 툴설치 1

표본추출을 위해 설치해야 되는 도구는 기본적으로 sort, shuf, gshuf가 있다. 기능적인 면을 떠나 대용량 파일의 경우 성능 속도가 도구를 선택하는 중요한 요인이다.

백만줄을 seq -f 'line %.0f' 1000000 명령어로 생성하여 표집한 경우 성능이 가장 좋은 것은 다음과 같은 순으로 정렬된다.

- 1. shuf: 0.090 초
- 2. 루비 2.0: 0.289 초
- 3. 펄 5.18.2: 0.589 초
- 4. 파이썬 : 1.342 초
- 6. sort -R:10.661 초
- 7. 스칼라: 24.229 초
- 8. 배쉬 루프 + sort : 32.593초

¹How can I shuffle the lines of a text file on the Unix command line or in a shell script?

따라서 shuf를 리눅스에서 gshuf를 맥에서 사용하면 최선의 성과를 얻을 수 있다.

gshuf가 맥의 경우 coreutils에 포함되어 있기 때문에 이를 설치해야 되는데, 이전에 brew link xz을 실행하고 바로 설치한다.

- \$ brew link xz
- \$ brew install coreutils

10.7.4 1% 표본 추출

gshuf, shuf 명령어는 -n 인자로 추출할 행을 수를 지정하면 자동으로 추출해주는데, 결과 를 리다이렉션하여 players_170000.txt 파일에 저장한다.

표본추출결과 데이터 크기를 $\frac{1}{1,000}$, $\frac{1}{10,000}$ 줄인 것이 확인된다.

```
$ gshuf -n 17000 players_result.txt > players_17000.txt
$ gshuf -n 170000 players_result.txt > players_170000.txt
$ ls -al
total 11236792
drwxr-xr-x+ 59 stat.... staff
                                    2006 10 12 18:35 .
drwxr-xr-x 5 root
                        admin
                                     170 1 15 2016 ...
-rwxrwxrwx 1 stat..... staff 1169766972 10 7 20:21 data_2016-10-05.zip
-rw-r--r--
            1 stat..... staff
                                  447091 10 12 18:35 players_17000.txt
            1 stat.... staff
                                  4468179 10 12 18:35 players_170000.txt
-rw-r--r--
-rw-r--r 1 stat.... staff 4578470987 10 5 23:18 players result.txt
```

제 11 장

챗GPT 데이터 과학

11.1 실습 데이터

palmerpenguins 패키지에 penguins 데이터셋이 붓꽃(iris) 데이터셋 대신 데이터 사이언스에서 많이 사용되고 있다. 챗GPT를 활용하여 유닉스 쉘 데이터분석을 위한 예제 데이터셋을 만들기 위해 펭귄 종별로 명칭을 달리하여 .csv 파일로 저장한다.

data/ 디렉토리 아래 penguin_adelie.csv, penguin_chinstrap.csv, data/penguin_gentoo.csv 파일 세개로 나눠진 것을 fs 패키지 dir_ls() 함수를 통해 확인된다.

```
as.character(.)

save_species <- function(species_name = "Adelie") {
  palmerpenguins::penguins %>%
    filter(str_detect(species, species_name)) %>%
    write_csv(glue::glue("data/penguin_{str_to_lower(species_name)}.csv"))
}

walk(species, save_species)

fs::dir_ls(path = "data/", glob="data/penguin_*.csv")

#> data/penguin_adelie.csv data/penguin_chinstrap.csv

#> data/penguin_gentoo.csv
```

11.2 펭귄종별 개체수

펭귄 종별로 .csv 파일을 나눈 이유는 실제 현장에서 다양하게 쪼개진 데이터를 하나로 묶어 분석을 수행하는 경우가 많은 일일이 데이터를 열어 분석가가 직접 분석하는 것은 손이 많이 가고 경우에 따라서는 실수가 들어가기 때문에 이를 방지할 필요가 있다.

펭귄 종별 개체수 빈도수를 계산하는 유닉스 쉘 프로그램을 작성해보자.

1 첫번째 프롬프트

"펭귄 종별로 구분된 palmer penguins csv 파일이 세개 data/ 폴더안에 있습니다. 각각의 파일명은 penguin_adelie.csv, penguin_chinstrap.csv, penguin_gentoo.csv 으로 종별 펭귄 개체수 빈도수를 계산하는 코드를 작성하세요."

첫번째 프롬프트를 작성하고 ShellGPT에 실행하면 다음과 같은 결과가 나온다. sgpt에서 제시한 유닉스 쉘 명령어를 해석하면 "이 셸 명령은 세 개의 서로 다른 CSV 파일에서 펭귄 데이터를 읽고, 종 열을 추출하고, 고유 값을 정렬 및 계산한 후 결과를 출력합니다."라는 취지로 작성된 것을 확인하고 이를 [E]xecute로 실행하면 기대했던 결과를 얻을 수 없음이 확인된다.

```
$ sgpt -s "펭귄 종별로 구분된 palmer penguins csv 파일이 세개 data/ 폴더안에
 ↔ 있습니다. \
> 각각의 파일명은 penguin_adelie.csv, penguin_chinstrap.csv,
□ penguin_gentoo.csv 으로 \
> 종별 펭귄 개체수 빈도수를 계산하는 코드를 작성하세요."
cat data/penguin_adelie.csv | tail -n +2 | cut -d ',' -f 2 | sort | uniq -c
cat data/penguin_chinstrap.csv | tail -n +2 | cut -d ',' -f 2 | sort | uniq -c
cat data/penguin_gentoo.csv | tail -n +2 | cut -d ',' -f 2 | sort | uniq -c
[E]xecute, [D]escribe, [A]bort: D
This shell command reads penguin data from three different CSV files, extracts
 4 the species column, sorts and counts the unique values, and outputs the

    results.

[E]xecute, [D]escribe, [A]bort: E
    44 Biscoe
    56 Dream
    52 Torgersen
    68 Dream
    124 Biscoe
```

두번째 프롬프트에서는 명확하게 펭귄종(species)이 위치한 칼럼을 특정하고 쉘 명령어를 작성하도록 지시하고 이를 바로 실행해보자.

┇ 두번째 프롬프트

"펭귄 종별로 구분된 palmer penguins csv 파일이 세개 data/ 폴더안에 있습니다. 각각의 파일명은 penguin_adelie.csv, penguin_chinstrap.csv, penguin_gentoo.csv 으로 종은 species 첫번째 칼럼에 위치하고, 종별 펭귄 개체수 빈도수를 계산하는 코드를 작성하세요."

원하는 결과가 나오기는 하지만 species도 각 1 빈도수로 출력되는데 이유는 .csv 파일에 칼럼명도 펭귄 개체로 인식해서 함께 빈도수를 계산했기 때문이다. 이에 대한 문제도 제기하고 프롬프트를 수정해보자.

```
$ sgpt -s "펭귄 종별로 구분된 palmer penguins csv 파일이 세개 data/ 폴더안에
↔ 있습니다. \
> 각각의 파일명은 penguin_adelie.csv, penguin_chinstrap.csv,
⊶ penguin_gentoo.csv 으로 \
> 종은 species 첫번째 칼럼에 위치하고, 종별 펭귄 개체수 빈도수를 계산하는 코드
→ 를 작성하세요."
cat data/penguin_adelie.csv | cut -d ',' -f 1 | sort | uniq -c
cat data/penguin_chinstrap.csv | cut -d ',' -f 1 | sort | uniq -c
cat data/penguin_gentoo.csv | cut -d ',' -f 1 | sort | uniq -c
[E]xecute, [D]escribe, [A]bort: E
   152 Adelie
     1 species
    68 Chinstrap
     1 species
   124 Gentoo
     1 species
```

첫번째 행이 칼럼명임을 명시해서 세번째 프롬프트를 작성한다. 하지만 제시된 프롬프 트만으로 깔끔한 결과를 제시하지 못하고 있다.

▮ 세번째 프롬프트

"펭귄 종별로 구분된 palmer penguins csv 파일이 세개 data/ 폴더안에 있습니다. 각각의 파일명은 penguin_adelie.csv, penguin_chinstrap.csv, penguin_gentoo.csv 으로 종은 species 첫번째 칼럼에 위치하고, 첫번째 행은 칼럼명이고, 종별 펭귄 개체수 빈도수를 계산하는 코드를 작성하세요."

```
$ sgpt -s "펭귄 종별로 구분된 palmer penguins csv 파일이 세개 data/ 폴더안에 다 있습니다. \
> 각각의 파일명은 penguin_adelie.csv, penguin_chinstrap.csv,
다 penguin_gentoo.csv 으로 \
```

> 종은 species 첫번째 칼럼에 위치하고, 첫번째 행은 칼럼명이고, 종별 펭귄 개체 △ 수 빈도수를 계산하는 코드를 작성하세요."

awk -F"," 'NR>1{a[\$1]++} END{for(i in a) print i,a[i]}' data/

penguin_adelie.csv data/penguin_chinstrap.csv data/penguin_gentoo.csv

[E]xecute, [D]escribe, [A]bort: D

Count the number of occurrences of the first column in three CSV files and

⇒ print the results.

[E]xecute, [D]escribe, [A]bort: E

Gentoo 124

Chinstrap 68

Adelie 152

species 2

맥락을 유지하여 원하는 결과를 얻을 수 있는 챗GPT 채팅 인터페이스를 사용하여 결과 를 도출할 수 있도록 작업해보자. 먼저 sgpt가 제시한 유닉스 쉘 스크립트를 챗GPT를 통 해 의미를 파악해보자.

₫ 코드해석 프롬프트

다음 유닉스 쉘 코드를 해석해주세요. awk -F"," 'NR>1{a[\$1]++} END{for(i in a) print i,a[i]}' data/penguin_adelie.csv data/penguin_chinstrap.csv data/penguin_gentoo.csv

주어진 awk 명령은 3개의 CSV 파일(penguin_adelie.csv, penguin_chinstrap.csv, penguin_gentoo.csv)에 대해 동작하는데, 이 파일들의 각 줄을 처리합니다. -F", "는 각 라인이 쉼표(,)로 구분된 필드로 구성된 CSV 형식임을 의미합니다.

- awk 스크립트의 각 부분을 분석하면 다음과 같습니다:
- NR>1 {a[\$1]++}: NR은 현재까지 읽은 라인 수를 나타내는 변수입니다. NR>1 이라는 조건은 첫 번째 행(즉, 헤더 행)을 건너뛰도록 합니다. 그 다음에 오는 {a[\$1]++}는 첫 번째 필드(\$1)의 값을 key로 사용하여 배열 a의 해당 요소의 값을 1 증가시킵니다. 다시 말해, 이 명령은 각 파일에서 첫 번째 필드의 값이 나타나는 횟수를 계산합니다.
- END {for(i in a) print i,a[i]}: 이 부분은 모든 입력을 처리한 후 실행되는 명령입

니다. END 부분에서는 for 반복문을 사용하여 배열 a의 모든 요소를 반복하고, 각 key(i)와 그에 대응하는 값(a[i])을 출력합니다.

결국, 이 awk 명령은 3개의 CSV 파일에서 첫 번째 필드의 각 값이 나타나는 횟수를 출력합니다. 첫 번째 행(헤더)은 건너뜁니다. 이 코드는 첫 필드가 범주형 변수(예: 특정 분류)인 경우 해당 변수의 각 값의 출현 빈도를 확인하는 데 유용할 수 있습니다.

species 2를 제거하는 유닉스 쉘 코드를 작성하는 것이 목표라서 이를 달성할 수 있도록 프롬프트에 직설적으로 작성한다.

i species 2 제거 프롬프트

쉘 코드를 실행하면 다음과 같은 결과가 나옵니다. species 2를 제외시키 도록 코드를 다시 작성해주세요.

Gentoo 124 Chinstrap 68 Adelie 152 species 2

주어진 출력에서 'species'라는 키 값이 있습니다. 이 값은 아마도 CSV 파일의 헤더에서 오는 값일 것입니다. 그렇다면, awk 명령을 수정하여 'species'를 무시하도록 할수 있습니다. 다음은 수정된 코드입니다:

```
awk -F"," 'NR>1 && $1 != "species" {a[$1]++} END{for(i in a) print
```

- i,a[i]}' data/penguin_adelie.csv data/penguin_chinstrap.csv data/
- → penguin_gentoo.csv

이 코드는 원래의 코드에서 추가된 && \$1 != "species" 조건을 통해, 첫 번째 필드의 값이 "species"인 행은 건너뛰게 됩니다. 이렇게 하면 'species'라는 키 값이 결과에서 제외되게 됩니다.

챗GPT를 통해 얻은 명령어를 복사하여 쉘 프롬프트에 붙여넣고 실행시키면 원하는 결과를 얻은 것이 확인된다.

```
$ awk -F"," 'NR>1 && $1 != "species" {a[$1]++} END{for(i in a) print i,a[i]}'
```

data/penguin_adelie.csv data/penguin_chinstrap.csv data/penguin_gentoo.csv

Gentoo 124 Chinstrap 68 Adelie 152

R 코드를 통해 세개로 쪼개진 .csv 파일이 목표를 달성했는지 검산해보자. 펭귄 종 순서는 다르지만 정확하게 펭귄종별 관측 빈도수가 일치함을 확인할 수 있다.

11.3 대용량 데이터

유닉스 쉘을 통해 데이터를 분석하게 되면 가장 큰 장점이 R/파이썬 프로그래밍 언어 혹은 엑셀, SAS/SPSS 등과 같은 프로그램을 열지 않고 바로 데이터 분석 작업을 수행할 수 있는 것을 꼽을 수 있다. 특히, 데이터 크기가 수백 MB넘거나 수십GB가 넘는 데이터도 신속하고 빠르게 주어진 컴퓨팅 자원을 활용하여 결과를 낼 수 있다는 것은 크나큰 장점이다.

11.3.1 오픈 데이터셋

챗GPT 프롬프트를 통해 실습에 필요한 적절한 데이터셋을 검색하여 바로 찾을 수 있다. 이런 점이 기존 구글검색과 큰 차이점이자 향후 구글 검색대신 챗GPT를 활용하여 업무 에 활용하는 사례가 늘어나게 되는 이유가 아닐까 싶다.

┇ 10 MB 이상 오픈 데이터셋

파일크기가 10 메가바이트 이상인 오픈 csv 데이터셋을 찾으세요

데이터셋을 찾는 과정에서 여러 출처를 확인했으나, 파일 크기가 10 메가바이트 이상인 CSV 데이터셋에 대한 구체적인 정보를 제공하는 웹사이트를 찾지 못했습니다. 하지만, 여러 공개 데이터 웹사이트가 존재하며, 이들 웹사이트에서 큰 CSV 파일을 찾을 수 있을 것입니다. 예를 들면, Data.gov, World Bank Open Data, Uniform Crime Reporting Program 등이 있습니다1.

따라서, 이러한 웹사이트에서 관심 있는 주제의 데이터셋을 검색하고, 필요에 따라 다운로드를 진행해 보시는 것이 좋겠습니다. 아쉽게도 현재로서는 파일의 크기를 미리 확인하는 방법이 제한적입니다. 한 가지 방법은 데이터셋을 다운로드 받아 직접 파일의 크기를 확인하는 것입니다.

제가 찾아본 결과 중에는 GitHub에 저장된 9.54 메가바이트 크기의 CSV 파일이 있었습니다. 이 파일의 URL은 다음과 같습니다:

... 중략

이 파일이 사용자의 요구를 충족시킬 수 있을지는 별도로 확인이 필요합니다. 더 구체적인 데이터셋을 찾기 위해서는 사용자의 관심사나 필요한 데이터의 종류 에 대한 추가 정보가 필요합니다. 이를 통해 저는 더 구체적인 검색을 진행하고 필 요한 데이터셋을 찾는 데 도움이 될 수 있습니다.

2020년부터 로스앤젤레스 시에서 발생한 범죄 사건이 담긴 범죄데이터를 DATA.GOV 1 에서 .csv 파일 형태로 다운로드 받는다. LA 범죄데이터에 대한 데이터 사전도 웹페이지에 잘 정리되어 있다. 2

Crime_Data_from_2020_to_Present.csv 파일은 237 MB로 수 GB크기를 갖지는 않지만 실습을 위한 대용량 데이터로 사용하기에 적합해보인다.

\$ du -sh data/*

237M data/Crime_Data_from_2020_to_Present.csv

8.0K data/penguin_adelie.csv

4.0K data/penguin_chinstrap.csv

 $^{^1} https://catalog.data.gov/dataset/crime-data-from-2020-to-present \\$

²https://data.lacity.org/Public-Safety/Crime-Data-from-2020-to-Present/2nrs-mtv8

```
8.0K data/penguin_gentoo.csv
```

20K data/penguins.csv

LA 범죄 데이터에 첫번째 궁금한 점은 LA에서 일어나는 범죄중에 우리나라와 비교하여 빈도수가 높은 범죄를 파악하는 것이다. 이를 위해서 데이터 사전에서 "Crm Cd", "Crm Cd Desc" 두 칼럼이 범죄유형코드와 범죄명이 기술되어 있음을 확인할 수 있다.

프롬프트를 직관적으로 작성하여 챗GPT 결과를 보고 이를 실제 데이터에 반영하여 결과를 확인해보자.

LA 범죄데이터 범죄유형 프롬프트

https://catalog.data.gov/dataset/crime-data-from-2020-to-present 웹 사이트에서 2020년부터 현재까지 LA 범죄데이터셋이 있습니다. 범죄유형 별("Crm Cd Des") 빈도수를 내림차순 정렬하는 쉘 코드를 작성해주세요.

먼저, 해당 데이터셋을 다운로드하고 CSV 형식이라고 가정하겠습니다. 그리고 "Crm Cd Des"라는 이름의 열이 범죄 유형을 나타내는 것으로 이해하겠습니다.

파일이 crime_data.csv라는 이름으로 다운로드되었다고 가정하면, 우리는 awk, sort, 그리고 uniq 명령어를 사용하여 범죄 유형별 빈도수를 계산하고 내림차순으로 정렬할 수 있습니다.

다음은 이를 수행하는 쉘 스크립트입니다:

```
awk -F ',' '{print $X}' crime_data.csv | sort | uniq -c | sort -nr
```

여기서 \$X는 "Crm Cd Des" 열의 위치입니다. 이 위치는 CSV 파일의 구조에 따라 다를 수 있으므로, 실제 열 번호로 대체해야 합니다. 예를 들어, "Crm Cd Des"가 CSV 파일의 5번째 열이라면, 스크립트는 다음과 같이 될 것입니다:

```
awk -F ',' '{print $5}' crime_data.csv | sort | uniq -c | sort -nr
```

- 이 스크립트는 다음을 수행합니다:
 - 1. awk -F', '{print \$5}' crime_data.csv: CSV 파일을 읽고 각 행에서 5번째 열(즉,

"Crm Cd Des")을 출력합니다.

- 2. sort: awk에서 출력한 범죄 유형을 알파벳순으로 정렬합니다.
- 3. uniq -c: 정렬된 범죄 유형의 리스트에서 고유한 항목을 찾아 각 항목의 빈도 수를 계산합니다.
- 4. sort -nr: 빈도수를 기준으로 결과를 내림차순으로 정렬합니다.
- 이 코드는 간단한 CSV 파일에 대해서는 잘 작동하지만, 데이터에 쉼표나 따옴표 등이 포함된 경우에는 제대로 작동하지 않을 수 있습니다. 이런 경우에는 보다 복잡한 파싱을 수행하는 도구(예: Python의 pandas 라이브러리)를 사용해야 할 수도 있습니다.

출력결과가 너무 길어 head 명령어로 범죄빈도가 높은 10개만 뽑아보자.

```
$ awk -F ',' '{print $10}' data/Crime_Data_from_2020_to_Present.csv | sort |
    uniq -c | sort -nr | head

95762 VEHICLE - STOLEN

78340 BATTERY - SIMPLE ASSAULT

63197 BURGLARY FROM VEHICLE

58921 "VANDALISM - FELONY ($400 & OVER

58497 THEFT OF IDENTITY

58189 BURGLARY

53785 "ASSAULT WITH DEADLY WEAPON

53735 THEFT PLAIN - PETTY ($950 & UNDER)

49939 INTIMATE PARTNER - SIMPLE ASSAULT

39442 THEFT FROM MOTOR VEHICLE - PETTY ($950 & UNDER)
```

동일한 작업을 R을 활용하여 검증해 보면 동일한 결과가 얻어진 것이 확인된다.

```
crimes <- read_csv("data/Crime_Data_from_2020_to_Present.csv")

crimes %>%
  janitor::clean_names() %>%
  count(crm_cd, crm_cd_desc, sort = TRUE)
```

#> # A tibble: 141 × 3					
#>		crm_cd	crm_cd_desc	n	
#>		<dbl></dbl>	<chr></chr>	<int></int>	
#>	1	510	VEHICLE - STOLEN	95762	
#>	2	624	BATTERY - SIMPLE ASSAULT	78340	
#>	3	330	BURGLARY FROM VEHICLE	63197	
#>	4	740	VANDALISM - FELONY (\$400 & OVER, ALL CHURCH VANDALISMS)	58921	
#>	5	354	THEFT OF IDENTITY	58497	
#>	6	310	BURGLARY	58189	
#>	7	230	ASSAULT WITH DEADLY WEAPON, AGGRAVATED ASSAULT	53785	
#>	8	440	THEFT PLAIN - PETTY (\$950 & UNDER)	53735	
#>	9	626	INTIMATE PARTNER - SIMPLE ASSAULT	49939	
#>	10	420	THEFT FROM MOTOR VEHICLE - PETTY (\$950 & UNDER)	39442	
#>	#	131 mc	pre rows		
#>	#	Use `print(n =)` to see more rows			

고급 데이터 분석을 위한 유닉스 쉘 스크립트를 작성하는 것은 R 혹은 파이썬 코드를 작성하는 것과 비교하여 난이도가 있지만 챗GPT의 도움을 받아 공동 개발하게 되면 비교적 수월하게 원하는 결과를 얻을 수 있다. 유닉스 쉘 데이터 분석 장점은 아무래도 특별한 도구(R/파이썬, 엑셀, SAS/SPSS 등)를 사용하지 않고도 컴퓨터라면 어느 컴퓨터에도 설치되어 있는 운영체제에서 간단한 분석부터 고급 분석작업까지 가볍고 빠르게 수행할 수 있는 장점이 있다. 챗GPT의 도입으로 데이터 사이언스에서 가장 큰 수혜주는 아마도 유닉스 쉘이 아닐가 싶다.

참고문헌

- [1] Ashish Vaswani **andothers**. Attention Is All You Need. 2017. arXiv: 1706.03762 [cs.CL].
- [2] Greg Wilson **andothers**. swcarpentry/shell-novice: Software Carpentry: the UNIX shell, June 2019. **version** v2019.06.1. **july** 2019. DOI: 10.5281/zenodo.3266823. URL: https://doi.org/10.5281/zenodo.3266823.
- [3] 이광춘 and 주용우. "사람과 인공지능의 일자리 경쟁 요인과 협업 방안". in 디지털 경영연구: 6.2 (2020), pages 39-50.

찾아보기

.bash_profile, 51	man, 46		
.evn, 7	nano, 14		
.gitignore, 7	nano, 14		
API Key, 6	OpenAI, 5		
3, 1	openai, 5		
Bash, 35	max_tokens, 21, 23		
bash	tempearature, 23		
-, 53	PATH, 37		
, 53	PowerShell, 11		
, 53	Prompt, 36		
cat, 15	pwd, 41		
cd, 49	Python, 14		
ChatGPT, 5	1 ytiloli, 14		
CLI, 34	REPL, 34		
cmd, 11	sgpt, 14		
Command Shell, 35	Shell, 35 ShellGPT, 13		
curl, 13			
find, 136	Software Carpentry, 5		
GPT, 21	tab completion, 57		
grep, 130	Unix		
GUI, 34	Shell, 33		
HPC, 33	venv, 6		
keywords, 22	wc, 140		
ls, 36, 43	WIMP, 34		
10, 50, 45	VV 11V11, JT		

WSL, 12

Debian, 12

Ubuntu, 12

데비안, 12

우분투, 12

고성능 컴퓨팅, 33

그래픽 사용자 인터페이스, 34

단축키, 53

루트 디렉토리, 42

명령 쉘, 35

명령-라인 인터페이스, 34

배쉬, 35

변수(variable), 37

선택옵션(options), 36

소프트웨어 카펜트리, 5

쉘, 35

쉘GPT, 13

쉘문법(Grammar of Shell), 38

스위치(switches), 36

스크립트(script), 38

업무 분장, 24

여론조사, 27

요약, 26

유닉스 쉘, 33

인자(argument), 36

자동화, 24

조개 껍데기, 35

챗GPT, 5

키워드, 24

탭 자동완성, 57

파이썬, 14

174 찾아보기

파일 시스템, 41

패러미터(parameter), 36

프롬프트, 36

플래그(flag), 36

현재 작업 디렉토리, 41

홈 디렉토리, 42