

데이터 과학을 지탱하는 기본기

한국 알 사용자회

2022-10-17

Contents

데이터 과학을 지탱하는 기본기	5
자동화	9
1 쉘(Shell) 소개	9
1.1 배경	9
1.2 쉘(Shell)	10
1.3 어떻게 생겼을까?	11
1.4 ls 와 플래그 의미 파악	12
1.5 어려운가요?	12
1.6 유연성과 자동화	13
1.7 Nelle 파이프라인 - 문제	13
2 파일과 폴더 넘나들기	15
2.1 도움말 얻기	17
2.2 cd 디렉토리 변경	20
2.3 상대/절대 경로	22
2.4 Nelle 파이프라인: 파일 구성	24
3 파일과 디렉토리 작업	27
3.1 파일과 디렉토리를 위한 좋은 명칭	28
3.2 파일과 폴더 이동	32
3.3 다수 파일과 폴더 작업	35
4 파이프와 필터	39
4.1 Nelle 파이프라인: 파일 확인하기	48
5 루프(Loops)	51
5.1 Nelle의 파이프라인: 많은 파일 처리하기	56
6 쉘 스크립트	63
6.1 Nelle 파이프라인: 스크립트 생성하기	68

7 파일, 문자, 디렉토리 등 찾기	73
버전제어와 협업	85
8 자동화된 버전제어	85
9 Git 구축 및 설정	89
10 저장소 생성	93
11 변경사항 추적	97
12 이력 탐색	109
13 추적대상에서 제외	117
14 GitHub 원격작업	121
15 협업 (Collaborating)	129
16 충돌 (Conflicts)	133
17 공개 과학 (Open Science)	141
18 라이선싱 (Licensing)	143
18.1 소프트웨어 라이선스	143
18.2 콘텐츠 라이선스	145
19 호스팅 (Hosting)	147
20 Git 추가설정	149
20.1 로컬 PC와 SSH 키(Key) 연결	149
20.2 SSH 공개키/비밀키 생성	149
20.3 첫 커밋(commit)	150
20.4 비밀번호 입력없이 푸쉬(push)	150
프로그래밍	153
21 R과 RStudio 소개	153
21.1 동기 모티브	153
21.2 워크샵 시작 전에	153
21.3 RStudio 소개	153
21.4 RStudio 내부 작업흐름	154
21.5 R 소개	155
21.6 계산기로 R 사용하기	156
21.7 수학 함수	157

21.8 다양한 객체 비교하기	158
21.9 변수와 대입	158
21.10 도전과제 1	160
21.1 벡터화(Vectorization)	160
21.11 환경 설정	161
21.12 팩키지	162
21.13 도전과제 2	163
21.14 도전과제 3	163
21.15 도전과제 4	164
22 RStudio 프로젝트 관리	165
22.1 들어가며	165
22.2 가능한 해결책	165
22.3 도전과제	166
22.4 프로젝트 구성 모범 사례	166
22.5 도전 과제 1	168
22.6 도전 과제 1	168
23 도움 청하기	171
23.1 도움말 파일 읽기	171
23.2 특수 연산자	172
23.3 팩키지 도움말 얻기	172
23.4 함수가 정확하게 기억나지 않을 때	172
23.5 어디서 시작해야 될지 아무 생각이 없을 때	172
23.6 코드가 동작않을 때: 동료에게 도움 구함	172
23.7 도전과제 1	173
23.8 도전과제 2	174
23.9 도전과제 3	174
23.10 도움 되는 웹사이트	175
24 자료구조	177
24.1 자료형	178
24.2 벡터와 자료형 강제변환	180
24.3 도전과제 1	184
24.4 데이터프레임	184
24.5 범주형	184
24.6 도전과제 2	185
24.7 리스트	186
24.8 도전과제 3	187
24.9 행렬	189
24.10 도전과제 4	190
24.11 도전과제 5	190
24.12 도전과제 6	191
24.13 도전과제 7	191
25 데이터프레임 탐색	193

25.1 행과 열을 추가	193
25.2 요인 (Factors)	194
25.3 도전과제 1	195
25.4 행 제거	196
25.5 칼럼 제거	197
25.6 dataframe 덧붙이기	197
25.7 도전과제 2	198
25.8 현실적인 예제	198
25.9 도전과제 3	200
25.10 도전과제 4	201
25.11 도전과제 5	201
26 부분집합 추출	203
26.1 색인 사용 요소 접근	203
26.2 요소 건너뛰고 제거	205
26.3 도전과제 1	206
26.4 명칭으로 부분집합 추출	206
26.5 논리 연산자 부분집합 추출	207
26.6 도전과제 2	208
26.7 특수값 처리하기	211
26.8 요인 부분집합 추출	211
26.9 행렬 부분집합 추출	212
26.10 도전과제 4	213
26.11 리스트 부분집합 추출	214
26.12 도전과제 5	215
26.13 도전과제 6	216
26.14 데이터프레임	216
26.15 도전과제 7	217
26.16 도전과제 8	218
27 제어 흐름	219
27.1 도전과제 1	221
27.2 연산 반복	222
27.3 도전과제 2	224
27.4 도전과제 3	225
27.5 도전과제 4	226
27.6 도전과제 5 - 고급	227
28 논문 품질 그래프 생성	229
28.1 도전과제 1	232
28.2 도전과제 2	232
28.3 계층(Layers)	233
28.4 도전과제 3	235
28.5 변환과 통계량	236
28.6 도전과제 4a	239
28.7 도전과제 4b	240

28.8 다중-창(Multi-Panel) 그림	240
29 벡터화(Vectorization)	245
29.1 도전과제 1	245
29.2 도전과제 2	246
29.3 도전과제 3	248
29.4 도전과제 4	249
30 함수 설명	251
30.1 함수 정의하기	251
30.2 도전과제 1	252
30.3 함수 결합	253
30.4 도전과제 2	253
30.5 방어적 프로그래밍	253
30.6 도전과제 3	255
30.7 함수조합	255
30.8 도전과제 4	258
30.9 도전과제 5	259
31 데이터 저장	261
31.1 그래프 저장하기	261
31.2 도전과제 1	262
31.3 데이터를 파일 저장	262
31.4 도전과제 2	263
32 plyr로 데이터프레임을 쪼개고 합치기	265
32.1 plyr 팩키지	266
32.2 도전과제 1	268
32.3 도전과제 2	272
32.4 대안 도전과제	273
33 dplyr 솜씨있게 조작	275
33.1 dplyr 팩키지	275
33.2 select() 사용	276
33.3 filter() 사용	276
33.4 도전과제 1	277
33.5 도전과제 1에 대한 해답	277
33.6 group_by()와 summarize() 사용	277
34 tidyr 솜씨있게 조작	283
34.1 시작하기	285
34.2 도전과제 1	285
34.3 gather()로 wide → long 전환	286
34.4 도전 과제 2	290
34.5 spread()로 'long' 형식 변환	290
34.6 도전과제 3	293
34.7 추가 학습	295

35 knitr 보고서 생성	297
35.1 데이터 분석 보고서	297
35.2 문학적 프로그래밍	297
35.3 R 마크다운 파일 생성	298
35.4 R 마크다운 구성요소	298
35.5 마크다운(Markdown)	299
35.6 도전과제	300
35.7 마크다운 추가기능	300
35.8 R 코드 덩어리	301
35.9 도전과제	301
35.1컴파일 작동방식	302
35.1덩어리 선택옵션	302
35.1도전과제	303
35.1회라인 R 코드	303
35.1도전과제	303
35.1기타 출력 선택옵션	303
35.1관련 자료	304
36 좋은 소프트웨어 작성법	305
36.1 프로젝트 폴더 구조화	305
36.2 가독성 높은 코드 생성	306
36.3 문서화	306
36.4 코드를 모듈화	306
36.5 문제를 잘게 쪼갠다	306
36.6 작성 코드 정상 수행	306
36.7 사람은 반복 금지	307
36.8 스타일 고집	307
데이터베이스	311
37 데이터베이스와 SQL 사용하기	311
38 SQLite 설치	313
38.1 설치	313
38.2 SQLite 설치	313
38.3 실습 데이터베이스 다운로드	314
38.4 SQLite DB 연결/설치 테스트	315
38.5 SQLite DB 나오는 법	315
39 변수/칼럼 선택하기	317
39.1 정의 몇가지	317
39.2 도전 과제	324
39.3 주요점	325
40 정렬, 중복 제거	327

CONTENTS	9
40.1 도전 과제	327
40.2 도전 과제	329
40.3 주요점	331
41 필터링 (Filtering)	333
41.1 도전 과제	336
41.2 주요점	337
42 새로운 값 계산하기	339
42.1 도전 과제	341
42.2 주요점	343
43 결측 데이터 (Missing Data)	345
43.1 도전 과제	348
43.2 주요점	348
44 집합(Aggregation)	349
44.1 도전 과제	355
44.2 주요점	355
45 데이터 결합하기	357
45.1 데이터 위생 (Data Hygiene)	360
45.2 도전 과제	362
45.3 주요점	362
46 데이터 생성과 변형	365
46.1 도전 과제	367
46.2 주요점	367
작업흐름 관리도구	371
47 자동화와 Make	371
47.1 사전 준비	371
48 들어가며	373
49 Makefiles	379
49.1 신규 규칙 두개 작성	384
50 자동 변수	387
50.1 의존성 간신하기	388
50.2 자동변수 규칙 작성	389
50.3 자동변수 요약	390
51 데이터와 코드 의존성	391
51.1 입력파일 간신	393

51.2 results.txt 의존성 testzipf.py	394
52 패턴 규칙	397
52.1 패턴 규칙 주요점	399
53 변수	401
54 함수	405
54.1 함수 요약	408
55 문서화 Makefile	409
56 결론	413
56.1 PNG 생성하기	413
56.2 아카이브 생성	415
56.3 Makefile 아카이빙	417
56.4 아카이브 디렉토리 touch	418

데이터 과학을 지탱하는 기본기

데이터 사이언스는 단일 도구나 언어로 구성되어 있지 않습니다. 데이터 사이언스는 유닉스 쉘, Git/GitHub, Make, SQL, R 프로그래밍 언어로 팀을 꾸려 급격히 진행되는 디지털 전환(Digital Transformation) 시대 불평등 해소와 디지털 경제 성장에 큰 기여를 하고 있습니다.

도구나 언어적인 측면에서 보면 운영체제 리눅스를 근간으로 하는 유닉스 쉘이 자동화를 담당하는 것을 시작으로 빅데이터를 가공하고 가치를 높일 수 있는 형태의 정형 데이터를 책임지는 SQL, 시간여행을 책임지는 Git, 협업과 공유를 담당하는 GitHub, 작업흐름을 담당하는 Make 그리고 기계와 대화를 담당하고 추상화 역할을 수행하는 프로그래밍 언어 R이 각자 역할을 하면서 데이터 사이언스 생태계를 구성하게 됩니다.



Figure 1: 데이터 과학을 지탱하는 도구

사단법인 한국 알(R) 사용자회는 디지털 불평등 해소와 통계 대중화를 위해 2022년 설립되었습니다. 오픈 통계 패키지 개발을 비롯하여 최근에 데이터 사이언스 관련 교재도 함께 제작하여 발간하는 작업을 수행하고 있습니다. 그 첫번째 결과물로 John Fox 교수님이 개발한 설치형 오픈 통계 패키지 Rcmdr(Fox 2016) (Fox and Bouchet-Valat 2021) (Fox 2005) 를 신종화 님께서 한글화 및 문서화에 10년 넘게 기여해주신 한국알사용자회 저작권을 흔쾌히 허락해 주셔서 설치형 오픈 통계 패키지 - Rcmdr로 세상에 나왔습니다.

두번째 활동을 여기저기 산재되어 있던 시각화 관련 자료를 묶어 **데이터 시각화(Data Visualization)**를 전자책 형태로 공개하였고, 데이터 분석 관련 저술을 이어 진행하게 되었습니다.

데이터 분석 언어 R에 관한 지식을 신속히 습득하여 독자들이 갖고 있는 문제에 접목시키고자 하시는 분은 한국 알(R) 사용자회에서 번역하여 공개한 R 신병훈련소(Bootcamp) 과정을 추천드립니다.

“데이터 과학을 지탱하는 기본기” 저작을 위해 소프트웨어/데이터 카펜트리(Software/Data Carpentry)의 원작내용을 번역(Wilson 2022)하고 필요한 경우 한국에서 고급 데이터 분석작업을 수행하기 위해 저자들의 경험을 녹여 제작한 출판물임을 밝혀둡니다.

“데이터 과학을 지탱하는 기본기” 저작물을 비롯한 한국 알(R) 사용자회 저작물은 크리에이티브 커먼즈 저작자표시-비영리-동일조건 변경 허락 (BY-NC-SA) 라이선스를 준용하고 있습니다.

관련 문의와 연락이 필요한 경우 한국 알(R) 사용자회 admin@r2bit.com 대표전자우편으로 연락주세요.

후원계좌

디지털 불평등 해소를 위해 제작중인 오픈 통계패키지 개발과 고품질 콘텐츠 제작에 큰 힘이 됩니다.

- 하나은행 448-910057-06204
- 사단법인 한국알사용자회

자동화

Chapter 1

쉘(Shell) 소개

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.

1.1 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, 마우스, 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,read-evaluate-print loop)**이다: 사용자가 명령어를 타이핑하고 엔터(enter)/반환(return)키를 입력하면, 컴퓨터가 읽고, 실행하고, 결과를 출력한다. 그리고 나면, 사용자는 다른 명령을 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 어렵거나 할 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 컴퓨터가 명령어를 읽어들이고 나서, 평가(혹은 실행)하고 출력결과를 화면에 뿌린다. 또 다른 명령어를 입력할 때까지 대기하는 루프를 반복하게 되서 그렇다.

상기 묘사가 마치 사용자가 직접 명령어를 컴퓨터에 보내고, 컴퓨터는 사용자에게 직접적으로 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(shell)로 불리는데 이유는 운영체제를 감싸서, 복잡성 일부를 숨겨서 운영체제와 더 단순하게 상호작용하게 만든다.

1.2 쉘(Shell)

쉘(Shell)은 다른 것과 마찬가지로 프로그램이다. 조금 특별한 것은 자신이 연산을 수행하기보다 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 클러치(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.

1.3 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 나타낸다. 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: **명령어(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 뚜렷하게 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 다를 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 원도우를 열고, `1s -F` / 명령어를 직접 타이핑한다. (공백과 대문자가 중요함으로 잊지 말자.) 원하는 경우 프롬프트도 변경해도 좋다.

1.4 `1s` 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 기록되어 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)” 단계는 두가지 부분으로 구성됨에 주목한다:

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

그리고 나서, 프롬프트를 출력하고 또다른 명령어를 입력받도록 대기한다.

Command not found 오류

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

```
$ 1s-F
-bash: 1s-F: command not found
```

일반적으로 명령어를 잘못 타이핑했다는 의미가 된다 - 이 경우, `1s` 와 `-F` 사이 공백을 빼먹어서 그렇다. 즉, `1s -F`와 같이 명령을 전달하면 의도한 바가 기계에 정확히 전달된다.

1.5 어려운가요?

GUI와 비교하여 컴퓨터와 상호작용하는데 있어 어려운 모형이고 학습하는데 노력과 시간이 다소 소요됩니다. GUI는 선택지를 보여주고, 사용자가 선택지 중에서 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.

1.6 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 클러칭(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.

1.7 Nelle 파이프라인 - 문제

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:

1. 서로 다른 300개 단백질의 상대적인 함유량을 측정하는 분석기계로 시료를 시험한다. 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 단백질에 대한 통계량과 다른 단백질에 대해 상응하는 통계량을 비교한다.
4. 결과를 작성한다. 그녀의 지도교수는 이달 말까지 이 작업을 정말로 마무리해서, 논문이 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 300×299/2). 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 수 있게 된다.

Chapter 2

파일과 폴더 넘나들기

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; `pwd`는 “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 솔직하게 되면 현재 위치를 다음과 같이 보여주게 된다:

```
$ pwd  
/Users/nelle
```

다음에서, 컴퓨터의 응답은 `/Users/nelle`으로 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\nelle`와 유사하게 보인다. (윈도우 버전마다 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/nelle`에서 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 다음과 같다:

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 디렉토리가 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 디렉토리 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 하단에서 Nelle 과학자 컴퓨터 계정과, 랩실 동료 미이라(Mummy)와 늑대인간(Wolfman) 디렉토리를 볼 수 있다.

미이라(Mummy) 파일은 /Users/imhotep 디렉토리에 저장되어 있고, 늑대인간(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 출력꼴을 생성할 수도 있다. ls으로 하여금 디렉토리 이름 뒤에 /을 추가하게 일러준다: 끝에 붙은 /은 디렉토리라는 것을 지칭한다. 설정에 따라 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/
```

2.1 도움말 얻기

`ls` 명령어에 뛸린 플래그가 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.

2.1.1 --help 플래그

배수 내부에서 동작하도록 작성된 배수 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.

<code>-a, --all</code>	do not ignore entries starting with <code>.</code>
<code>-A, --almost-all</code>	do not list implied <code>.</code> and <code>..</code>
<code>--author</code>	with <code>-l</code> , print the author of each file
<code>-b, --escape</code>	print C-style escapes for nongraphic characters
<code>--block-size=SIZE</code>	scale sizes by SIZE before printing them; e.g., ' <code>--block-size=M</code> ' prints sizes in units of 1,048,576 bytes; see SIZE format below
<code>-B, --ignore-backups</code>	do not list implied entries ending with <code>~</code>
<code>-c</code>	with <code>-lt</code> : sort by, and show, ctime (time of last modification of file status information); with <code>-l</code> : show ctime and sort by name; otherwise: sort by ctime, newest first

...

```

-X                      sort alphabetically by entry extension
-Z, --context           print any security context of each file
-1                      list one file per line. Avoid '\n' with -q or -b
--help                 display this help and exit
--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: <<http://www.gnu.org/software/coreutils/>>
 Full documentation at: <<http://www.gnu.org/software/coreutils/ls>>
 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.
```

2.1.2 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 탈바꿈하게 된다. 만약 운이 좋은 경우 상용법에 대한 예제도 포함되어 있다.

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

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

웹상의 매뉴얼 페이지

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

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

더많은 `ls` 플래그 탐색

`-l`, `-h` 플래그를 붙여 `ls` 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.

`ls`와 사용되는 `-l` 플래그는 `long`을 축약한 것으로 파일/디렉토리 명칭 뿐만 아니라 파일 크기, 최종 변경 시간 같은 부가정보가 출력된다. `-h` 플래그는 “human readable” 사람이 읽기 편한 형태로 파일크기를 지정한다. 예를 들어, 5369 대신에 5.3K이 화면에 출력된다.

재귀적으로 시간순으로 목록 출력

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

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

각 디렉토리의 파일/디렉토리가 가장 마지막 시간 변경순으로 정렬되어 출력된다.

여기서 흔 디렉토리가 하위 디렉토리(sub-directories)가 포함된것을 알 수 있다. 슬래시(/)가 붙지 않는 명칭을 갖는 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) `Desktop`으로 구성된다. `Desktop` 인자는 `ls`로 하여금 현재 작업 디렉토리가 아닌 바탕화면 디렉토리 내용을 출력하도록 지정하는 역할을 수행한다:

```
$ ls -F Desktop
data-shell/
```

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

2.2 cd 디렉토리 변경

지금 확인했듯이, 배수 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 것s는 것이 가능하듯이, 흄 디렉토리에 파일 수백개를 저장하는 것도 가능하다. 하지만, 이런 접근법은 자멸하는 전략이나 마찬가지다.

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

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

```
$ ls -F Desktop/data-shell
creatures/           molecules/          notes.txt        solar.pdf
data/                north-pacific-gyre/ pizza.cfg      writing/
```

둘째로, 다른 디렉토리로 위치를 실제로 바꿀 수 있다. 그렇게 하면 더 이상 흄 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 data 디렉토리로 이동해 보자. 다음 명령어를 쭉 이어서 실행하게 되면 목적지에 도달할 수 있다:

```
$ cd Desktop
$ cd data-shell
$ cd data
```

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

```
$ pwd
/Users/nelle/Desktop/data-shell/data

$ ls -F
amino-acids.txt   elements/      pdb/          salmon.txt
animals.txt        morse.txt    planets.txt    sunspot.txt
```

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

```
$ cd data-shell
```

```
-bash: cd: data-shell: 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 **부모**를 의미한다. 물론, cd .. 명령어를 실행하고 나서 pwd를 실행하게 되면 /Users/nelle/Desktop/data-shell로 되돌아간다:

```
$ pwd  
/Users/nelle/Desktop/data-shell
```

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 또 다른 특별한 디렉토리로, “현재 작업 디렉토리(current working directory)”를 의미한다. 충복되어 불필요해 보일 수 있지만, 곧 .에 대한 사용법을 학습할 것이다.

대부분의 명령라인 도구에서 플래그 다수를 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 ls에만 속하는 것이 아니고; 모든 프로그램에서 같은 방식으로 해석된다. 예를 들어, /Users/nelle/data 디렉토리에 있을 때, ls .. 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.

2.3 상대/절대 경로

컴퓨터에 파일시스템을 돌아다니는데 기본 명령어는 `pwd`, `ls`, `cd`을 들 수 있다. 지금까지 사용했던 햄던 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 제시한다!

```
$ 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/data-shell/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 `cd`와 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/data-shell/data
```

```
$ cd /Users/nelle/Desktop/data-shell
```

`pwd`와 `ls -F` 명령어를 실행하게 되면 원하던 디렉토리로 제대로 이동되었는지 확인이 가능하다.

단축(Shortcuts) 두개 더

쉘을 ~ (틸드) 문자를 경로의 시작으로 해석해서 “현재 사용자 홈 디렉토리”를 의미하게 된다. 예를 들어, Nelle의 홈 디렉토리가 `/Users/nelle`이라면, `~/data`은 `/Users/nelle/data`와 동치가 된다. 경로명에 첫 문자로 있을 때만 이것이 동작한다: `here/there/~/elsewhere` | `here/there/Users/nelle/elsewhere`이 되는 것은 아니다. 따라서, `cd ~`을 홈 디렉토리로 변경하는데 사용한다.

또 다른 단축은 대쉬(-) 문자다. cd는 - 문자를 지금 있는 이전 디렉토리로 변역한다. 이 방법이 전체 경로를 기억하고 있다가 타이핑하는 것보다 더 빠르다. 이를 통해 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.

동일 작업을 수행하는 수많은 방법 - 절대 경로 vs. 상대 경로

/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 ..

해답 풀이 1. No: .은 현재 디렉토리를 나타냄. 2. No: /는 루트 디렉토리를 나타냄. 3. No: Amanda 홈 디렉토리은 /Users/amanda임. 4. No: ../../..은 두 단계 거슬러 올라간다; 즉, /Users에 도달함. 5. Yes: ~은 사용자 홈 디렉토리를 나타남; 이 경우 /Users/amanda이 됨. 6. No: 현재 디렉토리 내부에 home 디렉토리가 존재하는 경우 home 디렉토리로 이동하게 됨. 7. Yes: 불필요하게 복잡하지만, 정답이 맞음. 8. Yes: 사용자 홈 디렉토리로 이동할 수 있는 단축키를 사용함. 9. Yes: 한 단계 위로 이동.

상대경로 해결

만약 pwd 명령어를 켰을 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/

해답 풀이

1. No: backup in /Users 디렉토리 내부에 backup 디렉토리가 있다.
2. No: /Users/thing/backup 디렉토리에 담긴 것을 출력한다.
하지만 ..으로 한 단계 상위 레벨 위치를 찾도록 요청했다.
3. No: 이전 해답을 참조한다.
4. Yes: ../backup/ 은 /Users/backup/을 지칭한다.

ls 독해 능력

상기 그림(도전과제 질문에 사용되는 파일 시스템)에 나온 디렉토리 구조를

상정한다. 만약 `pwd` 명령어를 켰을 때 화면에 `/Users/backup`이 출력되고, `-r` 인자는 `ls` 명령어가 역순으로 화면에 출력하게 한다면, 어떤 명령어가 다음을 화면에 출력할까요?

```
pnas_sub/ pnas_final/ original/
1. `ls pwd`
2. `ls -r -F`
3. `ls -r -F /Users/backup`
4. #2 #3, , #1 .
```

해답풀이 1. No: `pwd` 는 디렉토리 명칭이 아님. 2. Yes: 디렉토리 인자가 없는 `ls` 명령어는 현재 디렉토리의 파일과 디렉토리를 화면에 출력함. 3. Yes: 절대 경로를 명시적으로 사용. 4. Correct: 상기 해설 참조.

2.4 Nelle 파이프라인: 파일 구성

파일과 디렉토리에 대해서 알았으니, Nelle은 단백질 분석기가 생성하는 파일을 구성할 준비를 마쳤다. 우선 `north-pacific-gyre` 디렉토리를 생성해서 데이터가 어디에서 왔는지를 상기하도록 한다. 2012-07-03 디렉토리를 생성해서 시료 처리를 시작한 날짜를 명기했다. Nelle은 `conference-paper`와 `revised-results` 같은 이름을 사용하곤 했다. 하지만, 몇년이 지난 후에 이해하기 어렵다는 것을 발견했다. (마지막 지푸라기는 `revised-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. 분석기 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 시킬 수도 있다. 만약 다음과 같이 타이핑하고:

```
$ ls nor
```

그리고 나서 텁(키보드에 텁 키)을 누르면, 자동으로 쉘이 디렉토리 이름을 자동완성 시켜준다:

```
$ ls north-pacific-gyre/
```

텅을 다시 누르면, Bash가 명령문에 2012-07-03/을 추가하는데, 왜냐하면 유일하게 가능한 자동완성조건이기 때문이다. 한번더 텁을 누려면 아무것도 수행하지 않는다. 왜냐하면 1520가지 경우의 수가 있기 때문이다; 텁을 두번 누르면 모든 파일 목록을 가져온다. 이것을 텁 자동완성(tab completion)이라고 부르고, 앞으로도 다른 많은 털에서도 많이 볼 것이다.

Chapter 3

파일과 디렉토리 작업

이제는 어떻게 파일과 디렉토리를 살펴보는지 알게 되었지만, 우선, 어떻게 파일과 디렉토리를 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 아무것도 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/ wri
```

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.

3.1 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 선택옵션으로 처리하기 때문이다.
3. 명칭에 문자, 숫자, . (마침표), - (대쉬) and _ (밑줄)을 고수한다. 명령라인 인터페이스에서 다른 많은 문자는 특별한 의미를 갖는다. 학습을 진행하면서 이를 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 `thesis`로 작업 디렉토리를 변경하자. Nano 텍스트 편집기를 실행해서 `draft.txt` 파일을 생성하자:

```
$ cd thesis
$ nano draft.txt
```

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 작업할 수 없다. `nano`를 워크샵에서 사용하는데 이유는 거의 누구나 훈련없이 사용할 수 있기 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 그래픽 편집기로 Gedit를 사용한다. 윈도우에서는 Notepad++를 사용하는 것도 좋다. 윈도우에는 (notepad)이라고 불리는 자체 내장 편집기도 있는데 nano 편집기와 마찬가지로 명령라인에서 바로 불러 실행될 수 있다.

어떤 편집기를 사용하든, 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 O 를 누름)를 눌러서 데이터를 디스크에 쓰면 저장된다: (저장하고자

하는 파일명을 입력하도록 독촉받게 되면 draft.txt 기본디폴트로 설정된 것을 받아들이고 엔터키를 친다.)

```
GNU nano 2.0.6          File: draft.txt          Modified

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

Figure 3.1: Nano in Action

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

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 표현은 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-O를 눌러 파일을 저장한다는 의미를 갖는다.

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

```
$ ls  
draft.txt
```

파일을 생성하는 다른 방법

nano 편집기를 사용해서 텍스트 파일을 생성하는 방법을 살펴봤다. 홈 디렉토리에서 다음 명령어를 실행해 보자:

```
$ cd  
$ touch my_file.txt
```

1. touch 명령어는 어떤 작업을 수행하는가? GUI 파일 탐색기를 사용해서 본인 훈 디렉토리를 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?

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, 출력결과를 채울 수 있는 파일이 존재하는지 검색한다. 이런 프로그램에게 `touch` 명령어는 빈 텍스트 파일을 효율적으로 생성할 수 있는 메커니즘을 제공한다는 점에서 유용하다.

`data-shell` 디렉토리로 돌아가서, 생성한 초안을 제거해서 `thesis` 디렉토리를 깔끔하게 정리하자:

```
$ cd thesis
$ rm draft.txt
```

상기 명령어는 파일을 제거한다(`rm`은 “remove”를 줄인 것이다.) `ls` 명령어를 다시 실행하게 되면, 출력결과는 아무 것도 없게 되는데 파일이 사라진 것을 확인시켜준다:

```
$ ls
```

삭제는 영원하다

유닉스에는 삭제된 파일을 복구할 수 있는 휴지통이 없다. (하지만, 유닉스에 기반한 대부분의 그래픽 인터페이스는 휴지통 기능이 있다) 파일을 삭제하면 파일시스템의 관리대상에서 빠져서 디스크 저장공간이 다시 재사용되게 한다. 삭제된 파일을 찾아 되살리는 도구가 존재하지만, 어느 상황에서나 동작한다는 보장은 없다. 왜냐하면 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 thesis`을 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
```

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.

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 디렉토리가 하위 디렉토리를 가지고 있다면, `rm -r`은 하위 디렉토리에도 같은 작업을 반복한다. 매우 편리하지만, 부주의하게 사용되면 피해가 엄청날 수 있다.

디렉토리 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 `nano`를 실행함을 주목하자. 이전에는 `thesis` 디렉토리로 가서 `draft.txt` 이름으로 `nano`를 실행했다.

```
$ pwd
/Users/nelle/Desktop/data-shell
```

```
$ mkdir thesis
$ nano thesis/draft.txt
$ ls thesis
draft.txt
```

3.2 파일과 폴더 이동

`draft.txt`가 특별한 정보를 제공하는 이름이 아니어서 `mv`를 사용하여 파일 이름을 변경하자. `mv`는 “move”의 줄임말이다:

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

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

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

목표 파일명을 명시할 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, `mv`는 디렉토리에도 동작한다 — 별도 `mkdir` 명령어는 없다.

`quotes.txt` 파일을 현재 작업 디렉토리로 이동합시다. `mv`를 다시 사용한다. 하지만 이번에는 두번째 매개변수로 디렉토리 이름을 사용해서 파일 이름을 바꾸지 않고, 새로운 장소에 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` 명령어가 `thesis` 디렉토리가 비었음을 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
```

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

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

`cp` 명령어는 `mv` 명령어와 거의 동일하게 동작한다. 차이점은 이동하는 대신에 복사한다는 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` 파일을 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` 폴더의 복사본은 찾아서 보여준다.

파일명이 뭐가 중요해?

Nelle의 파일 이름이 “무엇.무엇”으로 된 것을 알아챘을 것이다. 이번 학습에서, 항상 `.txt` 확장자를 사용했다. 이것은 단지 관례다: 파일 이름을 `mythesis` 혹은 원하는 무엇이든지 작명할 수 있다. 하지만, 대부분의 사람들은 두 부분으로 구분된 이름을 사용하여 사람이나 프로그램이 다른 유형의 파일임을 구분하도록 돋는다. 이름에 나온 두번째 부분을 **파일 확장자(filename extension)**라고 부르고, 파일에 어떤 유형의 데이터가 담고 있는지 나타낸다. `.txt` 확장자는 텍스트 파일임을, `.pdf`는 PDF 문서임을, `.cfg` 확장자는 어떤 프로그램에 대한 구성정보를 담고 있는 형상관리 파일임을 내고, `.png` 확장자는 PNG 이미지 등등을 나타낸다.

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

`whale.mp3`처럼 고래 PNG 이미지 이름을 갖는 파일을 고래 노래의 음성파일로 변환하는 마술은 없다. 설사 누군가 두번 클릭할 때, 운영체제가 음악 재생기로 열어 실행할 수는 있지만 동작은 되지 않을 것이다.

파일 이름 바꾸기

데이터를 분석하는데 필요한 통계 검정 목록을 담고 있는 `.txt` 파일을 현재 디렉토리에 생성했다고 가정하자; 파일명은 `statstics.txt`. 파일을 생성하고 저장한 후에 곰곰히 생각해 보니 파일명 철자가 틀린 것을 알게 되었다! 틀린 철자를 바로잡고자 하는데, 다음 중 어떤 명령어를 사용해야 하는가? 1. `cp statstics.txt statistics.txt` 2. `mv statstics.txt statistics.txt` 3. `mv statstics.txt .` 4. `cp statstics.txt .`

해답 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 제시하고 있지는 않고 있다; 동일한 파일명은 생성될 수 없다. 4. No, 마침표(.)는 파일을 복사할 디렉토리를 나타내지 새로운 파일명을 제시하고 있지는 않고 있다; 동일한 파일명은 생성될 수 없다.

이동과 복사 아래 보여진 일련의 명령문에 뒤에 `ls`명령어의 출력값은 무엇일까요?

```
$ 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
```

해답 `/Users/jamie/data` 디렉토리에서 출발해서, `recombine` 이름의 디렉토리를 새로 생성한다. 두번째 행은 `proteins.dat` 파일을 새로 만든 폴더 `recombine`으로 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/data`에 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 위치한다.

3.3 다수 파일과 폴더 작업

다수 파일을 복사하기 이번 연습문제에서는 `data-shell/data` 디렉토리에서 명령어를 테스트한다. 아래 예제에서, 파일명 다수와 디렉토리명이 주어졌을 때 `cp` 명령어는 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 수행하는가?

```
$ ls -F
amino-acids.txt  animals.txt  backup/  elements/  morse.txt  pdb/  planets.txt  salmon.txt  sunsp
$ cp amino-acids.txt animals.txt morse.txt
```

해답 하나이상 파일명 다음에 디렉토리명이 주어지게 되면(즉, 목적지 디렉토리는 마지막 인자에 위치해야 한다.), `cp` 명령어는 파일을 해당 디렉토리에 복사한다.

연달아 파일명이 세게 주어지면, `cp` 명령어는 오류를 던지는데 이유는 마지막 인자로 디렉토리를 기대했기 때문이다.

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

와일드 카드(Wildcards)

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

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

혹은 `p5.pdb`을 매칭하지만 (`molecules` 디렉토리에 두 파일이 있다면), `propane.pdb`은 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.pi`도 매칭한다(첫번째 '*'은 어떤 문자도 매칭할 수가 없음). 하지만 `quality.practice`은 매칭할 수 없는데 이유는 'p'로 시작하지

않고, `preferred.p`도 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. 예를 들어, `molecules` 디렉토리(.pdb 확장자로 끝나는 파일만 모여있다.)에 `ls *.pdf`를 타이핑하게 되면, `*.pdf`으로 불리는 파일이 없다고 오류 메시지를 출력한다. 하지만, 일반적으로 `wc`과 `ls` 명령어는 와일드카드 표현식과 매칭되는 파일명 목록을 보게 되고 와일드카드 자체가 아니다. 다른 프로그램은 아니지만, 쉘은 와일드카드를 확장한 것을 다룬다는 점에서 직교 설계(orthogonal design)의 또 다른 사례로 볼 수 있다.

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-dataset_overview.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. > 해답 >> $ 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/
>
```

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 출력을 생성할까요?

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

해답

```
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.

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 `raw`와 `processed`가 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/ 2016-05-20-data/
$ cp -r 2016-05-18-data/ 2016-05-20-data/
$ rm -r -i 2016-05-20-data/
```

해답

첫번째 명령어들이 해당 목적을 달성한다. 먼저 재귀적으로 데이터 폴더를 복사한다. 그리고 나서 `rm` 명령어 두번 사용해서 복사한 디렉토리의 모든 파일을 제거한다. 쉘은 * 와일드카드로 매칭되는 모든 파일과 하위디렉토리를 확장하도록 한다.

두번째 명령어들은 순서가 잘못되었다: 복사하지 않는 파일을 살게하고 나서 재귀 복사 명령어로 디렉토리를 복사했다.

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

Chapter 4

파이프와 필터

몇가지 기초 유닉스 명령어를 배웠기 때문에, 마침내 쉘의 가장 강력한 기능을 살펴볼 수 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 0 혹은 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 -l을 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 `-c`를 사용할 수 있다.

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 웰로 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. `wc`가 출력하는 모든 것은 `lengths.txt` 파일에 대신 들어간다.) `ls lengths.txt`을 통해 파일이 존재하는 것을 확인한다:

```
$ ls lengths.txt
```

```
lengths.txt
```

`cat lengths.txt`을 사용해서 화면으로 `lengths.txt`의 내용을 보낼 수 있다. `cat`은 “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

```

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 더 유용한데 `$ less lengths.txt`와 같이 사용한다. 파일을 화면 단위로 출력한다. 아래로 내려가려면 스페이스바를 누르고, 뒤로 돌아가려면 `b`를 누르면 되고, 빠져 나가려면 `q`를 누른다.

이제 sort 명령어를 사용해서 파일 내용을 정렬합니다.

sort -n 명령어는 어떤 작업을 수행할까?

다음 파일 행을 포함하고 있는 파일에 sort 명령어를 실행하면:

```
10
2
19
22
6
```

출력결과는 다음과 같다:

```
10
19
2
22
6
```

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

```
2
6
10
19
22
```

인수 -n이 왜 이런 효과를 가지는지 설명하세요.

해답

-n 플래그는 알파벳 정렬이 아닌, 숫자 정렬하도록 명시한다.

-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.txt를 사용해서 wc 실행결과를 lengths.txt에 넣었듯이, 명령문 다음에 > sorted-lengths.txt를 넣음으로서, 임시 파일이름인 sorted-lengths.txt에 정렬된 목록 정보를 담을 수 있다. 이것을 실행한 다음에, 또 다른 head 명령어를 실행해서 sorted-lengths.txt에서 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 20`은 처음 20개 행만을 지정한다. `sorted-lengths.txt`이 가장 작은 것에서부터 큰 것으로 정렬된 파일 길이 정보를 담고 있어서, `head`의 출력 결과는 가장 짧은 행을 가진 파일이 되어야만 된다.

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. 문자열을 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. > 해답 > > 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.
{: .solution}

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.

해답 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 파일에 넣어 `tail -2`를 실행해야 한다.

이것이 혼란스럽다면, 좋은 친구네요: `wc`, `sort`, `head` 명령어 각각이 무엇을 수행하는지 이해해도, 중간에 산출되는 파일에 무슨 일이 진행되고 있는지 따라가기는 쉽지 않다. `sort`와 `head`을 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 명령문의 출력결과를 오른쪽 명령문의 입력값으로 사용된다는 뜻을 전달한다. 컴퓨터는 필요하면 임시 파일을 생성하거나, 한 프로그램에서 주기억장치의 다른 프로그램으로 데이터를 복사하거나, 혹은 완전히 다른 작업을 수행할 수도 있다; 사용자는 알 필요도 없고 관심을 가질 이유도 없다.

어떤 것도 파일을 연속적으로 사슬로 엮어 사용하는 것을 막을 수는 없다. 즉, 예를 들어 또 다른 파일을 사용해서 `wc`의 출력결과를 `sort`에 바로 보내고 나서, 다시 처리 결과를 `head`에 보낸다. `wc` 출력결과를 `sort`로 보내는데 파일을 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
```

또 다른 파일을 사용해서 `wc`의 출력결과를 `sort`에 바로 보내고 나서, 다시 처리 결과를 `head`로 보내게 되면 전체 파일라인은 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 x 에 3을 곱하고 로그를 취하는 것과 같다.” 이번 경우는, `*.pdb`의 행수를 세어서 정렬해서 첫부분만 계산하는 것이 된다.

명령문을 파일로 연결하기

현재 작업 디렉토리에, 최소 행수를 갖는 파일을 세개 찾고자 한다. 아래 열거된 어떤 명령어 중 어떤 것이 원하는 파일 3개를 찾아줄까?

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

해답 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 외에 프로그램 출력값이 또 다른 프로그램에 파이프되어 들어가는 것이 가능하게 한다.

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

`wc -l *.pdb > lengths`을 실행할 때 여기서 일어나는 것을 설명하면 다음과 같다. `wc` 프로그램을 실행할 새로운 프로세스를 생성하라고 쉘이 컴퓨터에 지시한다. 파일이름을 인자로 제공했기 때문에 표준입력 대신 `wc`는 인자에서 입력값을 읽어온다. `>`을 사용해서 출력값을 파일로 방향변경 했기 때문에, 쉘은 프로세스의 표준 출력결과를 파일에 연결한다.

`wc -l *.pdb | sort -n`을 실행한다면, 쉘은 프로세스 두개를 생성한다. (파이프 프로세스 각각에 대해서 하나씩) 그래서 `wc`과 `sort`은 동시에 실행된다. `wc`의 표준출력은 직접적으로 `sort`의 표준 입력으로 들어간다; `>`같은 방향변경이 없기 때문에 `sort`의 출력은 화면으로 나가게 된다. `wc -l *.pdb | sort -n | head -1`을 실행하면, 파일에서 `wc`에서 `sort`로, `sort`에서 `head`을 통해 화면으로 나가게 되는 데이터 흐름을 가진 프로세스 3개가 있게 된다.

이 간단한 아이디어가 왜 유닉스가 그토록 성공적이었는지를 보여준다. 다른 많은 작업을 수행하는 거대한 프로그램을 생성하는 대신에, 유닉스 프로그래머는 각자가 한 가지 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`, `sort` 같은 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 수행하고 표준출력에 쓴다.

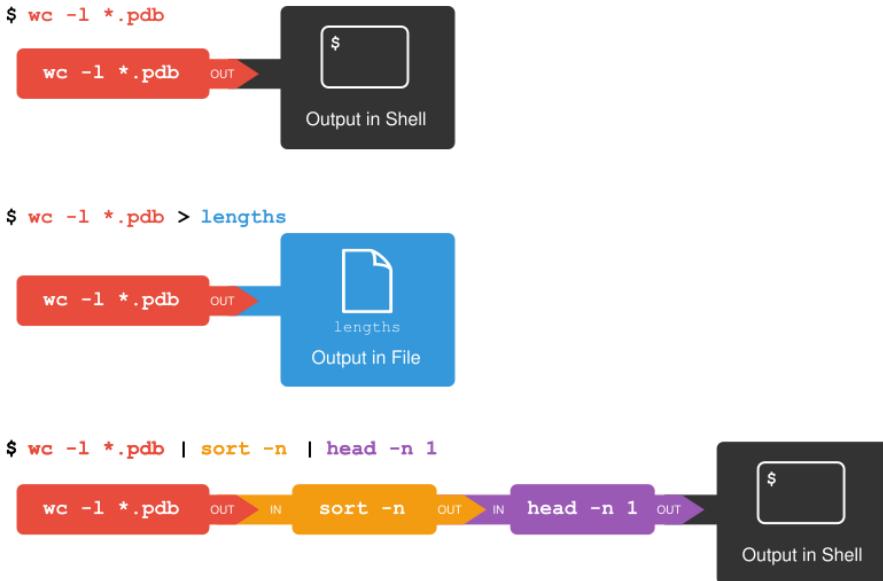


Figure 4.1: 방향변경과 파이프

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.

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.

< 기호의 의미하는 것은 무엇인가?

(다운로드 예제 데이터를 갖고 있는 최상위) data-shell 디렉토리로 작업 디렉토리를 변경한다. 다음 두 명령어 차이는 무엇인가?

```
$ wc -l notes.txt
$ wc -l < notes.txt
```

해답 < 기호는 입력을 방향변경을 해서 명령어로 전달한다.

상기 예제 모두에서, 쉘은 입력에서 `wc` 명령어를 통해 행수를 반환한다. 첫번째 예제에서, 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-D
.
```

3 “

`uniq`가 왜 인접한 중복 행만을 단지 제거한다고 생각합니까?

명령문 `uniq`는 입력으로부터 인접한 중복된 행을 제거한다. 예를 들어, `salmon.txt` 파일에 다음이 포함되었다면,

```
coho
coho
steelhead
coho
steelhead
steelhead
```

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

```
coho
steelhead
coho
steelhead
```

`uniq`가 왜 인접한 중복 행만을 단지 제거한다고 생각합니까? (힌트: 매우 큰 파일을 생각해보세요.) 모든 중복된 행을 제거하기 위해, 파이프로 다른 어떤 명령어를 조합할 수 있을까요? > 해답 > > \$ sort salmon.txt | uniq >

파이프 독해능력

`data-shell/data` 폴더에 `animals.txt`로 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.

해답 `head` 명령어는 `animals.txt` 파일에서 첫 5 행을 추출한다. 그리고 나서, `tail` 명령어로 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
```

파이프 구성하기

이전 연습문제에 사용된 `animals.txt` 파일을 가지고 다음 명령어를 실행한다:

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

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

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

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

해답

```
$ cut -d , -f 2 animals.txt | sort | uniq
{: .language-bash}
```

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?

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

해답 정답은 5. 정답을 이해하는데 어려움이 있으면, (data-shell/data 디렉토리에 위치한 것을 확인한 후) 명령어 전체를 실행하거나, 파이프라인 일부를 실행해 본다.

4.1 Nelle 파이프라인: 파일 확인하기

앞에서 설명한 것처럼 Nelle은 분석기를 통해 시료를 시험해서 17개 파일을 north-pacific-gyre/2012-07-03 디렉토리에 생성했다. 빠르게 건전성 확인하기 위해, 흄디렉토리에서 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, 다시 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].txt`을 사용하여 파일을 조심해서 선택하기로 한다. 언제나 그렇듯이, '*'는 임의 숫자의 문자를 매칭한다. [AB] 표현식은 'A' 혹은 'B'를 매칭해서 Nelle이 가지고 있는 유호한 데이터 파일 모두를 매칭한다.

와일드카드 표현식(Wildcard Expressions)

와일드카드 표현식은 매우 복잡할 수 있지만, 종종 다소 장황할 수 있는 비용을 지불하고 간단한 구문만 사용해서 작성하기도 한다. [data-shell/north-pacific-gyre/2012-07-03 디렉토리를 생각해 보자: *\[AB\].txt 와일드카드 표현식은 A.txt 혹은 B.txt으로 끝나는 모든 파일을 매칭시킨다. 이 와일드카드 표현식을 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 와일드카드 표현식은 오류가 나지 않는데 어떤 상황에서 본인 표현식은 오류 메시지를 출력하는가?

해답 1.

```
$ ls *A.txt
$ ls *B.txt
```

2. 새로운 명령어에서 나온 출력결과는 명령어가 두개라 구분된다.

The output from the new commands is separated because there are two commands.

3. A.txt로 끝나는 파일이 없거나 B.txt로 끝나는 파일이 없는 경우
그렇다.

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 **

해답 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.

Chapter 5

루프(Loops)

반복적으로 명령어를 실행하게 함으로써 자동화를 통해서 루프는 생산성 향상에 핵심이 된다. 와일드카드와 텝 자동완성과 유사하게, 루프를 사용하면 타이핑 상당량(타이핑 실수)을 줄일 수 있다. 와일드카드와 텝 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 생긴다.

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    #
```

```
> 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(iteration이라고도 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 첫번째 돌 때 \$filename 은 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 원소가 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.

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?

해답 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
```

두번째 코드 블록은 루프를 돌 때마다 다른 파일을 출력한다. datafile 파일 변수값이 \$datafile을 통해 평가되고 ls 명령어를 사용해서 파일 목록을 출력하게 된다.

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

프롬프트 따라가기

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.

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 꼬집어내는 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, (temperature같은) 오해를 줄 수 있는 이름은 오해를 불러일으켜서 독자가 생각하기에 당연히 프로그램이 수행해야 할 작업을 프로그램이 수행하지 못하게 할 가능성을 높인다.

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.

해답 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.

해답 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.

`data-shell/creatures` 디렉토리에서 예제를 계속해서 학습해보자. 다음에 좀더 복잡한 루프가 있다:

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

쉘이 `*.dat`을 전개해서 쉘이 처리할 파일 리스트를 생성한다. 그리고 나서 루프 **몸통(loop body)** 부분이 파일 각각에 대해 명령어 두개를 실행한다. 첫 명령어 `echo`는 명령 라인 매개변수를 표준 출력으로 화면에 뿌려준다. 예를 들어:

```
$ echo hello there
```

상기 명령은 다음과 같이 출력된다:

```
hello there
```

이 사례에서, 쉘이 파일 이름으로 `$filename`을 전개했기 때문에, `echo $filename`은 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, `$filename`이 `basilisk.dat`으로 전개될 때 루프 처음에 쉘이 프로그램으로 인식한 `basilisk.dat`를 실행하려고 하기 때문이다. 마지막으로, `head`와 `tail` 조합은 어떤 파일이 처리되는 81-100줄만 선택해서 화면에 뿌려준다. (파일이 적어도 100줄로 되었음을 가정)

```
::: {#shell-loop-space .rmdcaution}
```

파일, 디렉토리, 변수 등 이름에 공백

공백 whitespace)을 사용해서 루프를 돌릴 때 리스트의 각 원소를 구별했다. 리스트 원소중 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
CAAGTGTCC
```

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.dat로 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 확인하기 어렵다. echo로 명령문 앞에 위치시킴으로써, 명령문 각각이 제대로 동작되고 있는 확인하는 것이 가능하다. 다음 도표를 통해서 스크립트가 동작할 때 어떤 작업이 수행하고 있는지 상술하고 있다. 또한 echo 명령어를 사려깊이 사용하는 것이 어떻게 훌륭한 디버깅 기술이 되는지도 보여주고 있다.

5.1 Nelle의 파이프라인: 많은 파일 처리하기

Nelle은 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 선택해서, 올바른 출력 파일이름을 생성할 수 있다는 점이다.

명령어를 반복적으로 타이핑하는 것은 귀찮은 일이지만, 더 걱정이 되는 것은 Nelle이 타이핑 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 echo명령어를 bash goostats으로 변경하고 백업한다:

```
$ 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 것처럼 보인다 — 출력이 아무것도 없다. 잠시뒤에 Nelle은 작성한 스크립트가 화면에 아무것도 출력하지 않아서, 실행되고 있는지, 얼마나 빨리 실행되는지에 대한 정보가 없다는 것을 깨닫는다. 컨트롤+C(Control-C)를 눌러서 작업을 종료하고, 반복할 명령문을 위쪽 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 소요된다고 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 한다.

역사(history)를 아는 사람은 반복할 수 있다.

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

```
$ history | tail -n 5
456  ls -l NENE0*.txt
457  rm stats-NENE01729B.txt.txt
458  bash goostats NENE01729B.txt stats-NENE01729B.txt
459  ls -l NENE0*.txt
460  history
```

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

다른 이력(history) 명령어

이력(history)에 접근하는 단축 명령어가 다수 존재한다.

- Ctrl-R 탄축키는 “reverse-i-search” 이력 검색모드로 입력한 텍스트와 매칭되는 가장 최근 명령어를 이력에서 찾아서 제시한다. Ctrl-R 단축키를 한번 혹은 그 이상 누르게 되면 그 이전 매칭을 검색해 준다.
- !! 명령어는 바로 직전 명령어를 불러온다. (키보드 위화살표를 사용하는 것보다 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 찾아준다. 키보드 위화살표를 눌러 명령라인을 편집하는 것보다 훨씬 빠르다.

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 나온 텍스트를 xylose.dat에 저장된다.
4. 위 어느 것도 아니다.

해답 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.

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.

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

시운전(Dry Run)

루프는 한번에 많은 작업을 수행하는 방식이다 — 만약 잘못된 것이 있다면, 한번에 실수를 대단히 많이 범하게 된다. 루프가 수행하는 작업을 점검하는 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
```

해답 두번째 버전을 실행하면 좋을 것이다. 달려 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 버전을 직접 실행해보고 출력결과를 살펴보자! `analyzed-* .pdb` 파일을 열어서 파일에 기록된 내용도 살펴본다.

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
```

해답 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 새로운 디렉토리가 쪽 생성된다.

직접 코드를 실행해서 어떤 디렉토리가 생성되는지 확인한다!

Chapter 6

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 쉘스크립트가 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-O`를 눌러 파일을 저장하고, 나노 편집기에서 `Ctrl-X`를 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 `bash`라서, 다음과 같이 다음 명령어를 실행시킨다:

```
$ bash middle.sh
```

```

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

```

아니나 다를까, 스크립트의 출력은 정확하게 파이프라인을 직접적으로 실행한 것과 동일하다.

텍스트 vs. 텍스트가 아닌 것 아무거나

종종 마이크로소프트 워드 혹은 리브르오피스 Writer 프로그램을 “텍스트 편집기”라고 부른다. 하지만, 프로그래밍을 할 때 조금 더 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.sh를 편집할 수 있지만, 단순히 명령어를 다시 타이핑하는 것보다 아마 시간이 더 걸릴 것이다. 대신에 middle.sh를 편집해서 좀 더 다양한 기능을 제공하도록 만들어보자:

```
$ nano middle.sh
```

나노 편집기로 octane.pdb를 \$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. 이제 스크립트를 다음과 같이 바꿔 실행해 보자:

```
$ bash middle.sh octane.pdb
```

```

ATOM      9  H      1     -4.502    0.681    0.785   1.00   0.00
ATOM     10  H      1     -5.254   -0.243   -0.537   1.00   0.00
ATOM     11  H      1     -4.357    1.252   -0.895   1.00   0.00
ATOM     12  H      1     -3.009   -0.741   -1.467   1.00   0.00
ATOM     13  H      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
```

```

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

ATOM		H		1	1.324	0.350	-1.332	1.00	0.00
ATOM	10	H		1	1.271	1.378	0.122	1.00	0.00
ATOM	11	H		1	-0.074	-0.384	1.288	1.00	0.00
ATOM	12	H		1	-0.048	-1.362	-0.205	1.00	0.00
ATOM	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
```

ATOM		H		1	-1.259	1.420	0.112	1.00	0.00
ATOM	15	H		1	-2.608	-0.407	1.130	1.00	0.00
ATOM	16	H		1	-2.540	-1.303	-0.404	1.00	0.00
ATOM	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 더 나쁘다.

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

```
$ wc -l *.pdb | sort -n
```

`wc -l`은 파일에 행갯수를 출력하고(`wc`는 ‘word count’로 `-l` 플래그를 추가하면 ‘count lines’의 의미가 됨을 상기한다), `sort -n`은 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, $2` 등등은 사용할 수 없는데, 이유는 얼마나 많은 파일이 있는지를 예단할 수 없기 때문이다. 대신에, 특수 변수 `$@`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
```

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`, `uniq`를 사용해서 각각의 파일별로 나오는 유일무이한 개체에 대한 목록을 화면에 출력하세요.

해답

```
# csv
#
```

```
# . .
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 -l | 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, 논문에 사용될 그래프를 스크립트가 생성. 필요하면 나중에 그래프를 다시 생성할 필요가 있어서, 파일에 명령어를 저장하고자 한다. 명령문을 다시 타이핑(그리고 잠재적으로 잘못 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.

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?

해답 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 `history`를 호출해서 재사용할 수 있게 하고, 출력을 깔끔하게 하기 위해 약간의 편집을 하고 나서, 쉘 스크립트로 저장하는 흐름을 탄다.

6.1 Nelle 파이프라인: 스크립트 생성하기

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

```
#  
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 처리할지를 결정하게 하는 것이다. 스크립트를 다음과 같이 작성할 수 있다:

```
# Site A, Site B  
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 실행할 수 없다. 좀더 모험적이라면, 스크립트를 변경해서 명령-라인 매개변수를 검증해서 만약 어떠한 매개변수도 제공되지 않았다면

NENE* [AB].txt을 사용하도록 바꿀 수도 있다. 물론, 이런 접근법은 유연성과 복잡성 사이에 서로 대립되는 요소 사이의 균형, 즉 트레이드오프(trade-off)를 야기한다.

쉘 스크립트의 변수

molecules 디렉토리에서, 다음 명령어를 포함하는 script.sh라는 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.

해답 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 웰이 '*.pdb'을 명령라인에서 확장하지 않는다. 이를 테면, 스크립트의 첫번째 인자는 '*.pdb'으로 전달되어 스크립트 내부에서 확장되어 head와 tail 명령어를 실행시키게 된다.

주어진 확장자 내에서 가장 긴 파일을 찾아낸다 인자로 디렉토리 이름과 파일이름 확장자를 갖는 `longest.sh`이름의 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 파일 이름을 화면에 출력한다.

해답

```
# :  
# 1.  
# 2.  
#  
wc -l $1/*.$2 | sort -n | ta
```

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 *.dat`를 실행할 때, `example.sh` 이름의 스크립트가 무엇을 수행하는지 설명하세요:

```
#      1
echo *.*

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

#      3
echo $@.pdb
```

해답 스크립트 1은 파일명에 구두점(.)이 포함된 모든 파일을 출력한다.

스크립트 2는 파일 확장자가 매칭되는 첫 3 파일의 내용을 화면에 출력시킨다. 쉘이 인자를 `example.sh` 스크립트에 전달하기 전에 와일드카드를 확장시킨다.

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

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

스크립트 디버깅

Nelle 컴퓨터 `north-pacific-gyre/2012-07-03` 디렉토리의 `do-errors.sh` 파일에 다음과 같은 스크립트가 저장되었다고 가정하자.

```
# Calculate stats for data files.
for datafile in "$@"
do
    echo $datafile
    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? > **해답** > `-x` 플래그를 사용하면 디버그 모드에서 bash를 실행시키게 된다. > 각 명령어를 행단위로 실행시키고 출력결과를 보여주는데, 오류를 특정하는데 도움이 된다. >

이번 예제에서 `echo` 명령어는 아무 것도 출력하지 않는 것을 볼 수 있다. > 루프 변수명의 철자가 잘못 타이핑 되어 있다. > `datfile` 변수가 존재하지 않아서 빈 문자열이 반환되었다.

Chapter 7

파일, 문자, 디렉토리 등 찾기

“구글(Google)”을 “검색”을 의미하는 동사로 많은 분들이 사용하는 것처럼 유닉스 프로그래머는 “grep”을 동일하게 사용한다. grep은 “global/regular expression/print(전역/정규표현식/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 1988년 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)을 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.

grep명령어에 -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 해당사항이 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 붙이는 -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 대소문자 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 타이핑한다:

```
$ 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:

<code>-E, --extended-regexp</code>	PATTERN is an extended regular expression (ERE)
<code>-F, --fixed-strings</code>	PATTERN is a set of newline-separated fixed strings
<code>-G, --basic-regexp</code>	PATTERN is a basic regular expression (BRE)
<code>-P, --perl-regexp</code>	PATTERN is a Perl regular expression
<code>-e, --regexp=PATTERN</code>	use PATTERN for matching
<code>-f, --file=FILE</code>	obtain PATTERN from FILE
<code>-i, --ignore-case</code>	ignore case distinctions
<code>-w, --word-regexp</code>	force PATTERN to match only whole words
<code>-x, --line-regexp</code>	force PATTERN to match only whole lines
<code>-z, --null-data</code>	a data line ends in 0 byte, not newline

Miscellaneous:

...

grep 사용

다음중 어떤 명령어가 다음 결과를 만들어낼까요?

and the presence of absence:

1. grep "of" haiku.txt
2. grep -E "of" haiku.txt
3. grep -w "of" haiku.txt
4. grep -i "of" haiku.txt

해답 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 찾을 수 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.

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 쉘스크립트를 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 찾아본다. man cut 명령어를 사용해서 한줄에 필드 하나 이상을 선택하는 방법을 살펴본다. data-shell/data/animal-counts/animals.txt 파일이 예제 파일로 제공되고 있다: > 해답 >> grep -w \$1 -r \$2 | cut -d : -f 2 | cut -d , -f 1,3 > \$1.txt >> 상기 쉘 스크립트를 다음과 같이 호출하면 된다: >> \$ bash count-species.sh bear . >

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. 하지만, 나는 Amy라고 확신한다. 운좋게도, 소설의 전체 텍스트를 담고 있는 LittleWomen.txt 파일이 있다(data-shell/writing/data/LittleWomen.txt). for 루프를 사용해서, 네자매 각각이 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.

해답

```
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.

Nelle의 writing 디렉토리는 haiku.txt로 불리는 파일 하나와, 하위 디렉토리 4개를 포함한다. thesis 디렉토리는 슬프게도 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,

`find`의 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 d`에서 `-type f`로 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.txt`만을 화면에 출력한다.
 문제는 명령문을 실행하기 전에, *같은 와일드카드 문자를 쉘이 전개하는 것이다.
 현재 디렉토리에서 `*.txt`을 전개하면 `haiku.txt`이 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, `grep`을 가지고 작업했던 것을 수행하자. 단일
 인용부호에 `*.txt`을 넣어서 쉘이 와일드카드 `*`을 전개하지 못하게 한다. 이런
 방식으로, `find` 명령어는 확장된 파일명 `haiku.txt`이 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, `ls`와 `find` 명령어를 사용해서 비슷한 작업을 수행하도록 만들 수 있다. 하지만, 정상 상태에서 `ls`는 가능한 모든 것을 목록으로 출력하는 반면에, `find`는 어떤 특성을 가진 것을 검색하고 보여준다는 점에서 차이가 난다.

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 -l` 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. `find`의 출력 결과가 파일 이름 4개, 즉, `./data/one.txt`, `./data/LittleWomen.txt`, `./data/two.txt`, `./haiku.txt`라서, 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.

`find`와 `grep`을 함께 사용하는 것이 일반적이다. 먼저 `find`가 패턴을 매칭하는 파일을 찾고; 둘째로 `grep`이 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:

```
$ grep "FE" $(find .. -name '*.pdb')
```

```
./data/pdb/heme.pdb:ATOM      25 FE          1      -0.924   0.535  -0.518
```

매칭후 빼내기

`grep` 명령어의 `-v` 옵션은 패턴 매칭을 반전시킨다. 패턴과 매칭하지 않는 행만 출력시킨다. 다음 명령어 중에서 어느 것이 `/data` 폴더에 `s.txt`로 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. None of the above.

해답 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 스프레드시트 셀에서 값을 찾아내는 것을 어떻게 작성할까? 세번째 선택지는 쉘과 텍스트 처리가 모두 한계를 가지고 있다는 것을 인지하고, 대신에 (R 혹은 파이썬 같은) 프로그램 언어를 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 확장함으로써 발전한다. (Civilization advances by extending the number of important operations which we can perform without thinking about them.)”

find 파이프라인 독해 능력

다음 쉘 스크립트에 대해서 무슨 것을 수행하는지 짧은 설명문을 작성하세요.

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

해답

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

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, 파일소유. `man find` 명령어를 사용해서 이를 살펴보고 나서, 지난 24시간 이내 ahmed 사용자가 변경시킨 모든 파일을 찾는 명령어를 작성한다.

힌트 1: `-type`, `-mtime`, `-user` 플래그 세개를 모두 사용해야 한다. 힌트 2: `-mtime` 값을 음수를 지정해야 된다 — 왜일까?

해답

Nelle의 홈이 작업 디렉토리라고 가정하고, 다음 명령어를 타이핑한다:

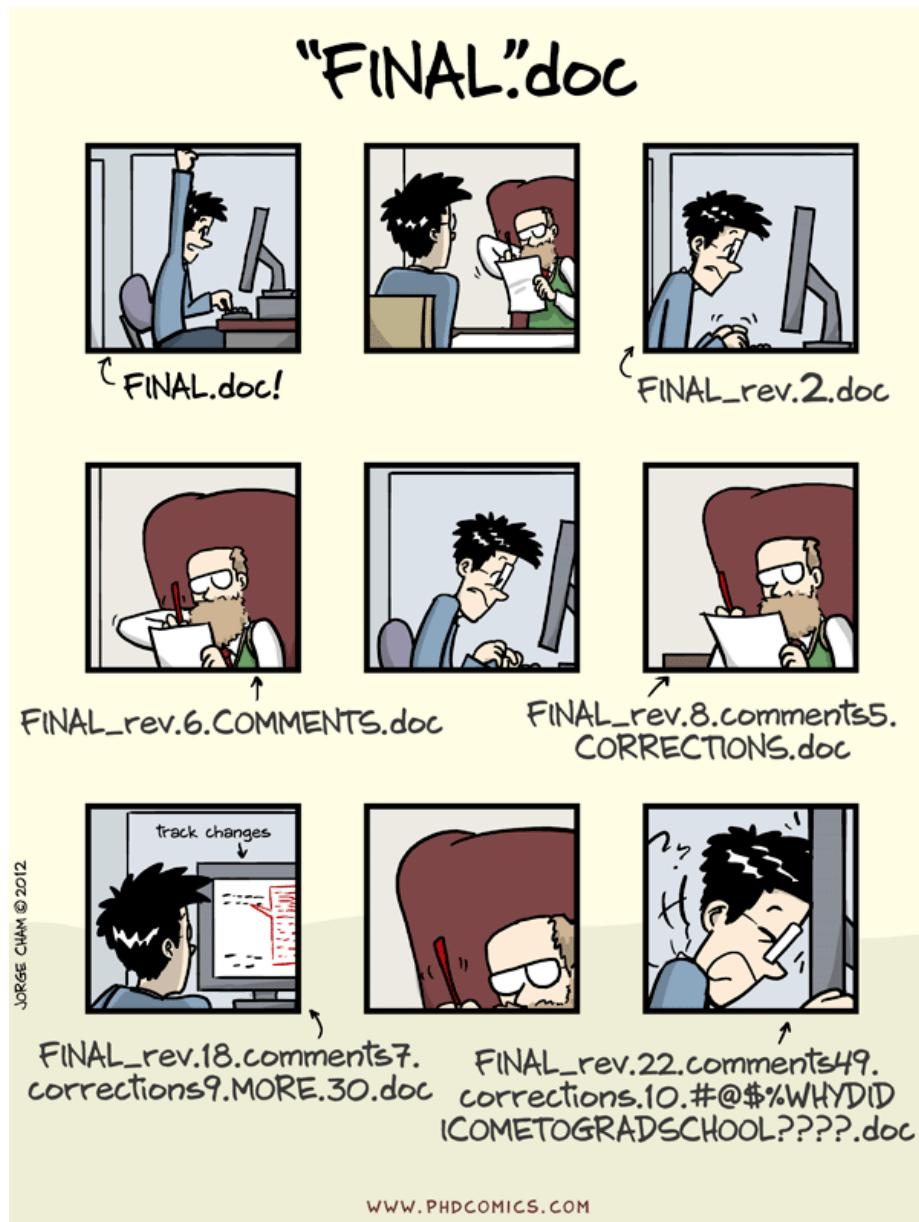
```
$ find ./ -type f -mtime -1 -user ahmed
```

버전제어와 협업

Chapter 8

자동화된 버젼제어

누군가 무엇을 했는지, 언제 했는지를 추적하기 위해서, 버젼제어를 어떻게 사용할 수 있는지 탐색해보자. 다른 사람과 협업을 하지 않더라도, 자동화된 버젼제어가 다음 상황보다 훨씬 더 낫다:



이전에 상기와 같은 상황에 처했었다: 같은 문서에 대해서 거의 동일한 다수 버전을 관리하는 것은 우스워 보인다. 일부 워드프로세서가 이런 상황을 좀더 잘 처리하도록 하는 기능이 있다. 예를 들어, 마이크로소프트 워드 “변경사항 추적(Track Changes)” 혹은 구글 닉스(Google Docs)의 버전 이력이 그것이다.

버전제어 시슬메은 문서의 기본 버전으로 시작하고 나서, 각 단계마다 변경한 이력을 저장한다. 테이프로 생각하면 쉽다: 테이프를 되감으면, 문서 시작한 지점으로 가고,

각 변경사항을 다시 돌리면 가장 최근 버전이 된다.

변경사항을 문서 그자체로부터 떨어진 것으로 생각하면, 동일 기반 문서에 다른 변경사항을 “재생(playback)”하고, 다른 문서 버전을 관리하는 것으로 간주할 수 있다. 예를 들어, 사용자 두명이 같은 문서에 독립적인 변경 작업을 수행할 수 있다.

만약 충돌나지 않으면, 심지어 동일 문서에서 두가지 변경사항을 작업할 수도 있다.

버전제어 시스템은 사용자를 대신해서 변경사항을 기록하고, 파일 버전을 생성하고 파일병합하는데 유용한 도구다. 버전제어 시스템은 어떤 변경사항을 다음 버전에 반영(커밋(commit))으로 불림)할지 결정하는 할 수 있게 하고, 커밋에 관한 유용한 메타정보를 보관한다. 특정 프로젝트와 프로젝트 메타정보에 대한 완전한 커밋이력은 저장소(repository)에 보관된다. 저장소는 협업하는 여러 동료 컴퓨터에 걸쳐 동기화될 수 있다.

버전제어 시스템의 오랜 역사

자동화된 버전제어 시스템이 새로운 것은 전혀 아니다. 1980년부터 RCS, CVS, Subversion 같은 도구가 존재했고, 많은 대기업에서 사용되고 있다. 하지만, 다양한 기능의 한계로 인해서 이들 중 다수는 이제 레거시 시스템(legacy system)으로 간주된다. 최근에 등장한 도구 Git과 Mercurial은 분산(distributed) 기능을 제공한다. 저장소를 굳이 중앙 서버에 둘 필요가 없다는 의미다. 이러한 최신 시스템에는 동시간에 동일한 파일에 다수 저작자가 작업하는 것을 가능하게 하는 강력한 병합(merge) 도구도 내장하고 있다.

논문 작성

- 논문을 작성하면서 정말 멋진 문단을 초안을 작성했지만, 나중에 망치게 되었다고 상상해 보자. 어떻게 정말 멋진 맹음말 버전이 포함된 문서를 되살릴 수 있을까? 가능하기도 할까?
- 공저자가 5명이라고 상상해보자. 공저자가 논문에 반영한 변경사항과 코멘트를 어떻게 관리할 수 있을까? 마이크로소프트 워드나 리브레오피스 Writer를 사용하는 경우, Track Changes 옵션을 사용해서 변경한 것을 반영하게 되면 어떻게 될까? 이러한 변경사항에 대한 이력은 갖고 있는가?

Chapter 9

Git 구축 및 설정

처음 Git를 새로운 컴퓨터에 사용할 때, 몇가지 설정이 필요하다. 다음에 Git을 시작할 때, 설정해야 되는 몇가지 사례가 나와있다:

- 이름과 전자우편 주소
- 선호하는 텍스트 편집기 설정
- 전역(즉, 모든 프로젝트)으로 이런 설정을 할지 여부

명령라인에서 Git 명령어는 다음과 같이 작성된다; `git verb options`, 즉, `git . verb` 가 실제로 수행하고자 하는 명령어가 되고, `options`는 `verb`에 필요할지도 모르는 추가 선택옵션 정보가 된다. 다음에 Dracula가 새로 구입한 노트북에 환경설정하는 방법이 나와있다:

```
$ git config --global user.name "Vlad Dracula"  
$ git config --global user.email "vlad@tran.sylvan.ia"
```

Dracula 대신에 본인 이름과 본인 전자우편 주소를 사용합니다. 사용자명과 전자우편 주소는 후속 Git 활동과 연관된다. 이것이 의미하는 바는 GitHub, BitBucket, GitLab, 혹은 Git 호스트 서버에 푸쉬하는 어떤 변경사항도 사용자명과 전자우편 주소를 담게되는 것을 의미한다.

줄마침(Line Endings)

다른 키보드 타이핑과 마찬가지로, 키보드로 Return를 치게 되면, 컴퓨터는 엔터값을 문자로 인코딩한다. 줄마침을 표현하기 위해서 운영체제마다 별도 문자를 사용한다. (개행 혹은 줄중단, 영어로 newline 혹은 line breaks를 들어봤을 수도 있다.) Git이 파일을 비교하는데 이러한 문자를 사용하기 때문에, 운영체제가 다른 컴퓨팅에서 파일을 편집할 때 예기치 않은 이슈가 발생될 수 있다. 이 문제는 금번 학습 범위를 넘어서는 것이지만, [on this GitHub page](#) 웹페이지에서 좀더 자세한 정보를 얻을 수 있다.

Git에서 줄마침을 인식하고 인코딩하는 방식을 변경하려면, `git config`에 `core.autocrlf` 명령을 사용한다. 권장되는 설정은 다음과 같다:

맥OS와 리눅스:

```
$ git config --global core.autocrlf input
```

윈도우:

```
$ git config --global core.autocrlf true
```

이번 학습에서, GitHub을 사용하게 되는데, 사용되는 전자우편주소는 GitHub 계정을 설정할 때 사용하는 것과 같은 것이 되어야 한다. 만약, 개인정보에 대해 걱정이 된다면, GitHub's instructions for keeping your email address private을 참조한다. GitHub에서 사적인 개인 전자우편주소를 선택하기로 했다면, user.email에 동일한 전자우편주소를 사용한다. 즉, username을 GitHub의 설정된 것으로 바꿔놓아 username@users.noreply.github.com게 된다. 나중에 git config 명령어를 사용해서 전자우편 주소를 변경할 수 있다.

Dracula도 자신이 선호하는 텍스트 편집기를 설정해야 하는데, 다음 표를 참조한다:

편집기	환경설정 명령어
Atom	\$ git config --global core.editor "atom --wait"
nano	\$ git config --global core.editor "nano -w"
BBEdit (Mac, with command line tools)	\$ git config --global core.editor "bbedit -w"
Sublime Text (Mac)	\$ git config --global core.editor "/Applications/Sublime\ Text.app/Contents/SharedSupport/bin/subl -n -w"
Sublime Text (Win, 32-bit install)	\$ git config --global core.editor "'c:/program files (x86)/sublime text 3/sublime_text.exe' -w"
Sublime Text (Win, 64-bit install)	\$ git config --global core.editor "'c:/program files/sublime text 3/sublime_text.exe' -w"
Notepad++ (Win, 32-bit install)	\$ git config --global core.editor "'c:/program files (x86)/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"
Notepad++ (Win, 64-bit install)	\$ git config --global core.editor "'c:/program files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"
Kate (Linux)	\$ git config --global core.editor "kate"
Gedit (Linux)	\$ git config --global core.editor "gedit --wait --new-window"
Scratch (Linux)	\$ git config --global core.editor "scratch-text-editor"
Emacs	\$ git config --global core.editor "emacs"

편집기	환경설정 명령어
Vim	\$ git config --global core.editor "vim"

원할 때마다 Git에 사용할 텍스트 편집기 환경설정을 다시 할 수 있다.

Vim 나가기

다수 프로그램에서 Vim이 기본설정된 편집기다. Vim을 예전에 사용한 적이 없고, 변경사항을 저장하지 않고 세션을 빠져나가고자 한다면, Esc 다음에, :q!를 타이핑하고 나서 Return를 친다. 변경사항을 저장하고 나가려면, Esc 다음에, :wq를 타이핑하고 Return을 친다.

앞서 실행한 상기 명령어는 한번만 실행하면 된다: --global 플래그는 Git으로 하여금 해당 컴퓨터에 본인 계정의 모든 프로젝트에 환경설정한 것을 사용하도록 한다.

본인이 설정한 환경설정 내용은 언제라도 다음 명령어를 입력하여 확인할 수 있다:

```
$ git config --list
```

원하는 만큼 환경설정을 바꿀 수도 있다: 편집기를 바꾸거나 전자우편주소를 갱신할 때 동일한 명령어를 사용하면 된다.

프록시(Proxy)

일부 네트워크에서 proxy를 사용할 필요가 있다. 이런 경우, Git에게 프록시에 대해 일러줘야 한다:

```
$ git config --global http.proxy proxy-url
$ git config --global https.proxy proxy-url
```

프록시를 비활성화 하는 경우, 다음 명령어를 사용한다.

```
$ git config --global --unset http.proxy
$ git config --global --unset https.proxy
```

Git 도움말과 매뉴얼

항상 기억할 것은 git 명령어를 잊은 경우, -h 선택옵션을 주어 명령어 목록을 볼 수 있고, --help를 사용해서 Git 매뉴얼도 이용할 수 있다:

```
$ git config -h
$ git config --help
```


Chapter 10

저장소 생성

Git 환경설정이 완료되면, Git를 사용할 수 있다. 행성 착륙선을 화성에 보낼 수 있는지 조사를 하고 있는 늑대인간과 드라큘라 이야기를 계속해서 진행해 보자.



Figure 10.1: motivatingexample

먼저 바탕화면/Desktop)에 작업할 디렉토리를 생성하고, 생성한 디렉토리로 이동하자:

```
$ cd ~/Desktop  
$ mkdir planets  
$ cd planets
```

그리고 나서, planets을 저장소(repository)로 만든다 — 저장소는 Git이 파일에 대한 버전정보를 저장하는 장소다:

```
$ git init
```

`git init` 명령어가 서브디렉토리(subdirectory)와 파일을 담고 있는 저장소를 생성하는데 주목한다 — `planets` 저장소 내부에 중첩된 별도 저장소를 생성할 필요는 없다. 또한, `planets` 디렉토리를 생성하고 저장소로 초기화하는 것은 완전히 서로 다른 과정이다.

`ls`를 사용해서 디렉토리 내용을 살펴보면, 변한 것이 아무것도 없는 것처럼 보인다:

```
$ ls
```

하지만, 모든 것을 보여주는 `-a` 플래그를 추가하면, Git은 `planets` 디렉토리 내부에 `.git`로 불리는 숨겨진 디렉토리를 생성한 것을 볼 수 있다:

```
$ ls -a
```

```
.. .git
```

Git은 `.git`이라는 특별한 하위 디렉토리에 프로젝트에 대한 정보를 저장한다. 여기에는 프로젝트 디렉토리 내부에 위치한 모든 파일과 서브 디렉토리가 포함된다. 만약 `.git`를 삭제하면, 프로젝트 이력을 모두 잊어버리게 된다.

모든 것이 제대로 설정되었는지를 확인을 하려면, Git에게 다음과 같이 프로젝트 상태를 확인 명령어를 던진다:

```
$ git status
```

```
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

다른 git 버전을 사용할 경우, 출력 결과물이 다소 다를 수도 있다.

Git 저장소를 생성할 장소

(이미 생성한 프로젝트) 행성에 대한 정보를 추적하면서, 드라큘라는 달에 관한 정보도 추적하고자 한다. `planets` 프로젝트와 관련된, 새로운 프로젝트 `moons`를 시작한다. 늑대인간의 걱정에 불구하고, Git 저장소 내부에 또다른 Git 저장소를 생성하려고 다음 순서로 명령어를 입력해 나간다:

```
$ cd ~/Desktop      #
$ cd planets        # planets      .
$ ls -a             # planets     .git
$ mkdir moons       # planets/moons   .
$ cd moons          # moons        .
$ git init          # Git    moons   .
$ ls -a             # Git    .git   .
```

`moons` 서브 디렉토리에 저장된 파일을 추적하기 위해 `moons` 디렉토리 안에서 `git init` 명령을 실행해야 할까?

해답

아니다. `moons` 서브 디렉토리에 Git 저장소를 만들 필요는 없다. 왜냐하면, `planets` 저장소가 이미 모든 파일, 서브 디렉토리, `planets` 디렉토리 아래 서브 디렉토리 파일 모두를 추적하기 때문이다. 따라서, 달에 관한 모든 정보를 추정하는데, 드라큘라는 `planets` 디렉토리 아래 `moons` 서브 디렉토리를 추가하는 것으로 충분하다.

추가적으로, 만약 Git 저장소가 중첩(nested)되면, Git 저장소는 서로 방해할 수 있다: 바깥 저장소가 내부 저장소 버전관리를 하게 된다. 따라서, 별도 디렉토리에 서로 다른 신규 Git 저장소를 생성하는게 최선이다.

디렉토리에 저장소가 서로 충돌하지 않도록 하려면, `git status` 출력물을 점검하면 된다. 만약, 다음과 같은 출력물이 생성되게 되면 신규 저장소를 생성하는 것이 권장된다:

```
$ git status

fatal: Not a git repository (or any of the parent directories): .git
```

git init 실수 올바르게 고치기

늑대인간은 드라큘라에게 중첩된 저장소가 중복되어 불필요한 이유와 함께 향후 혼란을 야기할 수 있는 이유를 설명했다. 드라큘라는 중첩된 저장소를 제거하고자 한다. `moons` 서브 디렉토리에 마지막으로 날린 `git init` 명령어 실행취소를 어떻게 할 수 있을까?

해답 - 주의해서 사용바람!

이러한 사소한 실수를 원복하고자, 드라큘라는 `planets` 디렉토리에서 다음 명령어를 실행하여 `.git` 디렉토리를 제거하기만 하면 된다:

```
$ rm -rf moons/.git
```

하지만, 주의한다! 디렉토리를 잘못 타이핑하게 되면, 보관해야하는 프로젝트 정보를 담고 있는 Git 이력 전체가 날아가게 된다. 따라서, `pwd` 명령어를 사용해서 현재 작업 디렉토리를 항상 확인한다.

Chapter 11

변경사항 추적

먼저 디렉토리 위치가 맞는 확인하자. `planets` 디렉토리에 위치해야 한다.

```
$ pwd
```

```
/home/vlad/Desktop/planets
```

`moons` 디렉토리에 여전히 있다면, `planets` 디렉토리로 되돌아간다.

```
$ pwd
```

```
/home/vlad/Desktop/planets/moons
```

```
$ cd ..
```

전진기지로서 화성의 적합성에 관한 기록을 담고 있는 `mars.txt` 파일을 생성한다. (파일 편집을 위해서 `nano` 편집기를 사용한다; 원하는 어떤 편집기를 사용해도 된다. 특히, 앞에서 전역으로 설정한 `core.editor`일 필요는 없다. 하지만, 파일을 새로 생성하거나 편집할 때 배쉬 명령어는 사용자가 선택한 편집기에 의존하게 된다. (`nano`일 필요는 없다.) 텍스트 편집기에 대한 활기로, The Unix Shell의 “Which Editor?” 부분을 참고한다.

```
$ nano mars.txt
```

`mars.txt` 파일에 다음 텍스트를 타이핑한다:

```
Cold and dry, but everything is my favorite color
```

`mars.txt` 파일은 이제 한 줄을 포함하게 되어서, 다음 명령어로 내용을 확인할 수 있다:

```
$ ls
```

```
mars.txt
```

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
```

다시 한번 프로젝트의 상태를 확인하고자 하면, 새로운 파일이 인지되었다고 Git이 일러준다:

```
$ git status
```

```
On branch master
```

```
Initial commit
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
    mars.txt
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

“untracked files” 메시지가 의미하는 것은 Git가 추적하고 있지 않는 파일 하나가 디렉토리에 있다는 것이다. git add를 사용해서 Git에게 추적관리하라고 일러준다:

```
$ git add mars.txt
```

그리고 나서, 올바르게 처리되었는지 확인한다:

```
$ git status
```

```
On branch master
```

```
Initial commit
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

```
    new file:   mars.txt
```

이제 Git은 mars.txt 파일을 추적할 것이라는 것을 알고 있지만, 커밋으로 아직 저장소에는 어떤 변경사항도 기록되지 않았다. 이를 위해서 명령어 하나 더 실행할 필요가 있다:

```
$ git commit -m "Start notes on Mars as a base"
```

```
[master (root-commit) f22b25e] Start notes on Mars as a base
```

```
1 file changed, 1 insertion(+)
```

```
create mode 100644 mars.txt
```

git commit을 실행할 때, Git은 git add를 사용해서 저장하려고 하는 모든 대상을 받아서 .git 디렉토리 내부에 영구적으로 사본을 저장한다. 이 영구 사본을 커밋(commit) (혹은 수정(revision))이라고 하고, 짧은 식별자는 f22b25e이다. (여러분의 커밋번호의 짧은 식별자는 다를 수 있다.)

`-m` (“message”를 의미) 플래그를 사용해서 나중에 무엇을 왜 했는지 기억에 도움이 될 수 있는 주석을 기록한다. `-m` 옵션 없이 `git commit`을 실행하면, Git은 `nano`(혹은 처음에 `core.editor`에서 설정한 다른 편집기)를 실행해서 좀더 긴 메시지를 작성할 수 있다.

좋은 커밋 메시지(Good commit messages) 작성은 커밋으로 만들어진 간략한(영문자 기준 50문자 이하) 변경사항 요약으로 시작된다. 일반적으로 메시지는 완전한 문장이 되어야 한다. 예를 들어, “If applied, this commit will”. 만약 좀 더 상세한 사항을 남기려면, 요약줄 사이에 빈줄을 추가하고 추가적인 내역을 적는다. 추가되는 공간에 왜 변경을 하는지 사유를 남기고, 어떤 영향을 미치는지도 기록한다.

이제 `git status`를 시작하면:

```
$ git status
```

```
On branch master
nothing to commit, working directory clean
```

모든 것이 최신 상태라고 보여준다. 최근에 작업한 것을 알고자 한다면, `git log`를 사용해서 프로젝트 이력을 보여주도록 Git에게 명령어를 보낸다:

```
$ git log
```

```
commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 09:51:46 2013 -0400
```

```
Start notes on Mars as a base
```

`git log`는 시간 역순으로 저장소의 모든 변경사항을 나열한다. 각 수정사항 목록은 전체 커밋 식별자(앞서 `git commit` 명령어로 출력한 짧은 문자와 동일하게 시작), 수정한 사람, 언제 생성되었는지, 커밋을 생성할 때 Git에 남긴 로그 메시지가 포함된다.

내가 작성한 변경사항은 어디있나?

이 시점에서 `ls` 명령어를 다시 실행하면, `mars.txt` 파일만 덩그러니 보게 된다. 왜냐하면, Git이 앞에서 언급한 `.git` 특수 디렉토리에 파일 변경 이력 정보를 저장했기 때문이다. 그래서 파일 시스템이 뒤죽박죽되지 않게 된다. (따라서, 옛 버전을 실수로 편집하거나 삭제할 수 없다.)

이제 드라큘라가 이 파일에 정보를 더 추가했다고 가정하자. (다시 한번 `nano` 편집기로 편집하고 나서 `cat`으로 파일 내용을 살펴본다. 다른 편집기를 사용할 수도 있고, `cat`으로 파일 내용을 꼭 볼 필요도 없다.)

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
```

git status를 실행하면, Git이 이미 알고 있는 파일이 변경되었다고 일러준다:

```
$ git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   mars.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

마지막 줄이 중요한 문구다: “no changes added to commit”. mars.txt 파일을 변경했지만, 아직 Git에게는 변경을 사항을 저장하려고 하거나 (git add로 수행), 저장소에 저장하라고 (git commit로 수행) 일러주지도 않았다. 이제 행동에 나서보자. 저장하기 전에 변경사항을 항상 검토하는 것은 좋은 습관이다. git diff를 사용해서 작업 내용을 두번 검증한다. git diff는 현재 파일의 상태와 가장 최근에 저장된 버전의 차이를 보여준다:

```
$ git diff
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..315bf3a 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,2 @@
 Cold and dry, but everything is my favorite color
 +The two moons may be a problem for Wolfman
```

출력 결과가 암호같은데 이유는 한 파일이 주어졌을 때 다른 파일 하나를 어떻게 재구성하는지를 일러주는 patch와 편집기 같은 도구를 위한 일련의 명령어라서 그렇다. 만약 해당 내역을 조각내서 쪼개다면:

- 첫번째 행은 Git이 신규 파일과 옛 버전 파일을 비교하는 유닉스 diff 명령어와 유사한 출력결과를 생성하고 있다.
- 두번째 행은 정확하게 Git이 파일 어느 버전을 비교하는지 일러준다; df0654a와 315bf3a은 해당 버전에 대해서 중복되지 않게 컴퓨터가 생성한 표식이다.
- 세번째와 네번째 행은 변경되는 파일 명칭을 다시한번 보여주고 있다.
- 나머지 행이 가장 흥미롭다. 실제 차이가 나는 것과 어느 행에서 발생했는지 보여준다. 특히 첫번째 열의 + 기호는 어디서 행이 추가 되었는지 보여준다.

변경사항 검토후에, 변경사항을 커밋(commit)하자.

```
$ git commit -m "Add concerns about effects of Mars' moons on Wolfman"
$ git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   mars.txt
```

no changes added to commit (use "git add" and/or "git commit -a")

이럴 수가, git add을 먼저 하지 않아서 Git이 커밋을 할 수 없다. 고쳐봅시다:

```
$ git add mars.txt
$ git commit -m "Add concerns about effects of Mars' moons on Wolfman"
```

```
[master 34961b1] Add concerns about effects of Mars' moons on Wolfman
 1 file changed, 1 insertion(+)
```

실제로 무엇을 커밋하기 전에 커밋하고자하는 파일을 먼저 추가하라고 Git이 주문하는데, 이유는 한번에 모든것을 커밋하지 싶지 않을 수도 있기 때문이다. 예를 들어, 작성하고 있는 논문에 지도교수 논문을 일부 인용하여 추가한다고 가정하자. 논문 중간에 인용되는 추가부분과 상응되는 참고문헌을 커밋하고는 싶지만, 결론 부분을 커밋하고는 싶지 않다. (아직 결론이 완성되지 않았다.)

이런 점을 고려해서, Git은 특별한 준비 영역(staging)이 있어서 현재 변경부분(change set)을 추가는 했으나 아직 커밋하지 않는 것을 준비 영역에서 추적하고 있다.

준비 영역(Staging area)

프로젝트 기간 동안에 걸쳐 발생된 변경사항에 대해 스냅사진을 찍는 것으로 Git을 바라보면, git add 명령어는 무엇이 스냅사진(준비영역에 놓는 것)에 들어갈지 명세하고, git commit 명령어는 실제로 스탠사진을 찍는 것이다. 만약 git commit을 타이핑할 때 준비된 어떤 것도 없다면, Git이 git commit -a 혹은 git commit --all 명령어 사용을 재촉한다. 사진을 찍으려고 모두 모이세요 하는 것과 같다. 하지만, 준비영역에 추가할 것을 명시적으로 하는 것이 항상 좋다. 왜냐하면 커밋을 했는데 잊은 것이 있을 수도 있기 때문이다. (스냅사진으로 돌아가서, -a 옵션을 사용했기 때문에 스냅사진에 들어갈 항목을 불완전하게 작성했을 수도 있다!) 수작업으로 준비영역에 올리거나, 원하는 것보다 많은 것을 올렸다면 “git undo commit”을 찾아보라.

파일 변경사항을 편집기에서 준비 영역으로, 그리고 장기 저장소로 옮기는 것을 살펴보자. 먼저, 파일에 행 하나를 더 추가한다:

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

```
$ git diff
```

```
diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

지금까지 좋다. 파일의 끝에 행을 하나 추가했다(첫 열에 +이 보인다). 이제, 준비영역에 변경 사항을 놓고, git diff 명령어가 보고하는 것을 살펴보자:

```
$ git add mars.txt
$ git diff
```

출력결과가 없다. Git이 일러줄 수 있는 것은 영구히 저장되는 것과 현재 디렉토리에 작업하고 있는 것에 차이가 없다는 것이다. 하지만, 다음과 같이 명령어를 친다면:

```
$ git diff --staged
```

```
diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

마지막으로 커밋된 변경사항과 준비 영역(Staging)에 있는 것과 차이를 보여준다. 변경사항을 저장하자:

```
$ git commit -m "Discuss concerns about Mars' climate for Mummy"
```

```
[master 005937f] Discuss concerns about Mars' climate for Mummy
1 file changed, 1 insertion(+)
```

현재 상태를 확인하자:

```
$ git status
```

```
On branch master
nothing to commit, working directory clean
```

그리고 지금까지 작업한 이력을 살펴보자:

```
$ git log
```

```
commit 005937fbe2a98fb83f0ade869025dc2636b4dad5
Author: Vlad Dracula <vlad@tran.sylvan.ia>
```

Date: Thu Aug 22 10:14:07 2013 -0400

Discuss concerns about Mars' climate for Mummy

commit 34961b159c27df3b475cfe4415d94a6d1fcd064d

Author: Vlad Dracula <vlad@tran.sylvan.ia>

Date: Thu Aug 22 10:07:21 2013 -0400

Add concerns about effects of Mars' moons on Wolfman

commit f22b25e3233b4645dabd0d81e651fe074bd8e73b

Author: Vlad Dracula <vlad@tran.sylvan.ia>

Date: Thu Aug 22 09:51:46 2013 -0400

Start notes on Mars as a base

단어 단위 차이분석(Word-based diffing)

경우에 따라서는 줄단위로 텍스트 차이 분석이 너무 자세하지 않을 수도 있다.
git diff 명령어에 --color-words 선택옵션이 유용할 수 있는데 이유는 색상을 사용해서 변경된 단어를 강조해서 표시해 주기 때문이다.

로그 페이지별 보기

화면에 git log 출력결과가 너무 긴 경우, git에 화면 크기에 맞춰 페이지 단위로 쪼개주는 프로그램이 제공된다. 페이지별 쪼개보기("pager")가 호출되면, 화면 마지막 줄에 프롬프트 대신에 :이 나타난다.

- 페이지(pager)에서 나오려면, Q를 타이핑한다.
- 다음 페이지로 이동하려면, Spacebar를 타이핑한다.
- 전체 페이지에서 특정 단어를 검색하려면, / 타이핑하고, and 특정단어를 검색하는 를 타이핑한다. 검색에 매칭되는 단어를 따라가려면 N을 타이핑한다.

로그 크기 제한걸기

git log가 전체 터미널 화면을 접수하는 것을 피하려면, -N 선택옵션을 적용해서 Git이 화면에 출력하는 커밋 숫자에 제한을 건다. 여기서 -N은 보고자 하는 커밋 갯수가 된다. 예를 들어 가장 마지막 커밋만 보려고 한다면 다음과 같이 타이핑한다:

\$ git log -1

commit 005937fbe2a98fb83f0ade869025dc2636b4dad5

Author: Vlad Dracula <vlad@tran.sylvan.ia>

Date: Thu Aug 22 10:14:07 2013 -0400

Discuss concerns about Mars' climate for Mummy

--oneline 선택옵션을 사용해서 출력되는 로그 메시지 크기를 줄일 수도 있다:

\$ git log --oneline

```
* 005937f Discuss concerns about Mars' climate for Mummy
* 34961b1 Add concerns about effects of Mars' moons on Wolfman
* f22b25e Start notes on Mars as a base
```

--oneline 선택옵션과 다른 선택옵션을 조합할 수도 있다. 유용한 조합 사례로 다음이 있다:

```
$ git log --oneline --graph --all --decorate

* 005937f Discuss concerns about Mars' climate for Mummy (HEAD, master)
* 34961b1 Add concerns about effects of Mars' moons on Wolfman
* f22b25e Start notes on Mars as a base
```

디렉토리

Git에서 디렉토리에 관해서 알아두면 좋을 두가지 사실.

1. Git은 그 자체로 디렉토리를 추적하지 않고, 디렉토리에 담긴 파일만 추적한다. 믿지 못하겠다면, 직접 다음과 같이 시도해 본다:

```
$ mkdir directory
$ git status
$ git add directory
$ git status
```

새로 생성된 directory 이름을 갖는 디렉토리가 git add 명령어로 명시적으로 추가했음에도 불구하고 untracked files 목록에 나오지 않고 있다. 이런 이유로 인해서 가끔 .gitkeep 파일을 보게 된다. .gitignore와 달리, 특별하지는 않고 유일한 목적은 디렉토리를 만들어 내어 Git이 저장소에 추가하도록 하는 역할만 수행한다. 사실 원하는 이름으로 파일명을 붙일 수 있다.

2. Git 저장소에 디렉토리를 생성하고 파일로 채워넣으면, 다음과 같이 디렉토리의 모든 파일을 추가할 수 있다:

```
git add <directory-with-files>
```

요약하면, 변경사항을 저장소에 추가하고자 할 때, 먼저 변경된 파일을 준비 영역(Staging)에 git add 명령어로 추가하고 나서, 준비 영역의 변경사항을 저장소에 git commit 명령어로 최종 커밋한다:

커밋 메시지 고르기

다음 중 어떤 커밋 메시지가 mars.txt 파일의 마지막 커밋으로 가장 적절할까요?

1. “Changes”
2. “Added line ‘But the Mummy will appreciate the lack of humidity’ to mars.txt”
3. “Discuss effects of Mars’ climate on the Mummy”

해답 1번은 충분히 기술되어 있지 못하고 커밋 목적이 불확실하다;
2번은 “git diff” 명령어를 사용한 것과 불필요하게 중복된다; 3번이 좋다: 짧고, 기술이 잘되어 있고, 피할 수 없게 명백하다(imperative).

Git에 변경사항 커밋하기

다음 중 어떤 명령어가 로컬 Git 저장소에 myfile.txt 파일 변경사항을 저장시키는 걸까?

1. \$ git commit -m "my recent changes"
2. \$ git init myfile.txt
\$ git commit -m "my recent changes"
3. \$ git add myfile.txt
\$ git commit -m "my recent changes"
4. \$ git commit -m myfile.txt "my recent changes"

해답

1. 파일이 이미 준비영역(staging)에 올라온 경우만 커밋이 생성된다.
2. 신규 저장소를 생성하게 된다.
3. 정답: 파일을 준비영역에 추가하고 나서, 커밋하게 된다.
4. myfile.txt 파일에 "my recent changes" 메시지를 갖는 커밋을 생성한다.

파일 다수를 커밋

준비영역(staging area)은 스냅샷 한번에 원하는 만큼 파일을 변경사항을 담아 낼 수 있다. 1. mars.txt 파일에 전진기지로 생각하는 금성(Venus)를 고려하고 있다는 결정을 담은 텍스트를 추가한다. 2. venus.txt 파일을 새로 생성해서 본인과 친구들에게 금성에 관한 첫생각을 담아낸다. 3. 파일 두개에 변경사항을 준비영역에 추가하고 커밋한다.

해답

먼저, mars.txt, venus.txt 파일에 변경사항을 기록한다:

```
$ nano mars.txt
$ cat mars.txt
```

Maybe I should start with a base on Venus.

```
$ nano venus.txt
$ cat venus.txt
```

Venus is a nice planet and I definitely should consider it as a base.

준비영역에 파일 두개를 추가한다. 한줄로 추가작업을 수행할 수 있다:

```
$ git add mars.txt venus.txt
```

혹은 명령어를 다수 타이핑하면 된다:

```
$ git add mars.txt
$ git add venus.txt
```

이제 파일을 커밋할 준비가 되었다. git status를 사용해서 확인하면, 커밋을 할 준비가 되었다:

```
$ git commit -m "Write plans to start a base on Venus"

[master cc127c2]
  Write plans to start a base on Venus
  2 files changed, 2 insertions(+)
  create mode 100644 venus.txt
```

bio 저장소

- bio라는 새로운 Git 저장소를 본인 로컬 컴퓨터에 생성한다.
- me.txt라는 파일로 본인에 대한 3줄 이력서를 작성한다. 변경사항을 커밋한다.
- 그리고 나서 한줄을 바꾸고, 네번째 줄을 추가하고 나서,
- 원래 상태와 갱신된 상태의 차이를 화면에 출력한다.

해답

필요하다면, planets 폴더에서 빠져나온다:

```
$ cd ..
```

bio 폴더를 새로 생성하고 bio 폴더로 이동한다:

```
$ mkdir bio
$ cd bio
```

git 명령어로 초기화한다:

```
$ git init
```

nano 혹은 선호하는 편집기를 사용해서 me.txt 파일에 본인 일대기를 작성한다. 파일을 추가하고 나서, 저장소에 커밋한다:

```
$ git add me.txt
$ git commit -m 'Adding biography file'
```

기술된 것(한줄 변경하고, 4번째 줄을 추가한다)처럼 파일을 변경한다. 원본 상태와 수정된 상태를 git diff 명령어를 사용해서 화면에 출력한다:

```
$ git diff me.txt
```

저자(Author)와 커미터(Committer)

매번 커밋을 할 때마다, Git은 이름을 두번 저장한다. 본인 이름이 저자(Author)와 커미터(Committer)로 기록된다. 마지막 커밋에 추가 정보를 Git에게 요구하면 확인이 가능하다:

```
$ git log --format=full
```

커밋할 때, 저자를 다른 누군가로 바꿀 수 있다:

```
$ git commit --author="Vlad Dracula <vlad@tran.sylvan.ia>"
```

커밋을 두개 생성한다: 하나는 --author 옵션을 갖는 것으로 저자로 동료이름을 반영한다. git log와 git log --format=full 명령어를 실행한다. 이런 방식이 동료와 협업하는 방식이 될 수도 있겠다고는 생각이 될 수 있다.

해법

```
$ git add me.txt
$ git commit -m "Update Vlad's bio." --author="Frank N. Stein <franky@monster.com>

[master 4162a51] Update Vlad's bio.
Author: Frank N. Stein <franky@monster.com>
1 file changed, 2 insertions(+), 2 deletions(-)

$ git log --format=full
commit 4162a51b273ba799a9d395dd70c45d96dba4e2ff
Author: Frank N. Stein <franky@monster.com>
Commit: Vlad Dracula <vlad@tran.sylvan.ia>

Update Vlad's bio.

commit aaa3271e5e26f75f11892718e83a3e2743fab8ea
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Commit: Vlad Dracula <vlad@tran.sylvan.ia>

Vlad's initial bio.
```


Chapter 12

이력 탐색

앞선 학습에서 살펴봤듯이, 식별자로 커밋을 조회할 수 있다. HEAD 식별자를 사용해서 작업 디렉토리의 가장 최근 커밋을 조회할 수 있다.

`mars.txt` 파일에 한번에 한줄씩 추가했다. 따라서, 눈으로 봐도 진행사항을 쉽게 추적할 수 있다. HEAD를 사용해서 추적작업을 수행해보자. 시작전에 `mars.txt` 파일에 변경을 가해보자.

```
$ nano mars.txt  
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color  
The two moons may be a problem for Wolfman  
But the Mummy will appreciate the lack of humidity  
An ill-considered change
```

이제, 변경된 사항을 살펴보자.

```
$ git diff HEAD mars.txt
```

```
diff --git a/mars.txt b/mars.txt  
index b36abfd..0848c8d 100644  
--- a/mars.txt  
+++ b/mars.txt  
@@ -1,3 +1,4 @@  
 Cold and dry, but everything is my favorite color  
 The two moons may be a problem for Wolfman  
 But the Mummy will appreciate the lack of humidity  
+An ill-considered change.
```

HEAD만 빼면, 앞서 살펴본 것과 동일하다. 이러한 접근법의 정말 좋은 점은 이전 커밋을 조회할 수 있다는 점이다. ~1(~)은 “틸드(tilde)”, 발음기호 [til-duh]을 추가해서 HEAD 이전 첫번째 커밋을 조회할 수 있다.

```
$ git diff HEAD~1 mars.txt
```

git diff 명령어를 사용해서 이전 커밋과 차이난 점을 보고자 한다면, HEAD~1, HEAD~2 표기법을 사용해서 조회를 쉽게 할 수 있다:

```
$ git diff HEAD~2 mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,4 @@
    Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
+An ill-considered change
```

git show를 사용해서도 커밋 메시지 분만 아니라 이전 커밋과 변경사항을 보여준다.
git diff는 작업 디렉토리와 커밋 사이 차이나는 부분을 보여준다.

```
$ git show HEAD~2 mars.txt
```

```
commit 34961b159c27df3b475cfe4415d94a6d1fcd064d
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:07:21 2013 -0400
```

Start notes on Mars as a base

```
diff --git a/mars.txt b/mars.txt
new file mode 100644
index 0000000..df0654a
--- /dev/null
+++ b/mars.txt
@@ -0,0 +1 @@
+Cold and dry, but everything is my favorite color
```

이런 방식으로, 연쇄 커밋 사슬을 구성할 수 있다. 가장 최근 사슬의 끝값은 HEAD로 조회된다: ~ 표기법을 사용하여 이전 커밋을 조회할 수 있다. 그래서 HEAD~1("head 마이너스 1"으로 읽는다.)은 "바로 앞선 커밋"을 의미하고, HEAD~123은 지금 있는 위치에서 123번째 이전 수정으로 간다는 의미가 된다.

커밋된 것을 git log 명령어로 화면에 뿌려주는 숫자와 문자로 구성된 긴 문자열을 사용하여 조회할 수도 있다. 변경사항에 대해서 중복되지 않는 ID로 "중복되지 않는(unique)"의 의미는 정말 유일하다는 의미다: 특정 컴퓨터에 있는 임의 파일 집합에 대한 모든 변경사항은 중복되지 않는 40-문자 식별자가 붙어있다. 첫번째 커밋은 ID로 f22b25e3233b4645dabd0d81e651fe074bd8e73b 0이 주어졌다. 그래서 다음과 같이 시도하자:

```
$ git diff f22b25e3233b4645dabd0d81e651fe074bd8e73b mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..93a3e13 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,4 @@
    Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
+An ill-considered change
```

올바른 정답이지만, 난수 40-문자로 된 문자열을 타이핑하는 것은 매우 귀찮은 일이다. 그래서 Git 앞의 몇개 문자만으로도 사용할 수 있게 했다:

```
$ git diff f22b25e mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..93a3e13 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,4 @@
    Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
+An ill-considered change
```

좋았어요! 파일에 변경사항을 저장할 수 있고 변경된 것을 확인할 수 있다. 어떻게 옛 버전 파일을 되살릴 수 있을까? 우연히 파일을 덮어썼다고 가정하자:

```
$ nano mars.txt
$ cat mars.txt
```

We will need to manufacture our own oxygen

이제 git status를 통해서 파일이 변경되었다고 하지만, 변경사항은 아직 준비영역(Staging area)에 옮겨지지 않은 것으로 확인된다:

```
$ git status

On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   mars.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

git checkout 명령어를 사용해서 과거에 있던 상태로 파일을 돌려 놓을 수 있다:

```
$ git checkout HEAD mars.txt
$ cat mars.txt
```

Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity

이름에서 유추할 수 있듯이, git checkout 명령어는 파일 옛 버전을 확인하고 갖고 나간다. 즉, 되살린다. 이 경우 HEAD에 기록된 가장 최근에 저장된 파일 버전을 되살린다. 좀 더 오래된 버전을 되살리고자 한다면, 대신에 커밋 식별자를 사용한다:

```
$ git checkout f22b25e mars.txt
$ cat mars.txt
```

Cold and dry, but everything is my favorite color

```
$ git status
```

```
# On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
# Changes not staged for commit:
  #   (use "git add <file>..." to update what will be committed)
  #   (use "git checkout -- <file>..." to discard changes in working directory)
  #
  #     modified:   mars.txt
  #
no changes added to commit (use "git add" and/or "git commit -a")
```

변경사항은 준비영역에 머물러 있는 것에 주목한다. 다시, git checkout 명령어를 사용해서 이전버전으로 되돌아 간다:

```
$ git checkout HEAD mars.txt
```

헤드(HEAD)를 잊지 말자

f22b25e 커밋 상태로 mars.txt 파일을 되돌리는데 앞서 다음 명령어를 사용했다.

```
$ git checkout f22b25e mars.txt
```

하지만 주의하자! checkout 명령어는 다른 중요한 기능을 갖고 있고 만약 타이핑에 오류가 있다면 의도를 Git이 오해할 수 있다. 예를 들어, 앞선 명령에서 mars.txt를 빼먹게 되면...

```
$ git checkout f22b25e
```

Note: checking out 'f22b25e'.

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

```
If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:
git checkout -b <new-branch-name>
HEAD is now at f22b25e Start notes on Mars as a base
```

“detached HEAD”는 “보기는 하지만 건드리지는 마시오”와 같다. 따라서 현재 상태에서 어떤 변경도 만들지 말아야한다. 저장소 지난 상태를 살펴본 후에 git checkout master 명령어로 HEAD를 다시 불린다.

실행 취소를 하는 변경을 하기 *전에** 저장소 상태를 확인하는 커밋 번호를 사용해야 한다는 것을 기억하는 것이 중요하다. 흔한 실수는 커밋 번호를 사용하는 것이다. 아래 예제에서는 커밋 번호가 f22b25e인 가장 최신 커밋(HEAD~1) 앞의 상태로 다시 되돌리고자 한다:

그래서, 모두 한군데 놓아보자:

흔한 사례 단순화

git status 출력결과를 주의깊이 읽게 되면, 힌트가 포함된 것을 볼 수 있다.

(use "git checkout -- <file>..." to discard changes in working directory)

출력결과가 언급하는 바는, 버전 식별자 없이 git checkout 명령어를 실행하게 되면 HEAD에 저장된 상태로 파일을 원복시킨다. 더블 대쉬 --가 필요한 경우는 명령어 자체로부터 복구해야 되는 파일명을 구별할 때다: 없는 경우, 커밋 식별자에 Git은 파일명을 사용한다.

파일이 하나씩 하나씩 옛 상태로 되돌린다는 사실이 사람들이 작업을 조직하는 방식에 변화를 주는 경향이 있다. 모든 것이 하나의 큰 문서로 되어있다면, 나중에 결론부분에 변경사항을 실행취소하지 않고, 소개부분에 변경을 다시 되돌리기가 쉽지 않다(하지만 불가능하지는 않다). 다른 한편으로 만약 소개부분과 결론부분이 다른 파일에 저장되어 있다면, 시간 앞뒤로 이동하기가 훨씬 쉽다.

파일 이전 버전 복구하기

정훈이가 몇주동안 작업한 파이썬 스크립트에 변경을 했고, 오늘 아침 정훈이가 작업한 변경사항이 스크립트를 “망가 먹어서” 더이상 실행이 되지 않는다. 복도 없이, 버그를 고치는데 1시간 이상 소모했다…

다행스럽게도, Git을 사용한 프로젝트 버전을 추적하고 있었다! 다음 아래 명령어 중 어떤 것이 data_cruncher.py로 불리는 파이썬 스크립트 가장 최근 버전을 복구하게 할까요?

1. \$ git checkout HEAD
2. \$ git checkout HEAD data_cruncher.py
3. \$ git checkout HEAD~1 data_cruncher.py
4. \$ git checkout <unique ID of last commit> data_cruncher.py
5. Both 2 and 4

커밋 되돌리기(Reverting a Commit)

정훈이는 동료와 함께 파이썬 코드를 협업해서 작성하고 있다. 그룹 저장소에 마지막으로 커밋한 것이 잘못된 것을 알게 되서, 실행취소하여 원복하고자 한다.

정훈이는 실행취소를 올바르게 해서 그룹저장소를 사용하는 모든 구성원이 제대로된 변경사항을 가지고 작업을 계속하길 원한다. `git revert [ID]` 명령어는 정훈이가 이전에 잘못 커밋했던 작업에 대해 실행취소하는 커밋을 새로 생성시킨다.

따라서, `git revert`는 `git checkout [ID]`와 다른데 이유는 `checkout`이 그룹저장소에 커밋되지 않는 로컬 변경사항에 대해서 적용된다는 점에서 차이가 난다. 정훈이가 `git revert`를 사용할 올바른 절차와 설명이 아래에 나와있다. 빠진 명령어가 무엇일까?

1. ‘_____ # 커밋 ID를 찾을 수 있도록 Git 프로젝트 이력을 살펴본다.
2. ID를 복사한다. (ID의 첫 문자 몇개만 사용한다. 예를 들어, 0b1d055).
3. `git revert [ID]`
4. 새로운 커밋 메시지를 타이핑한다.
5. 저장하고 종료한다.

작업흐름과 이력 이해하기

다음 마지막 명령의 출력결과는 무엇일까?

```
$ cd planets
$ echo "Venus is beautiful and full of love" > venus.txt
$ git add venus.txt
$ echo "Venus is too hot to be suitable as a base" >> venus.txt
$ git commit -m "Comment on Venus as an unsuitable base"
$ git checkout HEAD venus.txt
$ cat venus.txt #this will print the contents of venus.txt to the screen
```

1. Venus is too hot to be suitable as a base
2. Venus is beautiful and full of love
3. Venus is beautiful and full of love
 Venus is too hot to be suitable as a base
4. Error because you have changed venus.txt without committing the changes

해법

정답은 2. 왜냐하면, `git add venus.txt`가 `Venus is too hot to be suitable as a base` 행을 추가하기 전에만 적용된다. `git checkout`이 실행될 때 반영이 되지 않아서 그렇다. `git commit` 명령어에 `-a` 플래그를 사용하게 되면 이런 손실을 막을 수 있다.

git diff 이해 확인하기

`git diff HEAD~3 mars.txt` 명령어를 고려해 보자. 이 명령어를 실행하게 되면 실행결과로 예상하는 바를 말해보세요. 명령어를 실행하게 되면 어떤 일이 발생하는가? 그리고 이유는 무엇인가?

또 다른 명령어 `git diff [ID] mars.txt`를 시도해 보자. 여기서, [ID]를 가장 최근 커밋 식별자로 치환한다. 무슨 일이 생길까? 그리고 실제로 생긴 일은 무엇인가?

준비 단계 변경사항(Staged Changes) 제거하기

`git checkout` 명령어를 통해서 준비영역으로 올라오지 않은 변경사항이 있을 때, 이전 커밋을 복구할 수 있었다. 하지만, `git checkout`은 준비영역에 올라왔지만, 커밋되지 않는 변경사항에 대해서도 동작한다. `mars.txt` 파일에 변경사항을 만들고, 변경사항을 추가하고 나서, `git checkout` 명령어를 사용하게 되면 변경사항이 사라졌는지 살펴보자.

변경 이력 탐색과 요약

변경 이력 탐색은 Git에 있어 중요한 부분 중의 하나로, 특히 커밋이 수개월 전에 이뤄졌다면, 올바른 커밋 ID를 찾는 것이 종종 크나큰 도전과제가 된다. `planets` 프로젝트가 50 파일 이상으로 구성되었다고 상상해 보자.

`mars.txt` 파일에 특정 텍스트가 변경된 커밋을 찾고자 한다. `git log`를 타이핑하게 되면 매우 긴 목록이 출력된다. 어떻게 하면 검색범위를 좁힐 수 있을까? `git diff` 명령어가 특정 파일만 탐색할 수 있단느 점을 상기하자.

예를 들어, `git diff mars.txt`. 이 문제에 유사한 아이디어를 적용해 보자.

```
$ git log mars.txt
```

불행하게도 커밋 메시지 일부는 매우 애매모호하다. 예를 들어, `update files`. 어떻게 하면 파일을 잘 검색할 수 있을까? `git diff`, `git log` 명령어 모두 매우 유용하다. 두 명령어 모두 변경이력의 다른 부분을 요약해준다. 둘을 조합하는 것은 가능할까? 다음 명령어를 실행해 보자:

```
$ git log --patch mars.txt
```

엄청 긴 출력 목록이 나타난다. 각 커밋마다 커밋 메시지와 차이가 쪽 출력된다. 질문: 다음 명령어는 무슨 작업을 수행할까요?

```
$ git log --patch HEAD~3 *.txt
```


Chapter 13

추적대상에서 제외

만약 Git가 추적하기 않았으면 하는 파일이 있다면 어떨까요? 편집기에서 자동 생성되는 백업파일 혹은 자료 분석 중에 생성되는 중간 임시 파일이 좋은 예가 된다. 몇개 마루타 더미(dummy) 파일을 생성하자:

```
$ mkdir results  
$ touch a.dat b.dat c.dat results/a.out results/b.out
```

그려면 Git은 다음을 보여준다:

```
$ git status  
  
On branch master  
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
  
    a.dat  
    b.dat  
    c.dat  
    results/  
nothing added to commit but untracked files present (use "git add" to track)
```

벼전 제어 아래 이런 파일을 놓는 것은 디스크 공간 낭비다. 더 좋은 않는 것은, 이런 파일을 모두 관리목록에 넣는 것이 실제로 중요변경사항을 관리하는데 집중하지 못하게 한다는 것이다. 그래서 Git에게 중요하지 않는 이런 파일을 무시하게 일러준다.

.gitignore라는 프로젝트 루트 디렉토리에 파일을 생성해서 무시할 것을 명기함으로써 해당작업을 수행한다:

```
$ nano .gitignore  
$ cat .gitignore
```

```
*.dat
results/
```

상기 패턴은 .dat 확장자를 갖는 임의 파일과 results 디렉토리에 있는 모든 것을 무시한다. (하지만, 이들 파일 중 일부가 이미 추적되고 있다면, Git은 계속 추적한다.)

.gitignore 파일을 생성하자마자, git status 출력결과는 훨씬 깨끗해졌다:

```
$ git status
```

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
  .gitignore
nothing added to commit but untracked files present (use "git add" to track)
```

이제 Git가 알아차리는 유일한 것은 새로 생성된 .gitignore 파일이 된다. 우리는 이들 파일을 추적하여 관리하지 않는다고 생각할 수도 있지만, 우리와 저장소를 공유하고 있는 다른 모든 사람도 우리가 추적관리하지 않는 동일한 것을 무시하고 싶을 것이다. .gitignore 를 추가하고 커밋하자:

```
$ git add .gitignore
$ git commit -m "Ignore data files and the results folder."
$ git status
```

```
# On branch master
nothing to commit, working directory clean
```

보너스로, .gitignore는 실수로 추적하고 싶지 않는 파일이 저장소에 추가되는 것을 피할 수 있게 돋는다:

```
$ git add a.dat
```

```
The following paths are ignored by one of your .gitignore files:
a.dat
Use -f if you really want to add them.
```

만약 .gitignore 설정에 우선해서 파일을 추가하려면, git add -f를 사용해서 강제로 Git에 파일을 추가할 수 있다. 예를 들어, git add -f a.dat. 추적관리되지 않는 파일의 상태를 항상 보려면 다음을 사용한다:

```
$ git status --ignored
```

```
On branch master
Ignored files:
  (use "git add -f <file>..." to include in what will be committed)

  a.dat
```

```
b.dat
c.dat
results/
nothing to commit, working directory clean
```

중첩된 파일 추적하지 않기

디렉토리 구조가 다음과 같다:

```
results/data
results/plots
```

어떻게 하면 results/plots 만 추적하지 않을 수 있을까? results/data 디렉토리는 추적한다.

해답

대부분의 프로그래밍 이슈와 마찬가지로, 이 문제를 해결하는 몇 가지 방식이 있다. results/plots 디렉토리 콘텐츠만 추적하지 않기로 한다면, .gitignore 파일에 /plots/ 폴더만 추적하지 않도록 다음과 같이 .gitignore 파일을 변경하면 된다:

```
results/plots/
```

대신에 /results/ 디렉토리에 모든 것을 추적하지 않지만, results/data만 추적하고자 한다면, results/ 폴더를 .gitignore 파일에 추가하고 results/data/ 폴더에 대해서 예외를 생성한다. 다음 도전과제가 이런 유형의 해법을 다루게 된다.

종종 ** 패턴이 사용하기 수월한데, 다수 디렉토리와 매칭을 지원한다. 예를 들어, **/results/plots/*은 루트 디렉토리에 results/plots 디렉토리를 추정하기 어렵게 한다.

특정 파일만 포함시키기

final.data 파일만 제외하고 모든 .data 파일은 추적하지 않고자 하면 어떻게 하면 될까? 힌트: ! (느낌표 연산자) 부호가 수행하는 작업을 알아본다.

해법

.gitignore 파일에 다음 두 줄을 추가한다:

```
*.data          # data      .
!final.data     # final.data .
```

느낌표 연산자가 앞서 제외된 항목을 포함시키게 한다.

디렉토리에 모든 파일 추적하지 않기

디렉토리 구조가 다음과 같다:

```
results/data/position/gps/a.data
results/data/position/gps/b.data
results/data/position/gps/c.data
```

```
results/data/position/gps/info.txt
results/plots
```

result/data/position/gps 디렉토리에 모든 .data 파일을 추적하지 않도록 .gitignore 파일에 규칙을 작성하는데 가장 짧은 규칙은 무엇일까? info.txt 파일은 추적하자.

해답

results/data/position/gps 디렉토리에 .data로 끝나는 모든 파일은 results/data/position/gps/*.data 규칙으로 매칭된다. results/data/position/gps/info.txt 파일은 확장자가 달라서 계속 추적된다.

적용 규칙 순서

.gitignore 파일에 다음 내용이 담겨있다:

```
*.data
!*.*.data
```

적용 결과는 어떻게 될까?

해답

! 연산자는 이전에 정의된 추적제외 패턴을 부정한다. .gitignore 파일에서 !*.data 규칙은 앞서 추적에서 제외한 .data 모든 파일 추적제외를 부정한다. 따라서, 어떤 것도 추적제외되지 않고, .data 파일 모두가 추적된다.

로그 파일

스크립트를 작성해서 log_01, log_02, log_03 형태의 중간 로그 파일이 많이 생성되었다. 로그 파일을 보관하고자 하지만, git으로 추적하고 싶지는 않다.

1. log_01, log_02 … 형태 모든 파일을 추적 제외하는 .gitignore 규칙을 하나 작성한다.
2. log_01 형태 마루타 파일을 생성해서 “추적제외 패턴”을 테스트한다.
3. 종국에 log_01 파일이 매우 중요하는 것을 알게 되어서 .gitignore 파일을 변경하지 않고 추적되게 추가한다.
4. 추적하기를 원하지 않지만, .gitignore를 통해서 추적제외할 수 있는 파일이 어떤 유형이 있는지 주변 동료와 상의하자.

해답

1. log_* 혹은 log* 규칙을 .gitignore 파일에 추가한다.
2. git add -f log_01 명령어를 사용해서 log_01 파일에 대한 추적을 수행한다.

Chapter 14

GitHub 원격작업

버전 제어(version control)는 다른 사람과 협업할 때 진정으로 다가온다. 우리는 이미 버전 제어를 위해 필요한 작업 대부분을 수행했다; 한가지 빠진 것은 한 저장소에서 다른 저장소로 변경사항을 복사하는 것이다.

Git 같은 시스템은 임의 두 저장소 사이에 작업을 옮길 수 있는 기능을 제공한다. 하지만, 실무에서 다른 사람의 노트북이나 PC보다는 중앙 허브에 웹 방식으로 하나의 원본을 두고 사용하는 것이 가장 쉽다. 대부분의 프로그래머는 프로그램 마스터 원본을 GitHub, BitBucket, GitLab 호스팅 서비스에 두고 사용한다; 이번 학습 마지막 부분에서 이러한 접근법의 장점과 단점을 살펴본다.

세상 사람들과 현재 프로젝트에서 변경한 사항을 공유하는 것에서부터 시작해보자. GitHub에 로그인하고 나서, 우측 상단 아이콘을 클릭해서 `planets` 이름으로 신규 저장소를 생성한다:

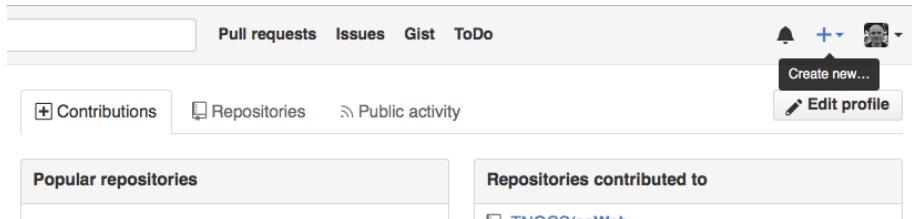


Figure 14.1: (1단계) GitHub 저장소 생성

저장소 이름을 “`planets`”으로 만들고 “Create Repository”를 클릭한다:

저장소가 생성되자 마자, GitHub는 URL을 가진 페이지와 로컬 저장소 환경설정 방법에 대한 정보를 화면에 출력한다:

다음 명령어가 실제로 GitHub 서버에서 자동으로 수행된 것이다:

```
$ mkdir planets
```

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner	<input style="width: 150px; height: 25px; border: 1px solid #ccc; border-radius: 4px; padding: 2px 5px;" type="text" value="mkuzak"/> / <input style="width: 150px; height: 25px; border: 1px solid #ccc; border-radius: 4px; padding: 2px 5px;" type="text" value="planets"/> ✓
<p>Great repository names are short and memorable. Need inspiration? How about supreme-octo-happiness.</p>	
<p>Description (optional)</p> <input style="width: 100%; height: 40px; border: 1px solid #ccc; border-radius: 4px; padding: 5px;" type="text"/>	
<p><input checked="" type="radio"/> Public Anyone can see this repository. You choose who can commit.</p>	
<p><input type="radio"/> Private You choose who can see and commit to this repository.</p>	
<p><input type="checkbox"/> Initialize this repository with a README This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.</p>	
<input style="border: 1px solid #ccc; border-radius: 4px; padding: 2px 5px;" type="button" value="Add .gitignore: None"/>	<input style="border: 1px solid #ccc; border-radius: 4px; padding: 2px 5px;" type="button" value="Add a license: None"/>
<input style="background-color: #4CAF50; color: white; border: none; border-radius: 4px; padding: 5px 15px; font-weight: bold; cursor: pointer; width: 150px;" type="button" value="Create repository"/>	

Figure 14.2: (2단계) GitHub 저장소 생성

[mkuzak / planets](#)

Code Issues 0 Pull requests 0 Wiki Pulse Graphs Settings

Unwatch 1 Star 0 Fork 0

Quick setup — if you've done this kind of thing before

or <https://github.com/mkuzak/planets.git>

We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# planets" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/mkuzak/planets.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/mkuzak/planets.git
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Figure 14.3: (3단계) GitHub 저장소 생성

```
$ cd planets
$ git init
```

`mars.txt` 파일을 추가하고 커밋한 이전 학습을 상기한다면, 로컬 저장소는 다음과 같이 도식적으로 표현할 수 있다:

이제 저장소가 두개로 늘어서, 도식적으로 표현하면 다음과 같다:

현재 로컬 저장소는 여전히 `mars.txt` 파일에 대한 이전 작업정보를 담고 있다. 하지만, GitHub의 원격 저장소에는 아직 어떠한 파일도 담고 있지는 않다:

다음 단계는 두 저장소를 연결하는 것이다. 로컬 저장소를 위해서 GitHub 저장소를 원격(remote)으로 만들어 두 저장소를 연결한다. GitHub의 저장소 홈페이지에 식별하는데 필요한 문자열이 포함되어 있다:



Figure 14.4: GitHub 저장소 URL 발견장소

SSH에서 HTTPS로 프로토콜(protocol)을 변경하려면 ‘HTTPS’ 링크를 클릭한다.

HTTPS vs. SSH

부가적인 설정이 필요하지 않아서 여기서는 HTTPS를 사용한다. 워크샵 후에 SSH 접근 설정을 원할지도 모른다. SSH 접근이 좀더 안전하다. GitHub, Atlassian/BitBucket, GitLab의 훌륭한 지도서 중 하나를 따라하는 것도 좋다. GitLab은 온라인 동영상도 제공한다.



Figure 14.5: GitHub 저장소 URL 변경

웹 브라우저에서 URL을 복사하고 나서, 로컬 컴퓨터 `planets` 저장소로 가서 다음 명령어를 실행한다:

```
$ git remote add origin https://github.com/vlad/planets.git
```

Make sure to use the URL for your repository rather than Vlad's: the only difference should be your username instead of vlad.

Vlad가 아니고 여러분 저장소의 URL을 사용했는지 확인한다: 유일한 차이점은 vlad 대신에 여러분의 사용자이름(username)이다.

`git remote -v` 실행해서 명령어가 제대로 작동했는지 확인한다:

```
$ git remote -v

origin  https://github.com/vlad/planets.git (push)
origin  https://github.com/vlad/planets.git (fetch)

origin 이름이 원격 저장소에 대한 로컬 별명이다. 원한다면 다른 명칭을 사용할 수도 있지만, origin 이름이 가장 일반적인 선택이다.
```

별명이 origin으로 설정되면, 다음 명령어가 변경사항을 로컬 저장소에서 GitHub 원격 저장소로 밀어 넣어 푸쉬(push)한다:

```
$ git push origin master

Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (9/9), 821 bytes, done.
Total 9 (delta 2), reused 0 (delta 0)
To https://github.com/vlad/planets
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

프록시(Proxy)

만약 연결된 네트워크가 프록시를 사용한다면, “Could not resolve hostname” 오류 메시지로 인해서 마지막 명령어가 실패할 가능성이 있다. 이 문제를 해결하기 위해서, 프록시에 대한 정보를 Git에 전달할 필요가 있다:

```
$ git config --global http.proxy http://user:password@proxy.url
$ git config --global https.proxy http://user:password@proxy.url
```

프록시를 사용하지 않는 또 다른 네트워크에 연결될 때, Git에게 프록시 기능을 사용하지 않도록 다음 명령어를 사용하여 일러준다:

```
$ git config --global --unset http.proxy
$ git config --global --unset https.proxy
```

비밀번호 관리자(password manager)

운영체제 위에 비밀번호 관리자(password manager)가 설정되어 있다면, 사용자이름(username)과 비밀번호(password)가 필요로 할 때, git push 명령어가 이를 사용하려 한다. “Git Bash on Windows”를 사용하면 기본 디폴트 행동이다. 관리자 비밀번호를 사용하는 대신에, 터미널에서 사용자이름과 비밀번호를 입력하려면, git push를 실행하기 전에 터미널에서 다음과 같이 타이핑한다:

```
$ unset SSH_ASKPASS
```

git uses SSH_ASKPASS for all credential entry에도 불구하고, SSH나 HTTPS를 경유하여 Git을 사용하든 SSH_ASKPASS을 설정하고 싶지 않을 수도 있다.

~/.bashrc 파일 하단에 unset SSH_ASKPASS을 추가해서 Git으로 하여금 사용자명과 비밀번호를 사용하도록 기본설정으로 둘 수도 있다.

이제 로컬 저장소와 원격 저장소는 다음과 같은 상태가 된다:

'-u' 플래그(flag)

Git 문서에서 git push과 함께 사용되는 -u 옵션을 볼 수 있다. git branch 명령어에 대한 --set-upstream-to 옵션과 동의어에 해당되는 옵션이다. 원격 브랜치를 현재 브랜치와 연결시키는데 사용된다. 그래서 git pull 명령어가 아무런 인자없이 사용될 수 있다. 원격 저장소가 설정되면, git push -u origin master 명령어만 실행시키면 연결작업이 완료된다.

또한, 원격 저장소에서 로컬 저장소로 변경사항을 풀(pull)해서 가져올 수도 있다:

```
$ git pull origin master
```

```
From https://github.com/vlad/planets
 * branch           master    -> FETCH_HEAD
 Already up-to-date.
```

이 경우 가져오기 하는 풀(pull)은 아무런 결과가 없는데, 이유는 두 저장소가 이미 동기화가 되어서다. 하지만, 만약 누군가 GitHub 저장소에 변경사항을 푸쉬했다면, 상기 명령어는 변경된 사항을 로컬 저장소로 다운로드한다.

GitHub GUI

GitHub 웹사이트에서 planets 저장소를 찾아간다. Code 탭아래 “XX commits”(“XX”는 숫자) 텍스트를 클릭한다. 각 커밋 우측의 버튼 세개 여기 저기 눌러보고, 클릭해 본다. 버튼을 눌러서 어떤 정보를 모을 수 있거나 탐색할 수 있는가? 쉘에서 동일한 정보를 어떻게 얻을 수 있을까?

해답 (클립보드 그림을 갖는) 가장 좌측 버튼은 클립보드에 커밋 식별자 전체를 복사한다. 쉘에서 git log 명령어가 각 커밋에 대한 전체 커밋 식별자를 보여준다.

중간 버튼을 클릭하게 되면, 특정 커밋으로 변경한 내용 전체를 확인할 수 있다. 녹색 음영선은 추가를 붉은색 음영선은 삭제를 의미한다. 쉘에서 동일한 작업을 git diff로 할 수 있다. 특히, git diff ID1..ID2(ID1와 ID2은 커밋 식별자다) 명령어(즉, git diff a3bf1e5..041e637)는 두 커밋 사이 차이를 보여준다.

가장 우측 버튼은 커밋 당시에 저장소 모든 파일을 보여준다. 쉘로 이런 작업을 수행하려면, 해당 시점의 저장소를 checkout 해야 한다. 쉘에서 git checkout ID(여기서 ID는 살펴보려고 하는 커밋 식별자) 명령어를 실행하면 된다. checkout 하게 되면, 나중에 저장소를 올바른 상태로 되돌려 놓아야 된다는 것을 기억해야 됩니다.

GitHub 시간도장(Timestamp)

GitHub에 원격저장소를 생성하라. 로컬 저장소의 콘텐츠를 원격 저장소로 푸쉬하라. 로컬 저장소에 변경사항을 만들고, 변경사항을 푸쉬하라.

방금 생성한 GitHub 저장소로 가서 GitHub 변경사항에 대한 시간도장(timestamps)을 살펴본다. GitHub이 시간정보를 어떻게 기록하는가? 왜 그런가?

해답 GitHub은 시간도장을 사람이 읽기 쉬운 형태로 표시한다(즉, “22 hours ago” 혹은 “three weeks ago”). 하지만, 시간도장을 이리저리 살펴보면, 파일의 마지막 변경이 발생된 정확한 시간을 볼 수 있다.

푸쉬(Push) vs. 커밋(Commit)

이번 학습에서, “git push” 명령어를 소개했다. “git push” 명령어가 “git commit” 명령어와 어떻게 다른가?

해답

변경 사항을 푸쉬하면, 로컬에서 변경한 사항을 원격 저장소와 상호협의하여 최신 상태로 갱신한다. (흔히 다른 사람 변경시킨 것을 공유하는 것도 이에 해당된다.) 커밋은 로컬 저장소만 갱신한다는 점에서 차이가 난다.

원격 설정 고치기

원격 URL에 오탈자가 발생되는 일이 실무에서 흔히 발생된다. 이번 연습문제는 이런 유형의 이슈를 어떻게 고칠 수 있느냐에 대한 것이다. 먼저 잘못된 URL을 원격(remote)에 추가하면서 시작해 보자.

```
git remote add broken https://github.com/this/url/is/invalid
```

git remote로 추가할 때 오류를 받았나요? 원격 URL을 적법한지 확인해 주는 명령어를 생각해 낼 수 있나요? URL을 어떻게 수정할 수 있을까요? (팁: git remote -h를 사용한다.) 이번 연습문제를 수행한 다음에 원격(remote)을 지워버리는 것을 잊지마자.

해답 원격(remote)을 추가할 때 어떤 오류 메시지도 볼 수 없다. (원격 remote 를 추가하는 것은 Git에게 알려주기만 할 뿐 아직 사용하지는 않았기 때문이다.) git push 명령어를 사용하자마자, 오류 메시지를 보게 된다. git remote set-url 명령어를 통해서 잘못된 원격 URL을 바꿔 문제를 해결하게 된다.

GitHub 라이선스와 README 파일

이번 학습에서 GitHub에 원격 저장소를 생성하는 방법을 학습했다. 하지만, GitHub 저장소를 초기화할 때 README.md 혹은 라이선스 파일을 추가하지 않았다. 로컬 저장소와 원격 저장소를 연결시킬 때 두 파일을 갖게 되면 무슨 일이 발생될 것으로 생각하십니까?

해답 이런 경우, 관련없는 이력때문에 병합 충돌(merge conflict)이 발생된다. GitHub에서 README.md 파일을 생성시키고 원격 저장소에서 커밋작업을 수행한다. 로컬 저장소로 원격 저장소를 풀(pull)로 땡겨오면, Git이 origin과 공유되지 않는 이력을 탐지하고 병합(merge)를 거부해 버린다.

```
$ git pull origin master
```

```
From https://github.com/vlad/planets
 * branch           master      -> FETCH_HEAD
```

```
* [new branch]      master    -> origin/master
fatal: refusing to merge unrelated histories

--allow-unrelated-histories 옵션으로 두 저장소를 강제로
병합(merge)시킬 수 있다. 이런 옵션을 사용할 때 주의한다. 병합하기
전에 로컬저장소와 원격저장소의 콘텐츠를 면밀히 조사한다.
```

```
$ git pull --allow-unrelated-histories origin master
```

```
From https://github.com/vlad/planets
 * branch            master    -> FETCH_HEAD
Merge made by the 'recursive' strategy.
 README.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```


Chapter 15

협업 (Collaborating)

다음 단계로, 짹을 이룬다. 한 사람이 “소유자”(연습을 시작하는데 사용될 GitHub 저장소 주인)가 되고, 다른 사람이 “협력자”(소유자 저장소를 복제해서 변경을 하는 사람)가 된다. 목표는 협력자가 변경사항을 소유자 저장소에 추가하는 것이다. 말미에는 역할을 바꿔서 두 사람 모두 소유자와 협력자의 역할을 수행한다.

혼자 훈련하기

혼자 힘으로 이번 학습을 쭉 진행해 왔다면, 두번째 터미널을 열어서 계속 진행할 수 있다.> 두번째 윈도우가 여러분의 협력자를 나타내고, 다른 컴퓨터에서 작업하고 있는 것으로 볼 수 있다. GitHub 접근권한을 다른 사람에게 줄 필요가 없어졌다. 왜냐하면 두 ‘파트너’ 모두 여러분이기 때문이다.

소유자가 협력자에게 접근권한을 부여할 필요가 있다. GitHub에서 오른쪽에 ‘setting’ 버튼을 클릭해서 협력자(Collaborators)를 선택하고, 파트너 이름을 입력한다.

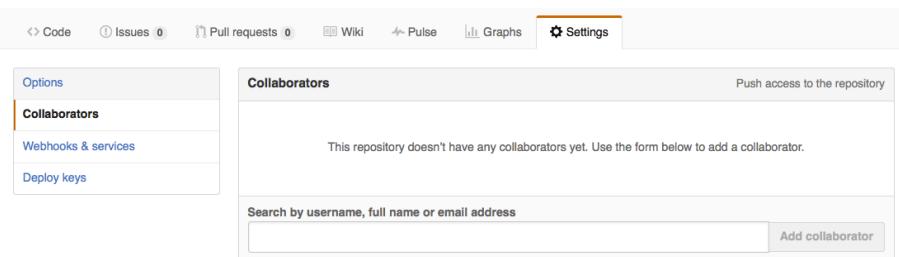


Figure 15.1: GitHub에 협업자(Collaborators) 추가

소유자 저장소에 접근 권한이 부여되면, 협력자(Collaborator)는 <https://github.com/notifications>으로 이동한다. 그곳에서 소유자 저장소의 접근을 받아들이면 된다.

다음으로 협력자(Collaborator)는 소유자 저장소 사본을 본인 컴퓨터로 내려받는다. 이런 작업을 “저장소 복제(clone a repo)”라고 부른다. 소유자의 저장소를 본인 바탕화면/Desktop 폴더에 클론하려면, 협력자는 다음 명령어를 입력한다:

```
$ git clone https://github.com/vlad/planets.git ~/Desktop/vlad-planets
'vlad'를 소유자 사용자이름(저장소를 소유하고 있는 사람)으로 바꾼다.
```

앞서 작업했던 것과 정확하게 동일한 방식으로, 협력자는 이제 소유자의 저장소 클론에서 변경을 마음대로 할 수 있다:

```
$ cd ~/Desktop/vlad-planets
$ nano pluto.txt
$ cat pluto.txt
```

It is so a planet!

```
$ git add pluto.txt
$ git commit -m "Add notes about Pluto"
```

```
1 file changed, 1 insertion(+)
create mode 100644 pluto.txt
```

그리고 나서, 변경사항을 GitHub의 소유자 저장소로 푸쉬한다:

```
$ git push origin master
```

```
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 306 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/vlad/planets.git
  9272da5..29aba7c  master -> master
```

주목할 점은 origin이라는 원격 저장소를 생성할 필요는 없다: 저장소를 복제(clone)할 때 Git이 자동으로 origin 이름을 붙여준다. (수작업으로 원격 설정을 할 때, 앞에서 왜 origin 이름을 사용한 것이 현명한 선택인 이유다.)

이제 GitHub 웹사이트에서 소유자 저장소를 살펴본다(아마도 웹브라우저 다시 고치기를 수행할 필요가 있을 수 있다.) 협력자가 신규 커밋을 한 것을 확인할 수 있다.

소유자 로컬 컴퓨터로 GitHub 원본 저장소의 변경사항을 다운로드하려면, 소유자는 다음과 같이 입력한다:

```
$ git pull origin master
```

```
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
```

```

Unpacking objects: 100% (3/3), done.
From https://github.com/vlad/planets
 * branch           master      -> FETCH_HEAD
Updating 9272da5..29aba7c
Fast-forward
  pluto.txt | 1 +
  1 file changed, 1 insertion(+)
  create mode 100644 pluto.txt

```

이제 저장소 3개 (소유자 로컬 저장소, 협력자 로컬 저장소, GitHub의 소유자 저장소) 모두 동기화 되었다.

기본적인 협업 작업흐름

실무에서 협업하는 저장소의 가장 최신 버전을 갖도록 확인하고 확인하는 것이 좋다. 어떤 변경을 하기 전에 `git pull` 명령어를 먼저 실행해야 한다. 기본적인 협업 작업흐름은 다음과 같다.

- `git pull origin master` 명령어로 본인 로컬 저장소를 최신상태로 갱신한다.
- 변경 작업을 수행하고 `git add` 명령어로 준비단계(staging area)로 보낸다.
- `git commit -m` 명령어로 변경사항을 커밋한다.
- GitHub에 `git push origin master` 명령어로 변경사항을 업로드한다.

상당한 변경사항을 포함한 단 한번의 커밋보다 작은 변화를 준 커밋을 많이 하는 것이 좋다: 작은 커밋이 가독성도 좋고 리뷰하기도 더 편하다.

역할을 바꾸고 반복한다.

역할을 바꿔서 전체 과정을 반복한다.

변경사항 리뷰

협력자에게 어떤 정보도 주지 않고 소유자가 저장소에 커밋을 푸쉬했다. 협력자는 명령라인으로 무엇이 변경되었는지 어떻게 알 수 있을까요?

해답

명령라인에서 협력자는 로컬 저장소에 원격 저장소 변경사항을 `git fetch origin master` 명령어를 사용해서 가져올 수 있다. 하지만, 그 자체로 병합(merge)되는 것은 아니다. `git diff master origin/master` 명령어를 실행해서, 협력자는 터미널에 변경사항을 확인할 수 있다.

GitHub에서도 협력자는 포크된 저장소로 가서 “This branch is 1 commit behind Our-Repository:master.” 메시지를 볼 수 있다. Compare 아이콘과 링크가 걸려있다. Compare 페이지에서 협력자는 base fork를 본인 저장소로 변경하고 나서, “compare across forks” 위에 링크를 클릭한다. 마지막으로 head fork를 주 저장소로 변경한다. 이 작업을 하게 되면 차이가 나는 모든 커밋을 볼 수 있게 된다.

GitHub에서 변경사항 주석(comment)달기

협력자는 소유자가 변경한 한 줄에 대해 질문을 가질 수 있고, 일부 제안사항도 있다.

GitHub으로 커밋 차이에 대해 주석을다는 것도 가능하다. 파란색 주석 아이콘(comment icon)을 클릭하면 주석 윈도우(comment window)을 열 수 있다.

협력자는 GitHub 인터페이스를 사용해서 코멘트와 제안을 남길 수 있다.

버전 이력, 백업, 그리고 버전 제어

일부 백업 소프트웨어는 파일 버전에 대한 이력을 기록하고 있다. 도한, 특정 버전을 복구하는 기능도 제공하고 있다. 이러한 기능이 버전 제어와 어떻게 다른가?

버전제어, Git, GitHub을 사용하는 좋은 점은 무엇인가?

Chapter 16

충돌 (Conflicts)

사람들이 병렬로 작업을 할 수 있게 됨에 따라, 누군가 다른 사람 작업영역에
발을 들여 넣을 가능성이 생겼다. 혼자서 작업할 경우에도 이런 현상이 발생한다:
소프트웨어 개발을 개인 노트북과 연구실 서버에서 작업한다면, 각 작업본에 다른
변경사항을 만들 수 있다. 버전 제어(version control)가 겹치는 변경사항을
해결(resolve)하는 툴을 제공함으로서, 이러한 충돌(conflicts)을 관리할 수 있게
돕는다.

충돌을 어떻게 해소할 수 있는지 확인하기 위해서, 먼저 파일을 하나 생성하자.
`mars.txt` 파일은 현재 두 협업하는 사람의 `planets` 저장소 사본에서는 다음과
같이 보인다:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

파트너 사본에만 한 줄을 추가하자:

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
This line added to Wolfman's copy
```

그리고 나서, 변경사항을 GitHub에 푸쉬하자:

```
$ git add mars.txt
$ git commit -m "Add a line in our home copy"
```

```
[master 5ae9631] Add a line in our home copy
 1 file changed, 1 insertion(+)

$ git push origin master

Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 352 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/vlad/planets
 29aba7c..dabb4c8  master -> master
```

이제 또다른 파트너가 GitHub에서 갱신(update)하지 않고, 본인 사본에 다른 변경사항을 작업한다:

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We added a different line in the other copy
```

로컬 저장소에 변경사항을 커밋할 수 있다:

```
$ git add mars.txt
$ git commit -m "Add a line in my copy"
```

```
[master 07ebc69] Add a line in my copy
 1 file changed, 1 insertion(+)
```

하지만, Git이 GitHub에는 푸쉬할 수 없게 한다:

```
$ git push origin master
```

```
To https://github.com/vlad/planets.git
! [rejected]          master -> master (non-fast-forward)
error: failed to push some refs to 'https://github.com/vlad/planets.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Git이 푸쉬를 거절한다. 이유는 로컬 브랜치로 반영되지 않는 신규 업데이트가 원격 저장소에 있음을 Git이 탐지했기 때문이다. 즉, 본인이 작업한 변경사항이 다른 사람이 작업한 변경사항과 충돌되는 것을 Git이 탐지해서, 앞에서 작업한 것을 뭉개지 않도록 정지시킨다. 이제 해야될 작업은 GitHub에서 변경사항을 풀(Pull)해서 가져오고, 현재 작업중인 작업본과 병합(merge)해서 푸쉬한다. 풀(Pull)부터 시작하자:

```
$ git pull origin master

remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From https://github.com/vlad/planets
 * branch           master      -> FETCH_HEAD
Auto-merging mars.txt
CONFLICT (content): Merge conflict in mars.txt
Automatic merge failed; fix conflicts and then commit the result.
```

`git pull` 명령어는 로컬 저장소를 갱신할 때 원격 저장소에 이미 반영된 변경사항을 포함시키도록 한다. 원격 저장소 브랜치에서 변경사항을 가져온(fetch) 후에, 로컬 저장소 사본의 변경사항이 원격 저장소 사본과 겹치는 것을 탐지해낸다. 따라서, 앞서 작업한 것이 뭉개지지 않도록 서로 다른 두 버전의 병합(merge)을 승인하지 않고 거절한 것이다. 해당 파일에 충돌나는 부분을 다음과 같이 표식해 놓는다:

```
$ cat mars.txt

Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
<<<<< HEAD
We added a different line in the other copy
=====
This line added to Wolfman's copy
>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d
```

<<<<< HEAD으로 시작되는 부분에 본인 변경사항이 나와있다. Git이 자동으로 =====을 넣어서 충돌나는 변경사항 사이에 구분자로 넣고, >>>>>기호는 GitHub에서 다운로드된 파일 내용의 마지막을 표시한다. (>>>>> 표시자 다음에 문자와 숫자로 구성된 문자열로 방금 다운로드한 커밋번호도 식별자로 제시한다.)

파일을 편집해서 표시자/구분자를 제거하고 변경사항을 일치하는 것은 전적으로 여러분에게 달려있다. 원하는 무엇이든지 할 수 있다: 예를 들어, 로컬 저장소의 변경사항을 반영하든, 원격 저장소의 변경사항을 반영하든, 로컬과 원격 저장소의 내용을 대체하는 새로운 것을 작성하든, 혹은 변경사항을 완전히 제거하는 것도 가능하다. 로컬과 원격 모두 교체해서 다음과 같이 파일이 보이도록 하자:

```
$ cat mars.txt

Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We removed the conflict on this line
```

병합을 마무리하기 위해서, 병합으로 생성된 변경사항을 `mars.txt` 파일에 추가하고 커밋한다:

```
$ git add mars.txt
$ git status

On branch master
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)
```

Changes to be committed:

```
modified:   mars.txt

$ git commit -m "Merge changes from GitHub"
```

[master 2abf2b1] Merge changes from GitHub

이제 변경사항을 GitHub에 푸쉬할 수 있다:

```
$ git push origin master
```

```
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 697 bytes, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/vlad/planets.git
  dabb4c8..2abf2b1  master -> master
```

Git이 병합하면서 수행한 것을 모두 추적하고 있어서, 수작업으로 다시 고칠 필요는 없다. 처음 변경사항을 만든 협력자 프로그래머가 다시 풀하게 되면:

```
$ git pull origin master
```

```
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 2), reused 6 (delta 2)
Unpacking objects: 100% (6/6), done.
From https://github.com/vlad/planets
 * branch      master      -> FETCH_HEAD
Updating dabb4c8..2abf2b1
Fast-forward
  mars.txt | 2 ++
  1 file changed, 1 insertion(+), 1 deletion(-)
```

병합된 파일을 얻게 된다:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
```

```
But the Mummy will appreciate the lack of humidity
We removed the conflict on this line
```

다시 병합할 필요는 없는데, 다른 누군가 작업을 했다는 것을 Git가 알기 때문이다.

충돌을 해소하는 Git 기능은 매우 유용하지만, 충돌해소에는 시간과 노력이 수반되고, 충돌이 올바르게 해소되지 않게 되면 오류가 스며들게 된다. 프로젝트 와중에 상당량의 충돌을 해소하는데 시간을 쓰고 있다고 생각되면, 충돌을 줄일 수 있는 기술적인 접근법도 고려해보는 것이 좋겠다.

- 좀 더 자주 upstream을 풀(Pull)하기, 특히 신규 작업을 시작하기 전이라면 더욱 그렇다.
- 작업을 구별하기 위해서 토픽 브랜치를 사용해서, 작업을 완료하면 마스터(master) 브랜치에 병합시킨다.
- 좀 더 작게 원자수준 커밋을 한다.
- 논리적으로 적절하다면, 큰 파일을 좀 더 작은 것으로 쪼갠다. 그렇게 함으로써 두 저작자가 동시에 동일한 파일을 변경하는 것을 줄일 수 있을 듯 싶다.

프로젝트 관리 전략으로 충돌(conflicts)을 최소화할 수도 있다:

- 동료 협력자와 누가 어떤 분야에 책임이 있는지 명확히 한다.
- 동료 협력자와 작업순서를 협의해서, 동일한 라인에 변경사항이 있을 수 있는 작업이 동시에 작업되지 않게 시간차를 둔다.
- 충돌이 문체변동(탭 vs 2 공백) 때문이라면, 프로젝트 관례를 수립하고, 코딩 스타일 도구(htmltidy, perltidy, rubocop 등)를 사용해서 필요한 경우 강제한다.

본인이 생성한 충돌 해소하기

강사가 생성한 저장소를 복제하세요. 저장소에 새 파일을 추가하고, 기존 파일을 변경하세요. (강사가 변경할 기준 파일이 어느 것인지 알려줄 것이다.) 강사의 말에 따라 충돌을 생성하는 연습을 위해서, 저장소에서 변경사항을 가져오도록 풀(Pull)하세요. 그리고 충돌을 해소하고 해결해 보세요.

텍스트 파일이 아닌 충돌

버전 제어 저장소의 이미지 파일이나 혹은 다른 텍스트가 아닌 파일에서 충돌이 발생할 때, Git는 무엇을 하나요?

해답

먼저 시도해 보자. 드라큘라가 화성 표면에서 사진을 찍어 mars.jpg로 저장했다고 가정한다.

화성 이미지 파일이 없다면 다음과 같이 더미 바이너리 파일을 생성할 수도 있다.

```
$ head --bytes 1024 /dev/urandom > mars.jpg
$ ls -lh mars.jpg
```

```
-rw-r--r-- 1 vlad 57095 1.0K Mar  8 20:24 mars.jpg
```

`ls` 명령어를 사용해서 파일 크기가 1 킬로바이트임이 확인된다.
`/dev/urandom` 특수 파일에서 불러온 임의 바이트로 꽉 차있다.

이제, 드라큘라가 `mars.jpg` 파일을 본인 저장소에 저장한다고 상정한다:

```
$ git add mars.jpg
$ git commit -m "Add picture of Martian surface"
```

```
[master 8e4115c] Add picture of Martian surface
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 mars.jpg
```

늑대인간도 비슷한 시점에 유사한 사진을 추가했다고 가정한다.
늑대인간의 사진은 화성하늘 사진인데, 이름도 `mars.jpg`로 동일하다.
드라큘라가 푸쉬하게 되면 유사한 메시지를 받게 된다:

```
$ git push origin master
```

```
To https://github.com/vlad/planets.git
! [rejected]          master -> master (fetch first)
error: failed to push some refs to 'https://github.com/vlad/planets.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

풀을 먼저한 뒤에 충돌나는 것을 해소한다는 것을 학습했다:

```
$ git pull origin master
```

이미지나 기타 바이너리 파일에 충돌이 생길 때, Git은 다음과 같은
메시지를 출력한다:

```
$ git pull origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/vlad/planets.git
 * branch            master      -> FETCH_HEAD
   6a67967..439dc8c  master      -> origin/master
warning: Cannot merge binary files: mars.jpg (HEAD vs. 439dc8c08869c342438f6dc4a2b
Auto-merging mars.jpg
CONFLICT (add/add): Merge conflict in mars.jpg
Automatic merge failed; fix conflicts and then commit the result.
```

이번에도 충돌 메시지가 `mars.txt`에 나온 것과 거의 동일하다. 하지만,
중요한 추가 라인 한줄이 있다:

```
warning: Cannot merge binary files: mars.jpg (HEAD vs. 439dc8c08869c342438f6dc4a2b615b05b93c
```

Git은 자동으로 텍스트 파일에 했던 것처럼 이미지 파일에 충돌지점 표식을 끼워넣을 수 없다. 그래서 이미지 파일을 편집하는 대신에, 간직하고자 하는 버전을 첫아웃(checkout)하고 나서 해당 버전을 추가(add)하고 커밋한다.

중요한 라인에, `mars.jpg` 두가지 버전에 대해서 커밋 식별자(commit identifier)를 Git이 제시하고 있다. 현재 작업 버전은 `HEAD`이고, 늑대인간 작업버전은 `439dc8c0...`이다. 본인 작업버전을 사용하고자 하면, `git checkout` 명령어를 사용한다:

```
$ git checkout HEAD mars.jpg
$ git add mars.jpg
$ git commit -m "Use image of surface instead of sky"
```

```
[master 21032c3] Use image of surface instead of sky
```

대신에 늑대인간 버전을 사용하려고 하면, `git checkout` 명령어를 늑대인간 `439dc8c0` 커밋 식별자와 함께 사용하면 된다:

```
$ git checkout 439dc8c0 mars.jpg
$ git add mars.jpg
$ git commit -m "Use image of sky instead of surface"
```

```
[master da21b34] Use image of sky instead of surface
```

이미지 모두 보관할 수도 있다. 동일한 이미지명으로 보관할 수는 없다는 것이 중요하다. 순차적으로 각 버전을 첫아웃(checkout)하고 나서 이미지명을 변경한다. 그리고 나서 이름을 변경한 버전을 추가한다. 먼저, 각 이미지를 첫아웃하고 이름을 변경하자:

```
$ git checkout HEAD mars.jpg
$ git mv mars.jpg mars-surface.jpg
$ git checkout 439dc8c0 mars.jpg
$ mv mars.jpg mars-sky.jpg
```

그리고 나서, `mars.jpg` 이전 파일을 삭제하고 신규 파일 두개를 추가한다:

```
$ git rm mars.jpg
$ git add mars-surface.jpg
$ git add mars-sky.jpg
$ git commit -m "Use two images: surface and sky"
```

```
[master 94ae08c] Use two images: surface and sky
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 mars-sky.jpg
rename mars.jpg => mars-surface.jpg (100%)
```

이제 화성 이미지 파일 두개가 저장소에서 확인되지만 `mars.jpg` 파일은 더이상 존재하지 않는다.

일반적인 작업 시간

원격 Git 저장소를 활용하여 공동 프로젝트로 작업하는 컴퓨터 앞에 앉아있다. 작업시간동안에 다음 동작을 취하지만, 작업순서는 다르다:

- 변경한다(make change): `numbers.txt` 텍스트 파일에 숫자 100을 추가.
- 원격 저장소 갱신시키기(Update remote): 로컬 저장소와 매칭되어 동기화시킴.
- 축하하기(Celebrate): 맥주로 성공을 자축함.
- 로컬 저장소 갱신시키기(Update local): 원격 저장소와 매칭되어 동기화시킴.
- 변경사항 준비영역으로 보내기(Stage change): 커밋대상으로 추가하기.
- 변경사항(Commit change): 로컬 저장소에 커밋하기

어떤 순서로 작업을 수행해야 충돌이 날 가능성을 최소화할 수 있을까? 아래표 action 칼럼에 순서대로 상기 명령어를 적어 본다. 일부 단계를 시작하는데 도움이 되도록 채워져 있다.

작업 순서를 정했으면, command 칼럼에 대응되는 명령어를 적어본다. 일부 단계를 시작하는데 도움이 되도록 채워져 있다.

order	action	command
1		
2		<code>echo 100 >> numbers.txt</code>
3		
4		
5		
6	Celebrate!	AFK

해답

order	action	command
1	Update local	<code>git pull origin master</code>
2	Make changes	<code>echo 100 >> numbers.txt</code>
3	Stage changes	<code>git add numbers.txt</code>
4	Commit changes	<code>git commit -m "Add 100 to numbers.txt"</code>
5	Update remote	<code>git push origin master</code>
6	Celebrate!	AFK

Chapter 17

공개 과학 (Open Science)

“공개(open)”의 반대는 “폐쇄(closed)”가 아니다. (The opposite of “open” isn’t “closed”.) “공개(open)”의 반대는 “망한(broken)” 것이다. (The opposite of “open” is “broken”.)

정보의 자유 공유는 과학에서 이상적일지 모르지만, 현실은 좀더 복잡하다. 현재, 보통 실무사례는 다음과 같다:

- 과학자가 데이터를 수집하고 학과에 가끔 백업되는 컴퓨터에 저장한다.
- 데이터를 분석하기 위해서 작은 프로그램을 작성하고 수정한다. (프로그램도 연구원의 로컬 노트북에 저장된다.)
- 적당한 분석 결과가 생성되자 마자, 작성해서 논문을 제출한다. 데이터를 논문에 포함할 수도 있다. (점점 많은 저널이 데이터를 요구한다.) 하지만, 아마도 프로그램 코드는 포함하지 않을 것이다.
- 시간이 흐른다.
- 저널에서는 연구원 분야의 익명으로된 소수의 사람들에게서 받아 검토(review) 결과를 보낸다. 검토 결과를 충족하도록 논문을 수정한다. 수정하는 동안에 앞서 작성한 프로그램, 스크립트를 변경해서 다시 제출한다.
- 좀더 많은 시간이 흐른다.
- 종국에 논문이 출판된다. 논문에 데이터 온라인 사본 링크를 포함할 수도 있다. 하지만, 논문은 유료로 돈을 내야만 접근 가능하다는 장벽(paywall)에 막혀있다: 개인 혹은 기관 접근 권한을 가진 사람만이 논문을 읽을 수 있다.

하지만, 점점 더 많은 과학자들에게, 프로세스는 다음과 같다:

- 과학자가 수집한 데이터가 수집되는 즉시, figshare 혹은 Zenodo 같은 공개 접근 저장소에 저장된다. 그리고 디지털 객체 식별자(Digital Object Identifier, DOI)가 부여된다. 혹은 데이터를 이미 게시하고 Dryad에 저장한다.
- 과학자가 작업물을 보관할 저장소를 GitHub에 생성한다.
- 분석작업을 수행하면서, 스크립트의 변경사항을 (아마도 몇몇 산출 결과도 포함해서) 저장소에 푸쉬한다. 논문을 위한 저장소를 다목적으로 사용한다; 이 저장소가 다른 동료 과학자와 협업하는 허브가 된다.

- 논문 상태에 만족할 정도로 진행되면, arXiv 혹은 다른 사전 출력 서비스에 게시하고, 다른 동료 과학자를 초대해서 피드백을 받는다.
- 피드백에 기초해서 저널에 논문을 마지막으로 제출하기 전 몇번의 수정사항을 게시할 수도 있다.
- 출판된 논문은 사전출판논문, 코드, 그리고 데이터 저장소의 링크를 포함한다. 그렇게 함으로써 다른 과학자가 본인 연구의 시작점으로 삼아서 연구를 쉽게 연결해서 수행할 수 있게 된다.

이러한 공개 연구 모형은 발견을 가속시킨다. 연구 작업이 더 많이 공개될수록, 더 많이 인용되고 재사용된다(the more widely it is cited and re-used). 하지만, 이런 방식으로 작업하고 연구하고자 하는 사람들은 실무에서 “공개(open)”가 정확하게 의미하는 바에 대해서 몇가지 결정을 내릴 필요가 있다. 공개 과학(Open Science)에 관한 다른 측면에 대해서 이 책을 참고한다.

이것이 버전 제어(version control)를 가르치는 (많은) 이유 중의 하나다. 버전제어가 꾸준히 사용될 때, 컴퓨터 작업에 대한 공유가능한 전자연구노트로 활동함으로써 “방법”에 대한 질문에 답을 한다:

- 누가 언제 무엇을 했는지를 포함해서, 작업에 대한 개념적 단계가 문서화된다. 모든 단계는 (커밋 ID)식별자로 도장이 찍힌다. 식별자는 의도와 목적을 갖는 중복되지 않고 유일하다.
- 정당성(reasonable), 아이디어, 다른 지적 작업에 대한 문서화를 이것에서 파생된 변경사항과 묶을 수 있다.
- 중복되지 않고 유일하며 복구가능한 방식으로 컴퓨터 작업 결과물을 얻어서 연구에 사용할 것을 조회할 수 있다.
- Git같은 분산된 버전제어 시스템으로, 버전제어 저장소는 영속성을 쉽게 얻을 수 있고, 전체 이력을 담아낼 수 있다.

코드를 인용가능하게 만들기

버전제어 저장소에 올라온 모든 것(데이터, 코드, 논문 등)은 인용가능한 객체로 변환시킬 수 있다. lesson 12: 인용(Citation)에서 인용하는 방법에 대해서 학습하게 된다.

내 작업을 어떻게 재현가능하게 만들 수 있을까?

연구실 동료중 한명에게 논문에 나온 내용과 웹으로만 최근에 본인이 성취한 결과를 재현할 수 있는지 물어본다. 동료 결과물 중 하나에 대해서도 같은 작업을 수행해 본다. 그리고 나서, 일하고 있는 연구실에 나온 결과물에 대해서도 시도를 해본다.

적절한 데이터 저장소를 찾는 방법?

2~3분정도 인터넷을 검색하고 앞에서 언급된 데이터 저장소를 조사해 본다: Figshare, Zenodo, Dryad. 전공분야에 따라, 본인 전공분야별로 잘 알려진 저장소가 도움이 될 수 있다. Nature에서 추천한 데이터 저장소도 유용할 수 있다.

주변 동료와 현재 작업에 사용하고 있는 데이터 저장소에 대해서 토론해 보고, 이유도 설명해 보자.

Chapter 18

라이선싱 (Licensing)

18.1 소프트웨어 라이선스

소스코드, 원고, 다른 창의적 저작물을 갖는 저장소가 공개될 때, 저장소 기반 디렉토리에 LICENSE 혹은 LICENSE.txt 파일을 포함해서 콘텐츠가 어떤 라이선스로 이용가능한지를 명확히 기술해야된다. 이유는 소스코드가 창의적 저작물로서, 자동적으로 지적재산(따라서 저작권)보호에 대상에 부합되기 때문이다. 자유로이 이용가능한 것으로 보여지거나, 명시적으로 광고되는 코드는 그런 보호를 유예하지 않는다. 따라서, 라이선스 문장이 없는 코드를 (재)사용하는 누구나 스스로 위험에 처하게 된다. 왜냐하면 소프트웨어 코드 저작자가 항상 일방향으로 재사용을 불법화할 수 있기 때문이다. 즉, 저작권 소유자가 당신을 저작권법 위반으로 고소할 수 있다.

라이선스는 그렇지 않다면 보유하지 못할 권리를 다른 사람(라이선스 허여자, licensee)에게 부여함으로써 이 문제를 해결한다. 어떤 조건아래서 무슨 권리가 부여될지는 라이선스마다 다소 차이가 난다. 독점적 라이선스와 대조하여, Open Source Initiative에서 공인된 오픈 라이선스(open licences)는 최소한 다음에 나온 권리를 모두 부여한다. 이런 권리를 오픈 소스 정의(Open Source Definition)로 부른다.:

1. 소스코드는 제약없이 이용가능하고, 사용되고, 재배포될 수 있다. 종합 배포의 일부로서도 포함된다.
2. 변형 혹은 다른 파생 저작물도 허락되고, 또한 재배포될 수 있다.
3. 이런 권리를 누가 받느냐의 질문이 차별의 조건이 되지 않는다. 예를 들어 상업적 혹은 학술적처럼 노력 분야에 의해서가 아님도 포함된다.

특히, 지금까지 라이선스 몇개가 인기를 얻고 있는데, choosealicense.com 웹사이트에서 본인 상황에 적합한 일반적인 라이선스를 선택하는데 도움이 된다. 주요한 고려사항에는 다음이 포함된다:

- 특허권을 주장하고자 하는가?

- 파생 저작물을 배포하는데 다른 사람도 소스코드를 배포하도록 강제할 것인가?
- 라이선싱하는 콘텐트가 소스코드인가?
- 이왕이면 소스코드도 라이선스할 것인가?

적절한 라이선스를 가장 잘 선택하는 것이 상당히 많은 가능한 조합이 있어 주눅이 들 수도 있다. 실무에서, 일부 라이선스만 지금까지 가장 인기가 있고, 다음이 그 범주에 포함된다:

- GNU 일반공중 라이선스 (GPL),
- MIT 라이선스,
- BSD 라이선스,
- 아파치 라이선스, 버전 2.0.

GPL은 다른 대부분의 공개소스 라이선스와 다른데, 전염성이 있는(*infective*) 특징이 있다: 코드의 수정된 버전을 배포하는 누구나 혹은 GPL 코드를 포함한 어느 것이든지, 자신의 코드도 동일하게 자유로이 공개가능하게 만들어야 한다.

흔히 사용되는 라이선서를 선택하는 것이 기여자나 사용자의 삶을 편하게 한다. 왜냐하면, 기여자나 사용자 모두 해당 라이선스에 친숙해서 사용할 때 상당한 양의 전문용어를 꼼꼼히 살펴볼 필요가 없기 때문이다.

Open Source Initiative와 Free Software Foundation 모두 좋은 선택이 될 수 있는 라이선스 목록을 유지관리하고 있다.

코드를 작성하는 과학자 관점에서 라이선싱과 라이선싱 선택지에 대한 전반적인 정보를 이 기사를 통해서 살펴볼 수 있다.

결국 가장 중요한 것은 라이선스가 무엇인지에 대해 분명한 문장이 있는지와 라이선스가 OSI와 FSF에서 승인되고 이미 검증된 것인지 여부다. 또한, 저장소에 공개된 것이 아닐지라도, 처음부터 최선으로 라이선스를 선택해야 된다. 결정을 미루는 것은 나중에 더 복잡하게 된다. 왜냐하면, 매번 새로운 협력자가 기여하기 시작하면, 협력자도 저작권을 갖게된다. 따라서, 라이선스를 고르자 마자, 승인을 득해야 할 필요가 있기 때문이다.

본인이 오픈 라이선스를 사용할 수 있나요?

여러분이 작성하고 있는 소프트웨어에 오픈소스 소프트웨어 라이선스를 적용할 수 있는지 알아본다. 여러분이 라이선스 적용을 일방적으로 할 수 있는가? 혹은 여러분의 기관이나 조직의 다른 사람에게서 허락이 필요한가? 만약 그렇다면 누굴까?

본인은 어떤 라이선스를 이미 승인했나요?

(금번 워크샵을 포함해서) 매일 사용하는 대다수 소프트웨어는 오픈-소스 소프트웨어로 출시되었다. 아래 목록 혹은 본인이 직접 고른 GitHub 사이트에서 프로젝트를 하나 고른다. 라이선스를 찾아(보통 LICENSE 혹은 COPYING 이름이 붙은 파일)보고, 소프트웨어 사용을 어떻게 제약하는지 살펴보자. 이번 세션에서 논의된 라이선스 중 하나인가? 차이점은 어떻게 나는가?

- Git, 소스코드 관리 도구
- CPython, 파이썬 언어 구현

- Jupyter, 웹기반 파이썬 노트북 프로젝트
- EtherPad, 실시간 협업 편집기

18.2 콘텐츠 라이선스

만약 저장소 콘텐츠가 소프트웨어가 아닌 데이터, 창의적 저작물(매뉴얼, 기술 보고서, 원고) 같은 연구제품이 포함되면, 소프트웨어를 위해 설계된 라이선스 대부분은 적합하지는 못하다.

- **데이터:** 대부분 국가사법권에서 데이터 유형 대부분은 자연에 대한 사실로 간주된다. 그럼으로, 저작권 보호를 받을 자격이 없다.(단, 아마도 사진과 의료영상정보 등은 예외) 따라서, 저작자 표시로 사회적 혹은 학자적 기대치를 알리려고, 저작권을 정의로 주장하는 방식으로 라이선스를 사용하는 것은 단지 법적으로 혼탁한 상황만 조장할 뿐이다. 크리에이티브 커먼즈 제로(Creative Commons Zero, CC0)처럼 공중도메인 권리포기를 지지하는 법적 표시를 분명히 하는 것이 더 낫다. Dryad 데이터 저장소는 사실 이를 요구하고 있다.
- **창의적 저작물(Creative works):** 매뉴얼, 보고서, 원고, 기타 창의적 저작물은 지적재산 보호 대상이 된다. 따라서 소프트웨어와 마찬가지로 자동으로 저작권으로 보호된다. 크리에이티브 커먼즈(Creative Commons) 조직이 기본 제약사항 4개를 조합해서 라이선스 집합을 마련했다:
 - 저작자 표시(Attribution): 파생 저작물에 대해서 최초 저작자의 이름, 출처 등의 정보를 반드시 표시해야 한다.
 - 변경 금지(No Derivative): 저작물을 복사할 수도 있으나 저작물을 변경 혹은 저작물을 이용하여 2차적 저작물로 제작을 금한다.
 - 동일조건변경허락(Share Alike): 2차적 저작물을 제작할 수 있으나, 2차적 저작물은 원래 저작물과 동일한 라이선스를 적용한다.
 - 비영리(Noncommercial): 저작물을 영리 목적으로 사용할 수 없음. 영리 목적을 위해서는 별도의 계약이 필요하다.

출처표시 (CC-BY)와 동일조건변경허락(CC-BY-SA) 라이선스만이 “오픈 라이선스”로 간주된다.

소프트웨어 카펜트리는 가능하면 폭넓게 재사용될 수 있도록 수업자료에 대해서는 CC-BY, 코드에는 MIT 라이선스를 사용한다. 다시 한번, 가장 중요한 것은 프로젝트 루트 디렉토리에 있는 LICENSE 파일에 라이선스가 무엇인지 분명하게 언급한다. 본인 프로젝트를 참조하는 방법을 기술하는데 CITATION 혹은 CITATION.txt 파일을 포함할 수도 있다. 소프트웨어 카펜트리 사례는 다음과 같다:

To reference Software Carpentry in publications, please cite both of the following:

Greg Wilson: "Software Carpentry: Lessons Learned". arXiv:1307.5448, July 2013.

```
@online{wilson-software-carpentry-2013,
  author      = {Greg Wilson},
  title       = {Software Carpentry: Lessons Learned},
  version     = {1},
```

```
date      = {2013-07-20},  
eprinttype = {arxiv},  
eprint     = {1307.5448}  
}
```

Chapter 19

호스팅 (Hosting)

저작물이나 작업을 공개하고자 하는 그룹에서 가지는 두번째 큰 질문은 코드와 데이터를 어디에 호스팅할지 정하는 것이다. 방법중 하나는 연구실, 학과, 혹은 대학이 서버를 제공하여 계정관리와 백업 등을 관리하는 것이다. 주된 장점은 누가 무엇을 소유하는지 명확하다. 특히 민감한 정보(예를 들어, 사람에 대한 실험정보 혹은 특히 출원에 사용될 수도 있는 정보)가 있다면 중요하다. 큰 단점은 서비스 제공 비용과 수명이다: 데이터를 수집하는데 10년을 보낸 과학자가 지금부터 10년 후에도 여전히 이용 가능하기를 원하지만, 학교 인프라를 지원하는 대부분의 연구기금의 수명이 턱없이 짧다.

또다른 선택지는 도메인을 구입하고 호스팅하는데 ISP(인터넷 서비스 제공자, Internet service provider)에 비용을 지불한다. 이 접근법은 개인이나 그룹에게 좀더 많은 제어권을 주고 학교나 기관을 바꿀 때 생기는 문제도 비켜갈 수 있다. 하지만, 위나 아래 선택지보다 초기 설정하는데 더 많은 시간과 노력이 요구된다.

세번째 선택지는 GitHub, BitBucket, 혹은 SourceForge 같은 공개 호스팅 서비스를 채용하는 것이다. 웹 인터페이스를 통해서 저장소 코드를 생성하고, 보고, 편집할 수 있게 한다. 이러한 서비스는 이슈추적, 위키 페이지, 이메일 통보, 코드 리뷰를 포함한 커뮤니케이션과 프로젝트 관리 도구도 제공한다. 이러한 서비스는 규모의 경제와 네트워크 효과로 모두 이익을 볼 수 있다: 즉, 동일한 표준을 갖는 작은 많은 서비스를 실행하는 것보다 큰 서비스 하나를 실행하는 것이 더 쉽다. 또한, 사람들이 협업하기도 더 쉽다. 대중적인 서비스를 사용하면 이미 동일한 서비스를 사용하는 커뮤니티와 본인 프로젝트를 연결하는데 도움이 된다.

예로서, 소프트웨어 카펜트리는 GitHub에 있어서, 해당 페이지에 대한 소스코드를 찾아볼 수 있다. GitHub 계정을 갖는 누구나 해당 페이지에 변경사항을 제안할 수 있다.

GitHub 저장소에서 Zenodo에 릴리스(release)를 연결하면 DOI를 부여할 수도 있다. 예를 들어, 10.5281/zenodo.57467이 “Git 소개”에 대해 주조된 DOI다.

규모가 크고 잘 정립된 서비스를 사용하는 것이 빠르게 강력한 도구의 장점을 흡수하는데 도움을 줄 수도 있다. 지속적 통합(Continuous integration, CI)이

그런 도구 중 하나로 자동으로 소프트웨어 빌드를 돌리고 코드가 커밋되거나 풀요청이 제출될 때마다 실행된다. 온라인 호스팅 서비스와 CI를 직접 통합이 의미하는 바는, 어떤 풀요청에도 해당 정보가 존재해서 코드 완결성과 품질 표준을 유지하는데 도움을 준다. 여전히 CI가 자가 구축한 호스팅 상황에도 이용가능하지만, 온라인 서비스 사용과 연계되면 초기설정과 유지보수 업무를 줄일 수 있다. 더욱이, 이러한 도구가 오픈소스 프로젝트에 무료로 제공되기도 한다. 사설 저장소에 대해서만 비용 일부를 지불하고 이용가능하다.

제도적 장벽 (Institutional Barriers)

공유가 과학에는 이상적이지만, 많은 기관에서 공유에 제약을 가한다. 예를 들어 잠재적으로 특허가능한 지적재산을 보호하는데 말이다. 만약 여러분이 그런 제약과 마주한다면, 특정 프로젝트 혹은 도메인에 예외를 요청하거나, 제도 협파를 통해서 더 공개된 과학을 지지하도록 좀더 앞서 나가는데 근본적인 동기에 관해 질의하는 것이 더 생산적일 수 있다.

본인 작업을 공개할 수 있을까?

본인 작업을 공개 저장소에 공개할 수 있는지 알아보자. 공개 작업을 일방적으로 할 수 있을까? 혹은 속한 조직의 누군가로부터 허락이 필요한가? 만약 그렇다면 조직의 누굴까?

본인 작업을 어디에 공개할 수 있을까?

본인 논문, 데이터, 소프트웨어를 공유하려면 이용가능한 저장소가 소속기관에 갖추어져 있는가? 소속기관 저장소는 arXiv, figshare, GitHub or GitLab와 같은 데이터 저장소 서비스와 비교하여 어떤 차이점이 있는가?

Chapter 20

Git 추가설정

20.1 로컬 PC와 SSH 키(Key) 연결

GitHub에 저장소(repository)를 만들고 여러 PC에서 작업을 진행할 경우 GitHub에 인증작업을 거쳐 진행하는 것이 여러모로 편리하다. 그중 하나의 방식이 공개키(public key)를 GitHub에 등록시켜 작업을 하는 것이 여기에 포함된다.

1. 먼저 윈도우를 사용할 경우 Git for windows를 다운로드 받아 설치한다.
2. ssh-keygen 명령어로 공개키/비밀키를 생성한다.
3. 생성된 공개키를 GitHub 계정에 등록시킨다.

20.2 SSH 공개키/비밀키 생성¹

SSH 공개키/비밀키를 생성시키고 이를 GitHub 홈페이지에 등록한다. 먼저 ssh-keygen 명령어에 매개변수 인자를 넣고 GitHub 전자우편주소도 함께 지정한다.

```
$ ssh-keygen -t rsa -C "your_email@example.com"
```

ssh-keygen 명령어로 생성된 키를 GitHub에 등록한다.

- 우측상단 [Settings] → [SSH and GPG keys] → [New SSH key]

[New SSH key]를 클릭하게 되면 Title, Key를 넣는 입력부분이 보인다. Title에 식별가능한 이름을 지정하고 앞서 생성한 id_rsa.pub 내용을 Key에 복사해서 붙여넣는다.

```
$ cat ~/.ssh/id_rsa.pub
```

```
ssh-rsa AAAxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

¹nickjolT (2017), “GitHub SSH 키 생성 및 등록하여 사용하기”

20.3 첫 커밋(commit)

인증작업을 완료한 후에 저장소에서 작업할 파일을 처음 커밋(commit)하는 경우 git add, git commit -m명령어를 이어서 전달시키게 되면 커밋을 때리는 사람이 누구인지를 등록하는 절차가 발생된다. git config를 통해 전자우편과 사용자명을 등록하게 되면 커밋을 정상적으로 진행시킬 수 있게 된다.

```
$ git config --global user.email "you@example.com"
$ git config --global user.name "Your Name"
```

20.4 비밀번호 입력없이 푸쉬(push)²

다음 단계로 비밀번호 없이 커밋된 내용을 GitHub에 전달하는 방법은 자격인증(credential) 캐싱을 통한 간단한 방법이 있다. 물론 처음에는 사용자명과 비번을 입력하는 과정을 필수적으로 거치게 된다.

```
$ git config credential.helper store
$ git push https://github.com/repo.git

Username for 'https://github.com': <USERNAME>
Password for 'https://<USERNAME>@github.com': <PASSWORD>

캐쉬에 시간제한을 두어서 7200초 즉 2시간 보안을 강화시킨다.

$ git config --global credential.helper 'cache --timeout 7200'
```

²Stackoverflow, “How do I avoid the specification of the username and password at every git push?”

프로그래밍

Chapter 21

R과 RStudio 소개

21.1 동기 모티브

과학은 다단계 과정이다: 실험을 설계하고 데이터를 수집하게 되면, 실제로 재미난 일이 시작된다. 이번 학습을 통해서 R과 RStudio를 사용해서 이런 과정을 시작하는 방법을 학습할 것이다. 원데이터(raw data)로 시작해서 탐색적 데이터 분석을 수행하고 분석결과를 시각화하는 방법을 학습할 것이다. 세월에 따른 국가별 인구 정보를 담고 있는 gapminder.org로부터 나온 데이터셋을 가지고 학습을 진행한다. 원데이터를 R로 불러올 수 있나요? 세네갈에 대한 인구를 시각화할 수 있나요? 아시아 대륙 국가에 대해 평균 소득을 계산할 수 있나요? 학습 말미에는 1분도 되지 않는 시간내에 모든 국가에 대해 인구수를 시각화할 수 있게 된다.

21.2 워크샵 시작 전에

노트북 컴퓨터에 최신 R과 RStudio가 설치되었는지 확인한다. 이것이 중요한데 이유는 설치된 R 버전이 최신이 아닌 경우 워크샵에 사용되는 팩키지가 제대로 설치되지 않거나 전혀 설치되지 않기 때문이다.

최신 R 버전 다운로드 RStudio 다운로드 및 설치

21.3 RStudio 소개

소프트웨어 카펜트리 R교육에 오신 것을 환영합니다.

이번 R 수업시간을 통해서, R 언어 기본기 뿐만 아니라 저녁이 있는 삶(make your life easier)을 가능토록 과학 프로젝트에 코드를 구조화하는 모범 활용사례도 교육한다.

도구로 RStudio 를 사용한다: 자유로이 사용가능하고, 무료이며, 오픈 소스 R과 통합된 개발 환경을 제공한다. RStudio는 편집기가 내장되어 있고, (서버 포함) 모든

플랫폼에서 동작하고, 버전 제어 및 프로젝트 관리 같은 많은 앞선 기능을 제공한다.

기본 배치

RStudio를 처음 열게 되면, 창 3개가 반갑게 여러분을 맞이한다:

- 인터랙티브 R 콘솔 : 좌측 전체
- 작업공간(Workspace)/이력 (History) : 우측상단 탭
- 파일(File)/그래프(Plot)/패키지(Package)/도움말(Help): 우측하단 탭

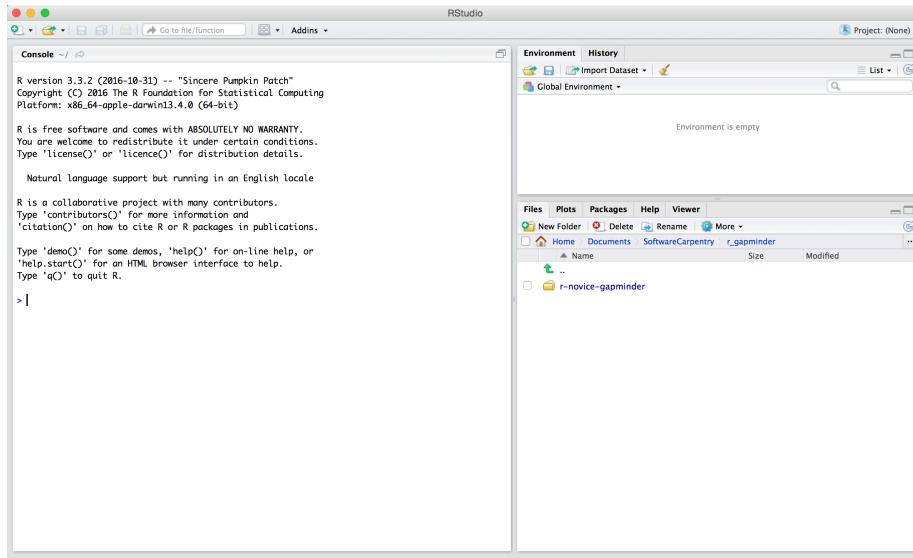


Figure 21.1: RStudio 배치화면

R 스크립트 같은 파일을 열게 되면, 편집창이 좌측 상단에 열린다.

21.4 RStudio 내부 작업흐름

RStudio 내부에서 작업하는 방식이 크게 두가지 있다.

1. 인터랙티브 R 콘솔에서 테스트하고 가지고 놀다가, 나중에 실행할 .R 파일에 복사해서 붙여넣는다.
 - 초기 시작하고 작은 테스트를 할 때 잘 작동한다.
 - 신속하게 노동 집약적인 개발이 된다.
2. .R 파일에서 작업을 시작하고, RStudio 명령어/단축키를 사용해서 현재 라인, 선택된 라인 혹은 변경된 라인을 인터랙티브 R 콘솔에 밀어넣어 실행한다.
 - 시작하는 매우 훌륭한 방식이다; 작성된 모든 코드가 나중에 저장된다.
 - RStudio 내부 혹은 R source() 함수를 사용해서 생성한 파일을 실행할 수 있다.

꿀팁: 코드 일부 실행하기

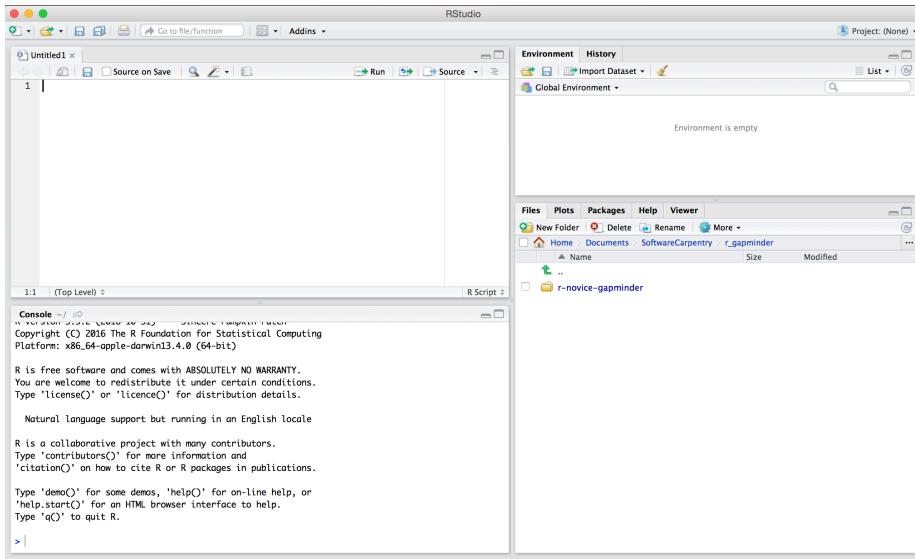


Figure 21.2: .R 파일을 연 RStudio 배치화면

편집기 창에서 코드를 실행하는데 있어 상당한 유연성을 RStudio가 제공한다. 버튼, 메뉴, 키보드 단축키, 세가지 방식이 있다. 현재 라인을 실행하려면,

1. 편집기창 상단 Run 버튼을 클릭한다.
2. “Code” 메뉴에서 “Run Lines”를 선택한다.
3. 리눅스 혹은 윈도우에서 Ctrl+ (Ctrl+Return), 맥 OS X에서 Command+ (⌘ +Return) 단축키를 누른다. (단축키는 버튼 위에 마우스를 올리면 볼 수 있다).

코드 블록을 실행하려면, 코드 블록을 선택하고 나서, Run 버튼을 누른다. 방금전에 실행한 코드 블록 내부 일부 코드를 변경했다면, 다시 코드 블록을 선택하고, Run 버튼을 누를 필요가 없다. 바로 옆에 있는 버튼(Re-run the previous region)을 누르면 된다. 이 버튼은 방금전에 변경한 내용을 포함해서 이전 코드 블록을 실행시킨다.

21.5 R 소개

R에서 상당한 시간을 R 인터랙티브 콘솔에서 사용한다. 이곳이 여러분이 작성한 모든 코드를 실행하는 곳으로 R 스크립트 파일에 추가하기 전에, 아이디어를 시험하는 유용한 환경이다. RStudio 콘솔은 R 명령-라인 환경에서 입력하는 곳과 동일하다.

R 인터랙티브 세션에서 보게 되는 첫번째 사항은 정보가 쭉 나오고 나서, “>” 가 나타나고 커서가 깜빡인다. 여러가지 면에서, 쉘 수업시간에 학습한 쉘 환경과 유사하다: **REPL** 루프(읽고, 평가하고, 출력하는 루프) 기본 아이디어 위에서 동작한다. 명령어를 입력하면, R이 명령어를 실행하고 나서, 결과를 반환한다.

21.6 계산기로 R 사용하기

R로 할 수 있는 가장 간단한 것이 산수다:

```
1 + 100
```

```
## [1] 101
```

R이 “[1]” 다음에 정답을 출력한다. 지금 당장 “[1]”에 대해 걱정하지 말자. 나중에 설명이 나와 있다. 지금은 일단 출력결과를 지칭한다고 생각한다.

배쉬(bash)처럼, 불완전한 명령어를 타이핑하면, R은 사용자가 명령어를 완성할 때까지 대기한다:

```
> 1 +
```

```
+
```

1+ 다음에 를 치게되면, R 세션이 “>” 대신에 “+”을 보여준다. 이것이 의미하는 바는 명령어가 완성될 때까지 대기한다는 것이다. 명령어를 취소하고자 한다면, 단순히 “Esc” 키를 치게 되면, 다시 “>” 프롬프트로 되돌아간다.

꿀팁: 명령문 취소

RStudio가 아니고 R 명령-라인을 사용하고 있다면, 명령문을 취소하는데 ESC 대신에 Ctrl+C(Ctrl+C) 사용이 필요하다. 또한, 이런 사실은 맥 사용자에도 동일하게 적용된다!

명령어 취소는 불완성된 명령어를 종료시키는데 그다지 유용하지는 않다: 명령어 취소 기능을 실행되고 있는 코드를 멈추는데도 사용할 수 있다. (예를 들어, 예상한 것보다 훨씬 더 오래 시간이 소요되는 경우) 혹은 현재 작성하고 있는 코드를 제거할 때도 사용한다.

계산기로 R을 사용할 때, 연산 순위는 초중등학교에서 배운 것과 동일하다.

가장 우선순위가 높은 것부터 낮은 순서는 다음과 같다:

- 괄호: (,)
- 멱승: ^ or **
- 나눗셈: /
- 곱셈: *
- 덧셈: +
- 뺏셈: -

```
3 + 5 * 2
```

```
## [1] 13
```

괄호를 사용해서 연산작업을 한데 묶는데, 이유는 의도한 바를 명확히 하거나, 기본설정과 차이가 날 때 평가순서를 강제하기 위해서다.

```
(3 + 5) * 2
```

```
## [1] 16
```

꼭 필요하지 않을 때 괄호가 사용되면 읽기 힘들어 지기도 하지만, 의도를 명확히 한다. 나중에 여러분이 작성한 코드를 다른 사람이 읽게 될을 기억하라.

```
(3 + (5 * (2 ^ 2))) # hard to read
3 + 5 * 2 ^ 2      # clear, if you remember the rules
3 + 5 * (2 ^ 2)    # if you forget some rules, this might help
```

상기 코드 각 라인 뒤에 텍스트를 “주석”이라고 부른다. 해쉬(혹은 번호기호)기호 # 다음에 오는 모든 것은 코드가 실행될 때 R에서 무시된다.

매우 작거나 큰 숫자는 과학 표기법을 따른다:

```
2/10000
```

```
## [1] 2e-04
```

상기 표기법은 10^{xx} 곱한 것을 축약한 것이다. 따라서, $2e-4$ 은 $2 * 10^{-4}$ 을 축약한 것이다.

숫자를 과학 표기법으로 작성할 수도 있다:

```
5e3 # Note the lack of minus here
```

```
## [1] 5000
```

21.7 수학 함수

R에는 수많은 내장 수학함수가 존재한다. 함수를 호출하려면, 함수명을 단순히 타이핑하고, 괄호를 열고 닫으면 된다. 괄호안에 타이핑하는 것을 함수 인자라고 부른다:

```
sin(1) #
## [1] 0.841
log(1) #
## [1] 0
log10(10) # 10
## [1] 1
exp(0.5) # e^(1/2)
## [1] 1.65
```

R에 나온 함수 모두를 기억해야 된다는 걱정은 하지마라. 구글이나 네이버, 다음에서 검색하면 된다. 함수명 앞 글자를 기억하고 있다면 RStudio에서 템 자동완성 기능을 사용한다.

템 자동완성이 R 자체 엔진보다 RStudio가 우월성을 갖는 분야로, 자동완성 기능이 함수, 함수 인자, 함수가 전달받을 수 있는 값을 더 쉽게 찾을 수 있게 한다.

명령문 앞에 ?을 타이핑하고 엔터를 치면 해당 명령어에 대한 도움말 페이지가 열린다. 명령어에 대한 기술과 동작방식에 대한 정보를 제공할 뿐만 아니라, 도움말 페이지 하단으로 스크롤해서 쭉 내리면 해당 명령어 사용법을 시연하는 코드 예제가 나와 있다. 나중에 예제를 통해서 살펴볼 것이다.

21.8 다양한 객체 비교하기

R에서 비교 작업도 수행할 수 있다:

```
1 == 1 # ("is equal to" )
## [1] TRUE

1 != 2 # ("is not equal to" )
## [1] TRUE

1 < 2 #
## [1] TRUE

1 <= 1 #
## [1] TRUE

1 > 0 #
## [1] TRUE

1 >= -9 #
## [1] TRUE
```

꿀팁: 숫자 비교

숫자 비교할 때 주의 사항: 정수가 아닌 경우, 절대로 == 비교를 사용하지 않는다. 정수는 구체적으로 자연수만 표현할 수 있는 자료형이다.

컴퓨터는 일정 정확도를 갖는 소수만 표현할 수 있다. 그래서, R로 출력할 때 같아 보이는 두 숫자는 사실 겉으로 드러나지 않는 표현이 다를 수 있다.

따라서, 작은 오차범위(컴퓨터 허용오차, Machine Numeric Tolerance)만큼 차이가 난다 대신에, all.equal 함수를 사용한다.

자세한 사항은 <http://floating-point-gui.de/> 웹사이트 참조.

21.9 변수와 대입

할당 연산자(Assignment Operator), <-를 사용해서 변수에 값을 저장할 수 있다. 예를 들어:

```
x <- 1/40
```

변수에 할당하면, 값을 출력하지 않음에 유의한다. 대신에, 나중에 사용하려고 변수라는 곳에 저장한다. x 변수에는 값 0.025가 담겨있다:

```
x
```

```
## [1] 0.025
```

좀더 구체적으로, 저장된 값은 부동소수점 수라고 불리는 해당 분수를 소수 근사한 것이다.

RStudio 우측 상단 창에서 Environment 탭을 클릭하면, 변수x와 값이 저장된 것을 확인할 수 있다. 변수 x는 숫자가 예상되는 어떤 연산작업에도 숫자자리에 사용될 수 있다:

```
log(x)
```

```
## [1] -3.69
```

변수가 다시 할당될 수도 있음에 주목한다:

```
x <- 100
```

x 변수에 0.025 값이 담겼지만, 현재는 담긴 값이 100 이 된다.

변수에 대입되는 값에는 대입되는 변수도 포함될 수 있다:

```
x <- x + 1 # RStudio x .
```

할당하는 우측편은 어떤 적법한 R 표현식도 될 수 있다. 우측편은 대입이 일어나기 전에 완전한 평가가 이루어 진다.

변수명에는 문자, 숫자, 밑줄, 구두점이 포함될 수 있다. 변수명이 숫자로 시작되거나, 공백이 있으면 안된다. 사람마다 긴 변수명에 대해 다른 관례를 사용한다. 다음에 관례가 나와있다.

- 단어.사이.구두점
- 단어_사이_밑줄
- 낙타대문자 : camelCaseToSeparateWords

어떤 관례를 사용하든 여러분 취향이지만, 일관성을 유지하라.

변수에 대입할 때, 할당 = 연산자 사용도 가능하다:

```
x = 1/40
```

하지만, R 사용자 사이 그다지 많이 활용되지 않는다. 가장 중요한 것은 사용하는 연산자에 일관성 유지하라. = 보다 <-를 사용하는 것이 덜 혼동스러운 경우가 있고, 커뮤니티에서 가장 흔히 사용되는 기호다. 그래서 추천하는 것은 <- 이다.

21.10 도전과제 1

다음 중 적법한 R 변수명은 어느것인가?

```
min_height
max.height
_age
.mass
MaxLength
min-length
2widths
celsius2kelvin
```

도전과제 1 해답

다음 사례는 R 변수로 사용될 수 있다:

```
min_height
max.height
MaxLength
celsius2kelvin
```

다음은 숨은 변수를 생성한다:

```
.mass
```

다음은 변수를 생성하는데 사용할 수 없다:

```
_age
min-length
2widths
```

21.11 벡터화(Vectorization)

인지해야 되는 마지막 사항은 R은 되었다는 사실이다. 변수와 함수는 값으로 벡터를 갖을 수 있다는 의미다. 예를 들어,

```
1:5
```

```
## [1] 1 2 3 4 5
2^(1:5)

## [1] 2 4 8 16 32
x <- 1:5
2^x

## [1] 2 4 8 16 32
```

놀랍도록 강력한 기능이다; 이점에 대해, 다음 수업에서 좀더 깊이 다룰 예정이다.

21.12 환경 설정

R 세션과 상호작용할 때 사용되는 몇 가지 유용한 명령어가 있다.

`ls` 명령어는 전역환경(작업하고 있는 R 세션)에 저장된 모든 변수와 함수 목록을 출력한다:

```
ls()
```

```
## [1] "con" "x"
```

꿀팁: 숨긴 객체

유닉스 쉘처럼, `ls`는 기본디폴트 설정으로 “.”으로 시작되는 함수와 변수를 숨긴다. 모든 객체 목록을 보려면, `ls(all.names=TRUE)` 타이핑하면 된다.

여기서 `ls` 명령어에 어떤 인자도 전달하지 않았음에 주목한다. 하지만, R에 함수를 호출하려면 괄호는 여전히 전달해야 된다.

`ls`만 그 자체로 타이핑하면, R은 해당 함수에 대한 소스코드만 출력한다!

```
ls
```

```
## function (name, pos = -1L, envir = as.environment(pos), all.names = FALSE,
##          pattern, sorted = TRUE)
## {
##   if (!missing(name)) {
##     pos <- tryCatch(name, error = function(e) e)
##     if (inherits(pos, "error")) {
##       name <- substitute(name)
##       if (!is.character(name))
##         name <- deparse(name)
##       warning(gettextf("%s converted to character string",
##                      sQuote(name)), domain = NA)
##     }
##   }
##   all.names <- .Internal(ls(envir, all.names, sorted))
##   if (!missing(pattern)) {
##     if ((l1 <- length(grep("[", pattern, fixed = TRUE))) &&
##         l1 != length(grep("]", pattern, fixed = TRUE))) {
##       if (pattern == "[") {
##         pattern <- "\\\\["
##         warning("replaced regular expression pattern '[' by '\\\\\\\\['")
##       }
##       else if (length(grep("[^\\\\\\\\]\\\\\\[<", pattern))) {
##         pattern <- sub("\\\\[<", "\\\\\\\\<", pattern)
##         warning("replaced '[<' by '\\\\\\\\[<' in regular expression pattern")
##       }
##     }
##   }
## }
```

```

##           grep(pattern, all.names, value = TRUE)
##     }
## else all.names
## }
## <bytecode: 0x7fe50f2a6eb0>
## <environment: namespace:base>
```

`rm` 명령어를 사용해서 더이상 사용할 필요가 없는 객체를 삭제한다:

```
rm(x)
```

작업 환경에 너무 많은 객체가 있고, 전부 삭제하고자 한다면, `rm` 함수에 `ls` 결과를 전달하면 된다:

```
rm(list = ls())
```

상기 예제에서, 명령어 두개를 조합했다. 연산 우선순위처럼, 가장안쪽 괄호 내부에 있는 것이 먼저 평가되고, 쭉 이어서 평가된다.

상기 예제에서, `ls` 결과가 `rm` 삭제 명령어 `list` 인자로 사용되도록 지정했다. 값을 이름으로 인자에 할당할 때, = 연산자를 사용해야만 한다!

대신에 <-가 사용되면, 의도하지 못한 역효과가 발생되거나, 오류 메시지만 얻게 된다:

```
rm(list <- ls())
```

```
## Error in rm(list <- ls()): ...
```

꿀팁: 경고 vs. 오류

R이 예상하지 못한 무언가 수행할 때 주위를 기울이다! R이 연산작업을 더이상 진행할 수 없을 때 상기와 같이 오류를 던진다.

다른 한편으로 경고는 일반적으로 함수는 실행된다. 하지만, 함수는 아마도 예상한 것처럼 동작하지 않을 것이다.

두가지 경우 몯, R이 출력해서 전해주는 메시지가 문제 해결에 대한 단서를 줄 것이다.

21.13 R 팩키지

팩키지를 작성하거나, 누군가 작성한 팩키지를 얻어 R에 함수추가도 가능하다. 현 저작시점에, CRAN(comprehensive R archive network)에는 13,000개 이상 팩키지가 있다. R과 RStudio 모두 팩키지 관리 기능이 있다:

- `installed.packages()` 타이핑하면, 어떤 팩키지가 설치되어 있는지 확인할 수 있다.
- `install.packages("packagename")` 타이핑하면, 팩키지를 설치할 수 있다. 여기서 `packagename`은 팩키지명칭이 된다.
- `update.packages()` 타이핑하면, 설치된 팩키지를 갱신할 수 있다.

- `remove.packages("packagename")` 타이핑하면, 팩키지를 제거한다.
- `library(packagename)` 명령어로 팩키지를 사용할 수 있게 한다.

21.14 도전과제 2

다음 프로그램에 나온 각 문장을 실행하게 되면, 각 변수 값에는 무슨 값이 담겨있을까요?

```
mass <- 47.5
age <- 122
mass <- mass * 2.3
age <- age - 20
```

도전과제 2에 대한 해답

```
mass <- 47.5
```

실행하게 되면 `mass` 변수에 47.5 값이 배정된다.

```
age <- 122
```

`age` 변수에 122 값이 배정된다.

```
mass <- mass * 2.3
```

상기 문장은 2.3을 47.5에 곱하여 `mass` 변수에 109.25 값이 배정된다.

```
age <- age - 20
```

기존 변수 `age` 122 값에 20을 빼서 변수 `age`에는 102 값이 할당된다.

21.15 도전과제 3

이전 도전과제 코드를 실행하고 난 후, `mass`와 `age`를 비교하는 코드를 작성한다. `mass`와 `age` 중 어느 것이 더 큰가?

도전과제 3에 대한 해답

상기 질문에 답하는 방식은 다음과 같이 `>` 을 사용하는 것이다:

```
mass > age
`~`
```

```
109.25 102      TRUE .
```

```
##    4 {#r-rstudio-chellenge-four}
```

```
mass age .
```

```
**     4      **
```

```
`rm`
```

```
rm(age, mass)
```

21.16 도전과제 5

ggplot2, plyr, gapminder 팩키지를 설치하라.

도전과제 5에 대한 해답

install.packages() 명령어를 사용해서 팩키지를 설치한다.

```
install.packages("ggplot2")
install.packages("plyr")
install.packages("gapminder")
` `` `
```

Chapter 22

RStudio 프로젝트 관리

22.1 들어가며

과학적 과정은 본질적으로 증분이다. 프로젝트 대부분은 아무렇게 적은 노트필기, 일부 코드, 그리고 나서, 원고작성, 그리고 나면 종국에 모든 것이 함께 섞여진다.

Managing your projects in a reproducible fashion does not just make your science reproducible, it makes your life easier.

— Vince Buffalo April 15, 2013

대부분은 다음과 같이 프로젝트를 구조화하는 경향이 있다:

항상 이런 방식을 회피해야 되는 이유는 많다:

1. 어느 데이터 버전이 원본이고, 어느 데이터 버전이 변경된 것인지 분간하기 정말 힘들다;
2. 다양한 확장자를 갖는 파일과 뒤섞일 때, 정말 엉망이 된다;
3. 아마도 실제 파일을 찾고, 해당 그래프를 생성하는데 사용된 정확한 프로그램 코드와 맞는 그림을 연결시키는데 시간이 엄청 많이 소요될 것이다.

프로젝트를 잘 배치하게 되면 궁극적으로 여러분의 삶을 편안하게 만들어 줄 것이다:

- 데이터 정합성을 보장할 것이다;
- 작성한 코드를 다른 사람(연구실 동료, 공동연구자, 지도교수)과 더 단순하게 공유할 수 있게 만든다;
- 논문 제출할 때 코드를 쉽게 업로드할 수 있게 한다;
- 휴가 뒤에, 프로젝트 백업을 더 손쉽게 한다.

22.2 가능한 해결책

다행스럽게도, 작업을 효과적으로 관리할 수 있게 도움이 되는 도구와 팩키지가 있다.

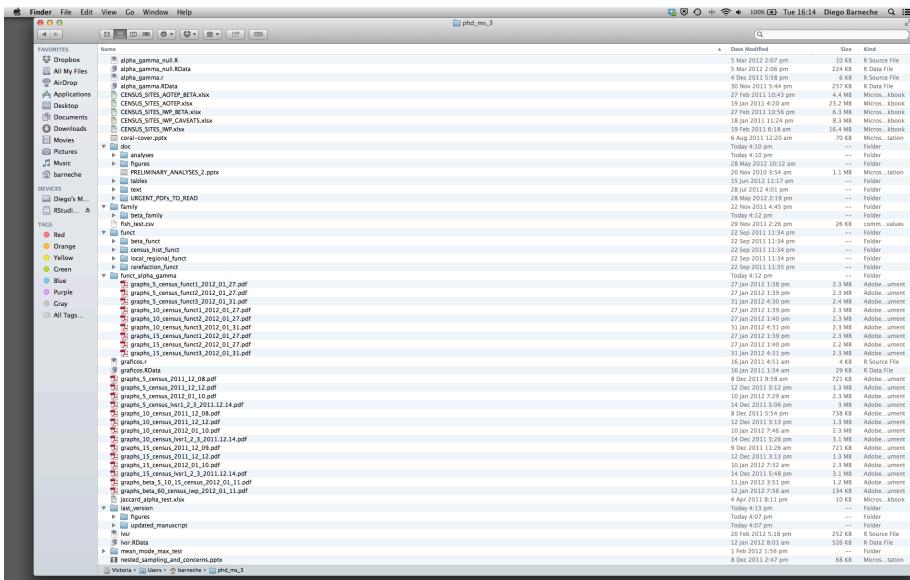


Figure 22.1: 혼한 프로젝트 디렉토리 구조

RStudio의 가장 강력하고 유용한 측면 중 하나가 프로젝트 관리 기능이다. 프로젝트 관리 기능을 사용해서, 모든 것이 갖춘 재현가능한 프로젝트를 생성한다.

22.3 도전과제

모든 것을 갖춘 프로젝트 생성해보자. RStudio에서 새로운 프로젝트를 생성한다:

1. “File” 메뉴를 클릭하고 나서, “New Project” 선택한다.
2. “New Directory”를 클릭한다.
3. “Empty Project”를 클릭한다.
4. 프로젝트를 저장할 디렉토리 명칭을 타이핑한다. 예를 들어, “my_project”.
5. “Create a git repository”에 대한 체크박스가 선택되었는지 확실히 한다.
6. “Create Project” 버튼을 클릭한다.

이제 프로젝트 디렉토리에서 R을 시작하거나, RStudio로 해당 프로젝트를 열게 되면, 프로젝트에 모든 작업은 해당 디렉토리에 완전히 담겨지게 된다.

22.4 프로젝트 구성 모범 사례

프로젝트를 구성하는 “가장 최선의” 방식이 없지만, 프로젝트 관리를 더 수월하게 하는데 준수해야 되는 일반적인 원칙이 몇가지 있다:

22.4.1 데이터는 읽기 전용

아마도 이것이 프로젝트 설정에 대한 가장 중요한 목적이다. 데이터는 일반적으로 수집하는데 시간이 많이 걸리고 비용이 많이 듈다. (엑셀처럼) 인터랙티브하게 데이터를 작업하게 되면, 필연적으로 데이터에 변형이 일어나고 데이터 출처와, 수집이 이루어진 뒤에 어떻게 변형되었는지 확인을 할 수 없게 된다. 따라서, 데이터를 “읽기-전용(Read-Only)”으로 다룬다.

22.4.2 데이터 정제

많은 경우에, 데이터가 “지저분하다”: 상당한 전처리 과정을 거쳐야 R 형식(혹은 다른 프로그래밍 언어)으로 유용하게 사용될 수 있다. 이런 작업이 “데이터 정제작업”(Data Munging, Data Wrangling)이라고 불린다. 별도 디렉토리에 데이터 정제 스크립트를 보관하고, “정제된” 데이터셋을 보관하는 두번째 “읽기-전용” 데이터 디렉토리를 생성하는 것도 유용하다.

22.4.3 자동생성 산출물은 일회용품

작성한 스크립트로 자동생성된 어떤 것이든 일회용품처럼 처리해만 된다: 작성한 스크립트가 모두 다시 자동생성할 수 있어야만 된다.

자동출력된 산출물을 관리하는 다른 방식은 많다. 각기 다른 분석마다 다른 하위디렉토리에 출력결과를 저장하는 것이 유용하다. 나중을 위해서 이런 접근법이 더 수월한데, 대부분의 분석이 탐색적이고, 최종 프로젝트에 채택되지도 않기 때문이고, 일부 분석 결과는 프로젝트 중간에 공유되기도 한다.

꿀팁: 과학적 컴퓨팅 위한 적정 관행

Good Enough Practices for Scientific Computing에서 프로젝트 구조화에 대한 다음과 같은 추천사항을 제시하고 있다:

1. 프로젝트명을 따서 자체 디렉토리를 갖도록 각 프로젝트를 배치한다.
2. doc 디렉토리에 프로젝트와 연관된 텍스트 문서를 배치한다.
3. data 디렉토리에 원본데이터와 메타데이터를 배치하고 results 디렉토리에 분석과정에서 생성되는 파일을 배치시킨다.
4. src 디렉토리에 프로젝트 스크립트와 프로그램 소스 코드를 배치하고, bin 디렉토리에 로컬 컴퓨터에서 컴파일 되거나 외부에서 가져온 프로그램을 배치한다.
5. 모든 파일이름을 콘텐츠 혹은 함수를 적절히 반영하도록 이름 짓는다.

22.4.4 재사용 함수 정의와 응용활용을 구별

R로 작업하는 가장 효과적인 방법이 인터랙티브 세션에서 여러가지 작업을 하다가, 제대로 동작하고 원하는 기능이 구현되면 .R 스크립트 파일로 명령어를 복사해서 넣는 것이다. history 명령어를 사용해서 지금까지 입력한 모든 명령어를 저장할 수도 있다. 하지만, 명령어 90%가 시행착오라 그다지 유용하지 않을 수 있다.

프로젝트가 새롭고 참신할 때, 스크립트 파일에 대체로 직접 실행되는 코드 라인이 많이 포함되어 있다. 코드 성숙도가 높아짐에 따라, 재사용 가능한 코드 덩어리를 함수로 뽑아낸다. 이런 함수를 별도 디렉토리에 몰아 넣는 것도 좋은 아이디어다; 여러 분석에 걸쳐 폭넓게 재사용되는 유용한 디렉토리와 분석 스크립트를 저장하는 디렉토리.

꿀팁: 중복 회피하기

다수 프로젝트에서 데이터와 분석 스크립트를 사용하는 경우가 있다. 일반적으로, 중복을 피해서, 공간도 절약하고, 여러 곳에 코드를 간접화하는 수고도 회피하고자 한다.

이런 경우, “기호 연결(Symbolic Link)”이 유용하다. 본질적으로 파일 시스템 어딘가 있는 파일에 대한 단축키다. 리눅스나 맥 OS X에서는 `ln -s` 명령어를 사용하고, 윈도우에서는 단축키를 생성하거나, `mklink` 명령어를 윈도우 터미널에서 사용한다.

22.4.5 데이터 디렉토리에 데이터 저장

이제 멋진 디렉토리 구조를 갖추어서, `data/` 디렉토리에 데이터 파일을 위치/저장한다.

22.5 도전 과제 1

`gapminder` 데이터를 웹사이트에서 다운로드 한다.

1. 파일을 다운로드한다 (CTRL + S, 마우스 우클릭 -> “Save as”, 혹은 File -> “Save page as”)
2. `gapminder-FiveYearData.csv` 파일명으로 저장된 것인지 확인한다.
3. 프로젝트 내부 `data/` 디렉토리에 파일을 저장한다.

나중에 데이터를 불러 적재하고 검사작업을 진행한다.

22.5.1 버전 제어

프로젝트와 버전 제어를 사용하는 것이 중요하다.

22.6 도전 과제 1

데이터를 R에 올리기 전에, 프롬프트 명령라인에서 데이터셋에 대한 전반적인 아이디어를 획득하는 것이 좋다. R에 데이터를 올리는 방법을 결정할 때 데이터셋에 대해 더 잘 이해하는 것이 수월하게 된다. 다음 질문에 답을 하는데 명령라인 쉘을 사용하라. 1. 파일 크기는 얼마나 되는가? 2. 데이터 행수는 얼마나 되는가? 3. 데이터 파일에는 어떤 유형의 값이 저장되어 있는가?

도전과제 2에 대한 해답

쉘에 다음 명령어를 실행한다.:

```
ls -lh data/gapminder_data.csv
## -rw-r--r-- 1 tidyverse staff    80K 10 17 19:31 data/gapminder_data.csv
파일 크기는 80K.
wc -l data/gapminder_data.csv
##      1705 data/gapminder_data.csv
1705 줄이다. 데이터는 다음과 같이 생겼다:
head data/gapminder_data.csv
## country,year,pop,continent,lifeExp,gdpPercap
## Afghanistan,1952,8425333,Asia,28.801,779.4453145
## Afghanistan,1957,9240934,Asia,30.332,820.8530296
## Afghanistan,1962,10267083,Asia,31.997,853.10071
## Afghanistan,1967,11537966,Asia,34.02,836.1971382
## Afghanistan,1972,13079460,Asia,36.088,739.9811058
## Afghanistan,1977,14880372,Asia,38.438,786.11336
## Afghanistan,1982,12881816,Asia,39.854,978.0114388
## Afghanistan,1987,13867957,Asia,40.822,852.3959448
## Afghanistan,1992,16317921,Asia,41.674,649.3413952
```

꿀팁: R Studio에서 명령라인

Tools -> Shell... 메뉴를 통해서 RStudio에서 셸을 띄울 수 있다.

Chapter 23

도움 청하기

23.1 도움말 파일 읽기

R과 모든 팩키지는 함수에 대한 도움말 파일을 제공한다. 네임스페이스(인터랙티브 R 세션)에 적재된 팩키지에 있는 특정 함수에 대한 도움말은 다음과 같이 찾는다:

```
?function_name  
help(function_name)
```

RStudio에 도움말 페이지에 도움말이 표시된다. (혹은 R 자체로 일반 텍스트로 표시된다)

각 도움말 페이지는 절(section)로 구분된다:

- 기술(Description): 함수가 어떤 작업을 수행하는가에 대한 충분한 기술
- 사용법(Usage): 함수 인자와 기본디폴트 설정값
- 인자(Arguments): 각 인자가 예상하는 데이터 설명
- 상세 설명(Details): 알고 있어야 되는 중요한 구체적인 설명
- 값(Value): 함수가 반환하는 데이터
- 함께 보기(See Also): 유용할 수 있는 연관된 함수.
- 예제(Examples): 함수 사용법에 대한 예제들.

함수마다 상이한 절을 갖추고 있다. 하지만, 상기 항목이 알고 있어야 하는 핵심 내용이다.

꿀팁: 도움말 파일 불러 읽어오기

R에 대해 가장 기죽게 되는 한 측면이 엄청난 함수 갯수다. 모든 함수에 대한 올바른 사용법을 기억하지 못하면, 엄두가 나지 않을 것이다. 운좋게도, 도움말 파일로 인해 기억할 필요가 없다!

23.2 특수 연산자

특수 연산자에 대한 도움말을 찾으려면, 인용부호를 사용한다:

```
? "<-"
```

23.3 팩키지 도움말 얻기

많은 팩키지에 “소품문(vignettes)”이 따라온다: 활용법과 풍부한 예제를 담은 문서. 어떤 인자도 없이, `vignette()` 명령어를 입력하면 설치된 모든 팩키지에 대한 모든 소품문 목록이 출력된다; `vignette(package="package-name")` 명령어는 `package-name` 팩키지명에 대한 이용 가능한 모든 소품문 목록을 출력하고, `vignette("vignette-name")` 명령어는 특정된 소품문을 얻다.

팩키지에 어떤 소품문도 포함되지 않는다면, 일반적으로 `help("package-name")` 명령어를 타이핑해서 도움말을 얻는다.

23.4 함수가 정확하게 기억나지 않을 때

함수가 어느 팩키지에 있는지 확신을 못하거나, 구체적인 철자법을 모르는 경우, 퍼지 검색(fuzzy search)을 실행한다:

```
??function_name
```

23.5 어디서 시작해야 될지 아무 생각이 없을 때

어떤 함수 혹은 팩키지가 필요한지 모르는 경우, CRAN Task Views 사이트가 좋은 시작점이 된다. 유지관리되는 팩키지 목록이 필드로 묶여 잘 정리되어 있다.

23.6 코드가 동작않을 때: 동료에게 도움 구함

함수 사용에 어려움이 있는 경우, 10 에 9 경우에 찾는 정답이 이미 Stack Overflow에 답글이 달려 있다. 검색할 때 [r] 태그를 사용한다:

원하는 답을 찾지 못한 경우, 동료에게 질문을 만드는데 몇 가지 유용한 함수가 있다:

```
?dput
```

`dput()` 함수는 작업하고 있는 데이터를 텍스트 파일 형식으로 덤프해서 저장한다. 그래서 다른 사람 R 세션으로 복사해서 붙여넣기 좋게 돋는다.

```
sessionInfo()
```

```
## R version 4.1.3 (2022-03-10)
## Platform: x86_64-apple-darwin20.6.0 (64-bit)
```

```

## Running under: macOS Big Sur 11.7
##
## Matrix products: default
## BLAS:    /usr/local/Cellar/openblas/0.3.20/lib/libopenblas-r0.3.20.dylib
## LAPACK:  /usr/local/Cellar/r/4.1.3/lib/R/lib/libRlapack.dylib
##
## locale:
## [1] ko_KR.UTF-8/ko_KR.UTF-8/ko_KR.UTF-8/C/ko_KR.UTF-8/C
##
## attached base packages:
## [1] stats      graphics   grDevices utils      datasets   methods    base
##
## other attached packages:
## [1]forcats_0.5.1  stringr_1.4.0  dplyr_1.0.9   purrr_0.3.4
## [5]readr_2.1.1    tidyr_1.2.0    tibble_3.1.8   ggplot2_3.3.5
## [9]tidyverse_1.3.1 DBI_1.1.1
##
## loaded via a namespace (and not attached):
## [1]tidyselect_1.1.2 xfun_0.28      haven_2.4.3     colorspace_2.0-2
## [5]vctrs_0.4.1     generics_0.1.2   htmltools_0.5.3 yaml_2.2.1
## [9]blob_1.2.2      utf8_1.2.2     rlang_1.0.4     pillar_1.8.1
## [13]glue_1.6.2     withr_2.5.0    bit64_4.0.5    dbplyr_2.1.1
## [17]modelr_0.1.8   readxl_1.3.1   lifecycle_1.0.1 munsell_0.5.0
## [21]gtable_0.3.0   cellranger_1.1.0 rvest_1.0.2     memoise_2.0.1
## [25]evaluate_0.14  knitr_1.36    tzdb_0.2.0     fastmap_1.1.0
## [29]fansi_1.0.3   broom_1.0.0   Rcpp_1.0.9     backports_1.4.0
## [33]scales_1.1.1   cachem_1.0.6   jsonlite_1.8.0 bit_4.0.4
## [37]fs_1.5.2       hms_1.1.1     digest_0.6.29  stringi_1.7.6
## [41]bookdown_0.24  grid_4.1.3    cli_3.3.0     tools_4.1.3
## [45]magrittr_2.0.3 RSQLite_2.2.12 crayon_1.5.1   pkgconfig_2.0.3
## [49]ellipsis_0.3.2 xml2_1.3.3    reprex_2.0.1   lubridate_1.8.0
## [53]assertthat_0.2.1 rmarkdown_2.11 httr_1.4.2    rstudioapi_0.13
## [57]R6_2.5.1       compiler_4.1.3

```

`sessionInfo()`는 R 현재 버전 정보와 함께 적재된 팩키지 정보를 출력한다. 이 정보가 다른 사람이 여러분 문제를 재현하고 디버그하는데 유용할 수 있다.

23.7 도전과제 1

`c` 연결함수에 대한 도움말을 살펴본다. 다음 명령어를 실행하면, 어떤 종류 벡터가 생성될 것으로 예상되는가:

```

c(1, 2, 3)
c('d', 'e', 'f')
c(1, 2, 'f')`
```

도전과제 1에 대한 해답

`c()` 함수는 벡터를 생성하는데 벡터내 모든 원소는 동일한 자료형이여야 한다. 첫번째의 경우 모든 원소는 숫자형이다. 두번째의 경우 모두 문자형이다. 세번째의 경우 문자형이다: 숫자형은 문자로 강제변환(coerced) 된다.

23.8 도전과제 2

`paste` 함수에 대한 도움말을 살펴본다. 나중에 이 함수를 사용할 것이다. `sep` 와 `collapse` 인자 사이에 차이는 무엇인가?

도전과제 2에 대한 해답

`paste()` 함수에 대한 도움말은 다음 명령어를 사용한다:

```
help("paste")
?paste
```

`sep`, `collapse` 함수 차이는 다소 난해하다. `paste` 함수는 임의 숫자 인자를 받는데 각각은 임의 길이를 갖는 벡터가 될 수 있다. `sep` 인자는 결합되는 요소들 사이에 문자를 지정한다; 기본디폴트로 공백이 된다. 반환되는 결과는 벡터로 된다. 반대로 `collapse`는 원소를 결합한 후에 해당 구분자를 이용하여 축약되어 하나의 문자열이 된다. 예를 들어,

```
paste(c("a", "b"), "c")
## [1] "a c" "b c"
paste(c("a", "b"), "c", sep = ", ")
## [1] "a,c" "b,c"
paste(c("a", "b"), "c", collapse = "|")
## [1] "a c|b c"
paste(c("a", "b"), "c", sep = ", ", collapse = "|")
## [1] "a,c|b,c"
```

(추가 정보가 필요한 경우, `paste` 도움말 페이지 하단에 예제를 참조한다. 혹은 `example('paste')` 명령어를 실행한다.)

23.9 도전과제 3

칼럼이 탭("\t")와 소수점(.)으로 구분되는 csv 파일을 불러올 수 있는 함수(연관된 모수)를 찾아내는데 도움말을 사용해보자. 소수점 구분자에 대한 확인이 중요한데 전세계 동료와 공동작업을 한다면 더욱 그렇다. 왜냐하면 다른 나라는 소수점

표기에 다른 관례를 사용하기 때문이다(예를 들어, 콤마 vs 점) 힌트: ??csv 명령어를 사용해서 csv 관련 함수를 찾아낸다.

도전과제 3에 대한 해답

소수점 구분자를 갖는 탭 구분 파일을 읽어 오는 표준 R 함수는 `read.delim()`이다. `read.table(file, sep="\t")` 명령어로 작업을 수행할 수 있다. `read.table()` 함수에 소수점이 기본설정된 구분자이며, 데이터파일에 hash (#) 문자를 갖가 포함되어 있다면 `comment.char` 인자도 변경해야 한다.

23.10 도움 되는 웹사이트

- Quick R
- RStudio cheat sheets
- Cookbook for R

Chapter 24

자료구조

R의 가장 강력한 기능중 하나는 표형식 데이터를 다룰 수 있는 능력이다 - 이미 스프레드쉬트나 CSV 파일을 갖고 있을 수 있다. data/ 디렉토리에 feline-data.csv 파일을 생성하면서 학습을 시작해 보자:

```
cats <- data.frame(coat = c("calico", "black", "tabby"),
                     weight = c(2.1, 5.0, 3.2),
                     likes_string = c(1, 0, 1))
write.csv(x = cats, file = "data/feline-data.csv", row.names = FALSE)
```

새로 생성한 feline-data.csv 파일 내부는 다음과 같다:

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
```

꿀팁: R에서 텍스트 파일 편집

대안으로, data/feline-data.csv 파일을 생성하는데 나노 편집기(Nano)를 사용하거나, RStudio File -> New File -> Text File 메뉴를 사용할 수 있다.

다음 명령어를 통해서 R로 데이터를 불러 가져온다:

```
cats <- read.csv(file = "data/feline-data.csv")
cats

##      coat weight likes_string
## 1  calico    2.1          1
## 2   black    5.0          0
## 3   tabby    3.2          1
```

read.table 함수를 사용해서 텍스트 파일에 저장된 표형식 데이터를 불러오는데 사용한다. 표형식 데이터는 CSV (csv = 콤마구분 값) 같은 구분 문자(punctuation

characters)로 칼럼이 구분된다. csv 파일에 데이터를 구분하는데 가장 일반적으로 사용되는 구분 문자가 탭과 콤마다. R에서 사용의 편리성을 위해서 `read.table` 함수에 두 가지 다른 버전 함수를 제공한다. `read.csv`가 콤마로 구분되는 데이터를 읽어들이는데 사용하고, `read.delim` 함수가 탭으로 구분되는 데이터를 읽어오는데 사용된다. `read.csv` 함수가 둘 중에 더 많이 사용된다. 필요한 경우 `read.csv`, `read.delim` 함수 모두 구분문자를 오버라이드해서 사용하는 것도 가능하다.

데이터프레임에 \$ 연산자를 사용해서 칼럼을 뽑아내면서 데이터셋에서 바로 탐색을 시작하는 것도 가능하다.

```
cats$weight
## [1] 2.1 5.0 3.2
cats$coat
## [1] "calico" "black"  "tabby"
칼럼에 다른 연산자 적용도 할 수 있다:
##      2 kg      :
cats$weight + 2
## [1] 4.1 7.0 5.2
paste("My cat is", cats$coat)
## [1] "My cat is calico" "My cat is black"  "My cat is tabby"
이번에 다음은 어떤가?
cats$weight + cats$coat
## Error in cats$weight + cats$coat:
```

상기 문장에서 발생된 것을 이해하는 것이 R로 데이터 분석을 성공적으로 수행하는데 중요하다.

24.1 자료형

2.1 더하기 "black"이 말이 되지 않기 때문에 마지막 문장이 오류를 뱉어낼 것이라고 추측한다면, 제대로 이해하고 있는 것이다. 이미 자료형(data type)으로 불리는 프로그래밍의 중요한 개념에 대한 직관을 갖추고 있는 것이다. 데이터 자료형이 무엇인지 다음을 통해 물어보게 된다:

```
typeof(cats$weight)
## [1] "double"
다섯 가지 주요 자료형이 있다: 실수형(double), 정수형(integer), 복소수형(complex), 논리형(logical), 문자형(character).
```

```
typeof(3.14)

## [1] "double"
typeof(1L) # The L suffix forces the number to be an integer, since by default R uses float numbers

## [1] "integer"
typeof(1+1i)

## [1] "complex"
typeof(TRUE)

## [1] "logical"
typeof('banana')

## [1] "character"
```

분석이 얼마나 복잡해지느냐에 관계없이, R에서 모든 데이터는 이러한 기본 자료형 중 하나로 해석된다. 이러한 엄격함이 정말로 중요한 결과를 잇하게 된다.

사용자가 또 다른 고양이에 대한 상세내용을 추가했고, 추가 정보는 `data/feline-data_v2.csv` 파일에 저장되어 있다.

```
file.show("data/feline-data_v2.csv")

coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
tabby,2.3 or 2.4,1
```

앞서와 마찬가지로 새로운 고양이 데이터를 불러와서 `weight` 칼럼에 데이터 자료형이 무엇인지 확인한다:

```
cats <- read.csv(file="data/feline-data_v2.csv")
typeof(cats$weight)
```

```
## [1] "character"
```

이러면 안되는데, 고양이 몸무게가 더이상 실수형 자료형이 아니다! 앞서 수행했던 동일한 수학연산을 취하게 되면 문제에 봉착한다:

```
cats$weight + 2
```

```
## Error in cats$weight + 2:
```

무슨 일이 일어난 걸까? R이 csv 파일을 불러올 때, 칼럼에 모든 것이 동일한 자료형이 되는 것을 요구한다; 칼럼에 모든 원소가 실수형으로 인식되지 않으면, 칼럼에 어떤 원소도 실수형이 될 수 없다. 고양이 데이터를 불러온 테이블을 데이터프레임(data.frame)이라고 부르고, 자료구조(data structure)로 불리는

첫번째 사례가 된다. 즉, 자료구조는 기본 자료형에서 R이 생성할 줄 아는 구조가 된다.

`class` 함수를 호출해서 데이터프레임인지를 알 수 있다:

```
class(cats)
```

```
## [1] "data.frame"
```

R에서 데이터를 성공적으로 사용하려면, 기본 자료구조가 무엇인지, 어떻게 동작하는지 이해할 필요가 있다. 지금으로서는 추가된 마지막 줄을 제거하고 좀더 살펴보도록 하자:

feline-data.csv:

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
```

RStudio로 다시 돌아와서:

```
cats <- read.csv(file="data/feline-data.csv")
```

24.2 벡터와 자료형 강제변환

강제변환(Type Coercion)을 보다 잘 이해하기 위해서, 또 다른 자료구조를 만나보자: 벡터(vector).

```
my_vector <- vector(length = 3)
my_vector
```

```
## [1] FALSE FALSE FALSE
```

R에서 벡터는 본질적으로 무언가 순서를 갖는 리스트로, 특별한 성질을 갖는데 벡터에 모든 것은 동일한 자료형을 갖는다는 점이다. 자료형을 지정하지 않게 되면, 기본디폴트 설정은 (logical)이 되거나; 자료형에 관계없이 공백터를 선언할 수 있다.

```
another_vector <- vector(mode='character', length=3)
another_vector
```

```
## [1] "" "" "
```

자료가 벡터인지 다음과 같이 확인한다:

```
str(another_vector)
```

```
## chr [1:3] "" "" "
```

상기 명령어로부터 다소 암호스러운 출력결과가 나오는데 해당 벡터에서 발견된 기본 자료형은 이 경우 `chr` 문자; 벡터에 나와있는 숫자는 벡터의 인덱스로 이 경우

[1:3]; 그리고 벡터에 실제로 들어있는 몇가지 예로, 이 경우 빈 문자열이 된다. 유사하게 다음을 실행하게 되면;

```
str(cats$weight)
## num [1:3] 2.1 5 3.2
```

cats\$weight도 벡터임을 알 수 있다 - R 데이터프레임으로 불러온 데이터 칼럼은 모두 벡터다, 그리고 이러한 연유로 칼럼에 있는 모든 원소는 동일한 자료형을 갖게 강제하는 이유가 된다.

토론 1

왜 R에서는 데이터 칼럼에 무엇을 넣는지에 대해서 고집스럽게 주장을 할까? 이러한 점은 어떻게 우리에게 도움이 될까?

토론 1

칼럼에 모든 것을 동일하게 둘으로써, 데이터에 관해서 단순한 가정을 할 수 있게 한다; 만약 칼럼의 첫번째 입력값이 숫자라면, 모든 입력값을 숫자로 해석할 수 있게 되고, 그렇게 함으로써 모든 것을 확인할 필요가 없게 된다. 깨끗한 데이터(clean data)라고 사람들이 회자할 때, 사람들이 의미하는 것이 이러한 일관성이다. 장기적으로 엄격한 일관성이 R에서 우리 삶을 풍요롭고 수월하게 만들 것이다.

결합 함수(`c()`)를 사용해서 벡터를 생성할 수 있다:

```
combine_vector <- c(2, 6, 3)
combine_vector
```

```
## [1] 2 6 3
```

지금까지 학습한 것을 바탕으로, 다음 문장은 어떤 결과를 출력하게 될까?

```
quiz_vector <- c(2, 6, '3')
```

이것을 자료형 강제변환(type coercion)라고 부른다. 이것이 많은 놀라움의 원천이고, 왜 기본 자료형에 대해서 인지하고 있어야 되는 이유가 되고, R이 해석하는 방식도 알아야 된다. R에서 혼합된 자료형(상기 예제는 숫자와 문자)의 경우 단하나의 벡터로 변환시킬 때, 모든 자료를 동일한 자료형으로 강제 변환시킨다. 다음을 생각해 보자.

```
coercion_vector <- c('a', TRUE)
coercion_vector
```

```
## [1] "a"      "TRUE"
another_coercion_vector <- c(0, TRUE)
another_coercion_vector
```

```
## [1] 0 1
```

강제 변환규칙은 다음과 같이 적용된다: $\text{character} \rightarrow \text{logical} \rightarrow \text{numeric} \rightarrow \text{integer} \rightarrow \text{double}$. 여기서 \rightarrow 표현은 다음으로 변환된다로 읽힌다. 이런 자동 변환규칙에 거슬러 자료형을 강제로 변환시키려면 `as.` 함수를 사용한다:

```
character_vector_example <- c('0', '2', '4')
character_vector_example

## [1] "0" "2" "4"

character_coerced_to_numeric <- as.numeric(character_vector_example)
character_coerced_to_numeric

## [1] 0 2 4

numeric_coerced_to_logical <- as.logical(character_coerced_to_numeric)
numeric_coerced_to_logical

## [1] FALSE TRUE TRUE
```

R이 기본 자료형을 다른 자료형으로 강제 변환할 때, 놀라운 일이 생겨난다! 자료형 강제변환에 대한 핵심사항은 차치하고 중요한 점은 다음과 같다: 본인 데이터가 생각한 바와 다르게 보인다면, 자료형 강제변환이 원인으로 지목되는 것이 당연하다; 벡터와 데이터프레임의 칼럼 자료형이 동일하도록 확실히 하라. 그렇지 않으면 끔찍한 놀라운 경험을하게 될 것이다!

하지만, 경우에 따라서는 자료형 강제변환이 매우 유용할 수도 있다! 예를 들어, `cats` 데이터프레임 `likes_string` 칼럼은 문자형이지만, 1과, 0이 실제로 `TRUE`와 `FALSE`를 표현한다는 것을 알고 있다. 이 경우 두 상태(`TRUE` 혹은 `FALSE`)를 갖는 논리형 자료형을 사용해야 한다. `as.logical` 함수를 사용해서 칼럼을 (`logical`)으로 ‘강제변환(coerce)’ 시킨다:

```
cats$likes_string

## [1] 1 0 1

cats$likes_string <- as.logical(cats$likes_string)
cats$likes_string
```

```
## [1] TRUE FALSE TRUE
```

결합 함수(`c()`)는 기존 벡터에 무언가 추가하는 역할을 수행한다:

```
ab_vector <- c('a', 'b')
ab_vector

## [1] "a" "b"

combine_example <- c(ab_vector, 'SWC')
combine_example

## [1] "a"     "b"     "SWC"
```

숫자 순열도 생성할 수 있다:

```
mySeries <- 1:10
mySeries

## [1] 1 2 3 4 5 6 7 8 9 10
seq(10)

## [1] 1 2 3 4 5 6 7 8 9 10
seq(1,10, by=0.1)

## [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4
## [16] 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9
## [31] 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.0 5.1 5.2 5.3 5.4
## [46] 5.5 5.6 5.7 5.8 5.9 6.0 6.1 6.2 6.3 6.4 6.5 6.6 6.7 6.8 6.9
## [61] 7.0 7.1 7.2 7.3 7.4 7.5 7.6 7.7 7.8 7.9 8.0 8.1 8.2 8.3 8.4
## [76] 8.5 8.6 8.7 8.8 8.9 9.0 9.1 9.2 9.3 9.4 9.5 9.6 9.7 9.8 9.9
## [91] 10.0
```

벡터에 관해 궁금한 점도 물어볼 수 있다:

```
sequence_example <- seq(10)
head(sequence_example, n=2)
```

```
## [1] 1 2
tail(sequence_example, n=4)
```

```
## [1] 7 8 9 10
length(sequence_example)
```

```
## [1] 10
class(sequence_example)
```

```
## [1] "integer"
typeof(sequence_example)
```

```
## [1] "integer"
```

마지막으로, 벡터의 각 원소에 명칭을 부여하는 것도 가능하다:

```
my_example <- 5:8
names(my_example) <- c("a", "b", "c", "d")
my_example
```

```
## a b c d
## 5 6 7 8
```

```
names(my_example)
```

```
## [1] "a" "b" "c" "d"
```

24.3 도전과제 1

1부터 26까지 숫자를 갖는 벡터를 생성하면서 시작해 보자. 생성한 벡터에 2를 곱해서 다시 자신에게 할당한다. 벡터에 A부터 Z까지 이름을 부여한다. (힌트: LETTERS라는 내장벡터가 있다.)

도전과제 1에 대한 해답

```
x <- 1:26
x <- x * 2
names(x) <- LETTERS
```

24.4 데이터프레임

데이터프레임(Data Frames)의 칼럼이 벡터라고 앞에서 언급했다:

```
str(cats$weight)
```

```
## num [1:3] 2.1 5 3.2
```

```
str(cats$likes_string)
```

```
## logi [1:3] TRUE FALSE TRUE
```

말이 된다. 다음은 어떤가?

```
str(cats$coat)
```

```
## Factor w/ 3 levels "black","calico",...: 2 1 3
```

24.5 범주형

또 다른 중요한 자료구조가 **범주형(factor)**이다. 범주형은 보통 문자 데이터처럼 생겼다. 하지만, 일반적으로 범주형 정보를 나타내는데 사용된다. 예를 들어, 연구중인 모든 고양이에 대한 색상을 문자열 벡터로 만들어보자:

```
coats <- c('tabby', 'tortoiseshell', 'tortoiseshell', 'black', 'tabby')
coats
```

```
## [1] "tabby"          "tortoiseshell" "tortoiseshell" "black"
```

```
## [5] "tabby"
```

```
str(coats)
```

```
## chr [1:5] "tabby" "tortoiseshell" "tortoiseshell" "black" "tabby"
```

문자열 벡터를 요인형으로 바꾸면 다음과 같다:

```
CATegories <- factor(coats)
class(CATegories)

## [1] "factor"
str(CATegories)

## Factor w/ 3 levels "black","tabby",...: 2 3 3 1 2
```

이제 R은 데이터에 3가지 가능한 범주가 있음을 파악하게 되었다 - 하지만, 놀라운 것도 함께 수행했다; 문자열을 출력하는 대신에, 숫자가 대량으로 출도되었다. R은 내부적으로 사람이 읽을 수 있는 범주를 숫자 인덱스로 치환시킨다. 이런 기능은 대다수 통계 계산에서 범주형 데이터를 숫자형으로 표현되는 기능을 활용하기 때문에 꼭 필요하다.

```
typeof(coats)

## [1] "character"
typeof(CATegories)

## [1] "integer"
```

24.6 도전과제 2

`cats` 데이터프레임에 요인형 칼럼이 있나요? 요인형 칼럼의 이름은 무엇인가요? `?read.csv` 명령어를 사용해서 텍스트 칼럼을 요인형 대신에 문자형으로 그대로 유지시키는 방법을 찾아내세요; 그리고 나서 `cat` 데이터프레임의 요인이 실제로 문자벡터임을 확인하는 명령문을 작성하시오.

도전과제 2에 대한 해답

해법으로 `stringAsFactors` 인자를 사용하면 된다.:

```
cats <- read.csv(file="data/feline-data.csv", stringsAsFactors=FALSE)
str(cats$coat)
```

또 다른 해법은 `colClasses` 인자를 사용해서 칼럼을 좀 더 면밀히 제어하는 것이다.

```
cats <- read.csv(file="data/feline-data.csv", colClasses=c(NA, NA, "character"))
str(cats$coat)
```

주의: 도움말 파일이 이해하기 어렵다는 학생이 다수 있다; 도움말 파일을 이해하기 어렵다는 것이 일반적이라서, 확신하지는 못하더라도, 문맥에 기초하여 최대한 추측하도록 용기를 주도록 한다.

모형 함수에서 기준 수준(baseline level)이 무엇인지 파악하는 것이 중요하다. 요인의 첫번째 범주로 가정하지만, 기본디폴트는 알파벳순으로 정해지게 되어 있다. 수준을 다음과 같이 지정해서 변경할 수 있다:

```
mydata <- c("case", "control", "control", "case")
factor_ordering_example <- factor(mydata, levels = c("control", "case"))
str(factor_ordering_example)

## Factor w/ 2 levels "control","case": 2 1 1 2
```

상기 경우 “control”를 1로, “case”를 2로 명시적으로 지정하도록 했다. 이러한 지정이 통계 모형 결과를 해석하는데 있어 매우 중요하다.

24.7 리스트

데이터 과학자로서 알고 있어야 되는 또 다른 자료구조가 (list)다. 리스트는 다른 자료형과 비교하여 몇가지 점에서 더 단순하다. 왜냐하면 원하는 무엇이든 넣을 수 있기 때문이다:

```
list_example <- list(1, "a", TRUE, 1+4i)
list_example

## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i

another_list <- list(title = "Numbers", numbers = 1:10, data = TRUE )
another_list
```

```
## $title
## [1] "Numbers"
##
## $numbers
## [1] 1 2 3 4 5 6 7 8 9 10
##
## $data
## [1] TRUE
```

데이터프레임에서 다소 놀라운 것을 이제 이해할 수 있다; 다음을 실행하게 되면 무슨 일이 발생될가:

```
typeof(cats)
```

```
## [1] "list"
```

데이터프레임은 ‘내부적으로(under the hood)’ 리스트라는 것을 알 수 있다 - 이유는 데이터프레임이 실제로 벡터와 요인으로 구성된 리스트이기 때문이다. 벡터와 요인으로 뒤섞인 칼럼을 붙잡아 두려면, 데이터프레임은 모든 칼럼을 유사한 표에 담을 수 있는 벡터보다 더 유연할 필요가 있다. 다른 말로, `data.frame`은 모든 벡터가 동일한 길이를 갖는 특별한 리스트로 정의할 수 있다.

`cats` 사례에서는 정수형, 숫자형, 논리형 변수로 구성된다. 이미 살펴봤듯이, 데이터프레임 각 칼럼은 벡터다.

```
cats$coat
```

```
## [1] calico black tabby
## Levels: black calico tabby
cats[,1]
```

```
## [1] calico black tabby
## Levels: black calico tabby
typeof(cats[,1])
```

```
## [1] "integer"
str(cats[,1])
```

```
## Factor w/ 3 levels "black","calico",...: 2 1 3
```

각 행은 다른 변수의 관측점(observation)으로 그 자체로 데이터프레임이다. 따라서, 서로 다른 자료형을 갖는 원소로 구성되어진다.

```
cats[1,]
```

```
##      coat weight likes_string
## 1 calico     2.1        TRUE
typeof(cats[1,])
```

```
## [1] "list"
str(cats[1,])
```

```
## 'data.frame':   1 obs. of  3 variables:
## $ coat       : Factor w/ 3 levels "black","calico",...: 2
## $ weight     : num 2.1
## $ likes_string: logi TRUE
```

24.8 도전과제 3

데이터프레임에서 변수와 관측점과 원소를 호출하는 미묘하지만 다른 방식이 존재한다:

- `cats[1]`

- `cats[[1]]`
- `cats$coat`
- `cats["coat"]`
- `cats[1, 1]`
- `cats[, 1]`
- `cats[1,]`

상기 예제를 시도해보고, 각각이 반환하는 것을 설명해 본다. 힌트: `typeof()` 함수를 사용해서 각각의 경우에 반환되는 것을 꼼꼼히 살펴본다.

도전과제 3에 대한 해답

```
cats[1]
```

```
##      coat
## 1 calico
## 2 black
## 3 tabby
```

데이터프레임을 벡터로 구성된 리스트로 간주할 수 있다. 단일 꺼쇠 `[1]`은 리스트의 첫번째 원소를 리스트로 반환한다. 이번 경우, 데이터프레임의 첫번째 칼럼이 된다.

```
cats[[1]]
```

```
## [1] calico black tabby
## Levels: black calico tabby
```

이중 꺼쇠 `[[1]]`은 리스트의 원소 내용물을 반환한다. 이번 경우, 리스트가 아닌 요인형 벡터로 첫번째 칼럼 내용물을 반환한다.

```
cats$coat
```

```
## [1] calico black tabby
## Levels: black calico tabby
```

명칭으로 항목을 꺼내는데 `$` 기호를 사용한다. `_coat_`가 데이터프레임의 첫번째 칼럼으로 요인형 벡터가 반환된다.

```
cats["coat"]
```

```
##      coat
## 1 calico
## 2 black
## 3 tabby
```

`["coat"]` 방식은 칼럼 인덱스를 명칭으로 바꾸고 동시에 꺼쇠를 사용한 경우다. 예제 1과 마찬가지로 반환되는 객체는 리스트가 된다.

```
cats[1, 1]
```

```
## [1] calico
## Levels: black calico tabby
```

단일 꺠쇠를 사용했는데 이번에는 행과 열의 좌표도 넣어 전달했다. 반환되는 객체는 첫번째 행, 첫번째 열에 교차하는 값이 된다. 반환되는 객체는 정수형이지만, 요인형 벡터의 일부분이라 정수값과 연관된 라벨 “calico”도 함께 출력한다.

```
cats[, 1]

## [1] calico black tabby
## Levels: black calico tabby
```

앞선 예제와 마찬가지로 꺠쇠를 하나만 사용했고, 행과 열 좌표도 전달했다. 행좌표를 지정하지 않은 경우, R에서 결측값은 해당 칼럼 벡터의 모든 원소로 해석된다.

```
cats[1, ]

##      coat weight likes_string
## 1 calico    2.1          TRUE
```

다시 한번, 꺠쇠를 하나만 사용했고, 행과 열 좌표도 전달했다. 칼럼 좌표가 지정되어 있지 않기 때문에, 첫번째 행의 모든 값을 포함하는 리스트가 반환된다.

24.9 행렬

마지막으로 중요한 자료형이 행렬(Matrices)이다. 0으로 가득찬 행렬을 다음과 같이 선언한다:

```
matrix_example <- matrix(0, ncol=6, nrow=3)

##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]     0     0     0     0     0     0
## [2,]     0     0     0     0     0     0
## [3,]     0     0     0     0     0     0
```

다른 자료구조와 마찬가지로, 행렬에 질문을 다음과 같이 던질 수 있다:

```
class(matrix_example)

## [1] "matrix" "array"
typeof(matrix_example)

## [1] "double"
str(matrix_example)

## num [1:3, 1:6] 0 0 0 0 0 0 0 0 0 0 ...
```

```
dim(matrix_example)

## [1] 3 6
nrow(matrix_example)

## [1] 3
ncol(matrix_example)

## [1] 6
```

24.10 도전과제 4

`length(matrix_example)` 실행결과는 어떻게 나올까? 시도해보자. 생각한 것과 일치하는가? 왜 그런가/ 왜 그렇지 않는가?

도전과제 4에 대한 해답

`length(matrix_example)` 실행결과는 어떻게 나올까?

```
matrix_example <- matrix(0, ncol=6, nrow=3)
length(matrix_example)
```

```
## [1] 18
```

행렬은 차원 속성이 추가된 벡터라서, `length()` 함수는 행렬의 전체 원소 갯수를 반환시킨다.

24.11 도전과제 5

또다른 행렬을 만들어 보자. 이번에는 1:50 숫자를 담고 있는 칼럼이 5, 행이 10일 행렬이다. `matrix()` 함수로 칼럼기준으로 혹은 행기준으로 채울 수 있나요? 행과 열을 바꿔 변경할 수 있는 방법을 찾아보자. (힌트: `matrix` 도움말 문서를 참조한다!)

도전과제 5에 대한 해답

또다른 행렬을 만들어 보자. 이번에는 1:50 숫자를 담고 있는 칼럼이 5, 행이 10일 행렬이다. `matrix()` 함수로 칼럼기준으로 혹은 행기준으로 채울 수 있나요? 행과 열을 바꿔 변경할 수 있는 방법을 찾아보자. (힌트: `matrix` 도움말 문서를 참조한다!)

```
x <- matrix(1:50, ncol=5, nrow=10)
x <- matrix(1:50, ncol=5, nrow=10, byrow = TRUE) # to fill by row
```

24.12 도전과제 6

이번 워크샵에서 다룬 각 섹션별 문자벡터를 포함하는 길이 2을 갖는 리스트를 생성하시오.

- 자료형(Data types)
- 자료구조(Data structures)

지금까지 살펴본 자료형(data type)과 자료구조(data structure)를 명칭으로 갖는 문자 벡터를 채워넣는다.

도전과제 6에 대한 해답

```
dataTypes <- c('double', 'complex', 'integer', 'character', 'logical')
dataStructures <- c('data.frame', 'vector', 'factor', 'list', 'matrix')
answer <- list(dataTypes, dataStructures)
```

주목: 칠판이나 벽에 자료형과 자료구조를 모두 적어두는 것이 도움이 될 수 있다 - 워크샵 동안 참여자들에게 기본 자료형과 구조의 중요성을 상기할 수 있기 때문이다.

24.13 도전과제 7

아래 행렬의 출력 결과를 생각해보자:

```
##      [,1] [,2]
## [1,]     4    1
## [2,]     9    5
## [3,]    10    7
```

이 행렬을 작성하는 올바른 명령어는 다음 중 무엇일까? 직접 타이핑하기 전에 각 명령어를 살펴보고, 정답을 생각해보자. 다른 명령어는 어떤 행렬을 만들어낼지도 생각해본다.

1. matrix(c(4, 1, 9, 5, 10, 7), nrow = 3)
2. matrix(c(4, 9, 10, 1, 5, 7), ncol = 2, byrow = TRUE)
3. matrix(c(4, 9, 10, 1, 5, 7), nrow = 2)
4. matrix(c(4, 1, 9, 5, 10, 7), ncol = 2, byrow = TRUE)

도전과제 7에 대한 해답

아래 행렬의 출력 결과를 생각해보자:

```
##      [,1] [,2]
## [1,]     4    1
## [2,]     9    5
## [3,]    10    7
```

이 행렬을 작성하는 올바른 명령어는 다음 중 무엇일까? 직접 타이핑하기 전에 각 명령어를 살펴보고, 정답을 생각해보자. 다른

명령어는 어떤 행렬을 만들어낼지도 생각해본다.

```
matrix(c(4, 1, 9, 5, 10, 7), ncol = 2, byrow = TRUE)
```

Chapter 25

데이터프레임 탐색

현 시점에서 모든 것을 살펴봤다: 마지막 수업에서, R의 모든 기본 자료형과 자료구조에 대한 여행을 마쳤다. 여러분이 수행하는 모든 작업은 이러한 도구를 조작하는 것이 된다. 하지만, 거의 대부분 쇼의 진정한 스타는 데이터프레임이다 - csv 파일에 정보를 불러와서 생성시킨 테이블. 이번 수업에서 데이터프레임으로 작업하는 방법에 대해 좀더 학습할 것이다.

25.1 행과 열을 추가

데이터프레임의 칼럼은 벡터라는 것을 배웠다. 따라서, 데이터는 칼럼에서 자료형의 일관성을 유지해야 한다. 이를테면, 칼럼을 새로 추가하려면 벡터를 새로 만들어서 시작한다:

```
library(tidyverse)

cats <- read_csv("data/feline-data.csv")
age <- c(2, 3, 5)
cats

## # A tibble: 3 x 3
##   coat    weight likes_string
##   <chr>    <dbl>      <dbl>
## 1 calico    2.1        1
## 2 black      5          0
## 3 tabby     3.2        1
```

age를 칼럼으로 다음과 같이 추가한다:

```
cbind(cats, age)

##       coat weight likes_string age
## 1 calico    2.1        1      2
```

```
## 2 black      5.0      0   3
## 3 tabby     3.2      1   5
```

데이터프레임의 행의 갯수와 다른 갯수를 갖는 age 벡터를 추가하게되면 추가되지 않고 오류가 발생됨에 주의한다:

```
age <- c(2, 3, 5, 12)
cbind(cats, age)
```

```
## Error in data.frame(..., check.names = FALSE): arguments imply differing number of
## Error in data.frame(..., check.names = FALSE): arguments imply differing number of
```

왜 정상동작이 되지 않을까? R은 테이블의 모든 행마다, 신규 칼럼에서도 원소 하나가 있길 원한다:

```
nrow(cats)
```

```
## [1] 3
```

```
length(age)
```

```
## [1] 2
```

그래서, 정상 동작하려면 `nrow(cats) = length(age)`이 되어야 한다. `cats` 콘텐츠를 새로운 데이터프레임으로 덮어써보자.

```
age <- c(2, 3, 5)
cats <- cbind(cats, age)
```

이제 행을 추가하면 어떻게 될까? 이미 데이터프레임의 행이 리스트라는 사실을 알고 있다:

```
newRow <- list("tortoiseshell", 3.3, TRUE, 9)
cats <- rbind(cats, newRow)
```

25.2 요인 (Factors)

살펴볼 것이 하나더 있다: `(factor)`에서 각기 다른 값을 `(level)`이라고 한다. 요인형 “coat” 변수는 수준이 3으로 구성된다: “black”, “calico”, “tabby”. R은 세가지 수준 중 하나와 매칭되는 값만 받아들인다. 완전 새로운 값을 추가하게 되면, 추가되는 신규 값은 NA가 된다.

경고 메시지를 통해서 coat 요인변수에 “tortoiseshell” 값을 추가하는데 성공하지 못했다고 알려준다. 하지만, 3.3 (숫자형), TRUE (논리형), and 9 (숫자형) 모두 weight, likes_string, age 변수에 성공적으로 추가된다. 왜냐하면 변수가 요인형이 아니라서 그렇다. “tortoiseshell”을 coat 요인변수에 성공적으로 추가하려면, 요인의 수준(level)로 “tortoiseshell”을 추가하면 된다:

```
levels(cats$coat)
## NULL
levels(cats$coat) <- c(levels(cats$coat), "tortoiseshell")
cats <- rbind(cats, list("tortoiseshell", 3.3, TRUE, 9))
```

대안으로, 요인형 벡터를 문자형 벡터로 변환시키면 된다; 요인변수의 범주를 잊게 되지만, 요인 수준을 조심스럽게 다룰 필요없이, 칼럼에 추가하고자 하는 임의 단어를 추가하면 된다:

```
str(cats)

## 'data.frame': 5 obs. of 4 variables:
## $ coat      : Factor w/ 1 level "tortoiseshell": NA NA NA 1 1
## $ weight    : num 2.1 5 3.2 3.3 3.3
## $ likes_string: num 1 0 1 1 1
## $ age       : num 2 3 5 9 9
cats$coat <- as.character(cats$coat)
str(cats)

## 'data.frame': 5 obs. of 4 variables:
## $ coat      : chr NA NA NA "tortoiseshell" ...
## $ weight    : num 2.1 5 3.2 3.3 3.3
## $ likes_string: num 1 0 1 1 1
## $ age       : num 2 3 5 9 9
```

25.3 도전과제 1

1. cats\$age 벡터에 7을 곱해서 human_age 벡터를 생성하자.
2. human_age를 요인형으로 변환시키자.
3. as.numeric() 함수를 사용해서 human_age 벡터로 다시 숫자형 벡터로 변환시킨다. 이제 7로 나눠서 원래 고양이 나이로 되돌리자. 무슨 일이 생겼는지 설명하자.

도전과제 1에 대한 해답

1. `human_age <- cats$age * 7`
2. `human_age <- factor(human_age). as.factor(human_age)`
works just as well.
3. `as.numeric(human_age)`을 실행하면 1 2 3 4 40이 된다.

왜냐하면 요인형 변수는 정수형(여기서 1:4)으로 자료를 저장하기 때문이다. 정수 라벨과 연관된 값은 여기서 28, 35, 56, 63이다. 요인형 변수를 숫자형 벡터로 변환시키면 라벨이 아니라 그 밑단의 정수를 반환시킨다. 원래 숫자를 원하는 경우, `human_age`를 문자형 벡터로 변환시키고 나서 숫자형 벡터로 변환시키면 된다.(왜 이방식은 정장 동작할까?) 실수로 숫자만 담긴 칼럼 어딘가 문자가 포함된 csv

파일로 작업할 때 이런 일이 실제로 종종 일어난다. 데이터를 불러 읽어올 때 `stringsAsFactors=FALSE` 설정을 잊지말자.

25.4 행 제거

이제 데이터프레임에 행과 열을 추가하는 방법을 알게 되었다 - 하지만, 데이터프레임에 “tortoiseshell” 고양이를 처음으로 추가하면서, 우연히 쓰레기 행을 추가시켰다:

```
cats
```

```
##           coat weight likes_string age
## 1       <NA>    2.1          1    2
## 2       <NA>    5.0          0    3
## 3       <NA>    3.2          1    5
## 4 tortoiseshell  3.3          1    9
## 5 tortoiseshell  3.3          1    9
```

데이터프레임에 문제가 되는 행을 마이너스해서 빼자:

```
cats[-4, ]
```

```
##           coat weight likes_string age
## 1       <NA>    2.1          1    2
## 2       <NA>    5.0          0    3
## 3       <NA>    3.2          1    5
## 5 tortoiseshell  3.3          1    9
```

-4, 다음에 아무것도 적시하지 않아서 4번째 행 전체를 제거함에 주목한다.

주목: 벡터 내부에 행 다수를 넣어 한번에 행을 제거할 수도 있다: `cats[c(-4, -5),]`

대안으로, NA 값을 갖는 모든 행을 제거시킨다:

```
na.omit(cats)
```

```
##           coat weight likes_string age
## 4 tortoiseshell  3.3          1    9
## 5 tortoiseshell  3.3          1    9
```

출력결과를 `cats`에 다시 대입하여 변경사항이 데이터프레임이 영구히 남도록 조치한다:

```
cats <- na.omit(cats)
```

25.5 칼럼 제거

데이터프레임의 칼럼도 제거할 수 있다. “age” 칼럼을 제거하고자 한다면 어떨까? 변수명과 변수 인덱스, 두가지 방식으로 칼럼을 제거할 수 있다.

```
cats[,-4]
```

```
##             coat weight likes_string
## 4 tortoiseshell    3.3          1
## 5 tortoiseshell    3.3          1
```

, -4 앞에 아무것도 없는 것에 주목한다. 모든 행을 간직한다는 의미를 갖는다.

대안으로, 색인명을 사용해서 칼럼을 제거할 수도 있다.

```
drop <- names(cats) %in% c("age")
cats[,!drop]
```

```
##             coat weight likes_string
## 4 tortoiseshell    3.3          1
## 5 tortoiseshell    3.3          1
```

25.6 dataframe 덧붙이기

데이터프레임(dataframe)에 데이터를 추가시킬 때 기억할 것은 칼럼은 벡터, 행은 리스트라는 사실이다. rbind() 함수를 사용해서 데이터프레임 두개를 본드로 붙이듯이 결합시킬 수 있다:

```
cats <- rbind(cats, cats)
cats
```

```
##             coat weight likes_string age
## 4 tortoiseshell    3.3          1    9
## 5 tortoiseshell    3.3          1    9
## 41 tortoiseshell   3.3          1    9
## 51 tortoiseshell   3.3          1    9
```

행명칭(rownames)이 불필요하게 복잡해져, 행명칭을 제거하면, 자동적으로 R이 순차적으로 행명칭을 부여시킨다.

```
rownames(cats) <- NULL
cats
```

```
##             coat weight likes_string age
## 1 tortoiseshell    3.3          1    9
## 2 tortoiseshell    3.3          1    9
## 3 tortoiseshell    3.3          1    9
## 4 tortoiseshell    3.3          1    9
```

25.7 도전과제 2

다음 구문을 사용해서 R 내부에서 직접 데이터프레임을 새로 만들 수 있다:

```
df <- data.frame(id = c("a", "b", "c"),
                  x = 1:3,
                  y = c(TRUE, TRUE, FALSE),
                  stringsAsFactors = FALSE)
```

다음 정보를 갖는 데이터프레임을 직접 제작해 보자:

- 이름(first name)
- 성(last name)
- 좋아하는 숫자

`rbind`를 사용해서 옆사람을 항목에 추가한다. 마지막으로 `cbind()` 함수를 사용해서 “지금이 커피시간인가요?”라는 질문의 답을 칼럼으로 추가한다.

도전과제 2에 대한 해답

```
df <- data.frame(first = c("Grace"),
                  last = c("Hopper"),
                  lucky_number = c(0),
                  stringsAsFactors = FALSE)
df <- rbind(df, list("Marie", "Curie", 238) )
df <- cbind(df, coffeetime = c(TRUE, TRUE))
```

25.8 현실적인 예제

지금까지 고양이 데이터로 가지고 데이터프레임 조작에 대한 기본적인 사항을 살펴봤다. 이제 학습한 기술을 사용해서 좀 더 현실적인 데이터셋을 다뤄보자. 앞에서 다운로드 받은 gapminder 데이터셋을 불러오자:

```
gapminder <- read.csv("data/gapminder_data.csv")
```

기타 팁

- 흔히 맞닥드리는 또 다른 유형의 파일이 탭구분자를 갖는 파일(.tsv)이다. 탭을 구분자로 명시하는데, "\\t"을 사용하고, `read.delim()` 함수로 불러 읽어온다.
- 파일을 `download.file()` 함수를 사용해서 인터넷으로부터 직접 본인 컴퓨터 폴더로 다운로드할 수 있다. `read.csv()` 함수를 실행해서 다운로드 받은 파일을 읽어온다. 예를 들어,

```
download.file("https://raw.githubusercontent.com/swcarpentry/r-novice-gapminder/gh-pages/data/gapminder/gapminder.csv",
              destfile = "data/gapminder_data.csv")
gapminder <- read.csv("data/gapminder_data.csv")
```

- 대안으로, `read.csv()` 함수 내부에 파일 경로를 웹주소를 치환해서

인터넷에서 직접 파일을 불러올 수도 있다. 이런 경우 로컬 컴퓨터에 csv 파일이 전혀 저장되지 않았다는 점을 주의한다. 예를 들어,

```
gapminder <- read.csv("https://raw.githubusercontent.com/swcarpentry/r-novice-gapminder/gh-pages/
```

- `readxl` 팩키지를 사용해서, 엑셀 스프레드시트를 평범한 텍스트로 변환하지 않고 직접 불러올 수도 있다.

gapminder 데이터셋을 좀더 살펴보자; 항상 가장 먼저 해야되는 작업은 `str` 명령어로 데이터가 어떻게 생겼는지 확인하는 것이다:

```
str(gapminder)
```

```
## 'data.frame': 1704 obs. of 6 variables:
## $ country : chr "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
## $ year    : int 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
## $ pop     : num 8425333 9240934 10267083 11537966 13079460 ...
## $ continent: chr "Asia" "Asia" "Asia" "Asia" ...
## $ lifeExp : num 28.8 30.3 32 34 36.1 ...
## $ gdpPercap: num 779 821 853 836 740 ...
```

`typeof()` 함수로 데이터프레임 칼럼 각각을 면밀히 조사할 수도 있다:

```
typeof(gapminder$year)
```

```
## [1] "integer"
```

```
typeof(gapminder$country)
```

```
## [1] "character"
```

```
str(gapminder$country)
```

```
## chr [1:1704] "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
```

데이터프레임 차원에 정보를 얻어낼 수도 있다; `str(gapminder)` 실행결과 `gapminder` 데이터프레임에 관측점 1704, 변수 6개가 있음을 상기한다. 다음 코드 실행결과는 무엇일까? 그리고 왜 그렇게 되는가?

```
length(gapminder)
```

```
## [1] 6
```

공정한 추측은 아마도 데이터프레임 길이가 행의 길이(1704)라고 보는 것이다. 하지만, 이번에는 다르다; 데이터프레임은 **벡터와 요인으로 구성된 리스트**라는 사실이다:

```
typeof(gapminder)
```

```
## [1] "list"
```

`length()` 함수는 6을 제시하는데, 이유는 `gapminder`가 6개 칼럼을 갖는 리스트로 만들어졌기 때문이다. 데이터셋에서 행과 열 숫자를 얻는데 다음 함수를 던져보자:

```
nrow(gapminder)
```

```
## [1] 1704
```

```
ncol(gapminder)
```

```
## [1] 6
```

혹은 한번에 보려면:

```
dim(gapminder)
```

```
## [1] 1704      6
```

또한, 모든 칼럼의 칼럼명이 무엇인지 파악하고자 하면 다음과 같이 질문을 던진다:

```
colnames(gapminder)
```

```
## [1] "country"     "year"        "pop"          "continent"    "lifeExp"     "gdpPercap"
```

현 단계에서, R이 제시하는 구조가 우리의 직관 혹은 예상과 부합되는지 물어보는 것이 중요하다; 각 칼럼에 대한 기본 자료형은 이해가 되는가? 만약 납득이 가지 않는다면, 후속 작업에서 나쁜 놀라운 사실로 전환되기 전에 문제를 해결해야 한다. 문제를 해결하는데, R이 데이터를 이해하는 방법과 데이터를 기록할 때 엄격한 일관성(strict consistency)의 중요성에 관해 학습한 것을 동원한다.

자료형과 자료구조가 타당해 보이게 되면, 데이터를 제대로 파고들어갈 시간이 되었다. `gapminder` 데이터 처음 몇줄을 살펴보자:

```
head(gapminder)
```

```
##       country year     pop continent lifeExp gdpPercap
## 1 Afghanistan 1952 8425333     Asia    28.8     779
## 2 Afghanistan 1957 9240934     Asia    30.3     821
## 3 Afghanistan 1962 10267083     Asia    32.0     853
## 4 Afghanistan 1967 11537966     Asia    34.0     836
## 5 Afghanistan 1972 13079460     Asia    36.1     740
## 6 Afghanistan 1977 14880372     Asia    38.4     786
```

25.9 도전과제 3

데이터 마지막 몇줄, 중간 몇줄을 점검하는 것도 좋은 습관이다. 그런데 어떻게 점검할 수 있을까? 중간 몇줄을 찾아보는 것이 너무 어렵지는 않지만, 임의로 몇줄을 추출할 수도 있다. 어떻게 할 수 있을까요?

** 도전과제 3에 대한 해답 마지막 몇줄을 점검하려면, R에 내장된 함수가 있어서 상대적으로 간단하다:

```
tail(gapminder)
```

```
tail(gapminder, n = 15)
```

데이터가 온전한지(혹은 관점에 따라 데이터가 온전하지 않은지)를 점검하는데 몇줄을 추출할 수 있을까요?

꿀팁: 몇가지 방법이 존재한다.

중첩함수(또다른 함수에 인자로 전달되는 함수)를 사용한 해법도 있다. 새로운 개념처럼 들리지만, 사실 이미 사용하고 있다. `my_dataframe[rows, cols]` 명령어는 데이터프레임을 화면에 뿌려준다. 데이터프레임에 행이 얼마나 많은지 알지 못하는데 어떻게 마지막 행을 뽑아낼 수 있을까? R에 내장된 함수가 있다. (의사) 난수를 얻어보는 것은 어떨까? R은 난수추출 함수도 갖추고 있다.

```
gapminder[sample(nrow(gapminder), 5), ]
```

분석결과를 재현가능하게 확실하게 만들려면, 코드를 스크립트 파일에 저장해서 나중에 다시 볼 수 있어야 한다.

25.10 도전과제 4

`file -> new file -> R script`로 가서, `gapminder` 데이터셋을 불러오는 R 스크립틀르 작성한다. `scripts/` 디렉토리에 저장하고 버전제어 시스템에도 추가한다. 인자로 파일 경로명을 사용해서 `source()` 함수를 사용해서 스크립트를 실행하라. (혹은 RStudio “source” 버튼을 누른다)

도전과제 4에 대한 해답 `scripts/load-gapminder.R` 파일에 담긴 내용물은 다음과 같다:

```
download.file("https://raw.githubusercontent.com/swcarpentry/r-novice-gapminder/gh-pages/_ep
              destfile = "data/gapminder_data.csv")
gapminder <- read.csv(file = "data/gapminder_data.csv")
```

스크립트를 실행시키면 데이터를 `gapminder` 변수에 적재시킨다:

```
source(file = "scripts/load-gapminder.R")
```

25.11 도전과제 5

`str(gapminder)` 출력결과를 다시 불러오자; 이번에는 `gapminder` 데이터에 대해 `str()` 함수가 출력하는 모든 것이 의미하는 바를 설명한다. 지금까지 학습한 요인, 리스트와 벡터 뿐만 아니라, `colnames()`, `dim()`과 같은 함수도 동원한다. 이해하지 못한 부분이 있다면, 주면 동료와 상의한다!

도전과제 5에 대한 해답

`gapminder` 객체는 다음 칼럼을 갖는 데이터프레임이다. - `country` `continent` 변수는 요인형 벡터 - `year` 변수는 정수형 벡터 - `pop`, `lifeExp`, `gdpPercap` 변수는 숫자형 벡터

Chapter 26

부분집합 추출

R에는 강력한 부분집합 연산자가 다수 구비되어 있다. 이를 완전히 익히게 되면 어떤 유형의 데이터셋에 대해서도 복잡한 연산을 수월하게 수행할 수 있게 된다.

어떤 유형의 객체에서 부분집합을 뽑아낼 수 있는 방식은 6가지가 있다. 다른 자료구조에 대한 부분집합을 뽑아내는 연산자는 3가지가 있다.

R의 핵심으로 가장 많은 일은 하는 것부터 시작해본다: 원자 숫자형 벡터(atomic vector)

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
x

##   a   b   c   d   e
## 5.4 6.2 7.1 4.8 7.5
```

원자 벡터(Atomic vectors)

R에서 문자열, 숫자, 논리 값을 갖는 간단한 벡터를 원자(atomic) 벡터라고 부르는데 이유는 더 이상 단순화할 수 없기 때문이다.

이제 가지고 놀 마루타 벡터를 생성하자. 해당 벡터 내용물을 순서에 넣는 방식은 무엇인가?

26.1 색인 사용 요소 접근

벡터 요소를 추출하는데, 대응되는 색인을 부여하는데, 1부터 시작된다:

```
x[1]
```

```
##   a
## 5.4
```

```
x[4]
```

```
## d
## 4.8
```

꺾쇠 괄호 연산자는 다른 어떤 함수와 비슷한다. 원자 벡터(행렬)에 대해, “n번째 요소를 뽑아낸다”라는 의미다.

한번에 다수 요소를 뽑아낼 수도 있다:

```
x[c(1, 3)]
```

```
## a c
## 5.4 7.1
```

혹은, 벡터 슬라이스로 뽑아낼 수도 있다:

```
x[1:4]
```

```
## a b c d
## 5.4 6.2 7.1 4.8
```

: 연산자는 왼쪽 요소부터 우측 요소까지 연속된 숫자를 생성한다. 예를 들어, x[1:4] 은 x[c(1,2,3,4)]와 동등하다:

```
1:4
```

```
## [1] 1 2 3 4
c(1, 2, 3, 4)
```

```
## [1] 1 2 3 4
```

동일한 원소를 여러번 추출하는 것도 가능하다:

```
x[c(1,1,3)]
```

```
## a a c
## 5.4 5.4 7.1
```

벡터를 벗어난 숫자를 뽑아내려고 하면, R은 결측값을 반환한다:

```
x[6]
```

```
## <NA>
## NA
```

길이 1을 갖는 벡터로 NA가 담겨있고, 명칭도 NA다.

0번째 요소를 뽑아내려고 하면, 공백터가 반환된다:

```
x[0]
```

```
## named numeric(0)
```

R에서 벡터 번호매기는 것은 1에서 시작

대다수 프로그래밍 언어(C와 파이썬)에서, 벡터 첫번째 요소는 색인 0을 갖는다. R에서, 첫번째 요소는 1이다.

26.2 요소 건너뛰고 제거

벡터 색인으로 음수를 사용하면, R은 명세된 숫자를 제외한 모든 요소를 반환한다:

```
x[-2]
```

```
##   a   c   d   e
## 5.4 7.1 4.8 7.5
```

다수 요소를 건너뛸 수도 있다:

```
x[c(-1, -5)] # or x[-c(1, 5)]
```

```
##   b   c   d
## 6.2 7.1 4.8
```

꿀팁: 연산작업 순서

초보자가 범하는 일반적인 실수는 벡터 슬라이스 건너뛰기 연산을 시도할 때 일어난다. 먼저 사람 대부분은 순열을 다음과 같이 부정연산을 통해 변경하려 한다:

```
x[-1:3]
```

다소 암호스런 오류가 제시된다:

```
## Error in x[-1:3]: only 0's may be mixed with negative subscripts
```

하지만, 연산작업 우선수위를 기억해보자. `:` 연산자는 사실 함수다. 그래서, 일어난 상황은 -1을 첫번째 인자로 받고, 두번째 인자로 3을 받아서, 연속된 숫자를 생성해낸다: `c(-1, 0, 1, 2, 3)`. 올바른 해법은 함수 호출을 괄호로 감싸는 것이다. - 연산자가 결과를 도출한다:

```
x[-(1:3)]
```

```
##   d   e
## 4.8 7.5
```

벡터에서 요소를 제거하려면, 결과를 다시 벡터에 대입할 필요가 있다:

```
x <- x[-4]
x
```

```
##   a   b   c   e
## 5.4 6.2 7.1 7.5
```

26.3 도전과제 1

다음과 같이 코드가 주어졌다:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

```
##   a   b   c   d   e
## 5.4 6.2 7.1 4.8 7.5
```

다음 출력결과를 산출하는 적어도 서로 다른 명령어 2개 제시해보자:

```
##   b   c   d
## 6.2 7.1 4.8
```

서로 다른 정답을 찾은 후에, 작업결과를 옆 사람과 비교한다. 서로 다른 전략을 취했나요?

도전과제 1에 대한 해답

```
x[2:4]
```

```
##   b   c   d
## 6.2 7.1 4.8
x[-c(1,5)]
```

```
##   b   c   d
## 6.2 7.1 4.8
x[c(2,3,4)]
```

```
##   b   c   d
## 6.2 7.1 4.8
```

26.4 명칭으로 부분집합 추출

색인 대신에 명칭을 사용해서, 요소를 뽑아낼 수 있다:

```
x <- c(a=5.4, b=6.2, c=7.1, d=4.8, e=7.5) # we can name a vector 'on the fly'
x[c("a", "c")]
```

```
##   a   c
## 5.4 7.1
```

명칭을 사용한 것이 객체에 대한 부분집합을 뽑아내는 훨씬 더 신뢰성 있는 방식이다: 다양한 요소 위치는 부분집합을 뽑아내는 연산자를 연결해서 적용할 때 종종 변경되지만, 명칭은 항상 동일하게 남게 마련이다!

26.5 논리 연산자 부분집합 추출

부분집합을 뽑아내는데 논리 벡터를 사용할 수도 있다:

```
x[c(FALSE, FALSE, TRUE, FALSE, TRUE)]
```

```
##   c   e
## 7.1 7.5
```

비교연산자(예를 들어, `>`, `<`, `==`)로 논리 벡터가 생성되기 때문에, 이를 사용해서 간결하게 벡터 부분집합을 추출할 수 있다: 다음 문장은 앞선 문장과 동일한 결과를 출력한다.

```
x[x > 7]
```

```
##   c   e
## 7.1 7.5
```

상기 문장을 분해하면, 첫째로 `x>7`을 평가해서, 논리벡터 `c(FALSE, FALSE, TRUE, FALSE, TRUE)`을 만들어내고, `TRUE` 값에 대응되는 벡터 `x` 원소를 추출하게 된다:

`==`을 사용해서 명칭으로 색인을 두어 추출하는 방식을 모사할 수 있다 (비교로 = 대신에 ==을 사용함을 기억한다):

```
x[names(x) == "a"]
```

```
##   a
## 5.4
```

꿀팁: 논리 조건 조합

We often want to combine multiple logical criteria. For example, we might want to find all the countries that are located in Asia or Europe **and** have life expectancies within a certain range. Several operations for combining logical vectors exist in R:

- `&`, the “logical AND” operator: returns `TRUE` if both the left and right are `TRUE`.
- `|`, the “logical OR” operator: returns `TRUE`, if either the left or right (or both) are `TRUE`.

You may sometimes see `&&` and `||` instead of `&` and `|`. These two-character operators only look at the first element of each vector and ignore the remaining elements. In general you should not use the two-character operators in data analysis; save them for programming, i.e. deciding whether to execute a statement.

- `!`, the “logical NOT” operator: converts `TRUE` to `FALSE` and `FALSE` to `TRUE`. It can negate a single logical condition (eg `!TRUE` becomes `FALSE`), or a whole vector of conditions (eg `!c(TRUE, FALSE)` becomes `c(FALSE, TRUE)`).

Additionally, you can compare the elements within a single vector using the `all` function (which returns TRUE if every element of the vector is TRUE) and the `any` function (which returns TRUE if one or more elements of the vector are TRUE). {`: .callout`}

26.6 도전과제 2

코드가 다음과 같이 주어져 있다:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

```
##   a   b   c   d   e
## 5.4 6.2 7.1 4.8 7.5
```

`x` 값에서 4보다 크고, 7보다 작은 것을 추출하는 코드를 작성해보자.

도전과제 2에 대한 해답

```
x_subset <- x[x<7 & x>4]
print(x_subset)
```
`::: {#subset-nonunique .rmdcaution}
** : (Non-unique names)**

(
 , --- , .
R
 , --- , .
)`
```

```
x <- 1:3
x
names(x) <- c('a', 'a', 'a')
x
x['a'] # only returns first value
x[names(x) == 'a'] # returns all three values
```
`:::
```

```
::: {#subset-operator-help .rmdcaution}
```

```
** : **
```

```
:  
`help("%in%")`    `?"%in%"` .  
:  
##          {#subset-skip-with-names}  
:  
R      ,       :  
x <- c(a=5.4, b=6.2, c=7.1, d=4.8, e=7.5) # we start again by naming a vector 'on the fly'  
x[!-a]  
``  
 , `!=`           :  
## Error: 0  
x[names(x) != "a"]  
``  
      , `~"a"`, `~"c"`,       ,       :  
x[names(x) != c("a", "c")]  
``  
R      ,       ,       ( `~"c"`,       )!  
`!=` ?  
### (Recycling) {#subset-recycle}  
:  
names(x) != c("a", "c")  
``  
`names(x)[3] != "c"`, `false`, R      `FALSE`, ?  
`!=`, R      .  
?  
![ ](assets/images/r/06-rmd-inequality.1.png)  
, * (recycle)* :  
:
```

```

![:     ](assets/images/r/06-rmd-inequality.2.png)

      , R `names(x)`      `c("a", "c")` ** * .
, `c("a", "c", "a", "c", "a")` . .
`"a"` `names(x)`      !=` `TRUE` .
,      5      2      ,
R .
`names(x)`      , R * * .
,
!
R      ( )      `%in%` .
`%in%`      `x`      "      ?" .
`!`      "in"      "not in"      :
x[! names(x) %in% c("a", "c") ]
```

3 {#r-subset-challenge-three}

 , gapminder `country` `continent` .
.
.
.
`TRUE` , `FALSE` .
?
.

seAsia <- c("Myanmar", "Thailand", "Cambodia", "Vietnam", "Laos")
read in the gapminder data that we downloaded in episode 2
gapminder <- read.csv("data/gapminder_data.csv", header=TRUE)
extract the `country` column from a data frame (we'll see this later);
convert from a factor to a character;
and get just the non-repeated elements
countries <- unique(as.character(gapminder$country))
```

`==` ;
`==` | ;
`%in%` .

(
) .
.

> ** 3 ** 
>
> - `countries==seAsia` * * .
>
```

```
>     `In countries == seAsia : longer object length is not a multiple of shorter object length`  

>     `seAsia`      `country` .  

>  

> -     ** ** ( , ) .  

>
```

```
(countries=="Myanmar" | countries=="Thailand" |  
countries=="Cambodia" | countries == "Vietnam" | countries=="Laos")
```

(혹은 `countries==seAsia[1] | countries==seAsia[2] | ...`). 이런 코딩방식은 올바른 정답을 주지만, 얼마나 어색한지 바로 알 수 있다. (만약 좀더 긴 목록에서 국가를 선택하게 되면 어떨까?)

- 이번 문제에 대한 최선의 방식은 `countries %in% seAsia`와 같이 코드를 작성하는 것이다. 정답이기도 하고 타이핑하기 쉽고 가독성도 높다.

26.7 특수값 처리하기

어느 지점에 다다르면, R 함수에 처리할 수 없는 결측값, 무한값, 정의되지 않는 값을 갖는 데이터와 마주하게 된다.

이런 유형의 데이터를 필터링하는데 사용되는 특수 함수가 있다:

- `is.na`는 벡터, 행렬, 데이터프레임에 포함된 NA 위치를 반환한다.
- 마찬가지로, `is.nan` 와 `is.infinite` 함수도 NaN 와 Inf 값에 대한 동일한 작업을 수행한다.
- `is.finite` 함수는 NA, NaN, Inf 값을 포함하지 않는 벡터, 행렬, 데이터프레임에 대한 모든 위치정보를 반환한다.
- `na.omit`은 벡터에서 모든 결측값을 필터링해서 제외시키다.

26.8 요인 부분집합 추출

지금까지 벡터 부분집합을 뽑아내는 다양한 방식을 탐색했다. 다른 자료구조에 대한 부분집합은 어떻게 뽑아낼 수 있을까?

요인 부분집합 뽑아내기는 벡터 부분집합 뽑아내기와 동일한 방식으로 동작한다.

```
f <- factor(c("a", "a", "b", "c", "c", "d"))  
f[f == "a"]  
  
## [1] a a  
## Levels: a b c d  
f[f %in% c("b", "c")]  
  
## [1] b c c
```

```
## Levels: a b c d
f[1:3]
```

```
## [1] a a b
## Levels: a b c d
```

중요한 주의점 하나는 건너뛰는 요소가 설사 해당 범주가 요인으로 존재하지 않더라도, 수준(level)을 제거하지 않는다는 점이다:

```
f[-3]
```

```
## [1] a a c c d
## Levels: a b c d
```

26.9 행렬 부분집합 추출

행렬의 경우도 [함수를 사용해서 부분집합을 뽑아낸다. 이번 경우에는 인자를 두개 사용한다: 첫번째 인자는 행에 적용되고, 두번째 인자는 칼럼에 적용된다:

```
set.seed(1)
m <- matrix(rnorm(6*4), ncol=4, nrow=6)
m[3:4, c(3,1)]
```

```
##          [,1]     [,2]
## [1,]  1.1249 -0.836
## [2,] -0.0449  1.595
```

첫번째 혹은 두번째 인자를 공백으로 남겨놓을 수도 있는데, 모든 행 혹은 칼럼을 각각 불러올 경우 사용한다:

```
m[, c(3,4)]
```

```
##          [,1]     [,2]
## [1,] -0.6212  0.8212
## [2,] -2.2147  0.5939
## [3,]  1.1249  0.9190
## [4,] -0.0449  0.7821
## [5,] -0.0162  0.0746
## [6,]  0.9438 -1.9894
```

행 혹은 칼럼 하나만 접근하고자 하면, R이 자동으로 결과값을 벡터로 전환시킨다:

```
m[3,]
```

```
## [1] -0.836  0.576  1.125  0.919
```

결과값을 행렬로 그대로 유지하고자 한다면, 세번째 인자를 명세할 필요가 있다: drop = FALSE:

```
m[3, , drop=FALSE]
```

```
##      [,1]  [,2]  [,3]  [,4]
## [1,] -0.836 0.576 1.12  0.919
```

벡터와 달리, 행렬 외부 행과 칼럼을 접근하고자 하면, R이 오류를 던진다:

```
m[, c(3,6)]
```

```
## Error in m[, c(3, 6)]:
```

꿀팁: 고차원 배열

다차원 배열을 다룰 때, [에 넘겨지는 각 인자가 차원에 대응된다. 예를 들어, 3D 배열에서 첫세개 인자는 각각 행, 열, 깊이 차원에 대응된다.

행렬을 보면 정말 자료형이 벡터라서, 단지 인자 하나로만 부분집합을 추출할 수도 있다:

```
m[5]
```

```
## [1] 0.33
```

보통 유용하지는 않다. 하지만, 행렬이 열우선형식(column-major format)으로 기본디폴트 설정으로 되어있음에 주목한다. 즉, 벡터 요소가 칼럼방향으로 배열된다는 것을 의미한다:

```
matrix(1:6, nrow=2, ncol=3)
```

```
##      [,1]  [,2]  [,3]
## [1,]     1     3     5
## [2,]     2     4     6
```

행렬을 행우선으로 쭉 펼치고자 한다면, byrow=TRUE를 사용한다:

```
matrix(1:6, nrow=2, ncol=3, byrow=TRUE)
```

```
##      [,1]  [,2]  [,3]
## [1,]     1     2     3
## [2,]     4     5     6
```

행과 칼럼 색인 대신에 행명칭(rownames)과 열명칭(column names)을 사용해서 배열 부분집합을 뽑아낼 수 있다.

26.10 도전과제 4

코드가 다음과 같이 주어져 있다:

```
m <- matrix(1:18, nrow=3, ncol=6)
print(m)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]     1     4     7    10    13    16
## [2,]     2     5     8    11    14    17
## [3,]     3     6     9    12    15    18
```

1. 다음 중 어떤 명령어가 값 11과 14를 추출하는 하는가?

- A. `m[2,4,2,5]` B. `m[2:5]` C. `m[4:5,2]` D. `m[2,c(4,5)]`

도전과제 4 해답

D

26.11 리스트 부분집합 추출

이제 몇가지 새로운 부분집합을 뽑아내는 연산자를 소개한다. 리스트 부분집합을 뽑아내는데 사용되는 함수가 세가지 있다: 원자벡터와 행렬에서 살펴본 `[`, 그리고 `[[`, `$`이 있다.

`[`을 사용하면, 항상 리스트만 반환한다. 리스트 부분집합을 뽑아내고자 하지만, 원소는 뽑아내고 싶지 않다면, 아마도 `[` 연산자를 사용할 것이다.

```
xlist <- list(a = "Software Carpentry", b = 1:10, data = head(iris))
xlist[1]
```

```
## $a
## [1] "Software Carpentry"
```

상기 명령어는 원소 하나만 갖는 리스트를 반환한다.

`[` 연산자를 사용해서 원자벡터에 적용한 그대로 리스트 원소를 부분집합으로 뽑아낼 수 있다. 하지만, 리스트가 재귀적으로 되어 있지 않다면, 비교 연산자는 동작하지 않는다. 이유는 비교 연산자가 데이터 구조 내부 개별 요소가 아닌, 리스트 각 요소에 내재한 자료구조로 되어있기 때문이다.

```
xlist[1:2]
```

```
## $a
## [1] "Software Carpentry"
##
## $b
## [1] 1 2 3 4 5 6 7 8 9 10
```

리스트 개별 원소를 추출하려면, 이중 꺠쇠 함수를 사용한다: `[[`.

```
xlist[[1]]
```

```
## [1] "Software Carpentry"
```

이제 결과값이 리스트가 아닌 벡터에 주목한다.

한번에 요소 하나이상을 추출할 수는 없다:

```
xlist[[1:2]]  
  
## Error in xlist[[1:2]]:  
요소를 건너뛰는 것도 사용할 수 없다:  
xlist[[-1]]  
  
## Error in xlist[[-1]]: invalid negative subscript in get1index <real>  
하지만, 명칭을 사용해서 요소에 대한 부분집합으로 뽑아내거나, 요소를 추출할 때  
사용할 수 있다:  
xlist[["a"]]  
  
## [1] "Software Carpentry"  
$ 함수는 명칭으로 요소를 뽑아내는데 사용되는 초간편 방법이다:  
xlist$data  
  
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1         5.1        3.5       1.4        0.2  setosa  
## 2         4.9        3.0       1.4        0.2  setosa  
## 3         4.7        3.2       1.3        0.2  setosa  
## 4         4.6        3.1       1.5        0.2  setosa  
## 5         5.0        3.6       1.4        0.2  setosa  
## 6         5.4        3.9       1.7        0.4  setosa
```

26.12 도전과제 5

리스트가 다음과 같이 주어져 있다:

```
xlist <- list(a = "Software Carpentry", b = 1:10, data = head(iris))
```

리스트와 벡터 부분집합을 추출하는 지식을 활용해서, xlist에서 숫자 2를 추출한다.
힌트: 숫자 2는 리스트 “b” 항목 내부에 담겨있다.

도전과제 5 해답

```
xlist$b[2]
```

```
## [1] 2
```

```
xlist[[2]][2]
```

```
## [1] 2
```

```
xlist[["b"]][2]
```

```
## [1] 2
```

26.13 도전과제 6

선형 모형이 다음과 같이 주어져 있다:

```
mod <- aov(pop ~ lifeExp, data=gapminder)
```

잔차 자유도를 추출하라. 힌트: `attributes()` 함수가 도움을 줄 것이다.

도전과제 6에 대한 해법

```
attributes(mod) ## `df.residual` is one of the names of `mod`  
mod$df.residual
```

26.14 데이터프레임

데이터프레임을 보면 내부는 리스트로 구성된 것이라는 점을 기억한다. 그래서 유사한 규칙이 적용된다. 하지만, 데이터프레임도 2차원 객체다:

[함수에 인자를 하나만 넣으면 리스트와 동일하게 동작한다. 즉, 각 리스트 요소는 칼럼에 대응된다. 작업 결과 나오는 객체는 데이터프레임이다:]

```
head(gapminder[3])
```

```
##          pop  
## 1 8425333  
## 2 9240934  
## 3 10267083  
## 4 11537966  
## 5 13079460  
## 6 14880372
```

유사하게, [[함수는 칼럼 한개만 추출하는데 동작된다:]

```
head(gapminder[["lifeExp"]])
```

```
## [1] 28.8 30.3 32.0 34.0 36.1 38.4
```

명칭으로 칼럼을 추출하는데 사용되는 편리한 단축어가 \$이다:]

```
head(gapminder$year)
```

```
## [1] 1952 1957 1962 1967 1972 1977
```

인자가 두개 있는 경우, [함수는 행렬에 대해서와 마찬가지로 동작한다:]

```
gapminder[1:3,]
```

	country	year	pop	continent	lifeExp	gdpPercap
## 1	Afghanistan	1952	8425333	Asia	28.8	779
## 2	Afghanistan	1957	9240934	Asia	30.3	821

```
## 3 Afghanistan 1962 10267083      Asia      32.0      853
```

행 하나만 부분집합으로 뽑아내면, 결과는 데이터프레임이 되는데 이유는 각 요소가 혼합된 자료형으로 구성되었기 때문이다:

```
gapminder[3,]
```

```
##       country year      pop continent lifeExp gdpPercap
## 3 Afghanistan 1962 10267083      Asia      32      853
```

하지만, 단일 칼럼에 대해서 결과는 벡터다. `drop = FALSE`를 세번째 인자로 넣으면 바꿀 수 있다.

26.15 도전과제 7

데이터프레임 부분집합을 뽑아내는 오류가 다음에 나와 있는데 이를 버그없이 수정하라:

1. 1957년에 수집된 관측점을 뽑아내라.

```
gapminder[gapminder$year == 1957,]
```

2. 1에서 4를 제외한 모든 칼럼을 뽑아내라.

```
gapminder[,-1:4]
```

3. 기대수명이 80세 이상 되는 행을 추출하라.

```
gapminder[gapminder$lifeExp > 80]
```

4. 첫번째 행과 4번째 5번째 칼럼(`lifeExp`, `gdpPercap`)을 뽑아내라.

```
gapminder[1, 4, 5]
```

5. 고급: 2002년과 2007년에 대한 정보를 담고 있는 행을 추출하라.

```
gapminder[gapminder$year == 2002 | 2007,]
```

도전과제 7에 대한 해법

데이터프레임 부분집합을 뽑아내는 오류가 다음에 나와 있는데 이를 버그없이 수정하라:

1. 1957년에 수집된 관측점을 뽑아내라.

```
# gapminder[gapminder$year == 1957,]
gapminder[gapminder$year == 1957,]
```

2. 1에서 4를 제외한 모든 칼럼을 뽑아내라.

```
# gapminder[,-1:4]
gapminder[,-c(1:4)]
```

3. 기대수명이 80세 이상 되는 행을 추출하라.

```
# gapminder[gapminder$lifeExp > 80]
gapminder[gapminder$lifeExp > 80,]
```

4. 첫번째 행과 4번째 5번째 칼럼(lifeExp, gdpPercap)을 뽑아내라.
(continent and lifeExp).

```
# gapminder[1, 4, 5]
gapminder[1, c(4, 5)]
```

5. 고급: 2002년과 2007년에 대한 정보를 담고 있는 행을 추출하라.

```
# gapminder[gapminder$year == 2002 | 2007,]
gapminder[gapminder$year == 2002 | gapminder$year == 2007,]
gapminder[gapminder$year %in% c(2002, 2007),]
```

26.16 도전과제 8

1. gapminder[1:20] 명령어는 왜 오류를 반환하는가? gapminder[1:20,]와 어떻게 다른가?
2. gapminder_small이라는 데이터프레임을 생성하는데 1에서 9까지 행과 19에서 23까지 행만 포함한다. 이 작업을 하나 혹은 두 단계로 작성한다.

도전과제 8에 대한 해답

1. gapminder는 데이터프레임이라, 이차원에서 부분집합을 추출할 필요가 있다. gapminder[1:20,] 명령어는 첫 20행과 모든 칼럼을 추출한다.

2.

```
gapminder_small <- gapminder[c(1:9, 19:23),]
```

Chapter 27

제어 흐름

종종 코딩할 때, 동작 흐름을 제어하고 싶을 때가 있다. 한 조건 혹은 조건 집합이 만족될 때만 동작이 일어나게 설정함으로써 이런 작업을 수행한다. 또 다른 방식으로, 특정 횟수만큼 동작이 일어나도록 설정할 수도 있다.

R에서 흐름을 제어하는 방식이 몇 가지 있다. 조건문에 대해서, 가장 흔히 사용되는 접근법이 루프 구성체(loop construct)다:

```
# if
if (condition is true) {
  perform action
}

# if ... else
if (condition is true) {
  perform action
} else { # that is, if the condition is false,
  perform alternative action
}
```

예를 들어, 변수 x가 특정값을 갖게 되면, 메시지를 출력하게 R에게 지시할 수도 있다:

```
x <- 8

if (x >= 10) {
  print("x is greater than or equal to 10")
}

x
```

```
## [1] 8
```

콘솔에 print 출력문이 아무것도 출력시키지 않는데, 이유는 x가 10보다 크지 않기 때문이다.

10보다 작은 값에 대해 메시지를 출력시키려면, else문을 추가시키면 된다.

```
x <- 8

if (x >= 10) {
  print("x is greater than or equal to 10")
} else {
  print("x is less than 10")
}
```

```
## [1] "x is less than 10"
```

else if 를 사용하면 다수 조건을 테스트할 수도 있다.

```
x <- 8

if (x >= 10) {
  print("x is greater than or equal to 10")
} else if (x > 5) {
  print("x is greater than 5, but less than 10")
} else {
  print("x is less than 5")
}
```

```
## [1] "x is greater than 5, but less than 10"
```

중요: R이 if()문을 내부 조건을 평가할 때, 논리 요소 즉, TRUE 혹은 FALSE를 찾는다. 초심자에게 있어 이런 점이 두통을 유발할 수도 있다. 예를 들어:

```
x <- 4 == 3
if (x) {
  "4 equals 3"
} else {
  "4 does not equal 3"
}
```

```
## [1] "4 does not equal 3"
```

x 벡터가 FALSE라서 not equal 메시지가 출력된다.

```
x <- 4 == 3
x
```

```
## [1] FALSE
```

27.1 도전과제 1

`if` 문을 사용해서 `gapminder` 데이터셋에서 2002년부터 어떤 레코드가 있는지 확인한 결과를 적절한 메시지로 출력하게 하자. 2012년에 대해서도 동일한 작업을 수행한다.

도전과제 1에 대한 해답

먼저 도전과제 1에 대한 해답을 살펴보자. `any()` 함수를 사용하지는 않는다. `gapminder$year` 벡터의 원소가 2002와 같은지 확인값을 갖는 논리 벡터를 생성시킨다:

```
gapminder[(gapminder$year == 2002),]
```

그리고 나서, 2002년과 대응되는 `gapminder` 데이터프레임 행의 숫자를 센다:

```
rows2002_number <- nrow(gapminder[(gapminder$year == 2002),])
```

2002년에 대한 레코드 존재는 `rows2002_number` 숫자가 하나 혹은 그 이상인지 확인하는 것과 같다.

```
rows2002_number >= 1
```

함께 담으면 다음과 같이 정리된다:

```
if(nrow(gapminder[(gapminder$year == 2002),]) >= 1){
  print("Record(s) for the year 2002 found.")
}
```

`any()` 함수로 상기 작업은 신속히 처리될 수 있다. 논리 조건을 다음과 같이 표현하여 작성할 수 있다:

```
if(any(gapminder$year == 2002)){
  print("Record(s) for the year 2002 found.")
}
```

다음과 같은 경고 메시지가 나오는 사람이 있나요?

작성한 조건문이 하나 이상 논리 요소를 갖는 벡터를 평가하게 되면, `if` 함수는 쭉 실행되지만, 첫번째 요소에 대한 조건만 평가한다. 따라서, 조건문이 길이 1이 되도록 확실히 할 필요가 있다.

꿀팁: `any()` 와 `all()` 함수

`any()` 함수는 벡터에 적어도 값 하나가 `TRUE` 가 되어야만 `TRUE` 값을 반환한다. 그렇지 않은 경우 `FALSE` 값을 반환한다. 이런 점은 `%in%` 연산자에 유사한 방식으로 적용된다. `all()` 함수는 명칭에서 나타나듯이, 벡터에 모든 값이 `TRUE` 인 경우에만 `TRUE` 값을 반환한다.

27.2 연산 반복

값을 담은 집합에 반복 작업을 수행할 때, 반복 순서가 중요하고, 집합에 속한 원소 각각에 대해 동일한 연산을 수행하는 경우, `for()` 루프가 제격이다. 앞선 쉘 수업에서 `for()` 루프를 살펴봤다. `for()` 루프는 가장 유연하게 루프를 돌리는 연산이지만, 유연성으로 인해 올바르게 사용하기도 가장 어렵다. 반복작업 순서가 중요하지 않은 경우, `for()` 루프 사용을 회피한다: 즉, 반복할 때마다 연산작업이 이전 반복작업 결과에 의존하는 경우.

`for()` 루프에 대한 기본 구조는 다음과 같다:

```
for(iterator in set of values){
    do a thing
}
```

예를 들어:

```
for(i in 1:10){
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

`1:10`이 즉석에서 벡터를 생성된다; 물론 다른 벡터도 반복시킬 수 있다.

두개 이상을 한번에 반복할 경우, 또 다른 `for()` 루프를 `for()` 루프내부에 중첩시킬 수도 있다.

```
for(i in 1:5){
    for(j in c('a', 'b', 'c', 'd', 'e')){
        print(paste(i,j))
    }
}

## [1] "1 a"
## [1] "1 b"
## [1] "1 c"
## [1] "1 d"
## [1] "1 e"
## [1] "2 a"
```

```
## [1] "2 b"
## [1] "2 c"
## [1] "2 d"
## [1] "2 e"
## [1] "3 a"
## [1] "3 b"
## [1] "3 c"
## [1] "3 d"
## [1] "3 e"
## [1] "4 a"
## [1] "4 b"
## [1] "4 c"
## [1] "4 d"
## [1] "4 e"
## [1] "5 a"
## [1] "5 b"
## [1] "5 c"
## [1] "5 d"
## [1] "5 e"
```

결과를 바로 출력하는 대신에, 루프 출력결과를 새로운 객체에 대입시킬 수도 있다.

```
output_vector <- c()
for(i in 1:5){
  for(j in c('a', 'b', 'c', 'd', 'e')){
    temp_output <- paste(i, j)
    output_vector <- c(output_vector, temp_output)
  }
}
output_vector
```

```
## [1] "1 a" "1 b" "1 c" "1 d" "1 e" "2 a" "2 b" "2 c" "2 d" "2 e" "3 a" "3 b"
## [13] "3 c" "3 d" "3 e" "4 a" "4 b" "4 c" "4 d" "4 e" "5 a" "5 b" "5 c" "5 d"
## [25] "5 e"
```

이러한 접근법은 유용할 수도 있지만, ‘실행결과를 키워나감’ (실행결과 객체를 점진적으로 키워나감) 이런 전략은 컴퓨터 계산 측면으로 보면 비효율적이다. 그래서, 많은 값을 반복할 때는 회피한다.

꿀팁: 실행결과를 키워나가지 말라!

초보자나 경험 많은 R 사용자 모두 저지르는 가장 큰 실수 중 하나가 실행결과 객체(벡터, 리스트, 행렬, 데이터프레임)를 루프를 돌리면서 키워나가는 것이다. 컴퓨터는 이런 것을 처리하는데 매우 좋지 못하다. 그래서 연산 속도가 급격히 늦어질 수 있다. 미리 적절한 차원을 갖는 빈 연산결과 저장 객체를 정의하는 것이 훨씬 낫다. 그래서, 상기처럼 실행결과를 저장할 행렬을 미리 알고 있다면, 5 칼럼과 5 열로 구성된 빈 행렬를 생성하고 나서, 매번 반복을 돌릴 때마다 적절한 위치에 실행결과를 저장한다.

- 고성능 R코드 작성과 성능비교

더 좋은 방식은 값을 채워넣기 전에 (빈) 출력결과를 저장할 객체를 정의하는 것이다. 이번 예제의 경우, 더 복잡해 보이지만, 훨씬 더 효율적이다.

```
output_matrix <- matrix(nrow=5, ncol=5)
j_vector <- c('a', 'b', 'c', 'd', 'e')
for(i in 1:5){
  for(j in 1:5){
    temp_j_value <- j_vector[j]
    temp_output <- paste(i, temp_j_value)
    output_matrix[i, j] <- temp_output
  }
}
output_vector2 <- as.vector(output_matrix)
output_vector2

## [1] "1 a" "2 a" "3 a" "4 a" "5 a" "1 b" "2 b" "3 b" "4 b" "5 b" "1 c" "2 c"
## [13] "3 c" "4 c" "5 c" "1 d" "2 d" "3 d" "4 d" "5 d" "1 e" "2 e" "3 e" "4 e"
## [25] "5 e"
```

꿀팁: While 루프

때로는 특정 조건이 만족될 때까지 연산작업을 반복할 필요도 있다. 이런 작업은 `while()` 루프로 수행한다.

```
while(this condition is true){
  do a thing
}
```

예제로, 0.1 보다 작은 난수가 나올 때까지 0과 1 사이 균등 분포(`runif()` 함수)에서 난수를 뽑아내는 `while` 루프코드가 다음에 나와 있다.

```
z <- 1
while(z > 0.1){
  z <- runif(1)
  print(z)
}
```

`while()` 루프가 항상 적절한 것은 아니다. 조건이 절대 만족되지 않는 경우가 있어, 무한 루프에 빠지지 않도록 특별히 유의해야 된다.

27.3 도전과제 2

`output_vector` 와 `output_vector2` 객체를 비교하라. 두 객체는 동일한가? 만약 동일하지 않다면, 왜 동일하지 않을까? 코드 마지막 블록을 변경해서 어떻게 `output_vector2` 를 `output_vector` 와 같도록 만들 수 있을까?

도전과제 2에 대한 해답

`all()` 함수를 사용해서 두 벡터가 동일한지 검사할 수 있다.

```
all(output_vector == output_vector2)
```

하지만, `output_vector` 벡터 모든 요소가 `output_vector2`에 있기도 하다:

```
all(output_vector %in% output_vector2)
```

바꿔서 검사해도 마찬가지다:

```
all(output_vector2 %in% output_vector)
```

따라서 `output_vector` 와 `output_vector2` 벡터 원소가 단지 다른 순서로 정렬된 사실을 확인하게 된다. 원소 배치가 틀어진 이유는 `as.vector()` 함수 연산이 입력행렬을 칼럼을 기준으로 작업했기 때문이다. `output_matrix` 행렬을 살펴보면, 행기준으로 원소를 배열시키면 됨을 알 수 있다. 해법은 `output_matrix` 입력 행렬을 전치시키는 것이다. `t()` 함수 혹은 입력 원소 순서를 밖는 것을 통해서 원하는 작업을 완수할 수 있다.

첫번째 해법은 원본 행렬을

```
output_vector2 <- as.vector(output_matrix)
```

다음과 같이 바꾸는 것이다.

```
output_vector2 <- as.vector(t(output_matrix))
```

두번째 해법은 다음과 같이 행렬 원소 위치를

```
output_matrix[i, j] <- temp_output
```

다음과 같이 바꾸는 것이다.

```
output_matrix[j, i] <- temp_output
```

27.4 도전과제 3

gapminder 데이터 루프를 대륙별로 돌리는 스크립트를 작성한다. 그리고 나서 평균 기대수명이 50년 보다 길거나 짧은지 결과를 출력한다.

도전과제 3에 대한 해답

1 단계: 대륙 벡터(`continent`)에서 유일무이한 값을 모두 추출한다.

```
gapminder <- read.csv("data/gapminder_data.csv")
unique(gapminder$continent)
```

2 단계: 각 대륙별로 루프를 돌리는데 gapminder 데이터에 `subset()` 함수로 각 대륙별 부분 데이터셋을 추출하고 평균 기대수명을 계산한다. 상기 작업과정을 다음과 같이 구현할 수 있다.

1. 'continent' 유일무이한 값 각각에 대해 루프를 돌린다.
2. 각 대륙별 부분 데이터셋별로 계산된 기대수명값을 임시 변수에 저장시킨다.(tmp)
3. 평균 기대수명을 계산해서 출력결과를 화면에 뿌려주는 형태로 사용자에게 반환시킨다:

```
for( iContinent in unique(gapminder$continent) ){
  tmp <- mean(subset(gapminder, continent==iContinent)$lifeExp)
  cat("Average Life Expectancy in", iContinent, "is", tmp, "\n")
  rm(tmp)
}
```

3 단계: 평균 기대수명이 50보다 적거나 큰 경우만 결과를 출력시킨다.
 이런 경우, 결과를 출력시키기 전에 if 조건을 추가한다. if 조건을 출력전에 추가해서 산출된 평균 기대수명이 기준치보다 높거나 낮은지를 평가하고 해당 조건에 맞춰 결과를 출력시킨다. 앞서 작성한 코드에 (3)을 반영하여 코드를 수정한다:

3a. 계산된 평균 기대수명 기준치(50년)보다 적은 경우 평균 기대수명이 기준치보다 낮은 대륙을 반환시킨다. 그렇지 않은 경우도, 평균 기대수명이 기준치보다 높은 대륙을 반환시키는 코드를 작성한다;

```
thresholdValue <- 50

for( iContinent in unique(gapminder$continent) ){
  tmp <- mean(subset(gapminder, continent==iContinent)$lifeExp)

  if(tmp < thresholdValue){
    cat("Average Life Expectancy in", iContinent, "is less than", thresholdValue)
  }
  else{
    cat("Average Life Expectancy in", iContinent, "is greater than", thresholdValue)
  } # end if else condition
  rm(tmp)
} # end for loop
```

27.5 도전과제 4

도전과제 3에서 나온 스크립트를 변경해서 각 국각별로 루프를 돌린다. 이번에는 예상수명이 50년보다 짧은지, 50년에서 70년 사이, 70년 이상인지 결과를 출력한다.

도전과제 4에 대한 해답

도전과제 해답을 변경하여 lowerThreshold, upperThreshold 두개 기준값을 추가시키고, if-else문을 확장시킨다.

```

lowerThreshold <- 50
upperThreshold <- 70

for( iCountry in unique(gapminder$country) ){
  tmp <- mean(subset(gapminder, country==iCountry)$lifeExp)

  if(tmp < lowerThreshold){
    cat("Average Life Expectancy in", iCountry, "is less than", lowerThreshold, "\n")
  }
  else if(tmp > lowerThreshold && tmp < upperThreshold){
    cat("Average Life Expectancy in", iCountry, "is between", lowerThreshold, "and", upp
  }
  else{
    cat("Average Life Expectancy in", iCountry, "is greater than", upperThreshold, "\n")
  }
  rm(tmp)
}

```

27.6 도전과제 5 - 고급

gapminder 데이터셋에서 각 국가별로 루프를 돌리는 스크립트를 작성한다. 국가명이 'B'로 시작하는지 테스트하고, 만약 평균 기대수명이 50년보다 작은 경우 선그래프로 시간에 대해 기대수명을 그래프를 그린다.

도전과제 5에 대한 해답

유닉스 쉘 학습에서 소개된 grep 명령어를 사용해서 "B"로 시작하는 국가를 찾는다. 먼저 "B"로 시작하는 국가를 찾는 방법을 이해하자. 유닉스 쉘 학습에 소개된 내용을 따라 다음과 같이 작성하고 싶을 것이다.

```
```r
grep("^B", unique(gapminder$country))
```

하지만, 상기 명령어를 평가하게 되면, "B"으로 시작되는 요인변수 country의 인덱스만 반환시킨다. 해당 값을 얻으려면, grep 명령어 내부에 value=TRUE 옵션을 추가시키면 된다:

```
grep("^B", unique(gapminder$country), value=TRUE)
```

candidateCountries 변수에 "B"로 시작되는 국가를 저장시킨다. 그리고 나서, 변수에 포함된 원소 각각을 루프 돌린다. 루프 내부에, 각 국가별로 평균 기대수명을 평가한다. 만약 평균 기대수명이 50보다 낮은 경우 연도별 평균 기대수명이 변하는 것을 Base 플롯으로 시각화한다:

```
thresholdValue <- 50
candidateCountries <- grep("^B", unique(gapminder$country), value=TRUE)

for(iCountry in candidateCountries){
 tmp <- mean(subset(gapminder, country==iCountry)$lifeExp)

 if(tmp < thresholdValue){
 cat("Average Life Expectancy in", iCountry, "is less than", thresholdValue)

 with(subset(gapminder, country==iCountry),
 plot(year,lifeExp,
 type="o",
 main = paste("Life Expectancy in", iCountry, "over time"),
 ylab = "Life Expectancy",
 xlab = "Year"
) # end plot
) # end with
 } # end for loop
 rm(tmp)
}

````
```

Chapter 28

논문 품질 그래프 생성

데이터를 도식화하는 것이 변수간 다양한 관계를 재빨리 탐색하는 최상의 방식 중 하나다.

R에는 세가지 주류 도식화 시스템이 존재한다: Base 도식화 시스템, lattice 팩키지, ggplot2 팩키지.

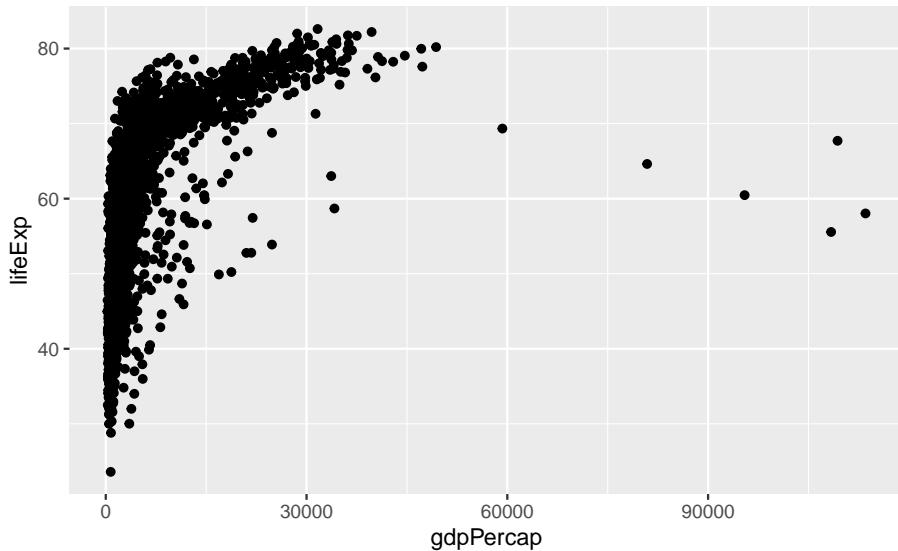
금일, ggplot2 팩키지를 학습할 것인데 이유는 논문 품질 그래프를 생성하는데 가장 효과적이기 때문이다.

ggplot2는 그래픽 문법(grammar of graphics)에 기반했다. 즉, 어떤 그래프도 동일한 구성요소 집합으로 표현된다: 데이터셋, 좌표 시스템, geoms 집합 - 데이터 점에 대한 시각적 표현.

ggplot2를 이해하는 핵심은 그림을 계층으로 사고하는 것이다: 포토샵(Photoshop), 일러스트레이터(Illustrator), 잉크스케이프(inkscape.) 같은 이미지 편집 프로그램으로 작업하는 것과 같다.

예제를 가지고 시작해본다:

```
library("ggplot2")
ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp)) +
  geom_point()
```

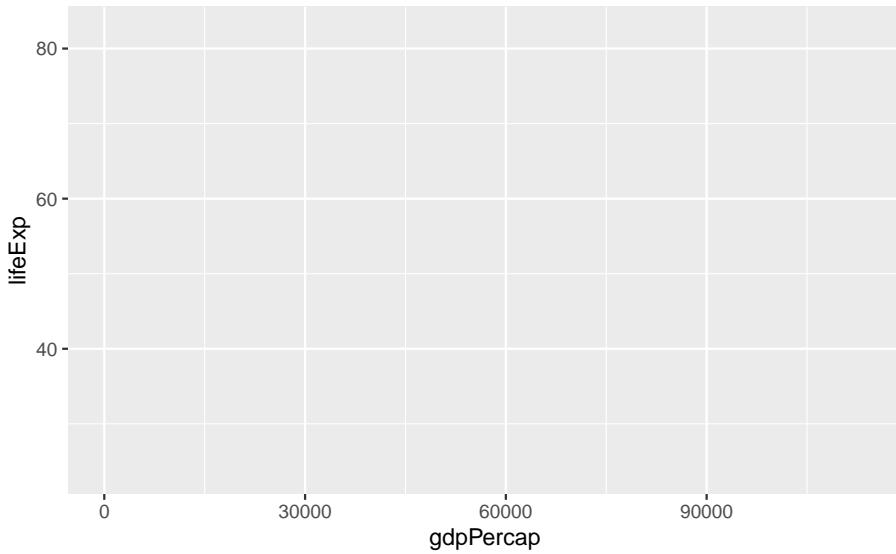


그래서, 처음으로 수행하는 작업은 `ggplot` 함수를 호출하는 것이다. 이 함수가 R에게 새로운 그림을 생성하고, `ggplot` 함수에 전달하는 어떤 인자도 해당 그림에 전역 선택옵션(그림에 있는 모든 계층에 적용)임을 전달한다.

`ggplot`에 인자를 두개 전달했다. 먼저, `ggplot`에 그림에 사용할 데이터가 무엇인지 전달한다; 번 예제에서 앞에서 불러온 gapminder데이터. 두번째 인자를 `aes`함수에 전달했는데, `ggplot`에게 데이터에 나온 변수를 도식화하는 그림의 미학적인 속성에 매핑하는 방법을 전달한다; 이번 경우에는 x와 y 위치. 여기서 `ggplot`에 gapminder데이터프레임 “`lifeExp`” 칼럼을 x-축에, “`gdpPercap`” 칼럼을 y-축에 도식화한다. 명시적으로 `aes`에 칼럼명을 전달(예를 들어, `x = gapminder[, “lifeExp”]`) . `ggplot` 함수가 데이터에 존재하는 칼럼을 식별할만큼 똑똑하기 때문이다!

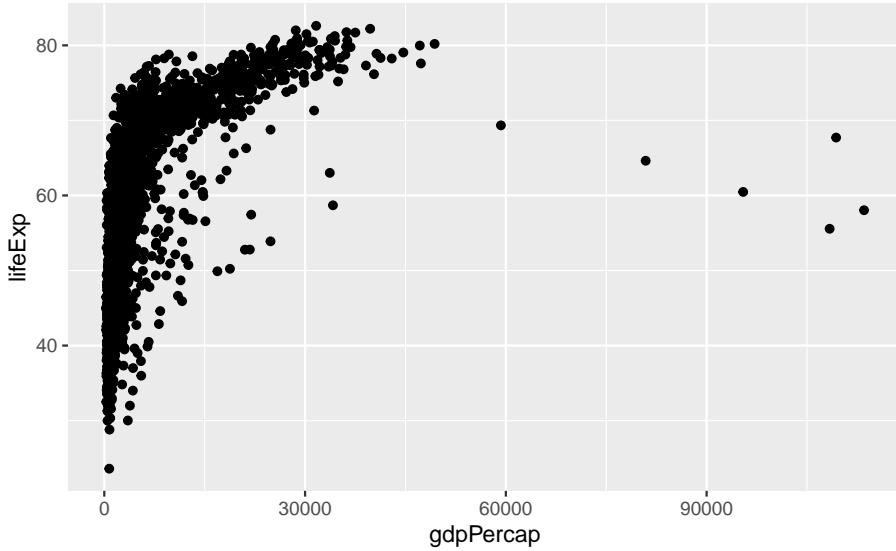
그 자체로, `ggplot`함수를 호출한다고 도식화가 바로 되는 것은 아니다:

```
ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp))
```



`ggplot` 함수에 데이터를 시각적으로 표현하는 방법을 전달할 필요가 있다. `geom` 계층을 추가해서 작업이 수행된다. 본 사례에서, `geom_point`를 사용했다; x와 y 사이 관계를 시각적으로 산점도 형태로 표현하도록 `ggplot`에게 전달한다:

```
ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp)) +
  geom_point()
```



28.1 도전과제 1

상기예제를 변경해서, 기대수명이 시간에 따라 어떻게 변해왔는지 시각화하는 그림을 생성한다:

```
ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp)) + geom_point()
```

힌트: gapminder 데이터셋에 “year”라는 칼럼이 있는데, x-축에 나타나야 된다.

도전과제 1에 대한 해답

한가지 해법은 다음과 같다:

```
ggplot(data = gapminder, aes(x = year, y = lifeExp)) + geom_point()
```



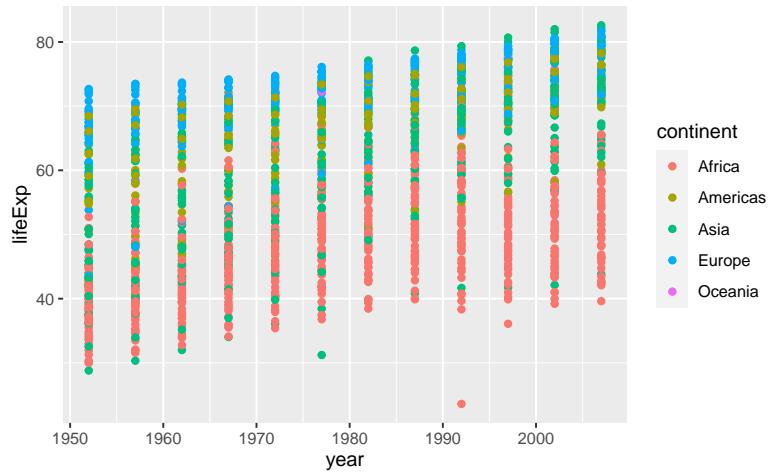
28.2 도전과제 2

이전 예제와 도전과제에서, `aes` 함수를 사용해서 `geom` 산점도로 x 와 y 지점을 각 점에 대해 표현했다. 변경할 수 있는 또다른 미학적 속성은 각 점에 대한 색깔이다. 앞선 도전과제 코드를 변경해서 “continent” 대륙별로 각 점에 색을 입힌다. 데이터에서 어떤 경향성을 볼 수 있는가? 예상했던 경향성인가?

도전과제 2에 대한 해답

앞선 예제와 도전과제에서 x 와 y의 각 점에 대한 좌표로 `geom` 산점도를 `aes` 함수에 사용했다. `aesthetic`의 또다른 특성으로 점에 대한 색상을 변경할 수 있다는 점이다. “continent” 칼럼을 색상(`color`)에 적용해서 이전 도전과제 코드에 접목시켜보자. 데이터에서 어떤 추세를 볼 수 있는가? 추세가 예상했던 것인가?

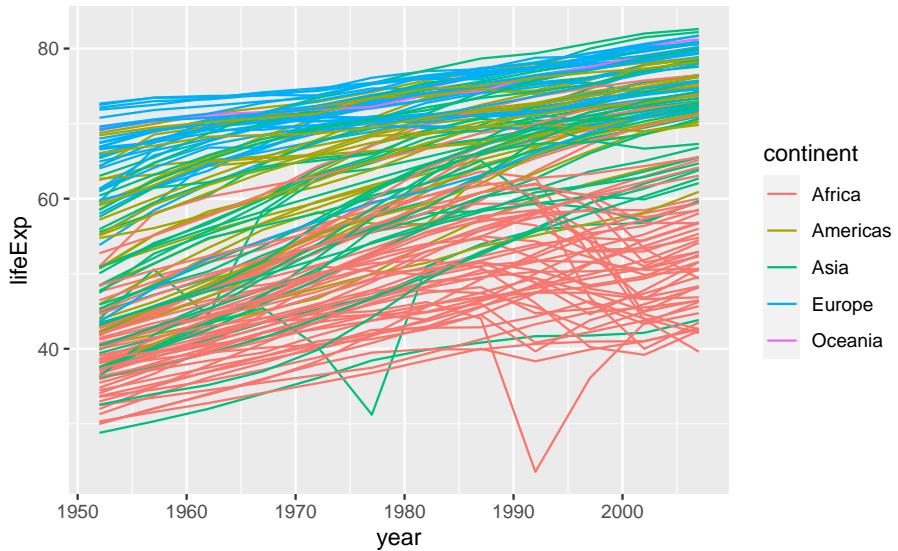
```
ggplot(data = gapminder, aes(x = year, y = lifeExp, color=continent)) +
  geom_point()
```



28.3 계층(Layers)

산점도가 아마도 시간에 따라 변하는 정보를 시각화하는데 최선은 아니다. 대신에, ggplot에 선그래프(line plot)로 데이터를 시각화한다:

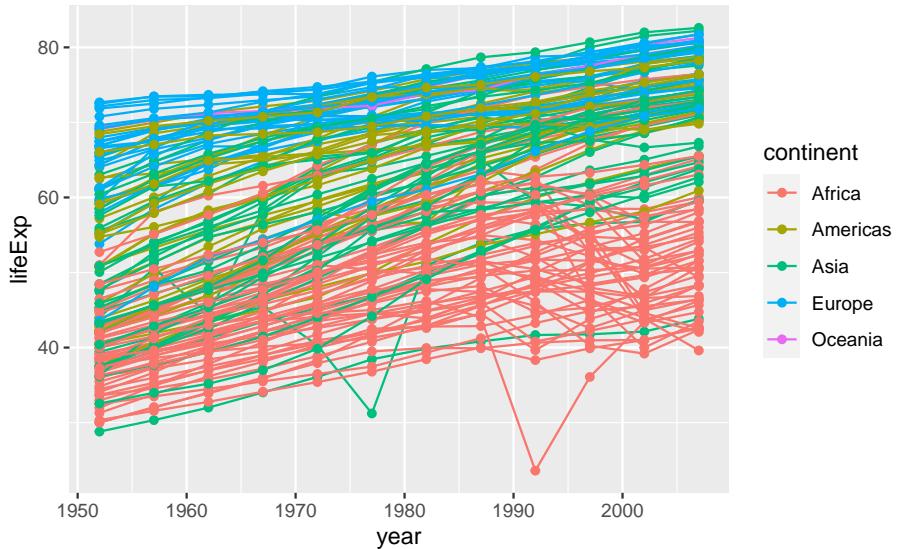
```
ggplot(data = gapminder, aes(x=year, y=lifeExp, by=country, color=continent)) +
  geom_line()
```



`geom_point` 계층을 추가하는 대신에, `geom_line` 계층을 추가했다. `aes`로 `by`를 추가해서, `ggplot`이 각 국가를 직선으로 연결해서 도식화한다.

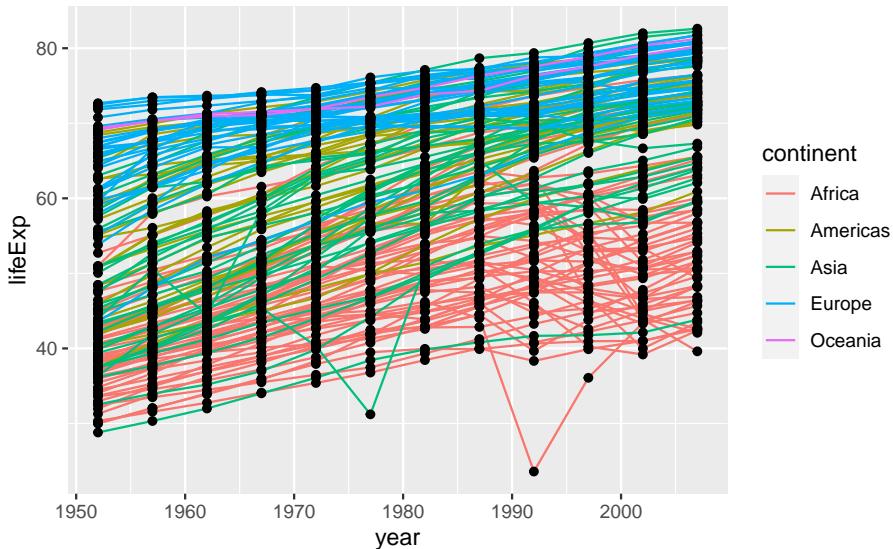
하지만, 직선과 점을 함께 시각화하려고 하면 어떨까? 단순히, 또 다른 계층을 그림에 추가하면 된다:

```
ggplot(data = gapminder, aes(x=year, y=lifeExp, by=country, color=continent)) +
  geom_line() + geom_point()
```



각 계층은 이전 계층 위에 도식화됨에 주목한다. 이번 예제에서, 점이 직선 위에 도식화되었다. 다음에 도식화한 산출물이 나와있다:

```
ggplot(data = gapminder, aes(x=year, y=lifeExp, by=country)) +
  geom_line(aes(color=continent)) + geom_point()
```



이번 예제에서, aesthetic인 색상 매팅이 ggplot에 전역으로 설정된 점 선택옵션에서 geom_line 계층으로 이동했다. 그래서, 해당 점에는 더이상 적용되지 않는다. 이제 분명하게 직선 위에 점이 도식화된 것을 확인할 수 있다.

꿀팁: aesthetic에 매팅대신에 값을 설정

지금까지 (색상같은) aesthetic 를 데이터의 변수로 매팅(mapping)해서 사용하는 법을 살펴봤다. 예를 들어, `geom_line(aes(color=continent))`을 사용하면, ggplot에서 자동으로 각 대륙별로 다른 색상을 입힌다. 그런데, 모든 선을 파란색으로 바꾸고자 하면 어떨까? `geom_line(aes(color="blue"))` 명령어가 동작해야 된다고 생각하지만, 사실은 그렇지 않다. 특정 변수에 대한 매팅을 생성하지 않았기 때문에, `aes()` 함수 밖으로 색상을 명시하는 부분을 예를 들어, `geom_line(color="blue")`와 같이 빼내기만 하면 된다.

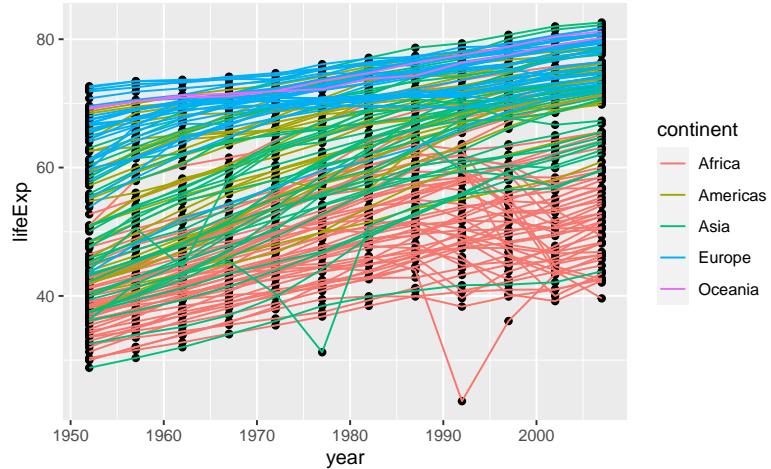
28.4 도전과제 3

앞선 예제에서 점과 직선 계층 코딩 순서를 뒤바꾼다. 어떻게 될까요?

도전과제 3에 대한 해답

앞선 예제에서 점과 직선 계층 코딩 순서를 뒤바꾼다. 어떻게 될까요?

```
ggplot(data = gapminder, aes(x=year, y=lifeExp, by=country)) +
  geom_point() + geom_line(aes(color=continent))
```

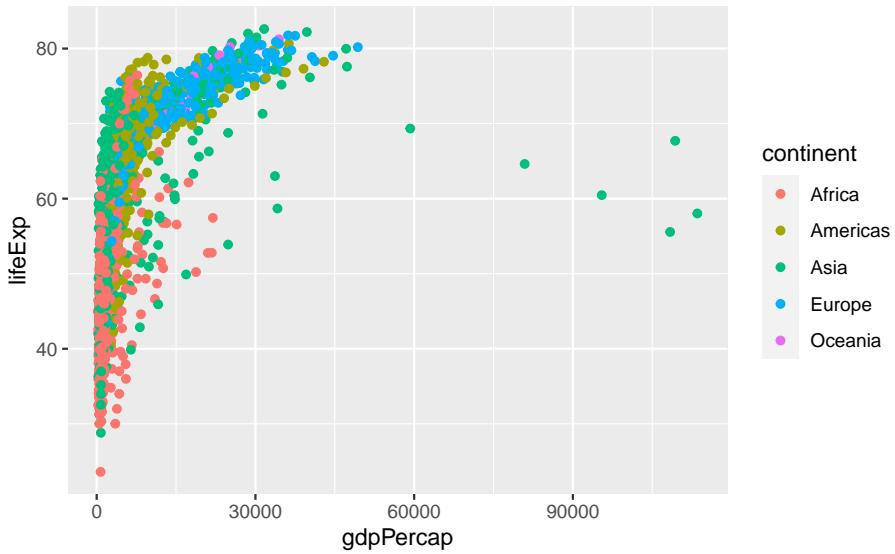


선이 점 위에 올라온다!

28.5 변환과 통계량

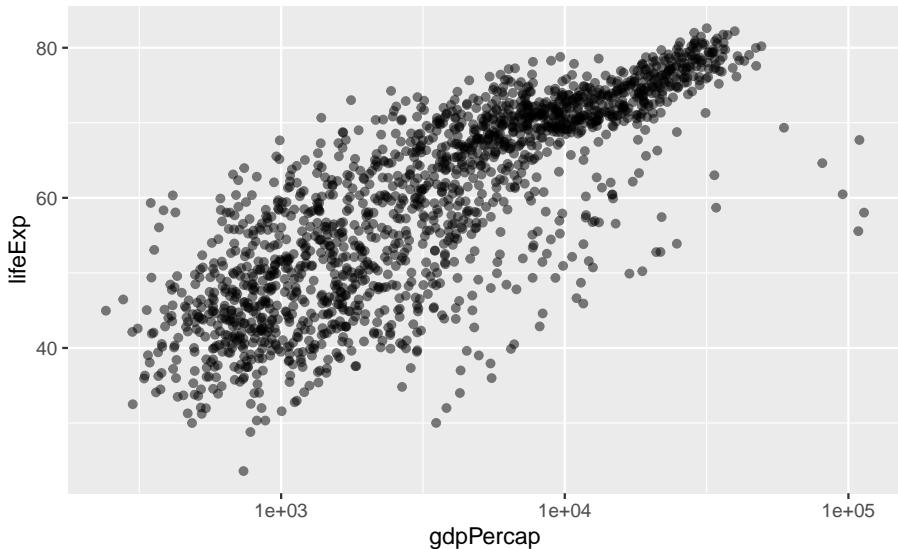
변환(Transformations)과 통계량(statistics)을 살펴보자. ggplot으로 데이터 위에 통계적 모형을 쉽게 겹치게 할 수 있다. 이를 시연하기 위해서, 첫번째 예제로 되돌아간다:

```
ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp, color=continent)) +
  geom_point()
```



현재, 일인당 GDP에 일부 심각한 이상점이 있어 점사이 내재된 관계를 보기 힘들다. scale 척도함수를 사용해서 y-축 척도를 변경한다. 이것을 통해 데이터 값과 aesthetic 시각값 사이 매핑을 제어한다. alpha 함수를 통해 투명도도 조정할 수 있다. 군집으로 많은 데이터가 모아진 경우 특히 투명도 조절을 유용하다.

```
ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp)) +
  geom_point(alpha = 0.5) + scale_x_log10()
```



그림에 렌더링하기 전에 \log_{10} 함수가 gdpPercap 칼럼값에 변환을 시켰다. 그래서,

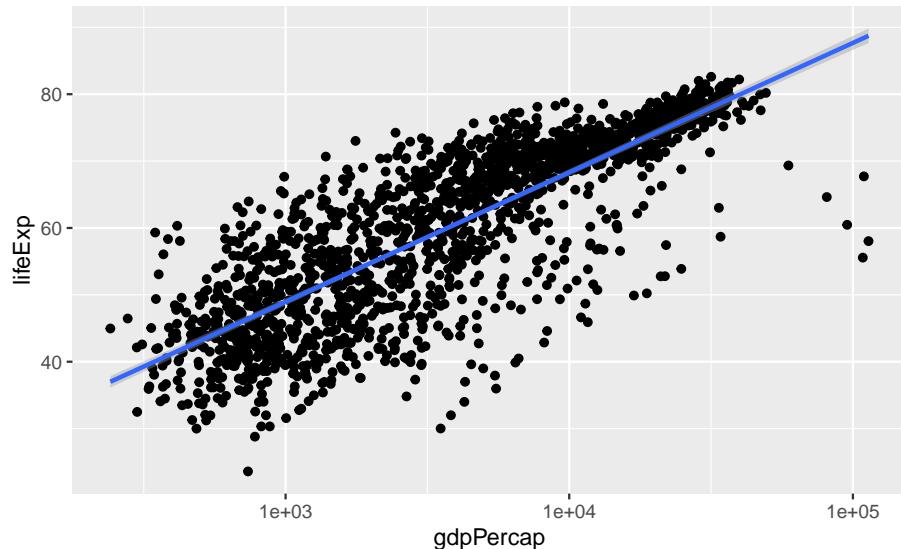
각 자리수 10은 변환된 척도에 1씩 증가에 대응된다. 예를 들어, 1인당 GDP 1,000은 y-축에 3, 10,000은 y-축에 4에 대응된다. 로그 변환은 x-축에 흘어진 데이터 시각화를 쉽게 도와준다.

꿀팁: `aesthetic`에 매핑대신에 값을 설정

`geom_point(alpha = 0.5)`을 사용한 것에 주목한다. 앞서 언급했듯이, `aes()` 함수 외부에서 설정된 것은 모든 점에 대해서 지정한 값이 적용되게 된다. 투명도 지정은 이 경우 원하는 바로 문제가 전혀 없다. 하지만, 다른 `aesthetic` 설정처럼, `alpha` 투명도를 데이터의 변수에 매핑시킬 수도 있다. 예를 들어, 각 대륙별로 다른 투명도를 적용시키고자 하면, `geom_point(aes(alpha = continent))` 코딩하는 것도 가능하다.

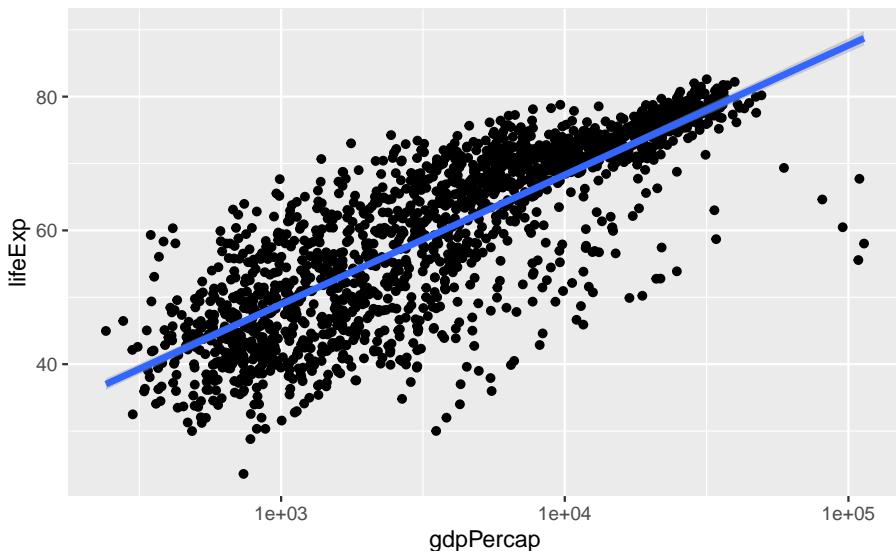
또다른 계층(`geom_smooth`)을 추가해서 관계를 단순히 적합시킬 수 있다:

```
ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() + scale_x_log10() + geom_smooth(method="lm")
```



굵은 선은 `geom_smooth` 계층에 `aesthetic` 크기를 설정해서 조정할 수 있다:

```
ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() + scale_x_log10() + geom_smooth(method="lm", size=1.5)
```



미학적인 항목을 명세할 수 있는 방식이 두개 있다. 바로 앞에서 `geom_smooth` 함수에 인자로 전달해서 크기에 대한 미학적인 설정을 했다. 앞에서는 `aes` 함수를 사용해서 데이터 변수와 시각적 표현 사이 매핑으로 정의했다.

28.6 도전과제 4a

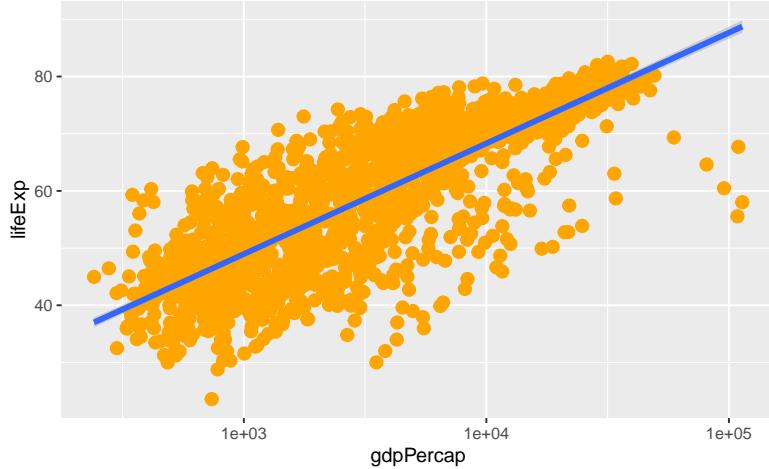
바로 앞 예제에서 점 계층에 나온 점 크기와 색상을 변경하라. **힌트:** `aes` 함수를 사용하지 않는다.

도전과제 4a에 대한 해답

바로 앞 예제에서 점 계층에 나온 점 크기와 색상을 변경하라.

힌트: `aes` 함수를 사용하지 않는다.

```
ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp)) +
  geom_point(size=3, color="orange") + scale_x_log10() +
  geom_smooth(method="lm", size=1.5)
```



28.7 도전과제 4b

도전과제 4a를 변경하는데, 점들이 다른 형태(shape)를 갖고 대륙별로 색상을 달리하는데 대륙별로 추세선을 반영한다.

힌트: 색상 인자를 aesthetic 내부로 위치시킨다.

도전과제 4b에 대한 해답

도전과제 4a를 변경하는데, 점들이 다른 형태(shape)를 갖고 대륙별로 색상을 달리하는데 대륙별로 추세선을 반영한다. 힌트: 색상 인자를 aesthetic 내부로 위치시킨다.

```
ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(size=3, shape=17) + scale_x_log10() +
  geom_smooth(method="lm", size=1.5)
```

28.8 다중-창(Multi-Panel) 그림

앞에서 그림 하나에 모든 국가에 대해 시간의 변화에 따른 기대수명 변화를 시각화했다. 대안으로, 패싯(facet) 창 계층을 추가해서, 그림을 여러개 창으로 조갤 수도 있다: “A” 혹은 “Z”로 시작하는 국가명을 갖는 나라만 집중해 보자.

팁(Tip)

데이터 부분집합(subset)을 추출하고 시작해 나간다. substr 함수를 사용해서 문자열의 일부를 뽑아낸다; 이 경우에, gapminder\$country 벡터의 시작과 끝 위치 문자가 된다. %in% 연산자를 통해서 조건을 만족하는 긴 코드 대신에 다중 비교를 간단히 수행한다. (이 경우,

```

starts.with %in% c("A", "Z") 코드는 starts.with == "A" |
starts.with == "Z")와 동일하다)

```
starts.with <- substr(gapminder$country, start = 1, stop = 1)
az.countries <- gapminder[starts.with %in% c("A", "Z"),]
ggplot(data = az.countries, aes(x = year, y = lifeExp, color=continent)) +
 geom_line() + facet_wrap(~ country)
```

`facet_wrap`    " (formula)"      , (`~`)
`gapminder`      .

##      {#r-ggplot-text}

x-      , y-      "Life expectancy"      .

**theme**
,      `labs()`      .
`aes()`      .
,      `color = "Continent"`      ,
(`fill`)  `fill = "MyTitle"`      .

ggplot(data = az.countries, aes(x = year, y = lifeExp, color=continent)) +
  geom_line() + facet_wrap( ~ country) +
  labs(
    x = "Year",           # x axis title
    y = "Life expectancy", # y axis title
    title = "Figure 1",    # main title of figure
    color = "Continent"    # title of legend
  ) +
  theme(axis.text.x=element_blank(), axis.ticks.x=element_blank())
```

{#r-ggplot-export}

`ggsave()` `ggplot` .
`ggsave()` (`width`, `height`, `dpi`)
,
`lifeExp_plot` ,
`ggsave()` `png` `results` .
(``results/` .)

```

```

lifeExp_plot <- ggplot(data = az.countries, aes(x = year, y = lifeExp, color=continent)
 geom_line() + facet_wrap(~ country) +
 labs(
 x = "Year", # x axis title
 y = "Life expectancy", # y axis title
 title = "Figure 1", # main title of figure
 color = "Continent" # title of legend
) +
 theme(axis.text.x=element_blank(), axis.ticks.x=element_blank())

ggsave(filename = "results/lifeExp.png", plot = lifeExp_plot, width = 12, height = 10,
```
`ggsave()`
`plot` , `ggplot` ( , `.`.png` ``.`.pdf` )
`device` .

`ggplot2` . RStudio [cheat sheet] [cheat]
, [ggplot2 ] [ggplot-doc]

, , Stack Overflow
!

[cheat]: http://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf
[ggplot-doc]: http://docs.ggplot2.org/current/

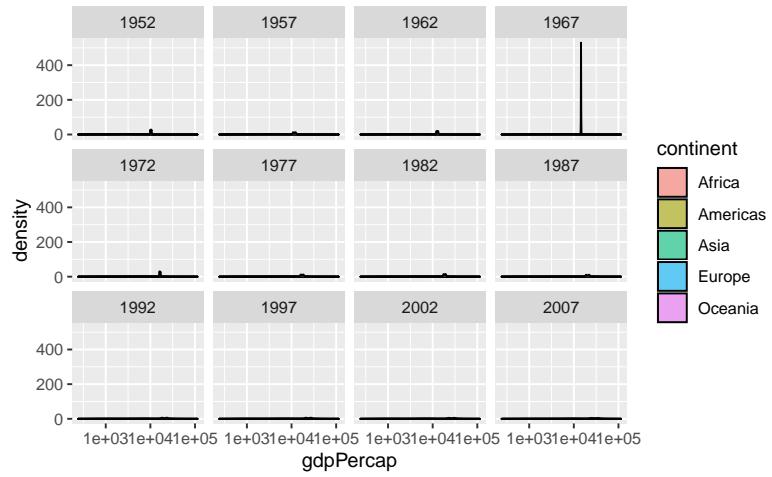
## 5 {#r-ggplot-challenge-five}

1 GPD
:
-
-x-
-
>

> ** 5 **
>
>     1 GPD
>
> :
> - x-
> -
>

```

```
ggplot(data = gapminder, aes(x = gdpPerCap, fill=continent)) +  
  geom_density(alpha=0.6) + facet_wrap( ~ year) + scale_x_log10()
```



Chapter 29

벡터화(Vectorization)

R 함수 대부분은 벡터화되었다. 즉, 한번에 각 요소에 대해 연산을 수행하도록 루프를 돌릴 필요없이 함수가 벡터 모든 요소에 대해 연산작업을 수행한다. 이렇게 되면 코드는 더욱 간결해지고, 가독성이 높아지고, 오류에 덜 노출된다.

```
x <- 1:4  
x * 2
```

```
## [1] 2 4 6 8
```

곱하기는 벡터 모든 요소에 일어난다.

두 벡터를 더할 수도 있다:

```
y <- 6:9  
x + y
```

```
## [1] 7 9 11 13
```

x 벡터 각 요소가 y 벡터 대응되는 요소에 더해진다:

```
x:  1  2  3  4  
     +  +  +  +  
y:  6  7  8  9  
-----  
    7  9 11 13
```

29.1 도전과제 1

gapminder 데이터셋 pop 칼럼에 벡터 연산을 시도해 본다. gapminder 데이터프레임에 신규 칼럼을 생성하는데, 백만명 단위로 인구정보를 표현한다. 데이터프레임에 head 혹은 tail 명령어를 적용해서 실제로 제대로 동작하는지 확인한다.

도전과제 1에 대한 해답

gapminder 데이터셋 pop 칼럼에 벡터 연산을 시도해 본다. gapminder 데이터프레임에 신규 칼럼을 생성하는데, 백만명 단위로 인구정보를 표현한다. 데이터프레임에 head 혹은 tail 명령어를 적용해서 실제로 제대로 동작하는지 확인한다.

```
gapminder$pop_millions <- gapminder$pop / 1e6
head(gapminder)
```

| | country | year | pop | continent | lifeExp | gdpPercap | pop_millions |
|------|-------------|------|----------|-----------|---------|-----------|--------------|
| ## 1 | Afghanistan | 1952 | 8425333 | Asia | 28.8 | 779 | 8.43 |
| ## 2 | Afghanistan | 1957 | 9240934 | Asia | 30.3 | 821 | 9.24 |
| ## 3 | Afghanistan | 1962 | 10267083 | Asia | 32.0 | 853 | 10.27 |
| ## 4 | Afghanistan | 1967 | 11537966 | Asia | 34.0 | 836 | 11.54 |
| ## 5 | Afghanistan | 1972 | 13079460 | Asia | 36.1 | 740 | 13.08 |
| ## 6 | Afghanistan | 1977 | 14880372 | Asia | 38.4 | 786 | 14.88 |

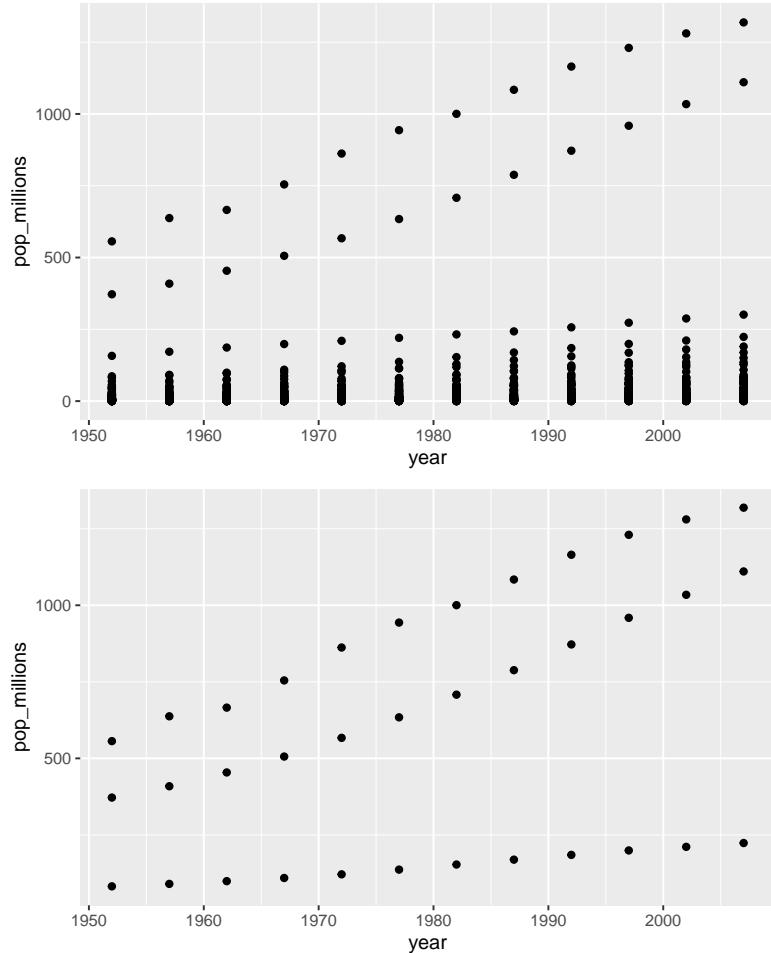
29.2 도전과제 2

그래프 하나에, 모든 국가에 대해 백만 단위로 인구를 연도별로 도식화한다. 어느 국가인지 식별하는 것은 신경쓰지 말자. 상기 연습문제를 반복하면서, 중국(China), 인도(India), 인도네시아(Indonesia)에 대해서만 도식화한다. 마찬가지로, 어느 국가인지 식별하는 것은 신경쓰지 말자.

도전과제 2에 대한 해답

연도별 백만단위로 인구수를 그래프로 표현한느데 앞서 학습한 내용을 상기한다.

```
ggplot(gapminder, aes(x = year, y = pop_millions)) +
  geom_point()
countryset <- c("China", "India", "Indonesia")
ggplot(gapminder[gapminder$country %in% countryset,],
       aes(x = year, y = pop_millions)) +
  geom_point()
```



비교 연산자, 논리 연산자, 그리고 많은 함수도 벡터화를 지원한다:

비교 연산자

```
x > 2
```

```
## [1] FALSE FALSE TRUE TRUE
```

논리 연산자

```
a <- x > 3 # or, for clarity, a <- (x > 3)
a
```

```
## [1] FALSE FALSE FALSE TRUE
```

꿀팁: 논리 벡터에 대한 유용한 일부 함수

`any()` 함수는 벡터 요소 어떤 것이든 TRUE 참이면, TRUE를 반환한다. `all()` 함수는

벡터 요소 모두가 TRUE 참이면, TRUE를 반환한다.

함수 대부분은 또한 벡터에 요소별(element-wise)로 연산작업을 수행한다:

함수(Functions)

```
x <- 1:4
log(x)
```

```
## [1] 0.000 0.693 1.099 1.386
```

벡터화 연산은 행렬(matrix)에 원소별로 연산작업을 수행한다:

```
m <- matrix(1:12, nrow=3, ncol=4)
m * -1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    -1   -4   -7  -10
## [2,]    -2   -5   -8  -11
## [3,]    -3   -6   -9  -12
```

꿀팁: 원소별(element-wise) 곱셈 vs. 행렬 곱셈

매우 중요: * 곱하기 연산은 요소별 곱셈 결과를 전달한다! 행렬 곱셈을 하려면, %*% 연산자를 사용한다:

```
m %*% matrix(1, nrow=4, ncol=1)

##      [,1]
## [1,]    22
## [2,]    26
## [3,]    30

matrix(1:4, nrow=1) %*% matrix(1:4, ncol=1)

##      [,1]
## [1,]    30
```

행렬 대수에 관한 더 많은 정보는 Quick-R reference guide을 참조한다.

29.3 도전과제 3

다음과 같은 행렬이 주어졌다:

```
m <- matrix(1:12, nrow=3, ncol=4)
m

##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

다음 명령어를 실행하면, 연산작업 결과가 어떻게 될지 생각한 것을 적어본다:

1. `m ^ -1`
2. `m * c(1, 0, -1)`
3. `m > c(0, 20)`
4. `m * c(1, 0, -1, 2)`

예상한 출력결과가 나왔나요? 만약 그렇지 않다면, 조교(helper)를 부르세요!

도전과제 3에 대한 해답

다음과 같은 행렬이 주어졌다:

```
m <- matrix(1:12, nrow=3, ncol=4)
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]     1     4     7    10
## [2,]     2     5     8    11
## [3,]     3     6     9    12
```

다음 명령어를 실행하면, 연산작업 결과가 어떻게 될지 생각한 것을 적어본다:

```
1. m ^ -1
##      [,1] [,2] [,3] [,4]
## [1,] 1.000 0.250 0.143 0.1000
## [2,] 0.500 0.200 0.125 0.0909
## [3,] 0.333 0.167 0.111 0.0833

2. m * c(1, 0, -1)
##      [,1] [,2] [,3] [,4]
## [1,]     1     4     7    10
## [2,]     0     0     0     0
## [3,]    -3    -6    -9   -12

3. m > c(0, 20)
##      [,1] [,2] [,3] [,4]
## [1,] TRUE FALSE TRUE FALSE
## [2,] FALSE TRUE FALSE TRUE
## [3,] TRUE FALSE TRUE FALSE
```

29.4 도전과제 4

다음 연속된 분수 순열 합계를 구하는데 관심이 있다:

```
x = 1/(1^2) + 1/(2^2) + 1/(3^2) + ... + 1/(n^2)
```

이 모두를 타이핑하는 것은 지루하고, n 값이 매우 큰 경우 불가능하다. 벡터화를 사용해서 n=100 일 때 x를 계산한다. n=10,000 일 때, 합은 얼마나 될까?

도전과제 4에 대한 해답

다음 연속된 분수 순열 합계를 구하는데 관심이 있다:

```
x = 1/(1^2) + 1/(2^2) + 1/(3^2) + ... + 1/(n^2)
```

이 모두를 타이핑하는 것은 지루하고, n 값이 매우 큰 경우 불가능하다. 벡터화를 사용해서 n=100 일 때 x를 계산한다. n=10,000 일 때, 합은 얼마나 될까?

```
sum(1/(1:100)^2)
```

```
## [1] 1.63
```

```
sum(1/(1:1e04)^2)
```

```
## [1] 1.64
```

```
n <- 10000
```

```
sum(1/(1:n)^2)
```

```
## [1] 1.64
```

함수를 사용해서 동일한 결과를 얻을 수도 있다:

```
inverse_sum_of_squares <- function(n) {
  sum(1/(1:n)^2)
}
```

```
inverse_sum_of_squares(100)
```

```
## [1] 1.63
```

```
inverse_sum_of_squares(10000)
```

```
## [1] 1.64
```

```
n <- 10000
```

```
inverse_sum_of_squares(n)
```

```
## [1] 1.64
```

Chapter 30

함수 설명

분석할 데이터셋이 하나라면, 파일을 엑셀같은 스프레드시트로 불러와서 간단한 통계량을 도식화하는 것이 아마도 훨씬 빠를 것이다. 하지만, gapminder 데이터는 주기적으로 갱신되며, 나중에 새로운 정보를 뽑아와서 분석작업을 다시 실행할 수도 있다. 또한, 미래 특정 시점에 다양한 곳에서 유사한 데이터를 얻어올 수도 있다.

이번 수업에서, 단일 명령어로 여러가지 연산작업을 반복하는 함수 작성방법을 학습할 것이다.

함수란 무엇인가?

함수는 연속된 연산작업을 모아 전체를 하나로 만들고, 향후 사용을 위해 보관된다. 함수는 다음 기능을 제공한다:

- 기억할 수 있고 나중에 호출할 수 있는 명칭
- 개별적인 연산을 기억할 필요에서 해방
- 사전에 정의된 입력과 예상되는 출력
- 더 큰 프로그래밍 환경에 풍부한 연계성

프로그래밍 언어 대부분의 기본 구성요소로서, 사용자 정의 함수는 어떠한 단일 추상화 못지 않는 중요한 “프로그래밍”으로 간주된다. 함수를 작성할 수 있다면, 여러분은 컴퓨터 프로그래마다:

30.1 함수 정의하기

새로운 R 스크립트 파일을 `functions/` 디렉토리에 `functions-lesson.R` 이름으로 저장하고 편집기에서 연다.

```
my_sum <- function(a, b) {  
  the_sum <- a + b  
  return(the_sum)  
}
```

화씨온도에서 캘빈온도로 변환하는 `fahr_to_kelvin()` 함수를 정의한다:

```
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}
```

`function` 출력을 대입하는 `fahr_to_kelvin()` 함수를 정의한다. 인자 명칭 리스트는 괄호 안에 담겨진다. 다음으로 함수 몸통부문(함수를 실행할 때 실행되는 문장)이 괄호({}) 내부에 담겨진다. 몸통부문 문장은 보통 공백 2 개로 들여쓰기한다. 들여쓰면, 코드 가독성이 높아지지만, 코드 동작에는 영향을 끼치지 않는다.

함수 생성을 요리책 저작하는 것처럼 생각하면 도움이된다. 먼저, 함수가 필요로 하는 “식재료(ingredient)”를 정의한다. 온도 변환기 함수에서, 함수에 사용할 식재료는 하나만 필요하다: “`temp`”. 식재료를 쭉 나열한 후에, 식재료로 무엇을 할지 기술한다; 이번 경우에, 식재료를 받아 일련의 수학 연산작업을 수행한다.

함수를 호출할 때, 함수에 전달되는 값이 변수에 대입된다. 그래서, 함수 내부에서 전달되는 값을 사용할 수 있다. 함수 내부에서 반환문(`return`)을 사용해서 호출한 쪽에 결과를 되돌려준다.

꿀팁:

R에만 있는 유일무이한 기능 하나가 반환문 `return`이 반드시 필요하지 않다는 점이다. R은 자동적으로 함수 몸통부문 마지막 줄에 있는 어떤 변수나 반환시킨다. 하지만 명확성을 위해서, 명시적으로 반환문 `return`을 정의한다.

함수를 실행해보자. 본인이 작성한 사용자 정의 함수 호출은 여타 다른 함수 호출과 차이가 없다:

```
#  
fahr_to_kelvin(32)  
  
## [1] 273  
#  
fahr_to_kelvin(212)  
  
## [1] 373
```

30.2 도전과제 1

`kelvin_to_celsius()` 함수를 작성해서 캘빈온도를 받아 섭씨온도를 반환한다.
힌트: 캘빈온도를 섭씨온도로 전환하려면, 273.15 을 뺀다.

도전과제 1에 대한 해답

`kelvin_to_celsius()` 함수를 작성해서 캘빈온도를 받아 섭씨온도를 반환한다.

```
kelvin_to_celsius <- function(temp) {
  celsius <- temp - 273.15
  return(celsius)
}
```

30.3 함수 결합

함수의 진정한 힘은 원하는 효과를 얻어 내는데 함수를 섞고, 매칭하고, 결합해서 더 큰 덩어리로 구현함에 있다.

화씨 온도에서 캠빈 온도로, 갤빈 온도에서 섭씨 온도로 온도를 전환하는 함수 두개를 정의한다:

```
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}

kelvin_to_celsius <- function(temp) {
  celsius <- temp - 273.15
  return(celsius)
}
```

30.4 도전과제 2

바로 위에 정의된 함수 두개를 재사용해서 화씨온도에서 바로 섭씨온도로 변환하는 함수를 정의한다. (혹은, 본인이 원하면 자신이 직접 작성한 함수를 사용한다.)

도전과제 2에 대한 해답

바로 위에 정의된 함수 두개를 재사용해서 화씨온도에서 바로 섭씨온도로 변환하는 함수를 정의한다.

```
fahr_to_celsius <- function(temp) {
  temp_k <- fahr_to_kelvin(temp)
  result <- kelvin_to_celsius(temp_k)
  return(result)
}
```

30.5 방어적 프로그래밍

R코드를 재사용 가능하고 모듈화 되도록 만드는데 함수 작성이 효율적인 방법을 제공한다는 점에서 이제야 참진가를 알게 되었다. 따라서, 함수가 의도된 방식으로 동작하도록 확실히 하는 것도 매우 중요하다. 함수 매개변수(인자) 검사는

방어적 프로그래밍(defensive programming) 개념과 연관되어 있다. 방어적 프로그래밍은 자주 조건을 검사하고, 무언가 잘못되면 오류를 던지도록 강요한다. 이러한 검사를 단언문(assertion statement)라고 부른다. 왜냐하면 다음으로 넘어가기 전에 조건이 TRUE 사실로 단언해야 되기 때문이다. 이러한 특성이 디버깅을 더 수월하게 하는데 이유는 오류가 최초 생겨난 곳에 대해 더 나은 아이디어를 제시해 주기 때문이다.

30.5.1 stopifnot() 조건 검사

화씨 온도를 켈빈 온도로 변환하는데 사용되는 자체 제작 함수 `fahr_to_kelvin()`을 다시 살펴보면서 시작해보자. 함수가 다음과 같이 정의되었다:

```
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}
```

함수가 의도한 대로 동작하기 위해서, `temp` 인자는 숫자형(numeric) 값이 되어야 한다; 그렇지 않은 경우, 온도를 변환하는 수학 연산 작업이 정상적으로 수행되지 않게 된다. 오류를 생성시키는데, `stop()` 함수를 사용한다. 예를 들어, `temp` 인자는 숫자형(numeric) 벡터가 되어야 하는데, `if` 문으로 조건을 검사해서 조건이 위반되면 오류를 던진다. 이를 반영하여 다음과 같이 함수 기능을 강화시킬 수 있다:

```
fahr_to_kelvin <- function(temp) {
  if (!is.numeric(temp)) {
    stop("temp must be a numeric vector.")
  }
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}
```

검사할 다수 조건과 인자가 있게 되면, 이 모두를 검사하는 코드가 많이 필요하게 된다. 다행히, `stopifnot()` 편의 함수가 R에서 지원된다. TRUE 참으로 평가되어야 하는 요구사항을 쭉 나열한다; FALSE 거짓인 것이 하나 발견되면 `stopifnot()` 함수는 오류를 던진다. 이러한 조건을 나열하는 것이 함수에 대한 부가 문서로서 부차적인 역할을 담당도 한다.

`fahr_to_kelvin()` 함수의 입력값을 검사하는 단언문을 추가해서 `stopifnot()` 함수로 방어적 프로그래밍을 시도해보자.

다음을 단언문으로 확인하자; `temp` 인자가 숫자 벡터다. 다음과 같이 코드를 작성하자:

```
fahr_to_kelvin <- function(temp) {
  stopifnot(is.numeric(temp))
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}
```

적절한 입력이 제시되면 정상적으로 동작한다.

```
#  
fahr_to_kelvin(temp = 32)  
  
## [1] 273  
  
부적절한 입력이 제시되면 즉시 실패한다.  
  
#  
fahr_to_kelvin(temp = as.factor(32))  
  
## Error in fahr_to_kelvin(temp = as.factor(32)): is.numeric(temp) is not TRUE
```

30.6 도전과제 3

방어적 프로그래밍을 사용해서, `temp` 인자를 부적절하게 제시하게 되면 `fahr_to_celsius()` 함수가 오류를 즉시 던지도록 작성한다.

도전과제 3에 대한 해답

`stopifnot()` 함수에 명시적으로 호출을 추가해서 앞서 정의한 함수를 확장시킨다. `fahr_to_celsius()` 함수는 다른 두 함수를 조합한 것이라서, 두번째 함수에도 검사를 추가하는 것은 중복이 된다.

```
fahr_to_celsius <- function(temp) {  
  stopifnot(is.numeric(temp))  
  temp_k <- fahr_to_kelvin(temp)  
  result <- kelvin_to_celsius(temp_k)  
  return(result)  
}
```

30.7 함수조합

이제 데이터셋에서 한 국가에 대한 국내총생산(GDP)를 계산하는 함수를 작성한다:

```
# , 1 GDP  
calcGDP <- function(dat) {  
  gdp <- dat$pop * dat$gdpPerCap  
  return(gdp)  
}
```

`function` 출력을 대입하는 `calcGDP()` 함수를 정의한다. 인자 명칭 리스트는 괄호 안에 담겨진다. 다음으로 함수 몸통부문 (함수를 실행할 때 실행되는 문장)이 괄호(`{}`) 내부에 담겨진다.

몸통부문 문장을 공백 2 개로 들여쓰기한다. 들여쓰면, 코드 가독성이 높아지지만, 코드 동작에는 영향을 끼치지 않는다.

함수를 호출할 때, 함수에 전달되는 값이 인자에 대입된다. 전달되는 값은 함수 몸통 내부에서 변수가 된다.

함수 내부에서, `return()` 함수를 사용해서 결과를 함수 밖으로 배출한다. 반환 `return()` 함수는 선택옵션이다: R은 자동으로 함수 마지막 줄에 실행되는 어떤 명령어든 결과를 반환한다.

```
calcGDP(head(gapminder))
```

```
## [1] 6.57e+09 7.59e+09 8.76e+09 9.65e+09 9.68e+09 1.17e+10
```

그다지 유용한 정보는 아니다. 인자를 일부 추가해서 연도별, 국가별 GDP를 추출한다.

```
# , 1 GDP
calcGDP <- function(dat, year=NULL, country=NULL) {
  if(!is.null(year)) {
    dat <- dat[dat$year %in% year, ]
  }
  if (!is.null(country)) {
    dat <- dat[dat$country %in% country,]
  }
  gdp <- dat$pop * dat$gdpPercap

  new <- cbind(dat, gdp=gdp)
  return(new)
}
```

별도 R 스크립트 파일에 상기 함수를 작성했다면(좋은 아이디어!), `source()` 함수를 사용해서 R 세션에 함수를 올려 적재할 수 있다:

```
source("functions/functions-lesson.R")
```

이제 함수에 상당히 많은 일을 하고 있네요. 쉬운 말로 풀어쓰면, 연도 `year` 인자가 비어있지 않는다면, 함수가 전달된 연도 정보로 부분집합 데이터를 생성하고 나서, 국가 `country` 인자가 비어있지 않는다면, 함수가 전달된 국가 정보로 부분집합 데이터를 생성한다. 그리고 나서, 연도와 국가 인자에서 추출된 부분집합에 대해 GDP를 계산한다. 그리고 나서, 함수가 GDP를 신규 칼럼으로 추가해서 부분집합으로 뽑아낸 데이터셋에서 추가하고 최종 결과로 반환한다. 숫자 벡터로 쭉 뽑혀진 것보다 출력결과가 훨씬 더 유용한 정보를 제공함을 알 수 있다.

연도를 인자로 명세할 때 어떤 일이 발생하는지 살펴보자:

```
head(calcGDP(gapminder, year=2007))
```

| | country | year | pop | continent | lifeExp | gdpPercap | gdp |
|-------|-------------|------|----------|-----------|---------|-----------|----------|
| ## 12 | Afghanistan | 2007 | 31889923 | Asia | 43.8 | 975 | 3.11e+10 |
| ## 24 | Albania | 2007 | 3600523 | Europe | 76.4 | 5937 | 2.14e+10 |
| ## 36 | Algeria | 2007 | 33333216 | Africa | 72.3 | 6223 | 2.07e+11 |
| ## 48 | Angola | 2007 | 12420476 | Africa | 42.7 | 4797 | 5.96e+10 |

```
## 60 Argentina 2007 40301927 Americas 75.3 12779 5.15e+11
## 72 Australia 2007 20434176 Oceania 81.2 34435 7.04e+11
```

혹은 특정 국가를 인자로 전달하면 어떻게 되는지 살펴보자:

```
calcGDP(gapminder, country="Australia")
```

```
##      country year      pop continent lifeExp gdpPercap      gdp
## 61 Australia 1952 8691212 Oceania   69.1    10040 8.73e+10
## 62 Australia 1957 9712569 Oceania   70.3    10950 1.06e+11
## 63 Australia 1962 10794968 Oceania   70.9    12217 1.32e+11
## 64 Australia 1967 11872264 Oceania   71.1    14526 1.72e+11
## 65 Australia 1972 13177000 Oceania   71.9    16789 2.21e+11
## 66 Australia 1977 14074100 Oceania   73.5    18334 2.58e+11
## 67 Australia 1982 15184200 Oceania   74.7    19477 2.96e+11
## 68 Australia 1987 16257249 Oceania   76.3    21889 3.56e+11
## 69 Australia 1992 17481977 Oceania   77.6    23425 4.10e+11
## 70 Australia 1997 18565243 Oceania   78.8    26998 5.01e+11
## 71 Australia 2002 19546792 Oceania   80.4    30688 6.00e+11
## 72 Australia 2007 20434176 Oceania   81.2    34435 7.04e+11
```

혹은, 연도와 국가를 모두 인자로 전달하면 어떻게 되는지 살펴보자:

```
calcGDP(gapminder, year=2007, country="Australia")
```

```
##      country year      pop continent lifeExp gdpPercap      gdp
## 72 Australia 2007 20434176 Oceania   81.2    34435 7.04e+11
```

함수 몸통부분을 살펴본다:

```
calcGDP <- function(dat, year=NULL, country=NULL) {
```

위에서 인자 두개 `year`, `country`를 추가했다. 함수 정의부에서 기본디폴트 인자를 모두 = 연산자를 사용해서 `NULL`로 설정했다. 이것이 의미하는 바는 사용자가 달리 인자를 설정하지 않는다면, 각 인자에 `NULL` 값을 인자로 넘기게 된다.

```
if(!is.null(year)) {
  dat <- dat[dat$year %in% year, ]
}
if (!is.null(country)) {
  dat <- dat[dat$country %in% country, ]
```

여기서, 각기 추가되는 인자가 `null`로 설정되었는지 검사한다. `null`이 아닌 경우, `null`이 아닌 인자로 부분집합 데이터로 뽑아내려고 `dat`에 저장된 데이터셋을 덮어쓴다.

추후에 좀더 유연하게 함수를 작성할 예정이다. 따라서, GDP를 다음 질문에 대해 답할 수 있다:

- 전체 데이터셋;

- 단일 연도;
- 단일 국가;
- 단일 연도와 단일국가 조합.

대신에 `%in%` 연산자를 사용해서, 여러 연도와 국가를 인자에 넘길 수 있다.

꿀팁: 값에 의한 전달

R에 함수는 거의 항상 데이터 사본을 생성해서 함수 몸통 내부에서 연산 작업을 수행한다. 함수 내부 `dat` 데이터를 변경할 때, 첫 인자로 전달한 원본 데이터가 아닌 `dat`에 저장된 `gapminder` 데이터셋 사본을 조작하는 것이다.

이를 “값에 의한 전달(pass-by-value)”라고 부르며, 코드 작성은 훨씬 더 안전하게 한다: 함수 몸통부문 내부에서 어떤 변경을 하든, 함수 몸통부문에 한정된 것이라는 것을 항상 확실히 할 수 있다.

꿀팁: 함수 유효범위(Function scope)

또한가지 중요한 개념이 유효범위(scoping)다: 함수 몸통부문 내부에서 생성하거나 변경한 어떤 변수(혹은 함수!)는 함수 실행 기간동안만 존재한다. `calcGDP` 함수를 호출할 때, 변수 `dat`, `gdp`, `new`는 함수 몸통 내부에서만 존재한다. 설사 인터랙티브 세션에서 동일한 명칭을 갖는 변수가 있더라도, 함수가 시행될 때, 어떤 방식으로도 변경되지 않는다.

```
gdp <- dat$pop * dat$gdpPerCap
new <- cbind(dat, gdp=gdp)
return(new)
}
```

마지막으로, 신규 부분집합 데이터셋에 GDP를 계산하고, 신규 데이터프레임을 생성하고 동일한 명칭을 갖는 칼럼을 추가한다. 즉, 나중에 함수를 호출할 때, 반환된 GDP 값에 대한 맥락을 찾아볼 수 있다는 것이다. 앞서 숫자 벡터보다 이런 점에서 훨씬 더 낫다.

30.8 도전과제 4

1987년 뉴질랜드 GDP를 계산하도록 GDP 함수를 테스트한다. 1952년 뉴질랜드 GDP와 얼마나 차이날까?

도전과제 4에 대한 해답

```
calcGDP(gapminder, year = c(1952, 1987), country = "New Zealand")
```

1987년 뉴질랜드 GDP: 65050008703

1952년 뉴질랜드 GDP: 21058193787

30.9 도전과제 5

`paste` 함수를 사용해서 텍스트를 결합할 수 있다, 즉:

```
best_practice <- c("Write", "programs", "for", "people", "not", "computers")
paste(best_practice, collapse = " ")
## [1] "Write programs for people not computers"
```

`text` 와 `wrapper`라는 벡터 인자 두개를 받는 `fence`라는 함수를 작성한다. 함수는 `wrapper`로 둘러싼 텍스트를 출력한다:

```
fence(text=best_practice, wrapper="***")
```

주목: `paste` 함수는 `sep`라는 인자를 갖는데, `sep` 인자는 텍스트 사이 구분자를 명세한다. 기본디폴트 설정은 공백이다: ”. `paste0` 함수에 대한 기본디폴트 설정은 공백이 없음이다.”“.

도전과제 5에 대한 해답

`text` 와 `wrapper`라는 벡터 인자 두개를 받는 `fence`라는 함수를 작성한다. 함수는 `wrapper`로 둘러싼 텍스트를 출력한다:

```
fence <- function(text, wrapper){
  text <- c(wrapper, text, wrapper)
  result <- paste(text, collapse = " ")
  return(result)
}
best_practice <- c("Write", "programs", "for", "people", "not", "computers")
fence(text=best_practice, wrapper="***")
## [1] "*** Write programs for people not computers ***"
```

꿀팁:

더 복잡한 연산작업을 수행할 때, 정말 잘 활용될 수 있는 R에만 있는 유일무이한 기능이 있다. 이러한 고급 개념으로 무장한 함수는 작성하지 않을 것이다. 향후, R로 함수를 작성하는데 무리가 없다면, R Language Manual과 Hadley Wickham 저서 Advanced R Programming 책에 포함된 환경을 참조한다. R은 프레임(frame) 대신에 “환경(environment)”을 용어로 사용한다.

꿀팁: 테스트와 문서화

함수를 테스트하고 문서화하는 것 모두 중요하다: 문서화를 하게 되면 여러분과 다른 독자가 작성된 함수 목적과 사용법을 이해하는데 도움이 된다.

작성된 함수가 생각한 대로 실제로 동작하는지 확실히 하는 것도 중요하다. 처음 시작할 때, 작업흐름은 아마도 다음과 같다:

1. 함수를 작성한다.
2. 함수 일부에 동작을 문서화하는 주석을 단다.
3. 소스 파일에 적재해서 불러온다.

4. 콘솔에서 예상한 대로 동작을 확인하는 실험을 한다.
5. 필요한 버그를 제거한다.
6. 깔끔하게 하고, 상기 과정을 반복한다.

함수에 대한 공식 문서는 별도 .Rd 파일에 작성되는데, 도움말 파일에서 보게 되는 문서로 변환된다.

roxygen2 팩키지는 R 프로그래머가 함수 코드와 함께 문서를 작성하게 돋고, 이 파일을 처리하게 되면 적절한 .Rd 파일로 떨군다. 더 복잡한 R 프로젝트 코드를 작성할 때, 문서를 작성하는 공식적인 방법으로 전환하면 된다. 공식 자동화 테스트는 testthat 팩키지를 사용해서 작성된다.

Chapter 31

데이터 저장

31.1 그래프 저장하기

`ggsave` 명령어를 사용해서, `ggplot2`에서 생성한 가장 최신 도식화 결과물을 저장하는 방법을 이미 살펴봤다. 다시금 상기시키기 위해 명령어를 적어보면 다음과 같다:

```
ggsave("My_most_recent_plot.pdf")
```

RStudio 내부에서 그래프로 저장할 경우, ‘Plot’ 윈도우에서 ‘Export’ 내보내기 버튼을 사용한다. 버튼을 클릭하면, .pdf, .png, .jpg 혹은 다른 이미지 형식으로 저장할지 선택옵션을 제시된다.

종종, ‘Plot’ 윈도우에 먼저 찍어보지 말고, 도표를 저장하고 싶을 때도 있다. 아마도 여러 페이지에 걸친 pdf 문서를 생성하고 싶을 것이다: 예를 들어, 각각은 다른 도표로 말이다. 혹은, 다수 파일에서 부분집합으로 데이터를 뽑아내고, 각 하위 데이터에 대해 도식화를 하고, 결과물을 도표로 저장하고자 한다. 하지만, 각각에 대해 ‘Export’ 내보내기 버튼을 클릭하려고 루프를 중단할 수는 없는 노릇이다.

이런 경우, 더 유연한 접근법을 사용할 수 있다. `pdf` 함수는 신류 pdf 장치를 생성한다. `pdf` 함수에 여러 인자를 사용해서, 크기와 해상도를 조절할 수 있다.

```
pdf("Life_Exp_vs_time.pdf", width=12, height=4)
ggplot(data=gapminder, aes(x=year, y=lifeExp, colour=country)) +
  geom_line() +
  theme(legend.position = "none")

# You then have to make sure to turn off the pdf device!
dev.off()
```

저장한 문서를 열어서 살펴본다.

31.2 도전과제 1

pdf 명령어를 다시 작성해서 pdf 파일에 두번째 페이지를 찍는다. 각 창 패널에 대륙별로 데이터를 패싯 도표(힌트: facet_grid 사용)로 출력한다.

도전과제 1에 대한 해답

```
pdf("Life_Exp_vs_time.pdf", width = 12, height = 4)
p <- ggplot(data = gapminder, aes(x = year, y = lifeExp, colour = country)) +
  geom_line() +
  theme(legend.position = "none")
p
p + facet_grid(. ~continent)
dev.off()
```

jpeg, png 등등 명령어도 다양한 형식으로 문서를 저장하는데, 유사하게 사용하면 된다.

31.3 데이터를 파일 저장

어느 시점이 되면, R에서 데이터를 내보내서 파일에 저장하기도 한다.

이런 목적으로 write.table 함수를 사용하는데, 앞서 살펴본 read.table 함수와 매우 유사하다.

데이터 정제 스크립트를 생성하자. gapminder 데이터에서 Australia 호주만 집중한다:

```
aust_subset <- gapminder[gapminder$country == "Australia",]

write.table(aust_subset,
            file="data/gapminder-aus.csv",
            sep=", "
)
```

쉘로 다시 전환해서, 모든 것이 정상인지 데이터를 살펴본다:

```
head data/gapminder-aus.csv

"country","year","pop","continent","lifeExp","gdpPercap"
"61","Australia",1952,8691212,"Oceania",69.12,10039.59564
"62","Australia",1957,9712569,"Oceania",70.33,10949.64959
"63","Australia",1962,10794968,"Oceania",70.93,12217.22686
"64","Australia",1967,11872264,"Oceania",71.1,14526.12465
"65","Australia",1972,13177000,"Oceania",71.93,16788.62948
"66","Australia",1977,14074100,"Oceania",73.49,18334.19751
"67","Australia",1982,15184200,"Oceania",74.74,19477.00928
```

```
"68", "Australia", 1987, 16257249, "Oceania", 76.32, 21888.88903
"69", "Australia", 1992, 17481977, "Oceania", 77.56, 23424.76683
```

음… 엄밀하게 보면 원하는 바는 아니다. 이 모든 인용부호는 어디서 왔을까? 또한 행번호도 보이는데 무의미하다.

도움말 파일을 살펴보고, 파일에 저장하는 방식을 변경해 본다.

```
?write.table
```

기본디플트 설정으로 데이터를 파일에 저장할 때, R은 자동으로 인용부호로 문자벡터를 감싼다. 행과 칼럼 명칭도 파일에 저장한다.

다음과 같이 고쳐본다:

```
write.table(
  gapminder[gapminder$country == "Australia",],
  file="data/gapminder-aus.csv",
  sep=",", quote=FALSE, row.names=FALSE
)
```

쉘 기술을 사용해서 다시 데이터를 살펴본다:

```
head data/gapminder-aus.csv
```

훨씬 좋아보인다!

31.4 도전과제 2

1990년 이후 수집된 데이터를 gapminder 데이터에서 부분집합으로 구성하는 데이터-경제 스크립트를 작성하라. 상기 스크립트를 사용해서, 작업한 신규 부분집합 데이터를 data/ 디렉토리에 파일로 저장한다.

도전과제 2에 대한 해답

```
write.table(
  gapminder[gapminder$year > 1990, ],
  file = "data/gapminder-after1990.csv",
  sep = ",", quote = FALSE, row.names = FALSE
)
```


Chapter 32

plyr로 데이터프레임을 쪼개고 합치기

앞서, 함수를 사용해서 코드를 단순화하는 방법을 살펴봤다. gapminder 데이터셋을 인자로 받아, (population)와 GDP를 곱해 GPD를 계산하는 calcGDP 함수를 정의했다. 추가적인 인자를 정의해서, year 연도별, country 국가별 필터를 적용할 수도 있다:

```
# Takes a dataset and multiplies the population column
# with the GDP per capita column.
calcGDP <- function(dat, year=NULL, country=NULL) {
  if(!is.null(year)) {
    dat <- dat[dat$year %in% year, ]
  }
  if (!is.null(country)) {
    dat <- dat[dat$country %in% country, ]
  }
  gdp <- dat$pop * dat$gdpPerCap

  new <- cbind(dat, gdp=gdp)
  return(new)
}
```

데이터 작업을 할 때, 흔히 마주치는 작업이 집단별 그룹으로 묶어 계산하는 것이다. 위에서, 단순히 두 칼럼을 곱해서 GDP를 계산했다. 하지만, 대륙별로 평균 GDP를 계산하고자 한다면 어떨까?

calcGDP를 실행하고 나서, 각 대륙별로 평균을 산출한다:

```
withGDP <- calcGDP(gapminder)
mean(withGDP[withGDP$continent == "Africa", "gdp"])
```

```
## [1] 2.09e+10
mean(withGDP[withGDP$continent == "Americas", "gdp"])

## [1] 3.79e+11
mean(withGDP[withGDP$continent == "Asia", "gdp"])

## [1] 2.27e+11
```

하지만, 그다지 멋있지는 않다. 그렇다. 함수를 사용해서, 반복되는 상당량 작업을 줄일 수 있었다. 함수 사용은 **멋있었다**. 하지만, 여전히 반복이 있다. 여러분이 직접 반복하게 되면, 일단 여러분 시간을 지금은 물론이고 그리고 나중에도 까먹게 되고, 잠재적으로 버그가 스며들 여지를 남기게 된다.

calcGDP처럼 유연성 있는 함수를 새로 작성할 수도 있지만, 제대로 동작하는 함수를 개발하기까지 상당량 노력과 테스트가 필요하다.

여기서 맞닥드린 추상화 문제를 “분할-적용-병합(split-apply-combine)” 전략이라고 부른다:

데이터를 집단으로 분할(split)하고, 이번 경우에 대륙, 해당 집단에 연산작업을 적용(apply)하고 나서, 선택옵션으로 결과를 묶어 병합(combine)한다.

32.1 plyr 팩키지

R을 예전에 사용했다면, apply 함수 가족에 친숙할 수 있다. R 내장함수도 동작이 잘 되지만, “분할-적용-병합” 문제를 해결하는데 사용되는 또다른 방법을 소개한다. plyr 팩키지는 이런 유형의 문제를 해결하는데 훨씬 사용자 친화적인 함수를 집합으로 제공한다.

이전 도전과제에서 plyr 팩키지를 설치했다. 이제 plyr 불러와서 적재한다:

```
library("plyr")
```

plyr 팩키지에는 리스트(lists), 데이터프레임(data.frames), 배열(arrays, 행렬, n-차원 벡터) 자료형 연산을 위한 함수가 있다. 함수 각각은 다음 작업을 수행한다:

1. **분할(split)**하는 연산
2. 순서대로 각각 쪼갠 것에 함수를 **적용(apply)**한다.
3. 데이터 객체로 출력 데이터로 다시 **병합(combine)**한다.

입력값으로 예상되는 자료구조와 출력값으로 반환되는 자료구조에 기반해서 함수명이 붙여졌다: [a]rray, [l]ist, [d]ata.frame. 첫번째 문자가 입력 자료구조, 두번째 문자가 출력 자료구조에 대응되고, 함수 나머지에 “ply”를 붙인다.

[a]rray, [l]ist, [d]ata.frame 를 입력과 출력에 조합하면 핵심 ****ply** 함수 9개가 산출된다. 분할과 적용 단계만 수행하고 병합단계를 거치지 않는 함수가 추가로 3개 있다. 입력 데이터 자료형에 NULL 출력값(_)으로 표현된다. (아래 테이블 참조)

여기서 “배열”에 대한 plyr가 다른 것과 다름에 주목한다. ply에 사용되는 배열은 벡터 혹은 행렬을 포함할 수 있다.

| | | Output | | | |
|-------|--------------------|--------|------------|-------|---------|
| | | array | data frame | list | nothing |
| Input | array | aaply | adply | alply | a_ply |
| | data frame | daply | ddply | dlply | d_ply |
| | list | laply | ldply | llply | l_ply |
| | n replicates | raply | rdply | rlply | r_ply |
| | function arguments | maply | mdply | mlply | m_ply |

Figure 32.1: 전체 apply 개요

xxply 함수(daply, ddply, llply, laply, ...) 각각은 동일한 구조를 갖고, 4가지 주요 기능을 갖는다:

```
xxply(.data, .variables, .fun)
```

- 함수명 첫글자는 입력 자료형, 함수 두번째 글자는 출력 자료형을 나타낸다.
- .data - 처리될 자료객체를 나타낸다.
- .variables - 분할(split)변수를 식별한다.
- .fun - 각 그룹 집단에 연산작업을 위해 호출되는 함수.

이제, 대륙별로 평균 GDP를 신속하게 계산할 수 있다:

```
ddply(
  .data = calcGDP(gapminder),
  .variables = "continent",
  .fun = function(x) mean(x$gdp)
)
```

```
##   continent      V1
## 1    Africa 2.09e+10
## 2  Americas 3.79e+11
## 3     Asia 2.27e+11
## 4   Europe 2.69e+11
```

```
## 5 Oceania 1.88e+11
```

방금전 코드에서 일어난 사건을 복기해 보자:

- ddply 함수에 data.frame 데이터프레임을 집어넣고(함수명이 d로 시작), 또다른 data.frame 데이터프레임을 반환한다(함수명 두번째 문자 d).
- 전달한 첫번째 인자는 연산작업을 실행하고자 하는 데이터프레임이다: 이번 경우에 gapminder 데이터. 먼저 calcGDP 함수를 호출해서, gapminder 데이터프레임에 gdp 칼럼을 추가한다.
- 두번째 인자가 분할 조건을 명시한다: 이번 경우에, “대륙(continent)” 칼럼이다. 부분집합(subsetting) 연산으로 이전에 수행했던 것처럼, 실제 칼럼 자체가 아니라, 칼럼명만 부여했음에 주목한다. plyr 팩키지에 자세한 구현이 되어 있어, 칼럼명만 전달하면 된다.
- 세번째 인자는 그룹 집단 각각에 적용하고자 하는 함수다. 이번 경우에는, 짧은 함수를 자체 정의했다: 데이터 각 부분집합은 함수 첫번째 인자, x에 저장되어 있다. 이것을 람다 함수라고 부른다: 어디서도 정의하지 않았기 때문에, 이름이 없는 무명함수다. ddply 함수가 호출되는 범위(scope)에만 존재한다.

출력 자료구조를 달리하면 어떨까?

32.2 도전과제 1

대륙별로 평균 기대수명을 계산하라. 어느 대륙이 가장 기대수명이 긴가? 어느 대륙이 가장 기대수명이 짧은가?

도전과제 1에 대한 해답

```
ddply(
  .data = gapminder,
  .variables = "continent",
  .fun = function(x) mean(x$lifeExp)
)

##   continent    V1
## 1      Africa 48.9
## 2    Americas 64.7
## 3      Asia 60.1
## 4     Europe 71.9
## 5  Oceania 74.3
```

오세아니아 대륙이 가장 길고 아프리카 대륙이 가장 짧다.

출력 자료구조를 달리하면 어떨까?

```
dlply(
  .data = calcGDP(gapminder),
  .variables = "continent",
```

```

.fun = function(x) mean(x$gdp)
)

## $Africa
## [1] 2.09e+10
##
## $Americas
## [1] 3.79e+11
##
## $Asia
## [1] 2.27e+11
##
## $Europe
## [1] 2.69e+11
##
## $Oceania
## [1] 1.88e+11
##
## attr(),"split_type")
## [1] "data.frame"
## attr(),"split_labels")
## continent
## 1 Africa
## 2 Americas
## 3 Asia
## 4 Europe
## 5 Oceania

```

동일한 함수를 호출했지만, 두번째 문자만 1로 변경했다. 그래서, 출력결과가 리스트로 반환됐다.

다수 칼럼을 지정해서 그룹별로 group by 할 수 있다:

```

ddply(
  .data = calcGDP(gapminder),
  .variables = c("continent", "year"),
  .fun = function(x) mean(x$gdp)
)

##    continent year      V1
## 1      Africa 1952 5.99e+09
## 2      Africa 1957 7.36e+09
## 3      Africa 1962 8.78e+09
## 4      Africa 1967 1.14e+10
## 5      Africa 1972 1.51e+10
## 6      Africa 1977 1.87e+10
## 7      Africa 1982 2.20e+10

```

```
## 8      Africa 1987 2.41e+10
## 9      Africa 1992 2.63e+10
## 10     Africa 1997 3.00e+10
## 11     Africa 2002 3.53e+10
## 12     Africa 2007 4.58e+10
## 13     Americas 1952 1.18e+11
## 14     Americas 1957 1.41e+11
## 15     Americas 1962 1.69e+11
## 16     Americas 1967 2.18e+11
## 17     Americas 1972 2.68e+11
## 18     Americas 1977 3.24e+11
## 19     Americas 1982 3.63e+11
## 20     Americas 1987 4.39e+11
## 21     Americas 1992 4.90e+11
## 22     Americas 1997 5.83e+11
## 23     Americas 2002 6.61e+11
## 24     Americas 2007 7.77e+11
## 25      Asia 1952 3.41e+10
## 26      Asia 1957 4.73e+10
## 27      Asia 1962 6.01e+10
## 28      Asia 1967 8.46e+10
## 29      Asia 1972 1.24e+11
## 30      Asia 1977 1.60e+11
## 31      Asia 1982 1.94e+11
## 32      Asia 1987 2.42e+11
## 33      Asia 1992 3.07e+11
## 34      Asia 1997 3.88e+11
## 35      Asia 2002 4.58e+11
## 36      Asia 2007 6.28e+11
## 37      Europe 1952 8.50e+10
## 38      Europe 1957 1.10e+11
## 39      Europe 1962 1.39e+11
## 40      Europe 1967 1.73e+11
## 41      Europe 1972 2.19e+11
## 42      Europe 1977 2.55e+11
## 43      Europe 1982 2.79e+11
## 44      Europe 1987 3.17e+11
## 45      Europe 1992 3.43e+11
## 46      Europe 1997 3.84e+11
## 47      Europe 2002 4.36e+11
## 48      Europe 2007 4.93e+11
## 49      Oceania 1952 5.42e+10
## 50      Oceania 1957 6.68e+10
## 51      Oceania 1962 8.23e+10
## 52      Oceania 1967 1.06e+11
## 53      Oceania 1972 1.34e+11
```

```

## 54   Oceania 1977 1.55e+11
## 55   Oceania 1982 1.76e+11
## 56   Oceania 1987 2.09e+11
## 57   Oceania 1992 2.36e+11
## 58   Oceania 1997 2.89e+11
## 59   Oceania 2002 3.45e+11
## 60   Oceania 2007 4.04e+11

daply(
  .data = calcGDP(gapminder),
  .variables = c("continent", "year"),
  .fun = function(x) mean(x$gdp)
)

##           year
## continent    1952    1957    1962    1967    1972    1977    1982
## Africa      5.99e+09 7.36e+09 8.78e+09 1.14e+10 1.51e+10 1.87e+10 2.20e+10
## Americas   1.18e+11 1.41e+11 1.69e+11 2.18e+11 2.68e+11 3.24e+11 3.63e+11
## Asia        3.41e+10 4.73e+10 6.01e+10 8.46e+10 1.24e+11 1.60e+11 1.94e+11
## Europe     8.50e+10 1.10e+11 1.39e+11 1.73e+11 2.19e+11 2.55e+11 2.79e+11
## Oceania    5.42e+10 6.68e+10 8.23e+10 1.06e+11 1.34e+11 1.55e+11 1.76e+11
##           year
## continent    1987    1992    1997    2002    2007
## Africa      2.41e+10 2.63e+10 3.00e+10 3.53e+10 4.58e+10
## Americas   4.39e+11 4.90e+11 5.83e+11 6.61e+11 7.77e+11
## Asia        2.42e+11 3.07e+11 3.88e+11 4.58e+11 6.28e+11
## Europe     3.17e+11 3.43e+11 3.84e+11 4.36e+11 4.93e+11
## Oceania    2.09e+11 2.36e+11 2.89e+11 3.45e+11 4.04e+11

```

for 루프 자리에 람다 함수를 사용할 수 있다(대체로 더 빠르다): for 루프 몸통부분을 람다 함수에 작성하면 된다:

```

d_ply(
  .data=gapminder,
  .variables = "continent",
  .fun = function(x) {
    meanGDPperCap <- mean(x$gdpPerCap)
    print(paste(
      "The mean GDP per capita for", unique(x$continent),
      "is", format(meanGDPperCap, big.mark=","))
  })
}
)

## [1] "The mean GDP per capita for Africa is 2,194"
## [1] "The mean GDP per capita for Americas is 7,136"
## [1] "The mean GDP per capita for Asia is 7,902"
## [1] "The mean GDP per capita for Europe is 14,469"

```

```
## [1] "The mean GDP per capita for Oceania is 18,622"
```

꿀팁: 숫자 출력하기

`format` 함수를 사용해서 메시지와 함께 숫자값을 “예쁘게” 출력할 수도 있다.

32.3 도전과제 2

대륙과 연도별로 평균 기대수명을 계산하시오. 2007년에 어느 것이 가장 짧고, 가장 긴가? 1952년과 2007년 사이에 가장 커다란 변화는 어느 것인가?

도전과제 2에 대한 해답

```
solution <- ddply(
  .data = gapminder,
  .variables = c("continent", "year"),
  .fun = function(x) mean(x$lifeExp)
)
solution_2007 <- solution[solution$year == 2007, ]
solution_2007

##    continent year    V1
## 12    Africa 2007 54.8
## 24   Americas 2007 73.6
## 36      Asia 2007 70.7
## 48    Europe 2007 77.6
## 60  Oceania 2007 80.7
```

오세아니아 대륙이 2007년 가장 긴 평균 기대수명을 갖는 반면 아프리카 대륙이 가장 짧다.

```
solution_1952_2007 <- cbind(solution[solution$year == 1952, ], solution_2007)
difference_1952_2007 <- data.frame(continent = solution_1952_2007$continent,
                                       year_1957 = solution_1952_2007[[3]],
                                       year_2007 = solution_1952_2007[[6]],
                                       difference = solution_1952_2007[[6]] - solution_2007)

##    continent year_1957 year_2007 difference
## 1    Africa     39.1     54.8     15.7
## 2   Americas     53.3     73.6     20.3
## 3      Asia     46.3     70.7     24.4
## 4    Europe     64.4     77.6     13.2
## 5  Oceania     69.3     80.7     11.5
```

아시아 대륙이 가장 큰 차를 보이는 반면 오세아니아 대륙이 가장 적은 차를 보인다.

32.4 대안 도전과제

실제로 실행하지 말고, 다음 중 어떤 코드가 대륙별 평균 기대수명을 계산하는가:

1.

```
ddply(  
  .data = gapminder,  
  .variables = gapminder$continent,  
  .fun = function(dataGroup) {  
    mean(dataGroup$lifeExp)  
  }  
)
```

2.

```
ddply(  
  .data = gapminder,  
  .variables = "continent",  
  .fun = mean(dataGroup$lifeExp)  
)
```

3.

```
ddply(  
  .data = gapminder,  
  .variables = "continent",  
  .fun = function(dataGroup) {  
    mean(dataGroup$lifeExp)  
  }  
)
```

4.

```
adply(  
  .data = gapminder,  
  .variables = "continent",  
  .fun = function(dataGroup) {  
    mean(dataGroup$lifeExp)  
  }  
)
```

도전과제에 대한 해답

대륙별 평균 기대수명을 4번째 R 코드로 계산할 수 있다.

Chapter 33

dplyr 솜씨있게 조작

데이터프레임을 솜씨있게 조작하는 것은 많은 과학연구원에게 많은 것을 의미한다. 특정 관측점(행) 혹은 변수(열)을 선택하거나, 특정 변수(들)로 데이터를 집단으로 그룹짓거나, 요약 통계량을 계산하기도 한다. 이런 연산작업에 정규 기본 베이스(Base) R 연산을 사용한다:

```
mean(gapminder[gapminder$continent == "Africa", "gdpPercap"])
## [1] 2194
mean(gapminder[gapminder$continent == "Americas", "gdpPercap"])
## [1] 7136
mean(gapminder[gapminder$continent == "Asia", "gdpPercap"])
## [1] 7902
```

하지만, 그다지 멋있지는 않은데, 이유는 상당한 반복이 상존하기 때문이다. 여러분이 직접 반복하게 되면, 일단 여러분 시간을 지금 물론이고, 나중에 소중한 시간을 깨먹게 되고, 더나아가 잠재적으로 버그가 스며들 여지를 남기게 된다.

33.1 dplyr 팩키지

운좋게도, dplyr 팩키지가 데이터프레임을 솜씨있게 조작하는데 있어 유용한 함수를 많이 제공한다. 이를 통해서, 위에서 언급된 반복을 줄이고, 실수를 범활 확률도 줄이고, 심지어 타이핑하는 수고도 줄일 수 있다. 보너스로, dplyr 문법은 훨씬 더 가독성도 높다.

가장 흔히 사용되는 6가지 함수 뿐만 아니라, 이런 함수를 조합하는데 사용되는 파이프 (%>%) 연산자 사용법도 다룬다.

1. select()

2. filter()
3. group_by()
4. summarize()
5. mutate()

이전 수업에서 팩키지를 설치하지 않았다면, 설치해서 직접 실습해보기 바란다:

```
install.packages('dplyr')
```

이제 팩키지를 불러와서 적재한다:

```
library("dplyr")
```

33.2 select() 사용

예를 들어, 데이터프레임에서 변수 일부만 뽑아서 작업해 나가고자 한다면, select() 함수를 사용한다. 이 함수는 선택한 변수만 갖도록 지정한다.

```
year_country_gdp <- select(gapminder, year, country, gdpPercap)
```

year_country_gdp 데이터프레임을 열게되면, year, country, gdpPercap 변수만 담겨있는 것을 보게 된다. 위에서는 문법이 사용되었지만, dplyr 팩키지의 장점은 파이프를 사용해서 함수 다수를 조합하는데 있다. 파이프 문법은 이전에 R에서 살펴봤던 것과는 사뭇 다르다. 위에서 파이프를 사용했던 것을 다시 작성해본다.

```
year_country_gdp <- gapminder %>% select(year, country, gdpPercap)
```

파이프를 사용해서 작성한 이유에 대한 이해를 돋기 위해서, 단계별로 살펴보자. 먼저 gapminder 데이터프레임을 불러오고 나서, %>% 파이프 기호를 사용해서 다음 작업단계(select() 함수)로 전달했다. 이번 경우에는 select() 함수에 데이터 객체를 명세하지 않았다. 이유는 이전 파이프로부터 건네받았기 때문이다. **재미난 사실:** 유닉스 쉘 강의에서 이미 파이프를 접해봤을 것이다. R에서 파이프 기호가 %>%인 반면, 쉘에서는 |을 사용한다. 하지만, 개념은 동일하다!

33.3 filter() 사용

이제 앞선 작업을 바탕으로 앞으로 작업을 진척시켜 보자. 유럽대륙만 갖고 작업하고자 한다. select 와 filter를 조합하면 된다.

```
year_country_gdp_euro <- gapminder %>%
  filter(continent=="Europe") %>%
  select(year, country, gdpPercap)
```

33.4 도전과제 1

명령어를 하나 (여러 행에 걸칠 수 있고, 파이프도 포함한다) 작성하는데, lifeExp, country, year 변수에 대해서 아프리카 대륙(African)만 갖는 데이터프레임을 작성한다. 하지만, 다른 대륙은 포함되면 안된다. 데이터프레임 행의 갯수는 얼마나 되는가? 그리고 이유는 무엇인가?

33.5 도전과제 1에 대한 해답

```
year_country_lifeExp_Africa <- gapminder %>%
  filter(continent=="Africa") %>%
  select(year, country, lifeExp)

{: .solution} {: .challenge}
```

지난번과 마찬가지로, gapminder 데이터프레임을 filter() 함수에 전달하고 나서, 필터링된 gapminder 데이터프레임 버전을 select() 함수에 전달한다. 주의: 연산순서가 이번 경우에 무척 중요하다. select() 함수를 먼저 실행하면, filter() 함수는 대륙 변수를 찾을 수 없는데, 이유는 이전 단계에서 제거했기 때문이다.

33.6 group_by()와 summarize() 사용

이제, 기본 베이스(base) R로 작업함으로써 실수를 범하기 쉬운 반복작업을 줄일 것으로 생각했지만, 현재까지 목표를 달성하지 못했다. 왜냐하면, 각 대륙마다 상기 작업을 반복해야 되기 때문이다. filter() 대신에, group_by()를 사용한다. filter()는 특정 기준을 만족하는 관측점만 넘겨준다(이번 경우: continent=="Europe"). group_by()는 본질적으로, 필터에서 사용할 수 있는 모든 유일무이한 기준을 사용할 수 있다. “

```
str(gapminder)

str(gapminder %>% group_by(continent))
```
`group_by()` (`grouped_df`) `gapminder` (`data.frame`)
`grouped_df` `list` , `list` `data.frame` ,
`continent` ().

<!-- -->
```

```

`summarize()` {#r-dplyr-summarize}

`group_by()` `summarize()`
 ,
 ,
 `group_by()` ,
 (
 `mean()` `sd()` `summarize()`

gdp_bycontinents <- gapminder %>%
 group_by(continent) %>%
 summarize(mean_gdpPercap=mean(gdpPercap))
``````

! [] (assets/images/r/13-dplyr-fig3.png)

continent mean_gdpPercap
<fctr>      <dbl>
1 Africa      2193.755
2 Americas    7136.110
3 Asia        7902.150
4 Europe      14469.476
5 Oceania     18621.609
``````

`gdpPercap` ,
``````

## 2 {#r-dplyr-challenge-two}

``````

,
?

> ** 2 **

>

lifeExp_bycountry <- gapminder %>%
 group_by(country) %>%
 summarize(mean_lifeExp=mean(lifeExp))

lifeExp_bycountry %>%
 filter(mean_lifeExp == min(mean_lifeExp) | mean_lifeExp == max(mean_lifeExp))
``````
```

```
    `dplyr`     `arrange()`  
`arrange()`  
`dplyr`  
`arrange()`     `desc()`  
  
lifeExp_bycountry %>%  
  arrange(mean_lifeExp) %>%  
  head(1)  
lifeExp_bycountry %>%  
  arrange(desc(mean_lifeExp)) %>%  
  head(1)  
...  
  
`group_by()`  
`year`   `continent`  
  
gdp_bycontinents_byyear <- gapminder %>%  
  group_by(continent, year) %>%  
  summarize(mean_gdpPercap=mean(gdpPercap))  
...  
  
  ,  
`summarize()`  
  ,  
  
gdp_pop_bycontinents_byyear <- gapminder %>%  
  group_by(continent, year) %>%  
  summarize(mean_gdpPercap=mean(gdpPercap),  
            sd_gdpPercap=sd(gdpPercap),  
            mean_pop=mean(pop),  
            sd_pop=sd(pop))  
...  
  
## `count()` `n()` {#r-dplyr-count}  
  
`dplyr`  
  2  
  , 2002  
`count()`  
  ,  
  `sort=TRUE`  
  
gapminder %>%  
  filter(year == 2002) %>%  
  count(continent, sort = TRUE)  
...
```

```

    , `n()`` . .
    , : .
gapminder %>%
  group_by(continent) %>%
  summarize(se_le = sd(lifeExp)/sqrt(n())))
```
;
`minimum`, `maximum`, `mean`, `se` . .

gapminder %>%
 group_by(continent) %>%
 summarize(
 mean_le = mean(lifeExp),
 min_le = min(lifeExp),
 max_le = max(lifeExp),
 se_le = sd(lifeExp)/sqrt(n())))
```
;

## `mutate()` {#r-dplyr-mutate}

`mutate()` ( ) .

gdp_pop_bycontinents_byyear <- gapminder %>%
  mutate(gdp_billion=gdpPercap*pop/10^9) %>%
  group_by(continent,year) %>%
  summarize(mean_gdpPercap=mean(gdpPercap),
            sd_gdpPercap=sd(gdpPercap),
            mean_pop=mean(pop),
            sd_pop=sd(pop),
            mean_gdp_billion=mean(gdp_billion),
            sd_gdp_billion=sd(gdp_billion))
```
;

`mutate()` {#r-dplyr-ifelse}

, `mutate()` .
`mutate()` `ifelse()` : , .
() , .

keeping all data but "filtering" after a certain condition
calculate GDP only for people with a life expectation above 25
gdp_pop_bycontinents_byyear_above25 <- gapminder %>%
 mutate(gdp_billion = ifelse(lifeExp > 25, gdpPercap * pop / 10^9, NA)) %>%

```

```

group_by(continent, year) %>%
 summarize(mean_gdpPercap = mean(gdpPercap),
 sd_gdpPercap = sd(gdpPercap),
 mean_pop = mean(pop),
 sd_pop = sd(pop),
 mean_gdp_billion = mean(gdp_billion),
 sd_gdp_billion = sd(gdp_billion))

updating only if certain condition is fulfilled
for life expectations above 40 years, the gdp to be expected in the future is scaled
gdp_future_bycontinents_byyear_high_lifeExp <- gapminder %>%
 mutate(gdp_futureExpectation = ifelse(lifeExp > 40, gdpPercap * 1.5, gdpPercap)) %>%
 group_by(continent, year) %>%
 summarize(mean_gdpPercap = mean(gdpPercap),
 mean_gdpPercap_expected = mean(gdp_futureExpectation))
```
```
`dplyr` `ggplot2` {#r-dplyr-ggplot}

`ggplot2` (facet)
 :
 # Get the start letter of each country
starts.with <- substr(gapminder$country, start = 1, stop = 1)
Filter countries that start with "A" or "Z"
az.countries <- gapminder[starts.with %in% c("A", "Z"),]
Make the plot
ggplot(data = az.countries, aes(x = year, y = lifeExp, color = continent)) +
 geom_line() + facet_wrap(~ country)
```
```
 ,
 (`starts.with`, `az.countries`)
`%>%` `dplyr` ,
`ggplot()` `.` ,
`%>%` ,
`ggplot()` `data =` ,
`dplyr`, `ggplot2` ,
 .

gapminder %>%
 # Get the start letter of each country
 mutate(startsWith = substr(country, start = 1, stop = 1)) %>%
 # Filter countries that start with "A" or "Z"
 filter(startsWith %in% c("A", "Z")) %>%
 # Make the plot

```

```

ggplot(aes(x = year, y = lifeExp, color = continent)) +
 geom_line() +
 facet_wrap(~ country)
```
`dplyr` :
gapminder %>%
  # Filter countries that start with "A" or "Z"
  filter(substr(country, start = 1, stop = 1) %in% c("A", "Z")) %>%
  # Make the plot
  ggplot(aes(x = year, y = lifeExp, color = continent)) +
  geom_line() +
  facet_wrap(~ country)
```
{#r-dplyr-challenge-advanced}

2002
,
.

** :** `dplyr` `arrange()`, `sample_n()`
`dplyr` .
.

> ** **
>

lifeExp_2countries_bycontinents <- gapminder %>%
 filter(year==2002) %>%
 group_by(continent) %>%
 sample_n(2) %>%
 summarize(mean_lifeExp=mean(lifeExp)) %>%
 arrange(desc(mean_lifeExp))
```
##      {#r-dplyr-advanced}

* [R for Data Science] (http://r4ds.had.co.nz/)
* [Data Wrangling Cheat sheet] (https://www.rstudio.com/wp-content/uploads/2015/02/)
* [Introduction to dplyr] (https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html)
* [Data wrangling with R and RStudio] (https://www.rstudio.com/resources/webinars/)

```

Chapter 34

tidyর 솜씨있게 조작

과학연구원들은 흔히 ‘wide’ 형식에서 ‘long’ 형식으로 혹은 역으로 데이터를 솜씨있게 조작해야 한다. ‘long’ 형식은 다음과 같이 정의된다:

- 각 칼럼이 변수.
- 각 행이 관측점

‘long’ 형식에서, 일반적으로 관측된 변수에 대해서 칼럼이 하나만 있고, 다른 칼럼은 ID 변수다.

‘wide’ 형식에서, 각 행은 흔히 관측점(site/subject/patient)이며, 동일한 자료형을 담고 있는 다수 관측변수를 갖게 된다. 시간이 경과함에 따라 반복되는 관측점이거나, 다수 변수의 관측점(혹은 둘이 혼합된 사례)일 수 있다. 데이터 입력이 더 단순하거나 일부 다른 응용사례에서 ‘wide’ 형식을 선호할 수 있다. 하지만, R 함수 다수는 ‘long’형식을 가정하고 설계되었다. 이번 학습을 통해서 원래 데이터 형식에 관계없이 데이터를 효율적으로 변환하는 방식을 학습한다.

wide vs long

| | ID | ID2 | A |
|----|----|-----|----|
| 1 | 1 | a1 | |
| 2 | 2 | a1 | |
| 3 | 3 | a1 | |
| ID | a1 | a2 | a3 |
| 1 | | 1 | a2 |
| 2 | | 2 | a2 |
| 3 | | 3 | a2 |
| 1 | | 1 | a3 |
| 2 | | 2 | a3 |
| 3 | | 3 | a3 |

데이터 형식은 주로 가독성에 영향을 준다. 사람에게, ‘wide’ 형식이 좀 더 직관적인데, 이유는 데이터 형상으로 인해 화면에 더 많은 데이터를 볼 수 있기 때문이다. 하지만, 컴퓨터에게 ‘long’ 형식이 더 가독성이 높고, 데이터베이스 형식에 훨씬 더 가깝다. 데이터프레임에 ID 변수는 데이터베이스 필드(Field)와 유사하고, 관측변수는 데이터베이스 값(Value)과 유사하다.

34.1 시작하기

(아마도 이전 학습에서 dplyr 팩키지는 설치했을 것이다) 먼저 설치하지 않았다면, 팩키지를 설치한다:

```
#install.packages("tidyverse")
#install.packages("dplyr")
```

팩키지를 불러와서 적재한다.

```
library("tidyverse")
library("dplyr")
```

먼저, gapminder 데이터프레임 자료구조를 살펴보자:

```
str(gapminder)

## 'data.frame': 1704 obs. of 6 variables:
## $ country : chr "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
## $ year    : int 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
## $ pop     : num 8425333 9240934 10267083 11537966 13079460 ...
## $ continent: chr "Asia" "Asia" "Asia" "Asia" ...
## $ lifeExp : num 28.8 30.3 32 34 36.1 ...
## $ gdpPercap: num 779 821 853 836 740 ...
```

34.2 도전과제 1

gapminder는 순수하게 ‘long’ 형식인가, ‘wide’ 형식인가, 혹은 두 가지 특징을 갖는 중간형식인가?

도전과제 1에 대한 해답

원 gapminder 데이터프레임은 두 가지 특징을 갖는 중간 형식이다. 데이터프레임에 다수 관측변수 (pop, lifeExp, gdpPercap)가 있다는 점에서, 순수한 긴형식은 아니라고 볼 수 있다.

gapminder 데이터셋처럼, 관측된 데이터에 대한 다양한 자료형이 있다. 대부분 순수 100% ‘long’ 혹은 순수 100% ‘wide’ 자료형식 사이 어딘가에 위치하게 된다. gapminder 데이터셋에는 “ID” 변수가 3개(continent, country, year), “관측변수”가 3개(pop, lifeExp, gdpPercap) 있다. 저자는 일반적으로 대부분의 경우에 중간단계 형식 데이터를 선호한다. 칼럼 1곳에 모든 관측점이 3가지 서로

다른 단위를 갖지 않음에도 불구하고 그렇다. 데이터프레임을 좀더 늘리는 연산은 거의 없다(예를 들어, ID변수 4개, 관측변수 1개).

흔히 벡터기반인 다수 R 함수를 사용할 때, 흔히 다른 단위를 갖는 값에 수학적 연산작업을 수행하지는 않느다. 예를 들어, 순수 ‘long’ 형식을 사용할 때, 인구, 기대수명, GDP의 모든 값에 대한 평균은 의미가 없는데, 이유는 상호 호환되지 않는 3가지 단위를 갖는 평균값을 계산하여 반환하기 때문이다. 해법은 먼저 집단으로 그룹지어서 데이터를 솜씨있게 다루거나 (dplyr 학습교재 참조), 데이터프레임 구조를 변경시키는 것이다. 주의: R에서 일부 도식화 함수는 ‘wide’ 형식 데이터에 더 잘 작동한다.

34.3 gather()로 wide → long 전환

지금까지, 깔끔한 형식을 갖는 원본 gapminder 데이터셋으로 작업을 했다. 하지만, ‘실제’ 데이터(즉, 자체 연구 데이터)는 절대로 잘 구성되어 있지 못하다. gapminder 데이터셋에 대한 wide 형식 버전을 갖고 시작해본다.

이곳에서 ‘wide’ 형태를 갖는 gapminder 데이터를 다운로드 받아서, 로컬 data 폴더에 저장시킨다.

데이터 파일을 불러와서 살펴보자. 주의: continent, country 칼럼이 요인형 자료형이 될 필요가 없다. 따라서 read.csv() 함수 인자로 stringsAsFactors를 거짓(FALSE)으로 설정한다.

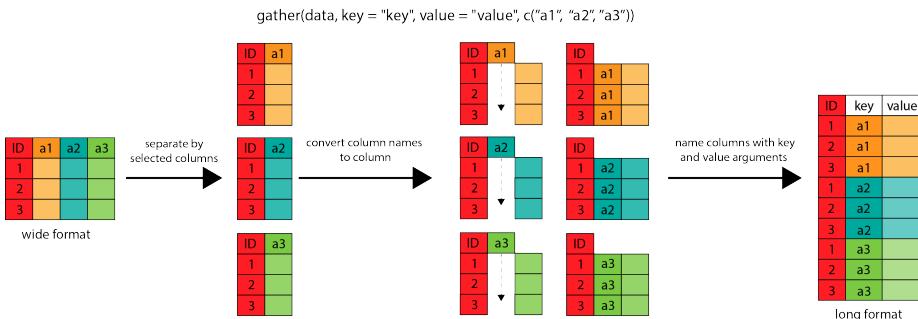
```
gap_wide <- read.csv("data/gapminder_wide.csv", stringsAsFactors = FALSE)
str(gap_wide)

## 'data.frame': 142 obs. of 38 variables:
## $ continent      : chr "Africa" "Africa" "Africa" "Africa" ...
## $ country       : chr "Algeria" "Angola" "Benin" "Botswana" ...
## $ gdpPercap_1952: num 2449 3521 1063 851 543 ...
## $ gdpPercap_1957: num 3014 3828 960 918 617 ...
## $ gdpPercap_1962: num 2551 4269 949 984 723 ...
## $ gdpPercap_1967: num 3247 5523 1036 1215 795 ...
## $ gdpPercap_1972: num 4183 5473 1086 2264 855 ...
## $ gdpPercap_1977: num 4910 3009 1029 3215 743 ...
## $ gdpPercap_1982: num 5745 2757 1278 4551 807 ...
## $ gdpPercap_1987: num 5681 2430 1226 6206 912 ...
## $ gdpPercap_1992: num 5023 2628 1191 7954 932 ...
## $ gdpPercap_1997: num 4797 2277 1233 8647 946 ...
## $ gdpPercap_2002: num 5288 2773 1373 11004 1038 ...
## $ gdpPercap_2007: num 6223 4797 1441 12570 1217 ...
## $ lifeExp_1952   : num 43.1 30 38.2 47.6 32 ...
## $ lifeExp_1957   : num 45.7 32 40.4 49.6 34.9 ...
## $ lifeExp_1962   : num 48.3 34 42.6 51.5 37.8 ...
## $ lifeExp_1967   : num 51.4 36 44.9 53.3 40.7 ...
## $ lifeExp_1972   : num 54.5 37.9 47 56 43.6 ...
```

```
## $ lifeExp_1977 : num 58 39.5 49.2 59.3 46.1 ...
## $ lifeExp_1982 : num 61.4 39.9 50.9 61.5 48.1 ...
## $ lifeExp_1987 : num 65.8 39.9 52.3 63.6 49.6 ...
## $ lifeExp_1992 : num 67.7 40.6 53.9 62.7 50.3 ...
## $ lifeExp_1997 : num 69.2 41 54.8 52.6 50.3 ...
## $ lifeExp_2002 : num 71 41 54.4 46.6 50.6 ...
## $ lifeExp_2007 : num 72.3 42.7 56.7 50.7 52.3 ...
## $ pop_1952 : num 9279525 4232095 1738315 442308 4469979 ...
## $ pop_1957 : num 10270856 4561361 1925173 474639 4713416 ...
## $ pop_1962 : num 11000948 4826015 2151895 512764 4919632 ...
## $ pop_1967 : num 12760499 5247469 2427334 553541 5127935 ...
## $ pop_1972 : num 14760787 5894858 2761407 619351 5433886 ...
## $ pop_1977 : num 17152804 6162675 3168267 781472 5889574 ...
## $ pop_1982 : num 20033753 7016384 3641603 970347 6634596 ...
## $ pop_1987 : num 23254956 7874230 4243788 1151184 7586551 ...
## $ pop_1992 : num 26298373 8735988 4981671 1342614 8878303 ...
## $ pop_1997 : num 29072015 9875024 6066080 1536536 10352843 ...
## $ pop_2002 : int 31287142 10866106 7026113 1630347 12251209 7021078 15929988 4048013 88
## $ pop_2007 : int 33333216 12420476 8078314 1639131 14326203 8390505 17696293 4369038 10
```

| wide format | | | | | | | | | | | | |
|-------------|---------|----------------|----------------|---------------|--------------|--------------|----------|----------|---------|-----|-----|-----|
| continent | country | gdpPercap_1952 | gdpPercap_1957 | gdpPercap_... | lifeExp_1952 | lifeExp_1957 | pop_1952 | pop_1957 | pop_... | ... | ... | ... |
| Africa | Algeria | | | | | | | | | | | |
| Africa | Angola | | | | | | | | | | | |
| ... | ... | | | | | | | | | | | |

깔끔한 중간 데이터 형식을 얻는 첫단추는 먼저 ‘wide’ 형식에서 ‘long’ 형식으로 변환하는 것이다. `tidyverse` 팩키지 `gather()` 함수는 관측 변수를 모아서(gather) 단일 변수로 변환한다.



```
gap_long <- gap_wide %>%
  gather(obs_type_year, obs_values, starts_with('pop'),
         starts_with('lifeExp'), starts_with('gdpPercap'))
str(gap_long)

## 'data.frame': 5112 obs. of 4 variables:
## $ continent : chr "Africa" "Africa" "Africa" "Africa" ...
```

```
## $ country      : chr  "Algeria" "Angola" "Benin" "Botswana" ...
## $ obstype_year: chr  "pop_1952" "pop_1952" "pop_1952" "pop_1952" ...
## $ obs_values   : num  9279525 4232095 1738315 442308 4469979 ...
```

위에서 파이프 구문을 사용했는데, 이전 수업에서 dplyr로 작업한 것과 유사하다. 사실, dplyr과 tidyverse는 상호 호환되어, 파이프 구문으로 dplyr과 tidyverse 팩키지 함수를 섞어 사용한다.

gather() 함수 내부에, 먼저 신규 ID 변수(obs_type_year)에 대한 명칭, 병합된 관측변수(obs_value)에 대한 명칭, 그리고 나서 이전 관측변수에 대한 명칭을 신규 칼럼명으로 부여한다. 모든 관측변수를 타이핑했지만, select() 함수처럼(dplyr 수업 참조), starts_with() 함수에 인자로 넣어, 원하는 문자열로 시작되는 모든 변수를 선택할 수도 있다. gather() 모음 필요없는 변수(예를 들어, ID 변수)를 식별하는데, - 기호를 사용하는 구문도 지원한다.

long format

| continent | country | obstype_year | obs_value |
|-----------|---------|----------------|-----------|
| Africa | Algeria | gdpPercap_1952 | |
| Africa | Algeria | gdpPercap_1957 | |
| Africa | Algeria | gdpPercap_... | |
| Africa | Algeria | lifeExp_1952 | |
| Africa | Algeria | lifeExp_1957 | |
| Africa | Algeria | lifeExp_... | |
| Africa | Algeria | pop_1952 | |
| Africa | Algeria | pop_1957 | |
| Africa | Algeria | pop_... | |
| Africa | Angola | gdpPercap_1952 | |
| Africa | Angola | gdpPercap_1957 | |
| Africa | Angola | gdpPercap_... | |
| Africa | Angola | lifeExp_1952 | |
| Africa | Angola | lifeExp_1957 | |
| Africa | Angola | lifeExp_... | |
| Africa | Angola | pop_1952 | |
| Africa | Angola | pop_1957 | |
| Africa | Angola | pop_... | |
| Africa | ... | gdpPercap_1952 | |
| Africa | ... | gdpPercap_1957 | |
| Africa | ... | gdpPercap_... | |
| Africa | ... | lifeExp_1952 | |
| Africa | ... | lifeExp_1957 | |
| Africa | ... | lifeExp_... | |
| Africa | ... | pop_1952 | |
| Africa | ... | pop_1957 | |
| Africa | ... | pop_... | |

```
gap_long <- gap_wide %>% gather(obstype_year,obs_values,-continent,-country)
str(gap_long)

## # 'data.frame': 5112 obs. of 4 variables:
##   $ continent : chr "Africa" "Africa" "Africa" "Africa" ...
##   $ country   : chr "Algeria" "Angola" "Benin" "Botswana" ...
##   $ obstype_year: chr "gdpPercap_1952" "gdpPercap_1952" "gdpPercap_1952" "gdpPercap_1952" ...
##   $ obs_values : num 2449 3521 1063 851 543 ...
```

이런 특수한 데이터프레임에는 별거 없어 보일 수도 있다. 하지만, ID 변수 하나, 규칙없는 변수명 관측변수 40개를 갖는 경우가 있을 수 있다. 이런 유연성은 시간을 상당히 절약해 준다!

이제 `obstype_year`은 정보가 두조각으로 나뉜다: 관측 유형(`pop,lifeExp, gdpPercap`)과 연도(`year`). `separate()` 함수를 사용해서 문자열을 다수 변수로 쪼갠다.

```
gap_long <- gap_long %>% separate(obstype_year,into=c('obs_type','year'),sep="_")
gap_long$year <- as.integer(gap_long$year)
```

34.4 도전 과제 2

`gap_long`을 사용해서, 각 대륙별로 평균 기대수명, 인구, 1인당 GDP를 계산한다.
힌트: `dplyr` 수업에서 학습한 `group_by()` 와 `summarize()` 함수를 사용한다.

도전과제 2에 대한 해답

```
gap_long %>% group_by(continent,obs_type) %>%
  summarize(means=mean(obs_values))

## # A tibble: 15 x 3
## # Groups:   continent [5]
##   continent obs_type     means
##   <chr>      <chr>       <dbl>
## 1 Africa     gdpPercap    2194.
## 2 Africa     lifeExp      48.9 
## 3 Africa     pop          9916003.
## 4 Americas   gdpPercap    7136.
## 5 Americas   lifeExp      64.7 
## 6 Americas   pop          24504795.
## # ... with 9 more rows
```

34.5 `spread()`로 ‘long’ 형식 변환

이제 작업을 이중점검 하도록, `gather()` 역함수(적절히 작성된 `spread()`)를 사용해서 관측변수를 다시 되돌린다. 그리고 나면 `gap_long`을 원래 중간 형식 혹은

'wide' 형식으로 퍼뜨린다. 중간 형식에서부터 시작해보자.

```
gap_normal <- gap_long %>% spread(obs_type,obs_values)
dim(gap_normal)

## [1] 1704     6
dim(gapminder)

## [1] 1704     6
names(gap_normal)

## [1] "continent" "country"    "year"        "gdpPercap"  "lifeExp"    "pop"
names(gapminder)

## [1] "country"    "year"        "pop"         "continent"   "lifeExp"    "gdpPercap"

이제, 최초 데이터프레임 gapminder와 동일한 차원을 갖는 중간 데이터프레임
gap_normal이 있다. 하지만, 변수 순서가 다르다. 순서를 수정하기 전에,
all.equal() 함수를 사용해서 동일한지 확인한다.

gap_normal <- gap_normal[,names(gapminder)]
all.equal(gap_normal,gapminder)

## [1] "Component \"country\": 1704 string mismatches"
## [2] "Component \"pop\": Mean relative difference: 1.63"
## [3] "Component \"continent\": 1212 string mismatches"
## [4] "Component \"lifeExp\": Mean relative difference: 0.204"
## [5] "Component \"gdpPercap\": Mean relative difference: 1.16"
head(gap_normal)

##   country year      pop continent lifeExp gdpPercap
## 1 Algeria 1952 9279525     Africa   43.1    2449
## 2 Algeria 1957 10270856     Africa   45.7    3014
## 3 Algeria 1962 11000948     Africa   48.3    2551
## 4 Algeria 1967 12760499     Africa   51.4    3247
## 5 Algeria 1972 14760787     Africa   54.5    4183
## 6 Algeria 1977 17152804     Africa   58.0    4910
head(gapminder)

##   country year      pop continent lifeExp gdpPercap
## 1 Afghanistan 1952 8425333     Asia    28.8    779
## 2 Afghanistan 1957 9240934     Asia    30.3    821
## 3 Afghanistan 1962 10267083     Asia    32.0    853
## 4 Afghanistan 1967 11537966     Asia    34.0    836
## 5 Afghanistan 1972 13079460     Asia    36.1    740
## 6 Afghanistan 1977 14880372     Asia    38.4    786
```

거의 다 왔다. 최초 데이터프레임은 country, continent, year 순으로 정렬되었다.

```
gap_normal <- gap_normal %>% arrange(country,continent,year)
all.equal(gap_normal,gapminder)
```

```
## [1] TRUE
```

훌륭하다! ‘long’ 형식에서 다시 중간 형식으로 돌아갔지만, 코드에 어떤 오류도 스며들지 않았다.

이제, ‘long’ 형식을 ‘wide’ 형식으로 변환하자. ‘wide’ 형식에서, country 국가와 continent 대륙을 ID 변수로 두고, 관측점을 3가지 측정값(pop,lifeExp,gdpPercap)과 시간(year)으로 쭉 뿐였다. 먼저, 모든 신규 변수(측정값 * 시간)에 대한 적절한 라벨을 생성할 필요가 있다. 또한, ID변수를 합쳐서 gap_wide를 정의하는 과정을 단순화할 필요가 있다.

```
gap_temp <- gap_long %>% unite(var_ID,continent,country,sep="_")
str(gap_temp)
```

```
## 'data.frame': 5112 obs. of 4 variables:
## $ var_ID    : chr "Africa_Algeria" "Africa_Angola" "Africa_Benin" "Africa_Botswana"
## $ obs_type   : chr "gdpPercap" "gdpPercap" "gdpPercap" "gdpPercap" ...
## $ year       : int 1952 1952 1952 1952 1952 1952 1952 1952 1952 ...
## $ obs_values: num 2449 3521 1063 851 543 ...
gap_temp <- gap_long %>%
  unite(ID_var,continent,country,sep="_") %>%
  unite(var_names,obs_type,year,sep="_")
str(gap_temp)
```

```
## 'data.frame': 5112 obs. of 3 variables:
## $ ID_var    : chr "Africa_Algeria" "Africa_Angola" "Africa_Benin" "Africa_Botswana"
## $ var_names  : chr "gdpPercap_1952" "gdpPercap_1952" "gdpPercap_1952" "gdpPercap_1952" ...
## $ obs_values: num 2449 3521 1063 851 543 ...
```

unite()를 사용해서, continent,country를 묶는 ID변수가 하나 생겼고, 변수명을 정의했다. 이제 파이프를 통해서 spread() 뿐만 아니라, 정의한 변수명을 정의했다. 이제 파이프를 통해서 spread() 뿐만 아니라, 정의한 변수명을 정의했다.

```
gap_wide_new <- gap_long %>%
  unite(ID_var,continent,country,sep="_") %>%
  unite(var_names,obs_type,year,sep="_") %>%
  spread(var_names,obs_values)
str(gap_wide_new)
```

```
## 'data.frame': 142 obs. of 37 variables:
## $ ID_var      : chr "Africa_Algeria" "Africa_Angola" "Africa_Benin" "Africa_Botswana"
## $ gdpPercap_1952: num 2449 3521 1063 851 543 ...
## $ gdpPercap_1957: num 3014 3828 960 918 617 ...
## $ gdpPercap_1962: num 2551 4269 949 984 723 ...
## $ gdpPercap_1967: num 3247 5523 1036 1215 795 ...
```

```

## $ gdpPercap_1972: num 4183 5473 1086 2264 855 ...
## $ gdpPercap_1977: num 4910 3009 1029 3215 743 ...
## $ gdpPercap_1982: num 5745 2757 1278 4551 807 ...
## $ gdpPercap_1987: num 5681 2430 1226 6206 912 ...
## $ gdpPercap_1992: num 5023 2628 1191 7954 932 ...
## $ gdpPercap_1997: num 4797 2277 1233 8647 946 ...
## $ gdpPercap_2002: num 5288 2773 1373 11004 1038 ...
## $ gdpPercap_2007: num 6223 4797 1441 12570 1217 ...
## $ lifeExp_1952 : num 43.1 30 38.2 47.6 32 ...
## $ lifeExp_1957 : num 45.7 32 40.4 49.6 34.9 ...
## $ lifeExp_1962 : num 48.3 34 42.6 51.5 37.8 ...
## $ lifeExp_1967 : num 51.4 36 44.9 53.3 40.7 ...
## $ lifeExp_1972 : num 54.5 37.9 47 56 43.6 ...
## $ lifeExp_1977 : num 58 39.5 49.2 59.3 46.1 ...
## $ lifeExp_1982 : num 61.4 39.9 50.9 61.5 48.1 ...
## $ lifeExp_1987 : num 65.8 39.9 52.3 63.6 49.6 ...
## $ lifeExp_1992 : num 67.7 40.6 53.9 62.7 50.3 ...
## $ lifeExp_1997 : num 69.2 41 54.8 52.6 50.3 ...
## $ lifeExp_2002 : num 71 41 54.4 46.6 50.6 ...
## $ lifeExp_2007 : num 72.3 42.7 56.7 50.7 52.3 ...
## $ pop_1952 : num 9279525 4232095 1738315 442308 4469979 ...
## $ pop_1957 : num 10270856 4561361 1925173 474639 4713416 ...
## $ pop_1962 : num 11000948 4826015 2151895 512764 4919632 ...
## $ pop_1967 : num 12760499 5247469 2427334 553541 5127935 ...
## $ pop_1972 : num 14760787 5894858 2761407 619351 5433886 ...
## $ pop_1977 : num 17152804 6162675 3168267 781472 5889574 ...
## $ pop_1982 : num 20033753 7016384 3641603 970347 6634596 ...
## $ pop_1987 : num 23254956 7874230 4243788 1151184 7586551 ...
## $ pop_1992 : num 26298373 8735988 4981671 1342614 8878303 ...
## $ pop_1997 : num 29072015 9875024 6066080 1536536 10352843 ...
## $ pop_2002 : num 31287142 10866106 7026113 1630347 12251209 ...
## $ pop_2007 : num 33333216 12420476 8078314 1639131 14326203 ...

```

34.6 도전과제 3

한 걸음 더 나아가, 국가, 연도, 측정값 3개를 쭉 뿌려 터무니 없는 `gap_ludicrously_wide` 형식 데이터를 생성한다. **힌트:** 신규 데이터프레임은 단지 행이 5개만 있다.

도전과제 3에 대한 해답

```

gap_ludicrously_wide <- gap_long %>%
  unite(var_names, obs_type, year, country, sep="_") %>%
  spread(var_names, obs_values)

```

이제, 엄청 ‘wide’ 형식 데이터프레임이 생겼다. 하지만, `ID_var` 변수는 더 유용할 수 있다. `separate()` 함수로 변수 2개로 구분하자.

```

gap_wide_betterID <- separate(gap_wide_new, ID_var, c("continent", "country"), sep="_")
gap_wide_betterID <- gap_long %>%
  unite(ID_var, continent, country, sep="_") %>%
  unite(var_names, obs_type, year, sep="_") %>%
  spread(var_names, obs_values) %>%
  separate(ID_var, c("continent", "country"), sep="_")
str(gap_wide_betterID)

## 'data.frame':   142 obs. of  38 variables:
## $ continent      : chr  "Africa" "Africa" "Africa" "Africa" ...
## $ country        : chr  "Algeria" "Angola" "Benin" "Botswana" ...
## $ gdpPercap_1952: num  2449 3521 1063 851 543 ...
## $ gdpPercap_1957: num  3014 3828 960 918 617 ...
## $ gdpPercap_1962: num  2551 4269 949 984 723 ...
## $ gdpPercap_1967: num  3247 5523 1036 1215 795 ...
## $ gdpPercap_1972: num  4183 5473 1086 2264 855 ...
## $ gdpPercap_1977: num  4910 3009 1029 3215 743 ...
## $ gdpPercap_1982: num  5745 2757 1278 4551 807 ...
## $ gdpPercap_1987: num  5681 2430 1226 6206 912 ...
## $ gdpPercap_1992: num  5023 2628 1191 7954 932 ...
## $ gdpPercap_1997: num  4797 2277 1233 8647 946 ...
## $ gdpPercap_2002: num  5288 2773 1373 11004 1038 ...
## $ gdpPercap_2007: num  6223 4797 1441 12570 1217 ...
## $ lifeExp_1952  : num  43.1 30 38.2 47.6 32 ...
## $ lifeExp_1957  : num  45.7 32 40.4 49.6 34.9 ...
## $ lifeExp_1962  : num  48.3 34 42.6 51.5 37.8 ...
## $ lifeExp_1967  : num  51.4 36 44.9 53.3 40.7 ...
## $ lifeExp_1972  : num  54.5 37.9 47 56 43.6 ...
## $ lifeExp_1977  : num  58 39.5 49.2 59.3 46.1 ...
## $ lifeExp_1982  : num  61.4 39.9 50.9 61.5 48.1 ...
## $ lifeExp_1987  : num  65.8 39.9 52.3 63.6 49.6 ...
## $ lifeExp_1992  : num  67.7 40.6 53.9 62.7 50.3 ...
## $ lifeExp_1997  : num  69.2 41 54.8 52.6 50.3 ...
## $ lifeExp_2002  : num  71 41 54.4 46.6 50.6 ...
## $ lifeExp_2007  : num  72.3 42.7 56.7 50.7 52.3 ...
## $ pop_1952     : num  9279525 4232095 1738315 442308 4469979 ...
## $ pop_1957     : num  10270856 4561361 1925173 474639 4713416 ...
## $ pop_1962     : num  11000948 4826015 2151895 512764 4919632 ...
## $ pop_1967     : num  12760499 5247469 2427334 553541 5127935 ...
## $ pop_1972     : num  14760787 5894858 2761407 619351 5433886 ...
## $ pop_1977     : num  17152804 6162675 3168267 781472 5889574 ...
## $ pop_1982     : num  20033753 7016384 3641603 970347 6634596 ...
## $ pop_1987     : num  23254956 7874230 4243788 1151184 7586551 ...
## $ pop_1992     : num  26298373 8735988 4981671 1342614 8878303 ...
## $ pop_1997     : num  29072015 9875024 6066080 1536536 10352843 ...

```

```
## $ pop_2002      : num  31287142 10866106 7026113 1630347 12251209 ...
## $ pop_2007      : num  33333216 12420476 8078314 1639131 14326203 ...
all.equal(gap_wide, gap_wide_betterID)
## [1] TRUE
다시 되돌아 왔다!
```

34.7 추가 학습

- R for Data Science
- Data Wrangling Cheat sheet
- Introduction to tidyverse
- Data wrangling with R and RStudio

Chapter 35

knitr 보고서 생성

35.1 데이터 분석 보고서

데이터 분석가는 동료 혹은 향후 참고자료로 자신의 작업결과를 문서화하는데 상당량의 보고서를 작성해서 분석과정과 출력결과를 기술한다.

처음 작업을 시작할 때, 온전히 본인 작업을 위해서 R 스크립트를 작성한다. 그리고 나서, 다양한 그래프가 첨부된 분석결과를 기술해서 동료에게 전자우편을 발송한다. 분석결과를 논의하는 과정에서 어느 그래프를 지칭하지에 대해 혼란이 종종 수반된다.

워드나 LaTeX으로 좀더 정형화된 보고서로 옮겨가도, 그림이 올바르게 보이도록 만드는데 상당한 시간을 소비한다. 대부분 페이지 나누기가 문제가 된다.

웹페이지(html 파일)를 생성하게 되면 모든 것이 훨씬 수월해진다. 이유는 하나의 긴 흐름이 되기 때문이다. 따라서, 한 페이지에 맞지 않는 긴 그림도 사용할 수 있게 된다. 스크롤링이 딱맞는 친구다.

35.2 문학적 프로그래밍

이상적으로 그런 분석 보고서가 재현가능한 문서가 된다: 만약 오류가 발견되거나, 추가적인 분석주제가 데이터에 추가되면, 다시 재컴파일 하게 되면, 신규 혹은 수정된 결과를 얻게 된다. (반대로, 워드나 한글 등 오피스로 작업하게 되면 그림을 다시 생성하고나서 문서에 붙여넣고 수작업으로 상세결과를 편집해야 한다.)

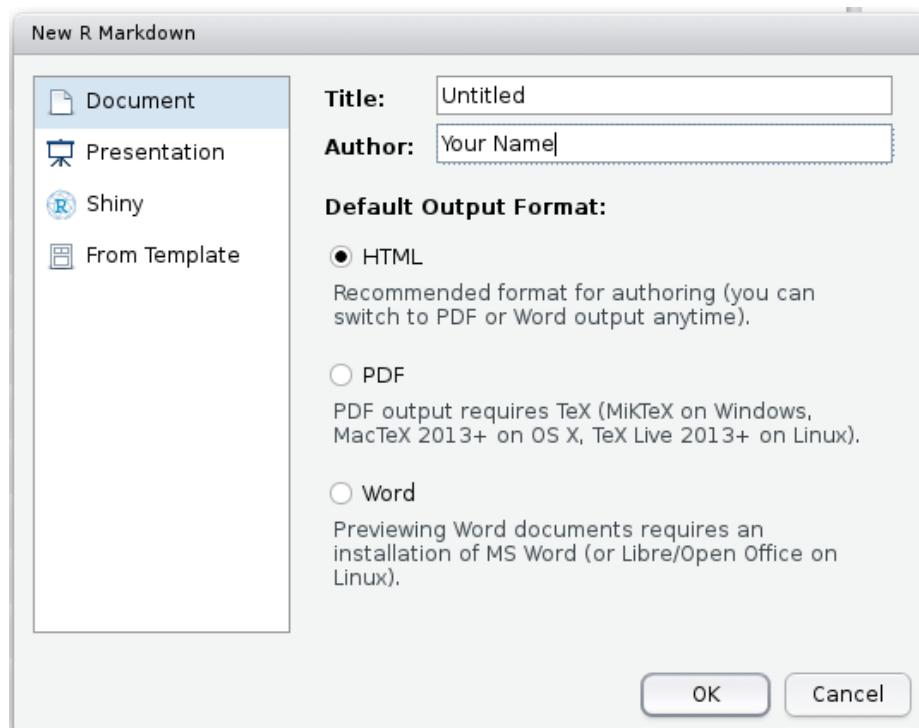
R에서 문학적 프로그래밍(literate programming)을 구현하는 주된 도구가 knitr다. knitr는 텍스트와 R 코드가 뒤섞인 문서를 생성할 수 있게 하는 역할을 수행한다. 문서가 knitr로 처리되면, R 코드가 실행되어 그래프와 분석결과가 문서에 삽입된다.

이런 유형의 아이디어를 문학적 프로그래밍(literate programming)이라고 부른다.

*knitr*는 R코드 뿐만 아니라 다양한 프로그래밍 언어를 텍스트와 뒤섞을 수 있도록 하지만, 기본적으로 R 마크다운을 추천한다. R 마크다운은 마크다운과 R코드를 섞어 쓸 수 있도록 한다. 마크다운은 가벼운 마크업 언어로 웹페이지를 생성하는데 사용된다.

35.3 R 마크다운 파일 생성

RStudio에서 File → New File → R Markdown을 클릭하면 다음과 같은 대화상자가 열린다:



기본설정(HTML 출력)된 대로 사용할 수도 있지만, 제목을 줄 수도 있다.

35.4 R 마크다운 구성요소

초기 설정된 텍스트 덩어리에 R에 대한 지시사항이 담겨있다: 제목, 저자명, 날짜, html 출력물(다른 말로, 웹페이지)을 생성할지가 포함된다.

```
---
title: "Initial R Markdown document"
author: "Karl Broman"
date: "April 23, 2015"
```

```
output: html_document
---
```

원하지 않는 경우, 상기 필드를 임의로 삭제할 수 있다. 엄밀히 말해서 인용부호는 상기 경우에 꼭 필요한 것은 아니다. 제독에 콜론(:)이 포함되는 경우 대체로 인용부호가 필요하다.

RStudio에 시작을 도와주도록 예제가 문서에 포함되어 있다. 다음과 같은 R 코드 덩어리가 포함되어 있는 점을 주목한다:

`knitr`로 실행되는 R 코드 덩어리로 R 코드 실행결과로 치환된다. 뒤에서 더 다루게 된다.

또한, 웹주소는 < > 꺠쇠 기호로 담아내고, ****Knit****처럼 별표 두개를 사용하기도 한다. 이것이 전형적인 Markdown 구문의 한 사례가 된다.

35.5 마크다운(Markdown)

HTML 코드로 작성하는 대신에 전자우편에서 문서를 작성하는 것처럼 텍스트에 마크업(markup)을 적용해서 웹페이지를 저작하는 시스템이 마크다운이다. 마크업 텍스트는 적절한 HTML코드로 치환되는 과정을 거쳐서 최종 HTML로 변환된다.

지금은 자동생성된 모든 코드를 삭제하고, 마크다운으로 저작을 시작해보자.

별표 두개를 사용해서 **굵게(bold)**할 수 있는데 **** (bold)**** 텍스트를 타이핑하면 된다. 밑줄 혹은 별표 한개를 사용해서 이탤릭 도 구현가능한데, _ 텍스트를 타이핑하면 된다.

다음과 같이 하이픈 혹은 별표를 적용해서 블릿 기호가 붙은 항목을 생성할 수 있다:

```
* bold with double-asterisks
* italics with underscores
* code-type font with backticks
```

혹은 다음과 같아도 가능하다:

```
- bold with double-asterisks
- italics with underscores
- code-type font with backticks
```

다음과 같이 웹페이지로 보여지게 된다:

- bold with double-asterisks
- italics with underscores
- code-type font with backticks

숫자를 사용하면 숫자가 붙은 항목도 생성이 가능하다. 원하는 만큼 동일한 숫자를 반복해서 적용하면 된다:

1. bold with double-asterisks
1. italics with underscores
1. code-type font with backticks

다음과 같이 보이게 된다.

1. bold with double-asterisks
2. italics with underscores
3. code-type font with backticks

기호를 각 라인 첫번째 적용하게 되면 다른 크기를 갖는 섹션 제목을 만들 수 있다:

```
#      (Title)
##    (Main section)
###   (Sub-section)
####    (Sub-sub section)
```

좌측 상단에 “Knit HTML” 버튼을 클릭하면 R 마크다운 문서를 HTML 웹페이지로 _컴파일_하게 된다. 바로 옆에 작은 물음표가 있음에 주목한다; 클릭하게 되면 “Markdown Quick Reference” (마크다운 구문) 뿐만 아니라 RStudio IDE에서 R 마크다운 문서도 참고할 수 있다.

35.6 도전과제

R 마크다운 문서를 생성한다. 모든 R 코드 덩어리를 삭제하고 마크다운 문서를 작성한다. (제목, 이탤릭 텍스트, 블릿 기호가 붙은 항목 등) 문서를 웹페이지로 변환시킨다.

도전과제에 대한 해답

RStudio에서 File > New file > R Markdown... 을 선택한다.
누름틀(Placeholder) 텍스트를 삭제하고 다음과 같이 작성한다.

```
# Introduction

## Background on Data

This report uses the *gapminder* dataset, which has columns that include:

* country
* continent
* year
* lifeExp
* pop
* gdpPercap

## Background on Methods
```

35.7 마크다운 추가기능

[text to show] (<http://the-web-page.com>)처럼 하이퍼링크를 걸 수도 있다.

`! [caption] (http://url/for/file)` 처럼 이미지를 삽입할 수도 있다.

F_2 처럼 아래첨자를 $F \sim 2$ 와 같이 넣을 수도 있고, F^2 처럼 윗첨자를 $F^{2\sim}$ 와 같이 넣을 수도 있다.

수식 작성법은 LaTeX을 참조해서 작성할 수 있는데, $\$ \$$ 와 $\$ \$ \$$ 을 사용해서 수식을 넣을 수 있다: $\$E = mc^2\$$.

```
$$y = \mu + \sum_{i=1}^p \beta_i x_i + \epsilon$$
```

35.8 R 코드 덩어리

마크다운은 흥미롭고 유용하기도 하지만, 실제 진정한 힘은 마크다운을 R 코드 덩어리와 뒤섞어 사용할 때 나온다. R 마크다운으로 말이다. R 마크다운이 처리되면서 R 코드도 실행된다; 그림이 생성되는 경우, 최종 문서에 자동으로 그림이 삽입되어 들어간다.

데이터를 불러오는 R 코드는 다음과 같이 생겼다:

즉, “`{r chunk_name}`” 사이에 R 코드 덩어리를 위치한다. R 코드 덩어리에 명칭을 부여하는 것이 좋은데 이유는 오류 수정을 유용하게 하고, 그래프가 생성되는 경우 파일명이 R 코드 덩어리에 지정한 명칭이 따라 붙기 때문이다.

35.9 도전과제

다음 R 코드를 추가한다.

- `ggplot2` 팩키지 불러오는 코드
- `gapminder` 데이터를 불러오는 코드
- 그래프를 생성하는 코드

도전과제 해답

```
```{r load-ggplot2}
library("ggplot2")
```

```{r read-gapminder-data}
gapminder <- read.csv("gapminder.csv")
```

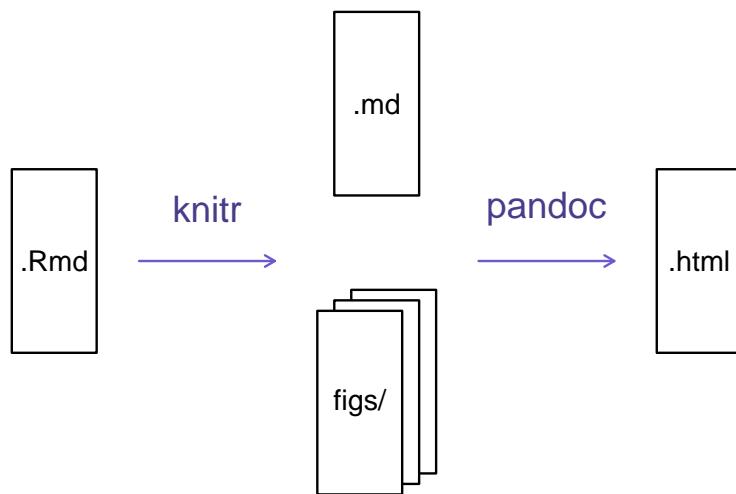
```{r make-plot}
plot(lifeExp ~ year, data = gapminder)
```

```

35.10 컴파일 작동방식

“Knit HTML” 버튼을 생성하게 되면, knitr 프로그램이 R 마크다운 문서를 처리해서, 일반 마크다운 문서가 생성된다(그림도 포함된다.): R 코드가 실행되면서 입력과 출력 모두에 대해 치환된다; 그림이 생성되는 경우, 그림에 대한 링크도 포함된다.

마크다운과 그림 문서는 pandoc 도구로 처리되어서, 마크다운 파일을 그림이 내장된 HTML 파일로 탈바꿈하게 된다.



35.11 덩어리 선택옵션

코드 덩어리(Chunk)가 어떻게 처리되는 방법에 대한 선택옵션이 다수 지원된다.

- echo=FALSE: R 코드가 출력되는 것을 방지.
- results="hide": 출력결과물이 문서에 뿌려지지 않도록 방지.
- eval=FALSE: 코드는 출력되지만, 평가되어 실행되는 것은 방지.
- warning=FALSE, message=FALSE: 경고와 메시지 출력을 숨김.
- fig.height, fig.width: 인치 단위로 그림 크기 높이와 폭을 지정.

다음과 같이 작성할 수도 있다:

전역 덩어리(chunk) 선택옵션을 사용하여 반복적으로 사용할 R마크다운 선택옵션을 다음과 같이 지정하는 것도 가능하다:

`fig.path` 선택옵션은 그림이 저장되는 장소를 지정하는데 사용한다. /이 중요한데 이유는 /을 지정하지 않으면 그림이 Figs으로 시작하는 파일명으로 표준 저장소에 저장된다.

작업 디렉토리에 R 마크다운 파일이 다수 존재하는 경우, `fig.path`를 사용해서

`fig.path="Figs/cleaning-", fig.path="Figs/analysis-"`처럼 그림 파일명에 접두어를 달리 지정할 수도 있다.

35.12 도전과제

그림 크기를 변경하고, 코드를 출력에서 감추도록, R 코드 덩어리 선택옵션을 설정해 본다.

도전과제 해답

```
```{r echo = FALSE, fig.width = 3}
plot(faithful)
```

```

35.13 인라인 R 코드

보고서의 모든 숫자를 재현가능하게 만들 수 있다. 인라인 코드를 작성할 때 ‘`r`’와 ‘`을`’ 사용한다. 예를 들어, ‘`r round(some_value, 2)`’. R 코드가 실행되어 코드가 결과 값으로 치환된다.

두줄이상에 걸쳐 인라인 코드 덩어리가 나눠지도록 작성하지는 말자.

아마도, 조금 큰 코드 덩어리의 경우 `include=FALSE`(`echo=FALSE`와 `results="hide"` 두가지 기능을 함께 보유)를 사용해서 연산과 정의를 처리하는 것도 가능하다.

이런 경우에 소수점 자리수에 대해서 2.0을 원하지만, `round(2.03, 1)` 코드는 2가 된다.

R/broman 팩키지에서 `myround` 함수가 도움이 될 수 있다.

35.14 도전과제

인라인 R 코드를 작성해 보자.

도전과제 해답

2 + 2 를 계산하는 인라인 R 코드: `2 + 2 = `r 2+2``.

35.15 기타 출력 선택옵션

R 마크다운 문서를 PDF나 워드 문서로 변환할 수도 있다. “Knit HTML” 버튼 옆에 작은 삼각형을 클릭하면 드롭다운 메뉴가 나타난다. 혹은 R 마크다운 `.Rmd` 파일의 헤더부분에 `pdf_document`, `word_document`으로 설정하면 된다.

꿀팁: PDF 문서 생성하기

- .pdf 문서를 생성하려면 추가로 소프트웨어를 설치해야 한다. 소프트웨어 설치 없이 .pdf 문서를 생성하게 되면 오류 메시지에 자세한 사항이 기술되어 있다.
 - TeX installers for Windows.
 - TeX installers for macOS.

35.16 관련 자료

- Knitr in a knutshell tutorial
- Dynamic Documents with R and knitr (book)
- R Markdown documentation
- R Markdown cheat sheet
- Getting started with R Markdown
- Reproducible Reporting
- The Ecosystem of R Markdown
- Introducing Bookdown

Chapter 36

좋은 소프트웨어 작성법

36.1 프로젝트 폴더 구조화

작업 프로젝트 폴더를 구조화

하위 폴더를 코드, 매뉴얼, 데이터, 바이너리, 출력 그래프 등으로 구분하여 프로젝트 폴더를 구조화시키고, 잘 조직화하고, 깔끔하게 한다. 완전 수작업으로 할 수도 있고, RStudio New Project 기능을 활용하거나 ProjectTemplate 같은 팩키지를 사용한다.

꿀팁: ProjectTemplate - 가능한 해결책

프로젝트 관리를 자동화하는 한 방식은 제3자 팩키지, ProjectTemplate을 설치하는 것이다. 해당 팩키지는 프로젝트 관리에 대한 이상적인 디렉토리 구조를 설정해 놓는다. 팩키지가 자동으로 분석 파일이나 작업흐름을 구성해서 구조화해 놓는다. RStudio 기본설정된 프로젝트 관리 기능과 Git을 섞어 사용하면, 작업을 기록할 뿐만 아니라, 동료 연구원과 작업산출물 공유를 가능케 한다.

1. ProjectTemplate을 설치한다.
2. 라이브러리를 불러 적재한다.
3. 프로젝트를 초기화한다.

```
install.packages("ProjectTemplate")
library(ProjectTemplate)
create.project("../my_project", merge.strategy = "allow.non.conflict")
```

ProjectTemplate과 기능에 대한 자세한 사항은 ProjectTemplate 홈페이지를 방문한다.

36.2 가독성 높은 코드 생성

코드 작성에 있어 가장 중요한 부분이 코드를 가독성 있고 이해가능하게 작성하는 것이다. 누군가 여러분이 작성한 코드를 골라 무슨 작업을 수행하는지 이해할 수 있어야 한다: 흔히 누군가는 6개월 후에 바로 당신이 될 수 있고, 만약 그렇게 작성하지 않았다면 과거 자기 자신 본인을 분명히 저주할 것이다.

36.3 문서화

문서화에서 왜(why) 그리고 무엇(what)은 OK, 어떻게(how)는 No.

처음 코드를 작성할 때, 주석은 명령어가 무엇을 수행하는지 기술한다. 왜냐하면, 여전히 학습중으로 개념을 명확히 하고, 나중에 다시 상기하는데 도움이 된다. 하지만, 이러한 주석은 나중에 작성한 코드가 어떤 문제를 해결하고자 하는지 기억을 하지 못하면 그다지 도움이 되지 못한다. 왜(why) 문제를 해결하려고 하는지, 그리고 어떤(what) 문제인지 전달하는 주석을 달려고 노력한다. 어떻게(how)는 그 다음에 온다: 정말 걱정하지 말아야 되는 사항은 구체적인 구현이다.

36.4 코드를 모듈화

소프트웨어 카펜트리에서 추천하는 것은 작성한 함수를 분석 스크립트와 구별해서 별도 파일에 저장시키는 것이다. 프로젝트 R세션을 열 때, source 함수로 불러올 수 있게 별도 파일로 저장한다.

분석 스크립트를 너저분하지 않게 하고, 유용한 함수 저장소를 프로젝트 분석 스크립트에 적재할 수 있게 함으로써 이러한 접근법이 깔끔하다. 또한 관련된 함수를 쉽게 무리지어 묶는다.

36.5 문제를 잘게 쪼갠다

문제를 한입크기 조각으로 쪼갠다.

처음 시작할 때, 문제 해결과 함수 작성은 어마어마한 작업이고, 코드를 쪼개는 것도 힘들다. 문제를 소화가능한 덩어리로 쪼개고, 나중에 구현에 관한 구체적인 사항을 걱정한다: 해결책을 코드로 작성할 수 있는 지점까지 문제를 더 작게 그리고 더 작은 함수로 계속 쪼개 나간다. 그리고 나서 다시 거꾸로 빌드해서 만들어 낸다.

36.6 작성 코드 정상 수행

작성한 코드가 올바른 작업을 수행하도록 만든다. 즉, 작성한 함수를 테스트해서 확실히 동작하게 만든다.

36.7 사람은 반복 금지

함수는 프로젝트 내부에서 재사용을 쉽게 한다. 프로젝트를 통해서 유사한 코드 라인 덩어리를 보게 되면, 대체로 함수로 옮겨져야 되는 대상을 찾은 것이다.

연산작업이 연속된 함수를 통해 실행되면, 프로젝트는 모듈로 만들기 쉽고, 변경하기 쉽다. 항상 특정한 입력값을 넣으면 특정한 출력값이 나오는 경우에 특히 그렇다.

36.8 스타일 고집

코드에 일관된 스타일을 지킨다.

데이터베이스

Chapter 37

데이터베이스와 SQL 사용하기

거의 모든 사람이 스프레드시트(spreadsheet) 사용했고, 거의 모든 사람이 종국에는 한계에 맞닥뜨렸다. 데이터셋이 더욱 복잡할수록, 데이터를 걸러내고, 다른 행과 열 사이에 관계를 표현하거나, 결측값을 다루기가 점점 어려워진다.

데이터베이스는 스프레드시트가 멈춘 곳에서 다시 시작한다. 만약 사용하고자 하는 것이 10여개의 숫자의 합이라면 데이터베이스는 사용하기가 간단하지는 않지만, 훨씬 큰 데이터셋에 훨씬 더 빨리 스프레드시트가 할 수 없는 많은 것을 수행할 수 있다. 그리고, 설사 데이터베이스를 스스로 생성할 필요는 없지만, 데이터베이스가 어떻게 동작하는지 파악하는 것은 우리가 사용하는 수 많은 시스템이 왜 그와 같은 방식으로 동작하는지 그리고 왜 특정한 방식으로 데이터를 구조화하려고 하는지도 이해를 준다.

Chapter 38

SQLite 설치

이번 학습은 다음 장에서 사용되는 예제 데이터베이스를 어떻게 설치하는지 설명한다. 다음의 지도사항을 따르기 위해서는 명령-라인을 사용하여 어떻게 디렉토리를 여기저기 이동하는지와 명령-라인에서 명령문을 어떻게 실행하는지 숙지할 필요가 있다. 이런 주제와 친숙하지 않다면, **유닉스 쉘(Unix Shell)** 학습을 참조하세요. 이후의 장에서 데이터베이스를 어떻게 생성하고 데이터를 채우는지 배울 것이지만, 먼저 SQLite 데이터베이스가 어떻게 동작하는지 설명을 할 필요가 있어서 데이터베이스를 선행하여 제공한다.

38.1 설치

인터랙티브하게 다음 학습을 수행하기 위해서는 설치 방법에 언급된 SQLite 를 참조하여 설치하세요.

그리고 , 여러분이 선택한 위치에 “software_carpentry_sql” 디렉토리를 생성하세요. 예를 들어

- 1) 명령 라인 터미널 윈도우를 여세요.
- 2) 다음과 같이 타이핑한다.

```
mkdir ~/swc/sql
```

- 3) 생성한 디렉토리로 현재 작업 디렉토리를 변경한다.

```
cd ~/swc/sql
```

38.2 SQLite 설치 ¹

SQLite Download Page에서 sqlite-tools-win32-x86-3200000.zip을 다운로드

¹SQLite3 설치 및 간단한 사용법

받는다. 압축을 풀면 황당하게 몇개 .exe 파일이 존재하는 황당함을 느낀다. 설치가 완료되었다.

```
$ ls
gen-survey-database.sql    sqlite3.exe           survey.db
sqldiff.exe                sqlite3_analyzer.exe
```

- sqlite3.exe: sqlite 실행파일
- gen-survey-database.sql: survey.db sqlite 데이터베이스를 생성시키는데 사용되는 스크립트
- survey.db: sqlite3.exe 명령어를 실행해서 gen-survey-database.sql 스크립트를 통해 생성된 데이터베이스

38.3 실습 데이터베이스 다운로드

(GitHub)에서 gen-survey-database.sql 파일을 어떻게 다운로드 받을까요?

~/swc/sql 디렉토리로 이동한 후에 그 디렉토리에서 GitHub 사이트 <https://github.com/swcarpentry/bc/blob/master/novice/sql/gen-survey-database.sql> SQL에 위치한 SQL 파일("gen-survey-database.sql")을 다운로드한다.

파일이 GitHub 저장소 내에 위치하고 있어서, 전체 Git 저장소(git repo)를 복제(cloning)하지 않고 단일 파일만 로컬로 가져온다. 이 목적을 달성하기 위해서, HTTP, HTTPS, FTP 프로토콜을 지원하는 명령-라인 웹크롤러(web-crawler) 소프트웨어 GNU Wget 혹은, 다양한 프로토콜을 사용하여 데이터를 전송하는데 사용되는 라이브러리이며 명령-라인 도구인 curl을 사용한다. 두 가지 도구 모두 크로스 플랫폼(cross platform)으로 다양한 운영체제를 지원한다.

Wget 혹은 curl을 로컬에 설치한 후에, 터미널에서 다음 명령어를 실행한다.

Tip: 만약 curl을 선호한다면, 다음 명령문에서 "wget"을 curl -O로 대체하세요.

```
root@hangul:~/swc/sql$ wget https://raw.githubusercontent.com/swcarpentry/bc/master/novice/sql/gen-survey-database.sql
```

상기 명령문으로 Wget은 HTTP 요청을 생성해서 github 저장소의 "gen-survey-database.sql" 원파일만 현재 작업 디렉토리로 가져온다.

성공적으로 완료되면 터미널은 다음 출력결과를 화면에 표시한다.

```
--2014-09-02 18:31:43-- https://raw.githubusercontent.com/swcarpentry/bc/master/novice/sql/gen-survey-database.sql
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 103.245.222.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|103.245.222.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3297 (3.2K) [text/plain]
Saving to: 'gen-survey-database.sql'

100%[=====] 3297 --:-- --
```

2014-09-02 18:31:45 (264 KB/s) - 'gen-survey-database.sql' saved [3297/3297]

이제 성공적으로 단일 SQL 파일을 가져와서, survey.db 데이터베이스를 생성하고 gen-survey-database.sql에 저장된 지시방법에 따라서 데이터를 채워넣는다.

명령-라인 터미널에서 SQLite3 프로그램을 호출하기 위해서, 다음 명령문을 실행한다.

```
root@hangul:~/swc/sql$ sqlite3 survey.db < gen-survey-database.sql
```

38.4 SQLite DB 연결/설치 테스트

생성된 데이터베이스에 연결하기 위해서, 데이터베이스를 생성한 디렉토리 안에서 SQLite를 시작한다. 그래서 ~/swc/sql 디렉토리에서 다음과 같이 타이핑한다.

```
root@hangul:~/swc/sql$ sqlite3 survey.db
```

sqlite3 survey.db 명령문이 데이터베이스를 열고 데이터베이스 명령-라인 프롬프트로 안내한다. SQLite에서 데이터베이스는 플랫 파일(flat file)로 명시적으로 열 필요가 있다. 그리고 나서 SQLite 시작되고 sqlite로 명령-라인 프롬프트가 다음과 같이 변경되어 표시된다.

```
SQLite version 3.20.0 2017-08-01 13:24:15
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

다음 출력결과가 보여주듯이 .databases 명령문으로 소속된 데이터베이스 이름과 파일 목록을 확인한다.

```
sqlite> .databases
seq   name           file
---  -----
0    main           ~/novice/sql/survey.db
```

다음과 같이 타이핑해서 필요한 “Person”, “Survey”, “Site” “Visited” 테이블이 존재하는 것을 확인한다. .table의 출력결과는 다음과 같다.

```
sqlite> .tables
Person  Site   Survey   Visited
```

38.5 SQLite DB 나오는 법

SQLite3 DB 명령-라인 인터페이스(CLI)를 어떻게 빠져나올까요?

SQLite3를 빠져나오기 위해서, 다음과 같이 타이핑한다.

```
sqlite> .quit
```


Chapter 39

변수/칼럼 선택하기

1920년 후반, 1930년 초반 William Dyer, Frank Pabodie, Valentina Roerich는 남태평양 도달불가능한 극(Pole of Inaccessibility)과 이어서 남극 대륙을 탐험했다. 2년 전에 이들의 탐험 기록이 Miskatonic 대학 창고 사물함에서 발견됐다. 기록을 스캔해서 OCR로 저장했고, 이제는 검색가능하고 분석이 용이한 방식으로 정보를 저장하고자 한다.

기본적으로 3가지 선택 옵션(텍스트 파일, 스프레드쉬트, 데이터베이스)이 있다. 텍스트 파일은 생성하기 가장 쉽고 버전 제어와 궁합이 맞지만, 검색과 분석 도구를 별도로 구축해야한다. 스프레드쉬트는 단순한 분석에는 적합하지만, 크고 복잡한 데이터셋을 매우 잘 다루지는 못한다. 그래서 데이터를 데이터베이스에 넣어서 어떻게 검색과 분석을 하는지 이번 학습에서 배울 것이다.

39.1 정의 몇가지

관계형 데이터베이스(relational database)는 테이블(tables)로 정렬된 정보를 저장하고 다루는 방식이다. 각 테이블은 데이터를 기술하는 필드(fields)로도 알려진 열(column)과 데이터를 담고 있는 레코드(records)로 알려진 행(row)으로 구성된다.

스프레드쉬트를 사용할 때, 이전 값에 기초하여 새로운 값을 계산할 때 공식을 셀(cell)에 넣어서 구한다. 데이터베이스를 사용할 때는 쿼리(queries, 질의)로 불리는 명령문을 데이터베이스 관리자(database manager)에게 보낸다. 데이터베이스 관리자는 사용자를 대신해서 데이터베이스를 다루는 프로그램이다. 데이터베이스 관리자는 쿼리가 명기하는 임의의 조회와 계산을 수행하고 다음 쿼리의 시작점으로 사용될 수 있는 테이블 형식으로 결과값을 반환한다.

모든 데이터베이스 관리자(IBM DB2, PostgreSQL, MySQL, Microsoft Access, SQLite)는 서로 다른 고유한 방식으로 데이터를 저장해서 한곳에서 생성된 데이터베이스는 다른 곳의 데이터베이스에서 직접적으로 사용될 수 없다. 하지만, 모든 데이터베이스 관리자는

데이터를 다양한 형식으로 가져오기(import)와 내보내기(export)를 지원한다. 그래서 한 곳에서 다른 곳으로 정보를 이동하는 것이 가능하다.

쿼리는 SQL로 불리는 언어로 작성된다. SQL은 “Structured Query Language”(구조적 질의 언어)의 약자다. SQL은 데이터를 분석하고 다시 조합할 수 있는 수백개의 다른 방식을 제공한다. 학습에서 일부를 살펴볼 것이지만, 이 일부가 과학자가 수행하는 일의 대부분을 처리할 것이다.

다음 테이블은 예제로 사용할 데이터베이스를 보여준다.

Person: 판독한 사람.

ident

personal

family

dyer

William

Dyer

pb

Frank

Pabodie

lake

Anderson

Lake

roe

Valentina

Roerich

danforth

Frank

Danforth

Site: 판독한 장소.

name

lat

long

DR-1

-49.85

-128.57

DR-3

-47.15

-126.72

MSK-4

-48.87

-123.4

Visited: 특정 사이트에서 판독한 시점.

ident

site

dated

619

DR-1

1927-02-08

622

DR-1

1927-02-10

734

DR-3

1939-01-07

735

DR-3

1930-01-12

751

DR-3

1930-02-26

752

DR-3

MSK-4

1932-01-14

844

DR-1

1932-03-22

Survey: 실제 판독.

taken

person

quant

reading

619

dyer

rad

9.82

619

dyer

sal

0.13

622

dyer

rad

7.8

622

dyer

sal

0.09

734

pb

rad

8.41

734

lake

sal

0.05

734

pb

temp

-21.5

735

pb

rad

7.22

735

sal

0.06

735

temp

-26.0

751

pb

rad

4.35

751

pb

temp

-18.5

751

lake

sal

0.1

752

lake

rad

2.19

752

lake

sal

0.09

752

lake

temp

-16.0

752

roe

sal

41.6

837

lake

rad

1.46

837

lake

sal

0.21

837

roe

sal

22.5

844

roe

rad

Table 39.1: 첫번째 SQL 쿼리

| family | personal |
|----------|-----------|
| Dyer | William |
| Pabodie | Frank |
| Lake | Anderson |
| Roerich | Valentina |
| Danforth | Frank |

Table 39.2: SQL 쿼리문은 대소문자 구분하지 않음

| family | personal |
|----------|-----------|
| Dyer | William |
| Pabodie | Frank |
| Lake | Anderson |
| Roerich | Valentina |
| Danforth | Frank |

11.25

3개 항목 (Visited 테이블에서 1개, Survey 테이블에서 2개) 은 붉은색으로 표기한 것을 주목하라. 왜냐하면 어떠한 값도 담고 있지 않아서 그렇다. 결측값(missing)은 추후 다룰 것이다. 지금으로서는 과학자의 이름을 화면에 표시하는 SQL을 작성하자. SQL select 문을 사용해서 원하는 칼럼이름과 원하는 테이블이름을 준다. 쿼리와 결과는 다음과 같다.

```
$ sqlite3 survey.db
SQLite version 3.35.4 2021-04-02 15:20:15
Enter ".help" for usage hints.
sqlite>
SELECT family, personal FROM Person;
```

쿼리 끝에 세미콜론(;)은 쿼리가 완료되어 실행준비 되었다고 데이터베이스 관리자에게 알려준다. 명령문과 칼럼 이름을 모두 소문자로 작성했고, 테이블 이름은 타이틀 케이스>Title Case, 단어의 첫 문자를 대문자로 표기)로 작성했다. 하지만 그렇게 반듯이 할 필요는 없다. 아래 예제가 보여주듯이, SQL은 **대소문자 구분하지 않는다**

```
SeLeCt FaMiLy, PeRsOnAl FrOm PeRsOn;
```

모두 소문자, 타이틀 케이스, 소문자 낙타 대문자(Lower Camel Case)를 선택하든지 관계없이 일관성을 가져라. 랜덤 대문자를 추가적으로 인지하지 않더라고 복잡한 쿼리는 충분히 그 자체로 이해하기 어렵다.

쿼리로 돌아가서, 데이터베이스 테이블의 행과 열이 특정한 순서로 저장되지 않는다는 것을 이해하는 것이 중요하다. 어떤 순서로 항상 표시되지만, 다양한

Table 39.3: 칼럼 교환

| personal | family |
|-----------|----------|
| William | Dyer |
| Frank | Pabodie |
| Anderson | Lake |
| Valentina | Roerich |
| Frank | Danforth |

Table 39.4: 칼럼명 중복 선택

| ident | ident | ident |
|----------|----------|----------|
| dyer | dyer | dyer |
| pb | pb | pb |
| lake | lake | lake |
| roe | roe | roe |
| danforth | danforth | danforth |

방식으로 제어할 수 있다. 예를 들어, 쿼리를 다음과 같이 작성해서 칼럼을 교환할 수 있다.

```
SELECT personal, family FROM Person;
```

혹은 심지어 칼럼을 반복할 수도 있다.

```
SELECT ident, ident, ident FROM Person;
```

손쉬운 방법으로, *을 사용해서 테이블의 모든 칼럼을 선택할 수도 있다.

```
SELECT * FROM Person;
```

39.2 도전 과제

- Site 테이블에서 사이트 이름만 선택하는 쿼리를 작성하세요.

Table 39.5: 모든 칼럼 선택

| ident | personal | family |
|----------|-----------|----------|
| dyer | William | Dyer |
| pb | Frank | Pabodie |
| lake | Anderson | Lake |
| roe | Valentina | Roerich |
| danforth | Frank | Danforth |

2. 많은 사람들이 쿼리를 다음과 같은 형식으로 작성한다.

```
SELECT personal, family FROM person;
```

혹은 다음과 같아도 작성한다.

```
select Personal, Family from PERSON;
```

읽기 쉽기 쉬운 스타일은 어느 것인가요? 이유는 무엇일까요?

39.3 주요점

- 관계형 데이터베이스는 정보를 테이블로 저장한다. 고정된 숫자의 칼럼과 변하기 쉬운 숫자의 레코드로 구성된다.
- 데이터베이스 관리자는 데이터베이스에 저장된 정보를 다루는 프로그램이다.
- 데이터베이스에서 정보를 추출하는데 SQL이라고 불리는 특화된 언어로 쿼리를 작성한다.
- SQL은 대소문자를 구별하지 않는다.

Chapter 40

정렬, 중복 제거

데이터는 종종 임여가 있어서, 쿼리도 종종 과잉 정보를 반환한다. 예를 들어, survey 테이블에서 측정된 수량 정보를 선택하면, 다음을 얻게된다.

```
$ sqlite3 survey.db
SQLite version 3.35.4 2021-04-02 15:20:15
Enter ".help" for usage hints.
sqlite>
select quant from Survey;
```

결과를 좀더 읽을 수 있게 만들기 위해서 쿼리에 `distinct` 키워드를 추가해서 중복된 출력을 제거한다.

```
select distinct quant from Survey;
```

하나 이상의 칼럼(예를 들어 survey 사이트 ID와 측정된 수량)을 선택한다면, 별개로 구별된 값의 쌍이 반환된다.

```
select distinct taken, quant from Survey;
```

양쪽 경우에 설사 데이터베이스 내에서 서로 인접하지 않더라도 모두 중복이 제거된 것을 주목하세요. 다시 한번, 행은 실제로 정렬되지는 않았다는 것을 기억하세요. 단지 정렬된 것으로 화면에 출력된다.

40.1 도전 과제

- Site 테이블에서 별개로 구별되는 날짜를 선택하는 쿼리를 작성하세요.

앞서 언급했듯이, 데이터베이스 레코드는 특별한 순서로 저장되지 않는다. 이것이 의미하는 바는 쿼리 결과가 반드시 정렬되어 있지 않다는 것이다. 설사 정렬이 되어 있더라도, 종종 다른 방식으로 정렬하고 싶을 것이다. 예를 들어 과학자의 이름

Table 40.1: SQL select 쿼리문

| |
|-------|
| quant |
| rad |
| sal |
| rad |
| sal |
| rad |
| sal |
| temp |
| rad |
| sal |
| temp |

Table 40.2: 중복 제거 쿼리문

| |
|-------|
| quant |
| rad |
| sal |
| temp |

Table 40.3: 칼럼 두개 중복 제거

| taken | quant |
|-------|-------|
| 619 | rad |
| 619 | sal |
| 622 | rad |
| 622 | sal |
| 734 | rad |
| 734 | sal |
| 734 | temp |
| 735 | rad |
| 735 | sal |
| 735 | temp |

Table 40.4: ident 칼럼 기준 정렬

| ident | personal | family |
|----------|-----------|----------|
| danforth | Frank | Danforth |
| dyer | William | Dyer |
| lake | Anderson | Lake |
| pb | Frank | Pabodie |
| roe | Valentina | Roerich |

Table 40.5: ident 칼럼 기준 역정렬

| ident | personal | family |
|----------|-----------|----------|
| roe | Valentina | Roerich |
| pb | Frank | Pabodie |
| lake | Anderson | Lake |
| dyer | William | Dyer |
| danforth | Frank | Danforth |

대신에 프로젝트 이름으로 정렬할 수도 있다. SQL에서 쿼리에 `order by` 절을 추가해서 간단하게 구현할 수 있다.

```
select * from Person order by ident;
```

디폴트로, 결과는 오름차순으로 정렬되어야 한다. (즉, 가장 적은 값에서 가장 큰 값 순으로 정렬된다.) `desc` ("descending")를 사용해서 역순으로도 정렬할 수 있다.

```
select * from person order by ident desc;
```

(그리고, `desc` 대신에 `asc`를 사용해서 오름차순으로 정렬하고 있다는 것을 명시적으로 표현할 수도 있다.)

한번에 여러 필드를 정렬할 수도 있다. 예를 들어, 다음 쿼리는 `taken` 필드를 오름차순으로 그리고 동일 그룹의 `taken` 값 내에서는 `person`으로 내림차순으로 결과를 정렬한다.

```
select taken, person from Survey order by taken asc, person desc;
```

만약 중복을 제거한다면 이해하기가 더 쉽다.

```
select distinct taken, person from Survey order by taken asc, person desc;
```

40.2 도전 과제

1. `Visited` 테이블에서 별개로 구별되는 날짜를 반환하는 쿼리를 작성하세요.
2. 성(family name)으로 정렬된 `Person` 테이블에 과학자의 성명 전부를 화면에

Table 40.6: 칼럼 기준 순정렬, 역정렬

| taken | person |
|-------|--------|
| 619 | dyer |
| 619 | dyer |
| 622 | dyer |
| 622 | dyer |
| 734 | pb |
| 734 | pb |
| 734 | lake |
| 735 | pb |
| 735 | NA |
| 735 | NA |

Table 40.7: 중복제거 칼럼 기준 순정렬, 역정렬

| taken | person |
|-------|--------|
| 619 | dyer |
| 622 | dyer |
| 734 | pb |
| 734 | lake |
| 735 | pb |
| 735 | NA |
| 751 | pb |
| 751 | lake |
| 752 | roe |
| 752 | lake |

출력하는 쿼리를 작성하세요.

40.3 주요점

- 데이터베이스 테이블의 레코드는 본질적으로 정렬되지 않는다. 만약 특정 순서로 정렬하여 표시하려면, 명시적으로 정렬을 명기하여야 한다.
- 데이터베이스의 값이 유일(unique)함을 보장하지는 않는다. 만약 중복을 제거하고자 한다면, 명시적으로 유일함을 명기하여야 한다.

Chapter 41

필터링 (Filtering)

데이터베이스의 가장 강력한 기능중 하나는 데이터를 필터(filter)하는 능력이다. 즉, 특정 기준에 맞는 레코드만 선택한다. 예를 들어, 특정 사이트를 언제 방문했는지 확인한다고 가정하자. 쿼리에 `where` 절을 사용해서 `Visited` 테이블로부터 레코드를 뽑아낼 수 있다.

```
select * from Visited where site='DR-1';
```

데이터베이스 관리자는 두 단계로 나누어 쿼리를 실행한다. 첫번째로, `where` 절을 만족하는 것이 있는지 확인하기 위해서 `Visited` 테이블의 각 행을 점검한다. 그리고 나서 무슨 칼럼을 표시할지 결정하기 위해서 `select` 키워드 다음에 있는 칼럼 이름을 사용한다.

이러한 처리 순서가 의미하는 바는 화면에 표시되지 않는 칼럼 값에 기반해서도 `where` 절을 사용해서 레코드를 필터링할 수 있다는 것이다.

```
select ident from Visited where site='DR-1';
```

데이터를 필터링하는데 불 연산자(Boolean Operators)를 사용할 수 있다. 예를 들어, 1930년 이후로 DR-1 사이트에서 수집된 모든 정보를 요청할 수도 있다.

```
select * from Visited where (site='DR-1') and (dated>='1930-00-00');
```

(각 테스트 주위의 괄호는 엄밀히 말해 필요하지는 않지만 쿼리를 좀더 읽기 쉽게 한다.)

Table 41.1: SQL 필터 쿼리문

| ident | site | dated |
|-------|------|------------|
| 619 | DR-1 | 1927-02-08 |
| 622 | DR-1 | 1927-02-10 |
| 844 | DR-1 | 1932-03-22 |

Table 41.2: SQL 칼럼 선택과 결합된 필터 쿼리문

| ident |
|-------|
| 619 |
| 622 |
| 844 |

Table 41.3: SQL 필터 부울연산 반영쿼리문

| ident | site | dated |
|-------|------|------------|
| 844 | DR-1 | 1932-03-22 |

대부분의 데이터베이스 관리자는 날짜에 대한 특별한 데이터 형식을 가진다. 사실 많이 있지만 두 가지 형식으로 볼 수 있다. 날짜 데이터 형식의 하나는 “May 31, 1971”와 같은 것이고, 다른 하나는 “31 days” 같은 기간에 대한 것이다. SQLite는 구분하지는 않는다. 대신에 SQLite는 날짜를 텍스트 (ISO-8601 표준 형식 “YYYY-MM-DD HH:MM:SS.SSSS”), 혹은 실수 (November 24, 4714 BCE 이후 지나간 일수), 혹은 정수 (1970년 1월 1일 자정 이후 초)로만 저장한다. 만약 복잡하게 들린다면, 그럴 수도 있다 하지만 스웨덴의 역사적인 날짜(historical dates in Sweden)를 이해하는 것만큼 복잡하는지는 않다.

Lake 혹은 Roerich가 무슨 측정을 했는지 알아내고자 한다면, or를 사용하여 이름에 테스트를 조합할 수 있다.

```
select * from Survey where person='lake' or person='roe';
```

다른 방식으로, in을 사용하여 특정 집합에 값이 있는지 확인할 수 있다.

```
select * from Survey where person in ('lake', 'roe');
```

and와 or를 조합할 수는 있지만, 어느 연산자가 먼저 수행되는지 주의할 필요가 있다. 만약 괄호를 사용하지 않는다면, 다음을 얻게 된다.

```
select * from Survey where quant='sal' and person='lake' or person='roe';
```

상기 결과는 Lake가 측정한 염분량과 Roerich가 측정한 임의 측정값이다. 대신에 아마도 다음과 같은 결과를 얻고자 했을 것이다.

```
select * from Survey where quant='sal' and (person='lake' or person='roe');
```

마지막으로 distinct와 where를 사용하여 두 번째 수준의 필터링을 한다.

```
select distinct person, quant from Survey where person='lake' or person='roe';
```

하지만, 기억하라. distinct는 처리될 때 선택된 칼럼에 표시되는 값에만 적용되고 전체 행에는 적용되지 않는다.

Table 41.4: SQL 필터 부울 선택(OR) 연산자 반영쿼리문

| taken | person | quant | reading |
|-------|--------|-------|---------|
| 734 | lake | sal | 0.05 |
| 751 | lake | sal | 0.10 |
| 752 | lake | rad | 2.19 |
| 752 | lake | sal | 0.09 |
| 752 | lake | temp | -16.00 |
| 752 | roe | sal | 41.60 |
| 837 | lake | rad | 1.46 |
| 837 | lake | sal | 0.21 |
| 837 | roe | sal | 22.50 |
| 844 | roe | rad | 11.25 |

Table 41.5: SQL 필터 가독성 높은 부울 선택(OR) 연산 쿼리문

| taken | person | quant | reading |
|-------|--------|-------|---------|
| 734 | lake | sal | 0.05 |
| 751 | lake | sal | 0.10 |
| 752 | lake | rad | 2.19 |
| 752 | lake | sal | 0.09 |
| 752 | lake | temp | -16.00 |
| 752 | roe | sal | 41.60 |
| 837 | lake | rad | 1.46 |
| 837 | lake | sal | 0.21 |
| 837 | roe | sal | 22.50 |
| 844 | roe | rad | 11.25 |

Table 41.6: SQL 필터 부울 선택(OR) 연산 적용순서

| taken | person | quant | reading |
|-------|--------|-------|---------|
| 734 | lake | sal | 0.05 |
| 751 | lake | sal | 0.10 |
| 752 | lake | sal | 0.09 |
| 752 | roe | sal | 41.60 |
| 837 | lake | sal | 0.21 |
| 837 | roe | sal | 22.50 |
| 844 | roe | rad | 11.25 |

Table 41.7: SQL 필터 괄호 적용 부울 선택(OR) 연산 적용순서

| taken | person | quant | reading |
|-------|--------|-------|---------|
| 734 | lake | sal | 0.05 |
| 751 | lake | sal | 0.10 |
| 752 | lake | sal | 0.09 |
| 752 | roe | sal | 41.60 |
| 837 | lake | sal | 0.21 |
| 837 | roe | sal | 22.50 |

Table 41.8: SQL 중복제거 선택문과 결합된 필터 부울 선택(OR) 연산

| person | quant |
|--------|-------|
| lake | sal |
| lake | rad |
| lake | temp |
| roe | sal |
| roe | rad |

방금전까지 수행하는 것은 대부분의 사람들이 어떻게 SQL 쿼리를 증가시키는지 살펴봤다. 의도한 것의 일부를 수행하는 단순한 것에서부터 시작했다. 그리고 절을 하나씩 하나씩 추가하면서 효과를 테스트했다. 좋은 전략이다. 사실 복잡한 쿼리를 작성할 때, 종종 유일한 전략이다. 하지만 이 전략은 빠른 회전(turnaround)시간에 달려있고 사용자에게는 정답을 얻게되면 인지하는 것에 달려있다. 빠른 회전시간을 달성하는 최선의 방법은 임시 데이터베이스에 데이터의 일부를 저장하고 쿼리를 실행하거나 혹은 합성된 레코드로 작은 데이터베이스를 채워놓고 작업을 하는 것이다. 예를 들어, 2천만 호주사람의 실제 데이터베이스에 쿼리를 작업하지 말고, 1만명 샘플을 뽑아 쿼리를 돌리거나 작은 프로그램을 작성해서 랜덤으로 혹은 그럴듯한 1만명 레코드를 생성해서 사용한다.

41.1 도전 과제

- 극에서 30°보다 고위도에 위치한 모든 사이트를 선택하고자 한다고 가정하자. 작성한 첫번째 쿼리는 다음과 같다.

```
select * from Site where (lat > -60) or (lat < 60);
```

왜 이 쿼리가 잘못된 것인지 설명하세요. 그리고 쿼리를 다시 작성해서 올바르게 동작하게 만드세요.

- 정규화된 염분 수치는 0.0에서 1.0 사이에 있어야 한다. 상기 범위 밖에 있는 염분수치를 가진 모든 레코드를 Survey 테이블에서 선택하는 쿼리를

작성하세요.

3. 만약 명명된 칼럼의 값이 주어진 패턴과 일치한다면 SQL 테스트 *column-name* like *pattern*은 참이다. “0 혹은 그 이상의 문자와 매칭”된다는 것을 의미하기 위해서 '%'문자를 패턴에 임의 숫자 횟수에 사용한다.

표현식

값

'a' like 'a'

True

'a' like '%a'

True

'b' like '%a'

False

'alpha' like 'a%'

True

'alpha' like 'a%p%'

True

표현식 *column-name* not like *pattern*은 테스트를 거꾸로 한다. like를 사용하여 사이트에서 'DR-something'으로 라벨이 붙지 않은 모든 레코드를 Visited에서 찾는 쿼리를 작성하세요.

41.2 주요점

- where를 사용해서 불 조건(Boolean conditions)에 따라 레코드를 필터링한다.
- 필터링이 전체 레코드에 적용되어서, 조건을 실제로 표시되지 않는 필드에 사용할 수 있다.

Chapter 42

새로운 값 계산하기

주의깊이 탐험 기록을 다시 정독한 뒤에, 탐험대가 보고한 방사선 측정치가 5%만큼 상향되어 수정될 필요가 있다는 것을 깨달았다. 저장된 데이터를 변형하기 보다는 쿼리의 일부분으로서 즉석에서 계산을 수행할 수 있다.

```
$ sqlite3 survey.db
SQLite version 3.35.4 2021-04-02 15:20:15
Enter ".help" for usage hints.
sqlite>
select 1.05 * reading from Survey where quant='rad';
```

쿼리를 실행하면, 표현식 `1.05 * reading`이 각 행마다 평가된다. 표현식에는 임의의 필드, 통상 많이 사용되는 연산자, 그리고 다양한 함수를 사용한다. (정확하게는 어느 데이터베이스 관리자를 사용되느냐에 따라 의존성을 띄게된다.) 예를 들어, 온도 측정치를 화씨에서 섭씨로 소수점 아래 두자리에서 반올림하여 변환할 수 있다.

Table 42.1: 신규 칼럼 생성 쿼리문

| 1.05 * reading |
|----------------|
| 10.31 |
| 8.19 |
| 8.83 |
| 7.58 |
| 4.57 |
| 2.30 |
| 1.53 |
| 11.81 |

Table 42.2: 신규 칼럼 반올림 적용 쿼리문

| taken | round(5*(reading-32)/9, 2) |
|-------|----------------------------|
| 734 | -29.7 |
| 735 | -32.2 |
| 751 | -28.1 |
| 752 | -26.7 |

Table 42.3: 접합연산자 적용 신규 칼럼 생성 쿼리문

| personal ' ' family |
|---------------------------|
| William Dyer |
| Frank Pabodie |
| Anderson Lake |
| Valentina Roerich |
| Frank Danforth |

```
select taken, round(5*(reading-32)/9, 2) from Survey where quant='temp';
```

다른 필드의 값을 조합할 수도 있다. 예를 들어, 문자열 접합 연산자 (string concatenation operator, ||)를 사용한다.

```
select personal || ' ' || family from Person;
```

first와 last 대신에 필드 이름으로 personal과 family를 사용하는 것이 이상해 보일지 모른다. 하지만, 문화적 차이를 다루기 위한 필요한 첫번째 단계다. 예를 들어, 다음 규칙을 고려해보자.

성명 전부(Full Name)

알파벳 순서

이유

Liu Xiaobo

Liu

중국 성이 이름보다 먼저 온다.

Leonardo da Vinci

Leonardo

“da Vinci” 는 “from Vinci”를 뜻한다.

Catherine de Medici

Medici

성(family name)

Jean de La Fontaine

La Fontaine

성(family name)이 “La Fontaine”이다.

Juan Ponce de Leon

Ponce de Leon

전체 성(full family name)이 “Ponce de Leon”이다.

Gabriel Garcia Marquez

Garcia Marquez

이중으로 된 스페인 성(surnames)

Wernher von Braun

von or Braun

독일 혹은 미국에 있는냐에 따라 달라짐

Elizabeth Alexandra May Windsor

Elizabeth

군주가 통치하는 이름에 따라 알파벳순으로 정렬

Thomas a Beckett

Thomas

시성된(canonized) 이름에 따라 성인 이름 사용

분명하게, 심지어 두부분 “personal”과 “family”으로 나누는 것도 충분하지 않다.

42.1 도전 과제

1. 좀더 조사한 뒤에, Valentina Roerich는 염도를 퍼센티지(%)로 작성한 것을 알게되었다. Survey 테이블에서 값을 100으로 나누어서 모든 염도 측정치를 반환하는 쿼리를 작성하세요.
2. union 연산자는 두 쿼리의 결과를 조합한다.

```
select * from Person where ident='dyer' union select * from Person where ident='roe' ;
```

union을 사용하여 앞선 도전과제에서 기술되어 수정된 Roerich가 측정한, Roerich만 측정한 염도 측정치의 통합 리스트를 생성하세요. 출력결과는 다음과 같아야 한다.

Table 42.4: union 연산자 적용 쿼리 두개 결합 결과

| ident | personal | family |
|-------|-----------|---------|
| dyer | William | Dyer |
| roe | Valentina | Roerich |

0.13
 622
 0.09
 734
 0.05
 751
 0.1
 752
 0.09
 752
 0.416
 837
 0.21
 837
 0.225

3. Visited 테이블에 사이트 식별자는 '-'으로 구분되는 두 부분으로 구성되어 있다.

```
select distinct site from Visited;
```

DR-1
 DR-3
 MSK-4

몇몇 주요 사이트 식별자는 두 문자길이를 가지고 몇몇은 3문자길이를 가진다. “in string” 함수 instr(X, Y)은 X 문자열에 문자열 Y가 첫번째 출현의 1-기반 인덱스를 반환하거나 Y가 X에 존재하지 않으면 0을 반환한다. 부분 문자열 함수 substr(X, I)은 인덱스 I에서 시작하는 문자열 X의 부분문자열을 반환한다. 상기 두 함수를 사용해서 유일한 주요 사이트 식별자를 생성하세요. (이 데이터에 대해서 작업된 리스트는 “DR”과 “MSK”만 포함해야 한다.)

42.2 주요점

- SQL은 쿼리의 일부로서 레코드의 값을 사용한 계산을 수행한다.

Chapter 43

결측 데이터 (Missing Data)

현실 세계 데이터는 결코 완전하지 않고 구멍은 항상 있다. `null`로 불리는 특별한 값을 사용하여 데이터베이스는 구멍을 표현한다. `null`은 0, `False`, 혹은 빈 문자열도 아니다.”아무것도 없음(none here)”을 의미하는 특별한 값이다. `null`을 다루는 것은 약간 특별한 기교와 신중한 생각을 요구한다.

시작으로 `Visited` 테이블을 살펴보자. 레코드가 8개 있지만 #752은 날짜가 없다. 혹은 더 정확히 말하면 날짜가 `null`이다.

```
$ sqlite3 survey.db
SQLite version 3.35.4 2021-04-02 15:20:15
Enter ".help" for usage hints.
sqlite>
select * from Visited;
```

`Null` 다른 값과는 다르게 동작한다. 만약 1930년 이전 레코드를 선택한다면,

Table 43.1: 결측값을 갖는 테이블

| ident | site | dated |
|-------|-------|------------|
| 619 | DR-1 | 1927-02-08 |
| 622 | DR-1 | 1927-02-10 |
| 734 | DR-3 | 1939-01-07 |
| 735 | DR-3 | 1930-01-12 |
| 751 | DR-3 | 1930-02-26 |
| 752 | DR-3 | NA |
| 837 | MSK-4 | 1932-01-14 |
| 844 | DR-1 | 1932-03-22 |

Table 43.2: 1930년 이전 레코드 선택

| ident | site | dated |
|-------|------|------------|
| 619 | DR-1 | 1927-02-08 |
| 622 | DR-1 | 1927-02-10 |

Table 43.3: 1930년 이후 레코드 선택

| ident | site | dated |
|-------|-------|------------|
| 734 | DR-3 | 1939-01-07 |
| 735 | DR-3 | 1930-01-12 |
| 751 | DR-3 | 1930-02-26 |
| 837 | MSK-4 | 1932-01-14 |
| 844 | DR-1 | 1932-03-22 |

```
select * from Visited where dated<'1930-00-00';
```

결과 2개를 얻게 되고, 만약 1930년 동안 혹은 이후 레코드를 선택한다면,

```
select * from Visited where dated>='1930-00-00';
```

결과를 5개 얻게되지만, 레코드 #752은 결과값 어디에도 존재하지 않는다. 이유는 `null<'1930-00-00'` 평가결과가 참도 거짓도 아니기 때문이다. `null`이 의미하는 것은 “알수가 없다”는 것이다. 그리고 만약 비교 평가식의 왼쪽편 값을 알지 못한다면, 비교도 참인지 거짓인지 알수가 없다. 데이터베이스는 “알수 없음”을 `null`로 표현하기 때문에, `null<'1930-00-00'`의 값도 사실 `null`이다. `null>='1930-00-00'`도 또한 `null`인데 왜냐하면 질문에 답을 할 수 없기 때문이다. 그리고, `where`절에 레코드는 테스트가 참인 것만 있기 때문에 레코드 #752은 어느 결과값에도 포함되지 않게 된다.

평가식만 `null`값을 이와 같은 방식으로 다루는 연산자는 아니다. `1+null`도 `null`이고, `5*null`도 `null`이고, `log(null)`도 `null`이 된다. 특히, 무언가를 = 과 != 으로 `null`과 비교하는 것도 `null`이 된다.

```
select * from Visited where dated=NULL;
```

```
select * from Visited where dated!=NULL;
```

`null` 인지 아닌지를 검검하기 위해서, 특별한 테스트 `is null`을 사용해야 한다.

Table 43.4: NULL 값 갖는 레코드 선택

| ident | site | dated |
|-------|------|-------|
| | | |

Table 43.5: NULL 값 갖지 않는 레코드 선택

| ident | site | dated |
|-------|------|-------|
|-------|------|-------|

Table 43.6: ‘is NULL’ 사용 NULL 값 갖는 레코드 선택

| ident | site | dated |
|-------|------|-------|
| 752 | DR-3 | NA |

```
select * from Visited where dated is NULL;
```

혹은, 역으로는 `is not null`을 사용한다.

```
select * from Visited where dated is not NULL;
```

`null` 값은 나타나는 곳마다 두통을 일으킨다. 예를 들어, Dyer가 측정하지 않은 모든 염분 정보를 찾는다고 가정하자. 다음과 같이 쿼리를 작성하는 것은 당연하다.

```
select * from Survey where quant='sal' and person!= 'lake';
```

하지만, 상기 쿼리 필터는 누가 측정을 했는지 모르는 레코드는 빠뜨린다. 다시 한번, 이유는 `person`이 `null`일 때, `!=`비교는 `null`값을 만들어서 레코드가 결과값에 있지 않게 된다. 만약 이런 레코드도 유지하려고 한다면, 명시적으로 검사를 추가할 필요가 있다.

```
select * from Survey where quant='sal' and (person!= 'lake' or person is null);
```

여전히 이러한 접근법이 맞는 것인지 아닌 것인지 판단할 필요가 있다. 만약 절대적으로 결과에 Lake가 측정한 어떠한 값도 포함하지 않는다고 확신한다면, 누가 작업을 한 것인지 모르는 모든 레코드를 제외할 필요가 있다.

Table 43.7: ‘is not NULL’ 사용 NULL 값 갖지 않는 레코드 선택

| ident | site | dated |
|-------|-------|------------|
| 619 | DR-1 | 1927-02-08 |
| 622 | DR-1 | 1927-02-10 |
| 734 | DR-3 | 1939-01-07 |
| 735 | DR-3 | 1930-01-12 |
| 751 | DR-3 | 1930-02-26 |
| 837 | MSK-4 | 1932-01-14 |
| 844 | DR-1 | 1932-03-22 |

Table 43.8: NULL 값이 갖는 문제

| taken | person | quant | reading |
|-------|--------|-------|---------|
| 619 | dyer | sal | 0.13 |
| 622 | dyer | sal | 0.09 |
| 752 | roe | sal | 41.60 |
| 837 | roe | sal | 22.50 |

Table 43.9: NULL 값 갖는 문제 명시적 해결

| taken | person | quant | reading |
|-------|--------|-------|---------|
| 619 | dyer | sal | 0.13 |
| 622 | dyer | sal | 0.09 |
| 735 | NA | sal | 0.06 |
| 752 | roe | sal | 41.60 |
| 837 | roe | sal | 22.50 |

43.1 도전 과제

- 날짜가 알려지지 않은 (즉 null) 항목은 빼고, 날짜 순으로 Visited 테이블에 있는 레코드를 정렬한 쿼리를 작성하세요.
- 다음 쿼리가 무슨 결과를 할까요?

```
select * from Visited where dated in ('1927-02-08', null);
```

상기 쿼리가 실질적으로 무엇을 생기게 할까요?

- 몇몇 데이터베이스 디자이너는 null 보다 결측 데이터를 표기하기 위해서 **보초값(sentinel value)**를 사용한다. 예를 들어, 결측 날짜를 표기하기 위해서 “0000-00-00” 날짜를 사용하거나 결측 염분치 혹은 결측 방사선 측정값을 표기하기 위해서 -1.0을 사용한다. (왜냐하면 실제 측정값이 음수가 될 수 없기 때문이다.) 이러한 접근법은 무엇을 단순화할까요? 이러한 접근법이 어떤 부담과 위험을 가져올까요?

43.2 주요점

- 데이터베이스는 결측 정보를 표현하기 위해서 null을 사용한다.
- null이 관계되는 산술 혹은 불 연산 결과도 null이다.
- null과 함께 안전하게 사용될 수 있는 유일한 연산자는 is null과 is not null이다.

Chapter 44

집합(Aggregation)

이제 데이터의 평균과 범위를 계산하고자 한다. Visited 테이블에서 모든 날짜 정보를 어떻게 선택하는지 알고 있다.

```
$ sqlite3 survey.db
SQLite version 3.35.4 2021-04-02 15:20:15
Enter ".help" for usage hints.
sqlite>
select dated from Visited;
```

하지만 조합하기 위해서는 `min` 혹은 `max` 같은 **집합 함수(aggregation function)**를 사용해야만 한다. 각 함수는 입력으로 레코드 집합을 받고 출력으로 단일 레코드를 만든다.

```
select min(dated) from Visited;
select max(dated) from Visited;
```

Table 44.1: SQL select 쿼리문

| dated |
|------------|
| 1927-02-08 |
| 1927-02-10 |
| 1939-01-07 |
| 1930-01-12 |
| 1930-02-26 |
| NA |
| 1932-01-14 |
| 1932-03-22 |

Table 44.2: 최소값 집합 함수 적용 쿼리문

| min(dated) |
|------------|
| 1927-02-08 |

Table 44.3: 최대값 집합 함수 적용 쿼리문

| max(dated) |
|------------|
| 1939-01-07 |

`min`과 `max`는 SQL에 내장된 단지 두개의 집합 함수다. 다른 세개는 `avg`, `count`, `sum`이 있다.

```
select avg(reading) from Survey where quant='sal';
select count(reading) from Survey where quant='sal';
select sum(reading) from Survey where quant='sal';
```

여기서 `count(reading)`을 사용했다. 하지만 `quant`를 단순히 쉽게 세거나 테이블의 다른 어떤 필드도 셀 수 있고 심지어 `count(*)`을 사용하기도 한다. 왜냐하면 `count()`함수가 값 자체보다는 얼마나 많은 값이 있는지에만 관심을 두기 때문이다.

SQL이 여러개의 집합연산도 한번에 수행한다. 예를 들어, 염분측정치의 범위도 알 수 있다.

```
select min(reading), max(reading) from Survey where quant='sal' and reading<=1.0;
```

출력결과가 놀라움을 줄 수도 있지만, 원 결과값과 집합 결과를 조합할 수도 있다.

```
select person, count(*) from Survey where quant='sal' and reading<=1.0;
```

왜 Roerich 혹은 Dyer가 아닌 Lake의 이름이 나타날까요? 답은 필드를 집합하지만 어떻게 집합하는지 말을 하지 않기 때문에 데이터베이스 관리자가 입력에서 실제 값을 고른다. 처음 처리된 것, 마지막에 처리된 것, 혹은 완전히 다른 무언가를 사용할 수도 있다.

또다른 중요한 사실은 집합할 어떠한 값도 없을 때, 집합 결과는 0 혹은 다른 임의의 값 보다 “알지 못한다(don't know)”가 된다.

Table 44.4: 평균값 집합 함수 적용 쿼리문

| avg(reading) |
|--------------|
| 7.2 |

Table 44.5: 개수 집합 함수 적용 쿼리문

| count(reading) |
|----------------|
| 9 |

Table 44.6: 합계 집합 함수 적용 쿼리문

| sum(reading) |
|--------------|
| 64.8 |

```
select person, count(*) from Survey where quant='sal' and reading<=1.0;
```

집합 함수의 마지막 중요한 한가지 기능은 매우 유용한 방식으로 나머지 SQL과는 일관되지 않다는 것이다. 만약 두 값을 더하는데 그중 하나가 null이면 결과는 null이다. 확장해서, 만약 한 집합의 모든 값을 더하기 위해서 sum을 사용하고 이들 중 임의의 값이 null이면, 결과도 또한 null이어야 한다. 하지만 집합함수가 null 값을 무시하고 단지 non-null 값만을 조합한다면 훨씬 더 유용하다. 명시적으로 항상 필터해야하는 대신에 이것의 결과 쿼리를 다음과 같이 작성할 수 있게 한다.

```
select min(dated) from Visited;
```

명시적으로 항상 다음과 같이 필터하는 쿼리를 작성할 필요가 없다.

```
select min(dated) from Visited where dated is not null;
```

한번에 모든 레코드를 집합하는 것이 항상 타당하지는 않다. 예를 들어, Gina가 데이터에 체계적인 편의(bias)가 있어서 다른 과학자의 방사선 측정치가 다른 사람의 것과 비교하여 높다고 의심한다고 가정하자. 다음 쿼리가 의도를 반영하여 동작하지 않는다는 것은 알고 있다.

```
select person, count(reading), round(avg(reading), 2)
from Survey
where quant='rad';
```

왜냐하면 데이터베이스 관리자가 각 과학자별로 구분된 집합하기 보다는 임의의 한명의 과학자 이름만 선택하기 때문이다. 단지 5명의 과학자만 있기 때문에, 다음과 같은 형식의 5개 쿼리를 작성할 수 있다.

```
select person, count(reading), round(avg(reading), 2)
from Survey
```

Table 44.7: 최소, 최대값 집합 함수 적용 쿼리문

| min(reading) | max(reading) |
|--------------|--------------|
| 0.05 | 0.21 |

Table 44.8: 원 결과값과 집합 함수를 적용한 쿼리문

| person | count(*) |
|--------|----------|
| dyer | 7 |

Table 44.9: NULL 값이 포함된 원데이터에 집합 함수를 적용한 쿼리문

| person | count(*) |
|--------|----------|
| dyer | 7 |

Table 44.10: NULL 값이 포함된 원데이터를 명시적으로 처리한 후 집합 함수를 적용한 쿼리문

| min(dated) |
|------------|
| 1927-02-08 |

Table 44.11: NULL 값이 포함된 원데이터를 명시적으로 처리하지 않는 집합 함수를 적용한 쿼리문

| min(dated) |
|------------|
| 1927-02-08 |

Table 44.12: 주의깊이 살펴볼 쿼리문

| person | count(reading) | round(avg(reading), 2) |
|--------|----------------|------------------------|
| dyer | 8 | 6.56 |

Table 44.13: 사람별로 작성한 쿼리문 문제 예제

| person | count(reading) | round(avg(reading), 2) |
|--------|----------------|------------------------|
| dyer | 2 | 8.81 |

Table 44.14: ‘group by’ 문을 사용해서 사람별로 작성한 쿼리문 예제

| person | count(reading) | round(avg(reading), 2) |
|--------|----------------|------------------------|
| dyer | 2 | 8.81 |
| lake | 2 | 1.83 |
| pb | 3 | 6.66 |
| roe | 1 | 11.25 |

```
where quant='rad'
and person='dyer';
```

하지만, 이러한 접근법은 성가시고, 만약 50명 혹은 500명의 과학자를 가진 데이터셋을 분석한다면, 모든 쿼리를 올바르게 작성할 가능성은 작다.

필요한 것은 데이터베이스 관리자가 `group by` 절을 사용해서 각 과학자별로 시간을 집합하도록 지시하는 것이다.

```
select person, count(reading), round(avg(reading), 2)
from Survey
where quant='rad'
group by person;
```

`group by`는 이름이 의미하는 것과 동일한 것을 정확하게 수행한다. 지정된 필드에 동일한 값을 가진 모든 레코드를 그룹으로 묶어서 집합을 각 배치별로 처리한다. 각 배치에 모든 레코드는 `person`에 동일한 값을 가지고 있기 때문에, 데이터베이스 관리자가 임의의 값을 잡아서 집합된 `reading` 값과 함께 표시하는지는 더 이상 문제가 되지 않는다.

한번에 다중 기준으로 정렬하듯이 다중 기준으로 묶어 그룹화할 수 있다. 예를 들어 과학자와 측정 수량에 따라 평균 측정값을 얻기 위해서, `group by` 절에 또 다른 필드만 추가한다.

```
select person, quant, count(reading), round(avg(reading), 2)
from Survey
group by person, quant;
```

그렇지 않으면 결과가 의미가 없기 때문에, `person`을 표시되는 필드 리스트에 추가한 것을 주목하라.

한단계 더 나아가 누가 측정을 했는지 알지 못하는 모든 항목을 제거하자.

Table 44.15: ‘group by’ 문을 확장하여 적용한 쿼리문 사례

| person | quant | count(reading) | round(avg(reading), 2) |
|--------|-------|----------------|------------------------|
| NA | sal | 1 | 0.06 |
| NA | temp | 1 | -26.00 |
| dyer | rad | 2 | 8.81 |
| dyer | sal | 2 | 0.11 |
| lake | rad | 2 | 1.83 |
| lake | sal | 4 | 0.11 |
| lake | temp | 1 | -16.00 |
| pb | rad | 3 | 6.66 |
| pb | temp | 2 | -20.00 |
| roe | rad | 1 | 11.25 |

Table 44.16: 사람별로 측정값을 정렬한 쿼리문 사례

| person | quant | count(reading) | round(avg(reading), 2) |
|--------|-------|----------------|------------------------|
| dyer | rad | 2 | 8.81 |
| dyer | sal | 2 | 0.11 |
| lake | rad | 2 | 1.83 |
| lake | sal | 4 | 0.11 |
| lake | temp | 1 | -16.00 |
| pb | rad | 3 | 6.66 |
| pb | temp | 2 | -20.00 |
| roe | rad | 1 | 11.25 |
| roe | sal | 2 | 32.05 |

```
select person, quant, count(reading), round(avg(reading), 2)
from Survey
where person is not null
group by person, quant
order by person, quant;
```

좀더 면밀하게 살펴보면, 이 쿼리는,

1. Survey테이블에서 person 필드가 null이 아닌 레코드를 선택한다.
2. 상기 레코드를 부분집합으로 그룹지어서 각 부분집합의 person과 quant의 값은 같다.
3. 먼저 person으로 부분집합을 정렬하고나서 quant로 각 하위 그룹내에서도 정렬한다.
4. 각 부분집합의 레코드 숫자를 세고, 각각 reading 평균을 계산하고, 각각 person과 quant 값을 선택한다. (모두 동등하기 때문에 어느 것인지는

문제가 되지 않는다.)

44.1 도전 과제

1. Frank Pabodie는 얼마 많이 온도 측정치를 기록했고 평균 값은 얼마인가요?
2. 집합 값의 평균은 값을 합한 것을 값의 갯수로 나눈 것이다. 값이 1.0, null, 5.0으로 주어졌을 때, avg 함수는 2.0 혹은 3.0을 반환하는 것을 의미하나요?
3. 각 개별 방사선 측정값과 평균값 사이의 차이를 계산하고자 한다. 쿼리를 다음과 같이 작성한다.

```
select reading - avg(reading) from Survey where quant='rad';
```

상기 쿼리가 무엇을 만드나요? 그리고 왜 그런가요?

4. group_concat(field, separator) 함수는 지정된 구분 문자(혹은 만약 구분자가 지정되지 않는다면 ',')를 사용하여 필드의 모든 값을 결합한다. 이 함수를 사용해서 과학자의 이름을 한줄 리스트로 다음과 같이 만드세요.

William Dyer, Frank Pabodie, Anderson Lake, Valentina Roerich, Frank Danforth
성씨(surname)으로 리스트를 정렬하는 방법을 제시할 수 있나요?

44.2 주요점

- 집합 함수는 많은 값을 조합해서 하나의 새로운 값을 만든다.
- 집합 함수는 null 값을 무시한다.
- 필터링 다음에 집합이 일어난다.

Chapter 45

데이터 결합하기

과거 기상 자료를 집계하는 웹사이트에 데이터를 제출해야 되어서, Gina는 위도, 경도, 날짜, 수량, 측정값 형식으로 자료를 체계적으로 만들 필요가 있다. 하지만, 위도와 경도 정보는 Site 테이블에 있는 반면에 측정 날짜 정보는 Visited 테이블에 있고, 측정값 자체는 Survey 테이블에 있다. 어떤 방식이든지 상기 테이블을 조합할 필요가 있다.

이러한 작업을 하는 SQL 명령어가 `join`이다. 어떻게 동작하는지 확인하기 위해서, Site와 Visited 테이블을 조인하면서 출발해보자.

```
$ sqlite3 survey.db
SQLite version 3.35.4 2021-04-02 15:20:15
Enter ".help" for usage hints.
sqlite>
select * from Site join Visited;
```

`join`은 두 테이블을 **벡터곱(cross product)**한다. 즉, 모든 가능한 조합을 표현하려고 한 테이블의 레코드 각각마다 다른 테이블의 각 레코드와 조인한다. Site 테이블에 3개 레코드가 있고, Visited 테이블에 8개 레코드가 있어서, 조인된 결과는 24개 레코드가 된다. 그리고, 각 테이블이 3개 필드가 있어서 출력은 6개의 필드가 된다.

조인이 수행하지 않은 것은 조인되는 레코드가 서로 관계가 있는지를 파악하는 것이다. 어떻게 조인할지 명시할 때까지 레코드가 서로 관계가 있는지 없는지 알 수 있는 방법은 없다. 이를 위해서 동일한 사이트 이름을 가진 조합에만 관심있다는 것을 명시하는 절(clause)을 추가한다.

```
select * from Site join Visited on Site.name = Visited.site;
```

`on`은 `where`와 같은 역할을 한다. 특정 테스트를 통과한 레코드만 간직한다. (`on`과 `where`의 차이점은 `on`은 레코드가 생성될 때 레코드를 필터링하는 반면에, `where`는 조인작업이 완료될 때까지 기다리고 난 뒤에 필터링을 한다.) 쿼리에 레코드를

Table 45.1: SQL join 쿼리문

| name | lat | long | ident | site | dated |
|------|-------|------|-------|-------|------------|
| DR-1 | -49.9 | -129 | 619 | DR-1 | 1927-02-08 |
| DR-1 | -49.9 | -129 | 622 | DR-1 | 1927-02-10 |
| DR-1 | -49.9 | -129 | 734 | DR-3 | 1939-01-07 |
| DR-1 | -49.9 | -129 | 735 | DR-3 | 1930-01-12 |
| DR-1 | -49.9 | -129 | 751 | DR-3 | 1930-02-26 |
| DR-1 | -49.9 | -129 | 752 | DR-3 | NA |
| DR-1 | -49.9 | -129 | 837 | MSK-4 | 1932-01-14 |
| DR-1 | -49.9 | -129 | 844 | DR-1 | 1932-03-22 |
| DR-3 | -47.1 | -127 | 619 | DR-1 | 1927-02-08 |
| DR-3 | -47.1 | -127 | 622 | DR-1 | 1927-02-10 |

Table 45.2: 키값을 명시한 SQL join 쿼리문

| name | lat | long | ident | site | dated |
|-------|-------|------|-------|-------|------------|
| DR-1 | -49.9 | -129 | 619 | DR-1 | 1927-02-08 |
| DR-1 | -49.9 | -129 | 622 | DR-1 | 1927-02-10 |
| DR-1 | -49.9 | -129 | 844 | DR-1 | 1932-03-22 |
| DR-3 | -47.1 | -127 | 734 | DR-3 | 1939-01-07 |
| DR-3 | -47.1 | -127 | 735 | DR-3 | 1930-01-12 |
| DR-3 | -47.1 | -127 | 751 | DR-3 | 1930-02-26 |
| DR-3 | -47.1 | -127 | 752 | DR-3 | NA |
| MSK-4 | -48.9 | -123 | 837 | MSK-4 | 1932-01-14 |

Table 45.3: 점표기법을 적용한 SQL join 쿼리문

| lat | long | dated |
|-------|------|------------|
| -49.9 | -129 | 1927-02-08 |
| -49.9 | -129 | 1927-02-10 |
| -49.9 | -129 | 1932-03-22 |
| -47.1 | -127 | NA |
| -47.1 | -127 | 1930-01-12 |
| -47.1 | -127 | 1930-02-26 |
| -47.1 | -127 | 1939-01-07 |
| -48.9 | -123 | 1932-01-14 |

추가하자 마자 데이터베이스 관리자는 두 다른 사이트에 관한 조합된 정보는 사용한 뒤에 버려버리고, 원하는 레코드만 남겨둔다.

조인 결과에 필드 이름을 명기하기 위해서 `table.field`를 사용한 것에 주목하세요. 이렇게 하는 이유는 테이블이 동일한 이름을 가질 수 있고 어느 필드를 언급하는지 좀 더 구체성을 끌 필요가 있다. 예를 들어, `person`과 `visited` 테이블을 조인한다면, 결과는 각각의 원래 테이블에서 `ident`로 불리는 필드를 상속한다.

이제는 조인에서 원하는 3개의 칼럼을 선택하려고 점 표기법(dotted notation)을 사용할 수 있다.

```
select Site.lat, Site.long, Visited.dated
from   Site join Visited
on     Site.name=Visited.site;
```

만약 두개의 테이블을 조인하는 것이 좋은 경우에, 많은 테이블을 조인하는 것은 더 좋아야한다. 더 많은 join 결과 의미없는 레코드 조합을 필터링해서 제거하는 더 많은 `on` 테스트를 단순히 추가해서 사실 쿼리에 임의 갯수의 테이블을 조인할 수 있다.

```
select Site.lat, Site.long, Visited.dated, Survey.quant, Survey.reading
from   Site join Visited join Survey
on     Site.name=Visited.site
and    Visited.ident=Survey.taken
and    Visited.dated is not null;
```

`Site`, `Visited`, `Survey` 테이블의 어느 레코드가 서로 대응되지는 분간할 수 있는데 이유는 각 테이블이 **기본키(primary keys)**와 **외래키(foreign keys)**를 가지고 있기 때문이다.. 기본키는 하나의 값 혹은 여러 값의 조합으로 테이블의 각 레코드를 유일하게 식별한다. 외래키는 또 다른 테이블에 있는 유일하게 레코드를 식별하는 하나의 값(혹은 여러 값의 조합)이다. 다르게 표현하면, 외래캐는 다른 테이블에 존재하는 테이블의 기본키다. 예제 데이터베이스에서 `Person.ident`는 `Person` 테이블의 기본키인 반면에, `Survey.person`은 외래키로 `Survey` 테이블의 항목과 `Person` 테이블의 항목을 연결한다.

Table 45.4: 다수 테이블을 확장하여 결합한 SQL join 쿼리문

| lat | long | dated | quant | reading |
|-------|------|------------|-------|---------|
| -49.9 | -129 | 1927-02-08 | rad | 9.82 |
| -49.9 | -129 | 1927-02-08 | sal | 0.13 |
| -49.9 | -129 | 1927-02-10 | rad | 7.80 |
| -49.9 | -129 | 1927-02-10 | sal | 0.09 |
| -47.1 | -127 | 1939-01-07 | rad | 8.41 |
| -47.1 | -127 | 1939-01-07 | sal | 0.05 |
| -47.1 | -127 | 1939-01-07 | temp | -21.50 |
| -47.1 | -127 | 1930-01-12 | rad | 7.22 |
| -47.1 | -127 | 1930-01-12 | sal | 0.06 |
| -47.1 | -127 | 1930-01-12 | temp | -26.00 |

Table 45.5: 행ID(rowid) 쿼리문

| rowid | ident | personal | family |
|-------|----------|-----------|----------|
| 1 | dyer | William | Dyer |
| 2 | pb | Frank | Pabodie |
| 3 | lake | Anderson | Lake |
| 4 | roe | Valentina | Roerich |
| 5 | danforth | Frank | Danforth |

대부분의 데이터베이스 디자이너는 모든 테이블은 잘 정의된 기본키가 있어야된다고 믿는다. 또한 이 키는 데이터와 떨어져서 만약 데이터를 변경할 필요가 있다면, 한 곳의 변경이 한 곳에만 변경을 만들어야만 한다. 이를 위한 쉬운 방법은 데이터베이스에 레코드를 추가할 때 임의의 유일한 ID를 각 레코드마다 추가하는 것이다. 실제로 이방법은 매우 흔하게 사용된다. “student numbers”, “patient numbers” 같은 이름을 ID로 사용하고, 몇몇 데이터베이스 시스템 혹은 다른 곳에서 원래 고유 레코드 식별자로 거의 항상 판명된다. 다음 쿼리가 시범으로 보여주듯이, 테이블에 레코드가 추가됨에 따라 SQLite는 자동으로 레코드에 숫자를 붙이고, 쿼리에서 이렇게 붙여진 레코드 숫자를 사용한다.

```
select rowid, * from Person;
```

45.1 데이터 위생 (Data Hygiene)

지금까지 조인이 어떻게 동작하는지 살펴봤으니, 왜 관계형 모델이 그렇게 유용한지 그리고 어떻게 가장 잘 사용할 수 있는지 살펴보자. 첫번째 규칙은 모든 값은 독립 요소로 분해될 수 없는 원자(atomic)적 속성을 지녀야 한다. 즉, 구별해서 작업하고자 하는 부분을 포함해서는 안된다. 하나의 칼럼에 전체 이름을 넣는 대신에 별도로 구별되는 칼럼에 이름과 성을 저장해서 이름 컴포넌트를 뽑아내는 부분 문자열 연산(substring operation)을 사용할 필요가 없다. 좀더 중요하게는,

별도로 이름을 두 부분으로 저장한다. 왜냐하면, 공백으로 쪼개는 것은 신뢰성이 약하다. “Eloise St. Cyr” 혹은 “Jan Mikkel Steubart” 같은 이름을 생각하면 쉽게 알 수 있다.

두번째 규칙은 모든 레코드는 유일한 기본키를 가져야한다. 내재적인 의미가 전혀 없는 일련번호가 될 수 있고, 레코드의 값중의 하나 (Person 테이블의 ident 필드), 혹은 Survey 테이블에서 심지어 모든 측정값을 유일하게 식별하는 (taken, person, quant) 삼중값의 조합도 될 수 있다.

세번째 규칙은 불필요한 정보가 없어야 한다. 예를 들어, Site테이블을 제거하고 다음과 같이 Visited 테이블을 다시 작성할 수 있다.

619

-49.85

-128.57

1927-02-08

622

-49.85

-128.57

1927-02-10

734

-47.15

-126.72

1939-01-07

735

-47.15

-126.72

1930-01-12

751

-47.15

-126.72

1930-02-26

752

-47.15

-126.72

null

837
 -48.87
 -123.40
 1932-01-14
 844
 -49.85
 -128.57
 1932-03-22

사실, 스프레드시트와 마찬가지로 각 행에 각 측정값에 관한 모든 정보를 기록하는 하나의 테이블을 사용할 수도 있다. 문제는 이와 같은 방식으로 조직된 데이터를 일관성있게 관리하는 것은 매우 어렵다. 만약 특정한 사이트의 특정한 방문 날짜가 잘못된다면, 데이터베이스에 다수의 레코드를 변경해야한다. 더 안좋은 것은 다른 사이트도 그 날짜에 방문되었기 때문에 어느 레코드를 변경할지 추정해야하는 것이다.

네번째 규칙은 모든 값의 단위는 명시적으로 저장되어야한다. 예제 데이터베이스는 그렇지 못해서 문제다.

Roerich의 염분치는 다른 사람의 측정치보다 수천배 크다. 하지만, 천단위 대신에 백만 단위를 사용하고 있는지 혹은 1932년 그 사이트에 염분에 이상 실제로 있었는지 알지못한다.

한걸음 물러나서 생각하자, 데이터와 저장하는데 사용되는 도구는 공생관계다. 테이블과 조인은 데이터가 특정 방식으로 잘 조직되었다면 매우 효과적이다. 하지만, 만약 특정 형태로 되어 있다면 효과적으로 다룰 수 있는 도구가 있기 때문에 데이터를 그와 같은 방식으로 조직하기도 한다. 인류학자가 말했듯이, 도구는 도구를 만드는 손을 만든다. (the tool shapes the hand that shapes the tool)

45.2 도전 과제

1. DR-1 사이트의 모든 방사선 측정치를 출력하는 쿼리를 작성하세요.
2. “Frank” 가 방문한 모든 사이트를 출력하는 쿼리를 작성하세요.
3. 다음 쿼리가 무슨 결과를 산출하는지 말로 기술하세요.

```
select Site.name from Site join Visited
on Site.lat<-49.0 and Site.name=Visited.site and Visited.dated>='1932-00-00';
```

45.3 주요점

- 모든 사실은 데이터베이스에서 정확하게 한번만 표현되어야 한다.

- 조인은 한 테이블의 레코드와 다른 테이블의 레코드를 모두 조합한 결과를 출력한다.
- 기본키는 테이블의 레코드를 유일하게 식별하는 필드값(혹은 필드의 집합)이다.
- 외래키는 또 다른 테이블의 기본키가되는 필드값(혹은 필드의 집합)이다.
- 테이블사이에 기본키와 외래키를 매칭해서 의미없는 레코드의 조합을 제거할 수 있다.
- 조인을 좀더 단순하고 효율적으로 만들기 위해서 키(key)는 원자값(atomic value)이 되어야 한다.

Chapter 46

데이터 생성과 변형

지금까지 어떻게 데이터베이스에서 정보를 추출하는지만 살펴봤다. 왜냐하면, 정보를 추가하는 것보다 정보를 조회하는 것이 더 자주 있는 일이기도 하고, 다른 연산자는 쿼리가 이해되어야만 의미가 통하기 때문이다. 만약 데이터를 생성하고 변형하고자 한다면, 다른 두쪽의 명령어를 공부할 필요가 있다.

첫번째 짹은 `create table`과 `drop table`이다. 두 단어로 작성되지만, 사실 하나의 단일 명령어다. 첫번째 명령어는 새로운 테이블을 생성한다. 인자는 테이블 칼럼의 이름과 형식이다. 예를 들어, 다음 문장은 survey 데이터베이스에 테이블 4개를 생성한다.

```
create table Person(ident text, personal text, family text);
create table Site(name text, lat real, long real);
create table Visited(ident integer, site text, dated text);
create table Survey(taken integer, person text, quant real, reading real);
```

다음 명령어를 사용하여 테이블 중의 하나를 제거할 수도 있다.

```
drop table Survey;
```

데이터를 제거할 때 매우 주의하라. 대부분의 데이터베이스는 변경사항을 되돌리는 기능을 제공하지만, 이러한 기능에 의존하지 않는 것이 더 낫다.

다른 데이터베이스 시스템은 테이블 칼럼의 다른 데이터 형식도 지원하지만, 대부분은 다음을 다음을 제공한다.

`integer`

부호있는 정수형

`real`

부동 소수점 실수

`text`

문자열

blob

이미지 같은 “이진 대형 개체”

대부분의 데이터베이스는 불(boolean)과 날짜/시간 값도 지원한다. SQLite는 불값을 정수 0 과 1 을 사용하고 날짜/시간은 앞선(earlier) 학습방식으로 표현한다. 점점 더 많은 데이터베이스가 위도와 경도 같은 지리정보 데이터 형식도 지원한다. 특정 시스템이 무슨 기능을 제공하고 제공하지 않는지 그리고 어떤 이름을 다른 데이터 형식에 부여하는지를 계속 파악하는 것은 끝없는 시스템 이식성에 대한 골치거리다.

테이블을 생성할 때, 칼럼에 몇가지 제약사항을 지정할 수 있다. 예를 들어, Survey 테이블에 대한 좀더 좋은 정의는 다음과 같이 될 것이다.

```
create table Survey(
    taken    integer not null, -- where reading taken
    person   text,           -- may not know who took it
    quant    real not null,  -- the quantity measured
    reading  real not null,  -- the actual reading
    primary key(taken, quant),
    foreign key(taken) references Visited(ident),
    foreign key(person) references Person(ident)
);
```

다시 한번, 정확하게 무슨 제약사항이 이용가능하고 어떻게 호출되는지는 어떤 데이터베이스 관리자를 사용하는지에 달려있다.

테이블이 생성되자마자, 다른 명령어 짹 insert와 delete를 사용하여 레코드를 추가하고 제거할 수 있다. insert 문의 가장 간단한 형식은 순서대로 값을 목록으로 나열하는 것이다.

```
insert into Site values('DR-1', -49.85, -128.57);
insert into Site values('DR-3', -47.15, -126.72);
insert into Site values('MSK-4', -48.87, -123.40);
```

또한, 다른 테이블에서 직접 값을 테이블에 삽입할 수도 있다.

```
create table JustLatLong(lat text, long text);
insert into JustLatLong select lat, long from site;
```

레코드를 삭제하는 것은 약간 난이도가 있다. 왜냐하면 데이터베이스가 내부적으로 일관성을 보장할 필요가 있기 때문이다. 만약 하나의 단독 테이블만 관심을 둔다면, 삭제하고자 하는 레코드와 매칭되는 where절과 delete문을 함께 사용한다. 예를 들어, Frank Danforth가 어떤 측정도 하지 않았다는 것을 인지하자마자, 다음과 같이 Person 테이블에서 Frank Danforth를 제거할 수 있다.

```
delete from Person where ident = "danforth";
```

하지만 대신에 Anderson Lake를 실수로 제거했다면 어떨까요? Survey 테이블은 Anderson Lake이 수행한 7개의 측정 레코드를 담고 있지만, 이것은 결코 일어나지

말아야 된다. `Survey.person`은 `Person` 테이블에 외래키이고, 모든 쿼리는 전자의 모든 값을 매칭하는 후자의 행이 있을 거라고 가정한다.

이러한 문제를 **참조 무결성(referential integrity)**이라고 부른다. 테이블 사이의 모든 참조는 항상 제대로 해결될 수 있도록 확인할 필요가 있다. 참조 무결성을 보증하는 한 방법은 기본키로 사용하는 레코드를 삭제하기 전에 외래키로 'lake'를 사용하는 모든 레코드를 삭제하는 것이다. 만약 데이터베이스 관리자가 이 기능을 지원한다면, **연쇄적인 삭제(cascading delete)**를 사용해서 자동화할 수 있다. 하지만, 이 기법은 여기서 다루는 학습 영역밖이다.

모든 것을 데이터베이스에 저장하는 대신 많은 응용프로그램은 하이브리드 저장 모델을 사용한다. 천체 이미지 같은 실제 데이터는 파일에 저장되는 반면에, 파일 이름, 변경된 날짜, 커버하는 하늘의 영역, 스펙트럼 특성, 등등 정보는 데이터베이스에 저장한다. 대부분의 음악 재생기(MP3 플레이어) 소프트웨어가 작성되는 방식이기도 하다. 응용프로그램 내부 데이터베이스는 MP3 파일을 기억하고 있지만, MP3 파일 자체는 디스크에 있다.

46.1 도전 과제

- `Survey.person`의 `null`인 모든 사용자를 문자열 '`unknown`'으로 대체하는 SQL문을 작성하세요.
- 동료중의 한명이 Robert Olmstead가 측정한 온도 측정치를 포함하는 다음과 같은 형식의 CSV 파일을 보내왔다.

```
Taken,Temp
619,-21.5
622,-15.5
```

`survey` 데이터베이스에 레코드로 추가하려고 CSV 파일을 읽고 SQL `insert`문을 출력하는 작은 파이썬 프로그램을 작성하세요. `Person` 테이블에 `Olmstead` 항목을 추가할 필요가 있을 것이다. 반복적으로 프로그램을 테스트하려면, SQL `insert or replace` 문을 자세히 살펴볼 필요도 있다.

- SQLite는 SQL 표준이 아닌 몇개 관리 명령어가 있다. 그중의 하나가 `.dump`로 데이터베이스를 다시 생성하는데 필요한 SQL 명령문을 출력한다. 또 다른 것은 `.load`로 `.dump`에서 생성된 파일을 읽어서 데이터베이스를 복원한다. 여러분의 동료중의 한명이 텍스트인 `dump` 파일을 버전 제어 시스템에 저장하는 것이 데이터베이스 변경사항을 추적하고 관리하는 좋은 방법이라고 생각한다. 이러한 접근법의 장점과 단점은 무엇일까요? (힌트: 레코드는 어느 특정한 순서로 저장되지 않는다.)

46.2 주요점

- 데이터베이스 테이블은 테이블 이름과 필드의 이름과 특성을 명시하는 쿼리를 사용해서 생성된다.

- 쿼리를 사용해서 레코드는 삽입, 갱신, 삭제될 수 있다.
- 모든 레코드가 유일한 기본키를 가질 때 데이터를 변경하는 것이 더 간단하고 안전하다.

작업흐름 관리도구

Chapter 47

자동화와 Make

Make는 파일을 읽어들이고, 특정 방식으로 읽어들인 파일을 처리하고, 처리결과를 파일에 적을 수 있는 명령어를 실행하는 도구다. 예를 들어, 소프트웨어 개발에서, Make를 사용해서 소스 코드를 컴파일하고 실행가능한 프로그램 혹은 라이브러리로 만들 수 있다. 하지만, Make를 사용해서 다음도 할 수 있다:

- 원데이터를 요약하는 분석용 데이터 파일을 얻는데, 원데이터 파일에 분석 스크립트를 실행한다.
- 그래프를 그려 도식화하는데, 필요한 데이터 파일에 시각화 스크립트를 실행한다.
- 텍스트 파일과 그림을 파싱하고 조합해서 논문을 자동 생성한다.

Make를 빌드 도구라고 부른다 — Make는 데이터 파일, 그래프, 논문, 프로그램 혹은 라이브러리를 빌드(build)한다. 원한다면, 기존에 존재하는 파일도 갱신할 수 있다.

Make는 Make가 생성한 파일과 이를 생성하는데 사용된 파일에 대한 의존성을 추적한다. 만약 원본 파일 (예를 들어, 데이터 파일) 중 하나가 변경되면, Make는 알아서 원본파일에 의존성을 갖는 파일(예를 들어, 그래프)을 재생성하고 갱신한다.

현재 빌드 도구는 많이 있다. 하지만, 시장에 나온 모든 빌드 도구는 Make와 같은 개념에 기초하고 있다.

유닉스 쉘에서 나온 `make`를 사용한다. 따라서, 쉘을 사용해서 디렉토리 목록을 살펴보고, 파일과 디렉토리를 생성, 복사, 삭제, 조회할 수 있고, 간단한 스크립트를 실행해본 이전 경험이 일부 필요하다.

47.1 사전 준비

이번 학습을 따라가는데 다음 파일을 다운로드한다:

1. `make-lesson.zip` 파일을 다운로드한다.

2. `make-lesson.zip` 파일을 배쉬 쉘(bash shell)을 경우해서 접근할 수 있는 디렉토리로 이동한다.
3. 배쉬 쉘 윈도우를 연다.
4. 파일을 다운로드한 디렉토리로 이동한다.
5. `make-lesson.zip` 압축 파일을 다음 명령어로 푼다:

```
$ unzip make-lesson.zip
```

6. 작업 디렉토리를 `make-lesson` 디렉토리로 변경한다:

```
$ cd make-lesson
```

Chapter 48

들어가며

소장하고 있는 책 중 지프의 법칙이 맞는지 관심이 있다고 가정해보자.

가장 자주 출현하는 단어는 두번째 가장 자주 출현하는 단어보다 거의 두배 나타난다. 이를 지프의 법칙, Zipf's Law이라고 부른다.

원데이터(책과 파이썬 스크립트)는 준비를 마쳤고 `head books/isles.txt` 명령어로 소장하고 있는 책 중 하나를 빠르게 살펴보자. 작업할 파이썬 스크립트와 데이터 파일은 다음과 같은 구조를 갖고 있다.

```
| - books
|   | - abyss.txt
|   | - isles.txt
|   | - last.txt
|   | - LICENSE_TEXTS.md
|   | - sierra.txt
|- plotcounts.py
|- countwords.py
|- testzipf.py
```

텍스트 파일을 읽어 들이고, 텍스트에 단어갯수를 세고, 출력을 데이터 파일에 저장하는 `countwords.py` 스크립트로 작성이 되어 있어 먼저 책의 단어 빈도수를 살펴보자.

```
$ python countwords.py books/isles.txt isles.dat
```

`head` 명령어를 사용해서 데이터 파일 첫 5행을 살펴본다면,

```
$ head -5 isles.dat
```

`isles.dat` 파일이 단어마다 한줄씩 구성된 것을 볼 수 있다.

```
the 3822 6.7371760973
of 2460 4.33632998414
and 1723 3.03719372466
```

```
to 1479 2.60708619778
a 1308 2.30565838181
```

각 행은, 단어 자체, 해당 단어 출현횟수, 출현횟수를 텍스트 파일에 전체 단어 갯수에 대한 백분율로 나타낸다. 또 다른 예제로:

```
$ python countwords.py books/abyss.txt abyss.dat
$ head -5 abyss.dat
```

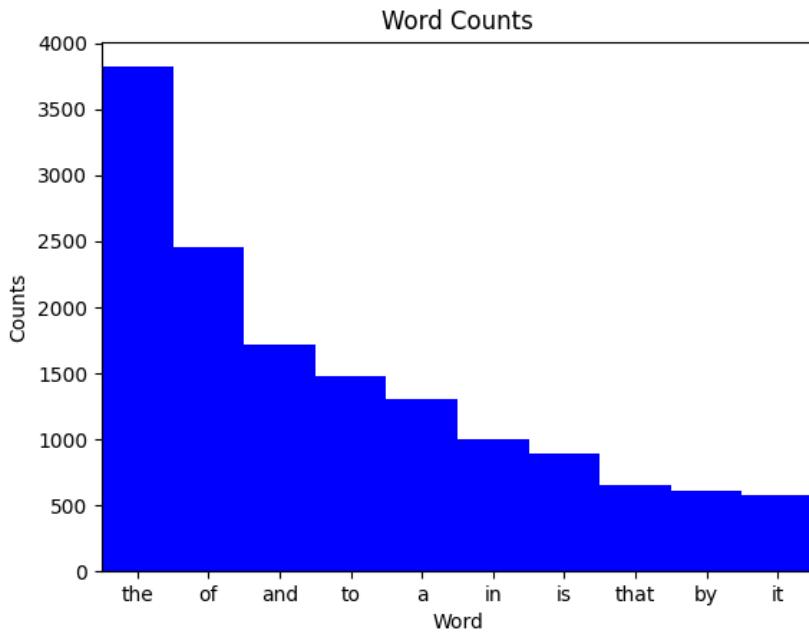
```
the 4044 6.35449402891
and 2807 4.41074795726
of 1907 2.99654305468
a 1594 2.50471401634
to 1515 2.38057825267
```

결과를 시각화하자. `plotcounts.py` 스크립트는 데이터 파일을 읽어들여, 가장 자주 출현하는 10 단어를 도식화한다:

```
$ python plotcounts.py isles.dat ascii
the #####
of #####
and #####
to #####
a #####
in #####
is #####
that #####
by #####
it #####
```

`plotcounts.py` 아스키가 아닌 그래프로 결과를 볼 수도 있다.

```
$ python plotcounts.py isles.dat show
```



윈도우를 닫아 그래프를 빠져나온다.

plotcounts.py로 그래프를 이미지(예, PNG 파일)로 저장할 수도 있다:

```
$ python plotcounts.py isles.dat isles.png
```

마지막으로 지프의 법칙을 앞서 분석한 책에 대해 실행해보자.

```
$ python testzipf.py abyss.dat isles.dat
```

| Book | First | Second | Ratio |
|-------|-------|--------|-------|
| abyss | 4044 | 2807 | 1.44 |
| isles | 3822 | 2460 | 1.55 |

상기 스크립트를 합하면, 작업흐름은 다음을 구현한 것이 된다:

1. 데이터 파일을 읽어들인다.
2. 해당 데이터 파일에 분석을 실시한다.
3. 분석 결과를 신규 파일에 저장한다.
4. 분석 결과에 대한 그래프를 그려 도식화한다.
5. 이미지 파일로 그래프를 저장해서, 논문에 삽입할 수 있게 된다.
6. 분석요약표를 제작한다.

지금까지 작업을 수행한 것처럼, 쉘 프롬프트에서 countwords.py 와 plotcounts.py 스크립트를 파일 한둘에 작업하는 것은 문제 없다. 하지만, 텍스트 파일이 5개, 10개, 20개가 된다면, 파이프라인에 작업량이 늘어나게 되면 어마어미한 작업량이

된다. 여기에 더해서 어느 누구도 심지어 30초 가량 소요된다고 하더라도 앓아서 명령이 종료되기를 원하지 않는다.

지루한 데이터 프로세싱 작업의 가장 일반적인 해법은 시작부터 종료까지 전체 작업과정을 담은 쉘스크립트를 작성하는 것이다.

예를 들어 나노(nano) 같은 텍스트 편집기를 사용해서 `run_pipeline.sh` 파일에 다음 사항을 추가한다.

```
# USAGE: bash run_pipeline.sh
# to produce plots for isles and abyss
# and the summary table for the Zipf's law tests

python countwords.py books/isles.txt isles.dat
python countwords.py books/abyss.txt abyss.dat

python plotcounts.py isles.dat isles.png
python plotcounts.py abyss.dat abyss.png

# Generate summary table
python testzipf.py abyss.dat isles.dat > results.txt
```

쉘 스크립트를 돌려 확인한다. 출력결과는 앞선 결과와 동일하다.

```
$ bash run_pipeline.sh
$ cat results.txt
```

쉘 스크립트를 작성하게 되면 재현성 관련 몇가지 문제를 해결해준다.

1. 명시적으로 파이프라인을 문서화해서 (미래 자신을 포함해서) 동료와 좀더 효율적으로 커뮤니케이션할 수 있게 돋는다.
2. `bash run_pipeline.sh` 명령어 하나를 타이핑함으로써 전체 분석을 재현할 수 있게 만들었다.
3. 따라서 반복적인 오탈자와 실수를 방지한다. 처음에는 제대로 동작하지 않을 수 있지만, 버그나 오류를 수정하고 나면 제대로 동작된 상태로 지속된다.

이와 같은 장점에도 불구하고 여전히 몇가지 단점도 존재한다.

`plotcounts.py` 파일로 생성되는 막대그래프 폭을 조정하는 경우를 상정해보자.

`plotcounts.py` 파일을 편집해서 막대 폭을 1.0 대신 0.8로 조정하다. `plot_word_counts` 함수 정의를 찾아 `width = 1.0` 을 `width = 0.8`으로 바꾼다.

이제 그레프를 다시 제작할 수 있다. `bash run_pipeline.sh` 스크립트만 다시 실행시키기만 하면 된다. 제대로 동작하기 때문에 큰 문제는 없지만, 단어 빈도수를 계산하는 부분이 실행에 오래 걸린다면 큰 문제가 아닐 수 없다. 단어 빈도수를 계산하는 루틴은 변경된 것이 없다; 따라서, 해당 로직을 다시 돌릴 필요는 없다.

대안으로 수작업으로 빈도수가 계산된 파일만 대상으로 그레프를 따로 제작한다.

```
for book in abyss isles; do
    python plotcounts.py $book.dat $book.png
done
```

하지만 이런 접근방법은 쉘 스크립트를 도입한 장점이 상당수 훼손되었다.

또다른 일반적인 방식은 run_pipeline.sh 스크립트 일부를 주석처리하여 실행되지 않게 하는 것이다.

```
# USAGE: bash run_pipeline.sh
# to produce plots for isles and abyss
# and the summary table for the Zipf's law tests.

# These lines are commented out because they don't need to be rerun.
#python countwords.py books/isles.txt isles.dat
#python countwords.py books/abyss.txt abyss.dat

python plotcounts.py isles.dat isles.png
python plotcounts.py abyss.dat abyss.png

# Generate summary table
# This line is also commented out because it doesn't need to be rerun.
#python testzipf.py abyss.dat isles.dat > results.txt
```

이렇게 작성하고 나서 bash run_pipeline.sh 명령어를 사용해서 수정한 쉘 스크립트를 돌려 원하는 결과를 얻어낸다.

하지만, 쉘 스크립트 일부를 주석처리해서 돌리고 나중에 다시 원복하는 방식은 번거롭고 데이터 처리 파이프라인이 복잡할 경우 오류가 발생할 개연성을 충분히 담게 된다.

Make(GNU Make로도 알려짐)는 빠르고, 자유롭게 사용 가능하고, 무료이며, 문서화가 잘된 빌드 관리자(build manager)다. Make는 1977년 벨 연구소 여름 인턴으로 근무한 Stuart Feldman에 의해 개발되었고, 오늘날까지 널리 사용되고 있다. Make는 분석을 실행하고, 결과를 도식화하는데 필요한 명령어를 실행한다. 쉘스크립트처럼, 단일 쉘 명령어를 통해, 복잡한 순서 명령어를 실행할 수 있다. 쉘스크립트와 달리, 명시적으로 파일 사이에 존재하는 의존성을 기록해서, 텍스트 파일이 변경될 때, 데이터 파일 혹은 이미지 파일을 언제 다시 재생성할지 결정할 수 있다. 새로운 파일을 생성하는데, 파일을 처리하는 일반적인 패턴을 따르는 어떤 명령어에도 Make를 사용할 수 있다. 예를 들어:

- 원데이터 파일에 분석 스크립트를 실행해서, 원 데이터를 요약하는 데이터를 생성한다.
- 상기 데이터에 시각화 스크립트를 실행해서, 그래프를 생성한다.
- 텍스트와 그래프를 파싱하고 조합해서 논문을 생성한다.
- 소스코드를 컴파일해서 실행 프로그램 혹은 라이브러리를 생성한다.

시중에 이용 가능한 빌드 도구가 많이 있다. 예를 들어, Apache ANT, doit, 윈도우용 nmake가 있다. 이런 빌드 도구를 사용하는 스크립트를 빌드하는 빌드 도구도 있다.

예를 들어, GNU Autoconf와 CMake가 있다. 어떤 도구가 최선인지는 요구사항, 사용 의도, 운영체제에 달려있다. 하지만, 거의 모든 빌드 도구는 Make와 동일한 기본 개념을 공유하고 있다.

거의 40살이 된 Make를 왜 사용할까?

오늘날, 고성능 컴퓨팅에 매우 일반적인 언어인 C 혹은 포트란 레거시 코드를 갖고 작업한 연구원은 아마도 Make를 접해봤을 것 같다.

재현 가능한 연구 작업흐름을 구현하거나, (R 혹은 파이썬을 사용해서) 자료분석과 시각화 작업을 자동화하거나, 텍스트를 가지고 표와 그래프를 조합해서 데출판을 위한 보고서와 논문을 작업하는데, Make를 사용했을 것이다.

Make의 기본적인 개념은 거의 모든 빌드도구에 공통이다.

GNU Make는 빠르고, 자유롭게 사용 가능하고, 무료이며, 문서화가 잘 정비되어 있고, 매우 인기가 있고, Make를 구현한 것 중 하나를 매우 잘 확장하고 있다. 이제부터, GNU Make에 집중한다. 따라서, Make를 언급하면, GNU Make를 의미한다.

Chapter 49

Makefiles

다음 내용을 갖는 Makefile로 불리는 파일을 생성한다:

```
# .  
isles.dat : books/isles.txt  
    python wordcount.py books/isles.txt isles.dat
```

간단한 빌드 파일로, Make를 사용할 때 Makefile로 불리우며 - Make가 실행하는 파일이다. 순차적으로 각 라인을 차차히 살펴보자:

- # 은 주석을 나타낸다. # 시작되는 어떤 텍스트나 끝 줄까지 Make가 무시한다.
- isles.dat 는 대상(target)으로, 생성되거나 빌드되는 파일이다.
- books/isles.txt 파일은 의존성(dependency)으로, 대상을 빌드하거나 갱신하는데 필요한 파일이 된다. 대상은 의존성을 0 혹은 여러개 갖을 수 있다.
- : 은 구분자로 대상과 의존성을 구별한다.
- python countwords.py books/isles.txt isles.dat 은 동작(action)으로, 의존성을 사용해서 대상을 빌드하거나 갱신하는데 실행되는 명령어가 된다. 대상은 의존성을 0 혹은 여러개 갖을 수 있다.
- 동작은 공백 8자가 아니라, 탭(TAB) 문자를 사용해서 들여쓰기 한다. 1970년부터 사용된 Make 유산이다.
- 대상, 의존성, 동작이 모여 규칙(rule)을 구성하게 된다.

상기 규칙은 python countwords.py 동작과 books/isles.txt 의존성을 사용해서 isles.dat 대상을 빌드하는 방식을 기술하고 있다.

아무것도 없는 처음 상태에서 시작하도록, 생성했던 .dat 와 .png 확장자를 갖는 파일을 삭제한다:

```
$ rm *.dat *.png
```

기본 디폴트 설정으로, Make는 Makefile로 불리는 Makefile을 찾는다. 그리고, Make를 다음과 같이 실행한다.

```
$ make
```

Make는 실행하는 동작을 화면에 출력한다:

```
python countwords.py books/isles.txt isles.dat
```

만약 다음 메시지를 보게 되면,

```
Makefile:3: *** missing separator. Stop.
```

동작 중 하나를 들여쓰는데 탭 문자 대신에 공백을 사용했기 때문이다.

기대한 결과가 나왔는지 확인해보자.

```
head -5 isles.dat
the 3822 6.737176097303014
of 2460 4.336329984135378
and 1723 3.0371937246606735
to 1479 2.607086197778953
a 1308 2.305658381808567
```

isles.dat 파일 첫 5행이 이전 실행결과와 정확하게 동일해야 된다.

Makefile을 Makefile로 호명할 필요는 없다. 하지만, 그밖의 다른 것으로 호명하려면, Make가 어디서 찾을 수 있는지 일러줄 필요가 있다. -f 플래그를 사용해서 파일명을 지정할 수 있다. 예를 들어:

```
$ make -f MyOtherMakefile
```

Makefile을 재설하면, Make가 다음과 같이 정보를 준다:

```
make: `isles.dat' is up to date.
```

상기와 같이 출력되는 이유는 대상 isles.dat 파일이 이제 생성되어서, Make가 다시 생성하기 않기 때문이다. 각동 방식을 살펴보기 위해서, 텍스트 파일 중 하나를 갱신한 척 해보자. 편집기에서 파일을 여는 대신에, 쉘 touch 명령어를 사용해서 시간도장(timestamp)을 갱신한다. 시간도장 갱신은 파일을 편집하면 발생되게 된다:

```
$ touch books/isles.txt
```

books/isles.txt 파일과 isles.dat 파일 시간도장(timestamp)을 비교하면,

```
$ ls -l books/isles.txt isles.dat
```

대상 isles.dat 파일이 의존성 books/isles.txt 파일보다 더 이전 파일임을 알게 된다:

```
-rw-r--r--    1 mjj      Administ   323972 Jun 12 10:35 books/isles.txt
-rw-r--r--    1 mjj      Administ   182273 Jun 12 09:58 isles.dat
```

Make를 다시 실행하면,

```
$ make
```

isles.dat 파일을 다시 생성시킨다:

```
python countwords.py books/isles.txt isles.dat
```

대상을 빌드하도록 Make가 요청받을 때, Make는 대상과 의존성 모두의 '최종 변경 시간'을 검사한다. 만약 대상과 비교해서 의존성이 갱신된 것이 있다면, 대상을 갱신하도록 동작을 다시 실행한다. 이러한 접근법을 사용해서 Make는 직간접적으로 변경된 파일만 다시 빌드작업을 수행한다. 이것을 **증분 빌드(incremental build)**라고 한다.

문서로 Makefiles

파일간 의존성과 함께 분석 단계 입력과 출력을 명시적으로 기록함으로써, Makefiles은 일종의 문서(documentation)로 역할을 수행하여 작업자가 기억해야 되는 것을 대폭 줄여준다.

Makefile 파일 후미에 또 다른 규칙을 추가하자:

```
abyss.dat : books/abyss.txt
            python wordcount.py books/abyss.txt abyss.dat
```

Make를 실행하면,

```
$ make
```

다음 결과를 얻게 된다:

```
make: `isles.dat' is up to date.
```

아무일도 발생하지 않는 데, 이유는 Make가 Makefile에서 찾게되는 첫번째 대상만 빌드하려고 하기 때문이다. 첫번째 대상이 **기본 디폴트 설정 대상(default target)**으로 isles.dat 파일이 되는데 이미 최신 상태다. 명시적으로 Make에게 abyss.dat 파일을 빌드한다는 의도를 일러줄 필요가 있다:

```
$ make abyss.dat
```

이제 다음 결과를 얻게 된다:

```
python countwords.py books/abyss.txt abyss.dat
```

"Up to Date" vs "Nothing to be Done"

Make가 이미 존재하고 가장 최신(up to date) 파일을 빌드하는 경우 Make는 다음 사항을 알려준다.

```
make: `isles.dat' is up to date.
```

Make가 이미 존재하지만 Makefile에 규칙(rule)이 없는 경우는, 다음 메시지를 Make가 알려준다.

```
$ make countwords.py
```

```
make: Nothing to be done for `countwords.py'.
```

up to date 가 의미하는 바는 Makefile에 파일(혹은 디렉토리)과 연관된 액션을 갖는 규칙이 있지만 파일이 가장 최신 상태임을 반증한다. Nothing to be done은 파일이 존재는 하지만 다음 사항을 나타내고 있다.

- Makefile이 해당 규칙이 없다.
- Makefile이 규칙은 있으나 액션을 갖고 있지 않다.

작업한 모든 파일을 제거하고 나서 명시적으로 전부 재생성할 수 있다. 이 작업으로 새로운 대상과 연관된 규칙 clean을 도입한다: .dat 파일처럼 자동생성된 파일을 제거하는 일반적인 명칭이다.

```
clean :
    rm -f *.dat
```

상기 규칙예제가 의존성을 갖지 않는 사례다. clean은 어떤 .dat 파일에 대해서도 의존성이 없다. 왜냐하면, 금방 삭제될 파일을 다시 생성하는 것이 무의미하기 때문이다. 파일이 존재하든 존재하지 않던 데이터 파일을 제거하고자 한다. 대상을 명세하고 Make를 실행하면,

```
$ make clean
```

다음 결과를 얻게 된다:

```
rm -f *.dat
```

clean으로 실제로 어떤 것도 빌드되지는 않는다. 오히려, 유용한 연속 동작을 실행하는데 사용하는 약칭이 된다. 매우 유용하지만, 이런 대상이 문제를 불러일으킬 수 있다. 예를 들어, 데이터 파일을 다시 생성하고, clean이라는 디렉토리를 생성하고 나서, Make를 실행한다:

```
$ make isles.dat abyss.dat
$ mkdir clean
$ make clean
```

다음 결과를 얻게 된다:

```
make: `clean' is up to date.
```

Make가 clean이라는 파일(혹은 디렉토리)을 찾는다. clean 규칙에는 어떤 의존성도 없기 때문에, clean이 빌드되어서 최신 상태라고 가정하게 된다. 그래서, 규칙에 나온 동작을 하나도 실행하지 않는다. clean을 약칭으로 사용할 때, make clean 명령어를 실행하면, 항상 해당 규칙을 Make가 실행하도록 일러줄 필요가 있다. Make에게 어떤 것도 빌드하지 않는 **가짜 대상(phony target)**이라고 일러줘서 이 문제를 해결한다. 대상을 .PHONY로 표식해서 해당 작업을 수행한다:

```
.PHONY : clean
clean :
    rm -f *.dat
```

Make를 실행하면,

```
$ make clean
```

다음 결과를 얻는다:

```
rm -f *.dat
```

모든 데이터 파일을 생성하도록 유사한 명령어를 추가할 수도 있다:

```
.PHONY : dats
dats : isles.dat abyss.dat
```

상기 규칙예제가 다른 규칙에 대상이 되는 의존성을 갖는 규칙이다. Make를 실행하면, 의존성이 존재하는지 검사하고, 만약 의존성이 없다면, 해당 파일을 생성하는 규칙이 있는지를 살펴본다. 만약 그런 규칙이 존재하면, 먼저 동작시키게 되고, 그렇지 않다면 Make가 오류를 발생시킨다.

의존성(Dependencies)

의존성을 다시 빌드하는 순서는 무작위다. 목록에 기입된 순서로 빌드된다고 가정하지 말아야 된다.

의존성은 방향성 있는 주기가 없는 그래프(directed acyclic graph) 형태를 띤다. 대상이 본인 혹은 의존성을 갖는 하나가 대상에 의존성을 갖을 수 없다.

동작을 갖지 않는 규칙의 한 사례이기도 하다. 필요한 경우, 순전히 의존성 빌드를 촉발하는데 사용되기도 하다.

만약 상기 규칙을 실행하면,

```
$ make dats
```

Make가 data 파일을 생성하게 된다:

```
python wordcount.py books/isles.txt isles.dat
python wordcount.py books/abyss.txt abyss.dat
```

dats를 다시 실행하면,

```
$ make dats
```

Make가 데이터 파일이 존재하는 것을 알게 된다:

```
make: Nothing to be done for `dats'.
```

이제 최종 Makefile은 다음과 같다:

```
# Count words.
.PHONY : dats
dats : isles.dat abyss.dat

isles.dat : books/isles.txt
    python wordcount.py books/isles.txt isles.dat

abyss.dat : books/abyss.txt
    python wordcount.py books/abyss.txt abyss.dat

.PHONY : clean
clean :
    rm -f *.dat
```

dats에 명기된 대상을 빌드하는데 관여된 의존성을 도식화했는데, Makefile에서 구현된 사항이 다음 그림에 나와 있다:

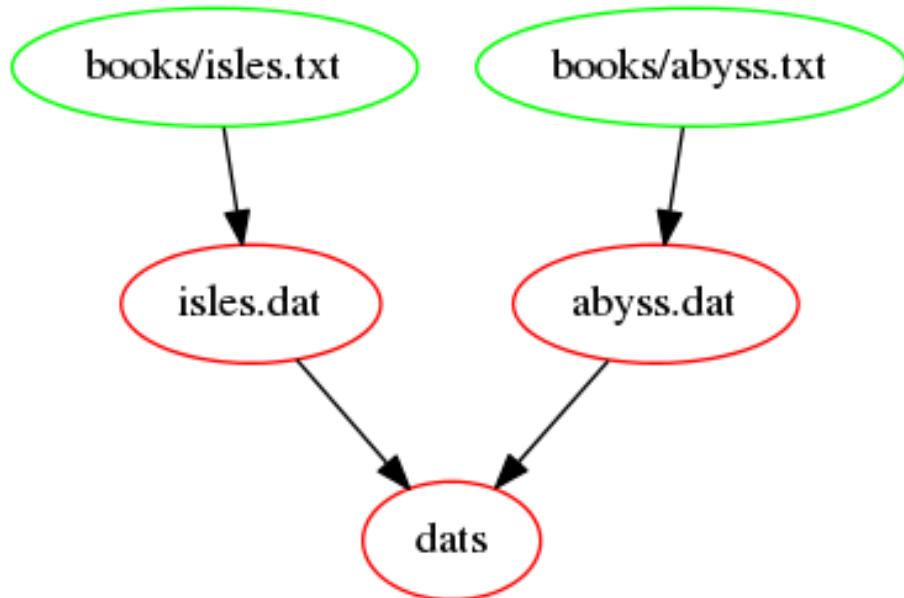


Figure 49.1: Makefile 내부에 표현된 의존성

49.1 신규 규칙 두개 작성

1. last.dat 파일에 대한 새로운 규칙을 작성하는데, books/last.txt 파일에서 생성된다.
2. dats 규칙을 상기 대상을 포함하도록 간신하라.
3. analysis.tar.gz 에 대한 새로운 규칙을 작성하는데, 데이터 파일을 보관하도록 압축파일이 생성된다. 규칙은 다음 작업이 필요하다:
 - 세가지 .dat 파일 각각에 의존성을 갖는다.
 - python testzipf.py abyss.dat isles.dat last.dat > results.txt 동작을 호출한다.
4. Makefile 상단에 규칙을 두어 디폴트기본 타겟이 되게 한다.
5. results.txt 파일을 제거하도록 clean을 간신한다.

시작 Makefile은 다음과 같다.

```

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat

isles.dat : books/isles.txt
    python countwords.py books/isles.txt isles.dat
  
```

```
abyss.dat : books/abyss.txt
    python countwords.py books/abyss.txt abyss.dat

.PHONY : clean
clean :
    rm -f *.dat

도전과제에 대한 해답

# Generate summary table.
results.txt : isles.dat abyss.dat last.dat
    python testzipf.py abyss.dat isles.dat last.dat > results.txt

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

isles.dat : books/isles.txt
    python countwords.py books/isles.txt isles.dat

abyss.dat : books/abyss.txt
    python countwords.py books/abyss.txt abyss.dat

last.dat : books/last.txt
    python countwords.py books/last.txt last.dat

.PHONY : clean
clean :
    rm -f *.dat
    rm -f results.txt
```

results.txt에 명기된 대상을 빌드하는데 관여된 의존성을 도식화했는데, Makefile에서 구현된 사항이 다음 그림에 나와 있다:

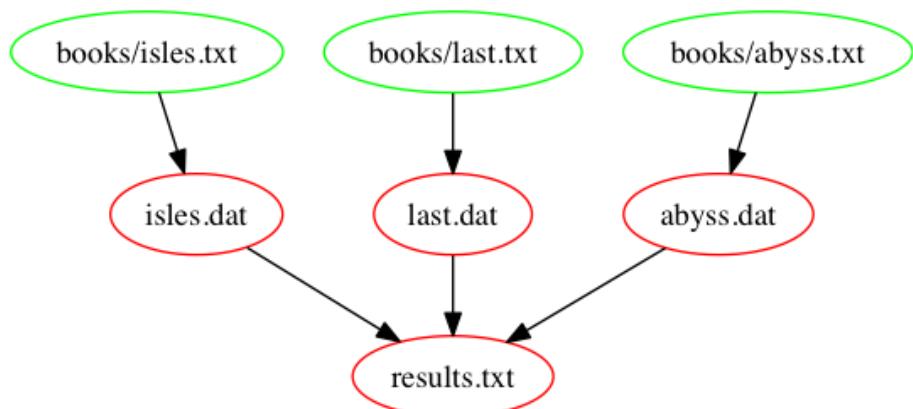


Figure 49.2: Makefile 내부에 표현된 results.txt 의존성

Chapter 50

자동 변수

앞선 수업 말미 연습문제를 푼 후, Makefile 은 다음과 같아 보인다:

```
# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

isles.dat : books/isles.txt
    python wordcount.py books/isles.txt isles.dat

abyss.dat : books/abyss.txt
    python wordcount.py books/abyss.txt abyss.dat

last.dat : books/last.txt
    python wordcount.py books/last.txt last.dat

# Generate archive file.
analysis.tar.gz : isles.dat abyss.dat last.dat
    tar -czf analysis.tar.gz isles.dat abyss.dat last.dat

.PHONY : clean
clean :
    rm -f *.dat
    rm -f analysis.tar.gz
```

Makefile에 중복이 엄청 많다. 예를 들어, 텍스트 파일과 데이터 파일 명칭이 Makefile 전역에 걸쳐 여러 곳에서 반복되고 있다. Makefile은 코드의 한 형태로, 어떤 코드에서나 그렇듯이 반복되는 코드는 문제가 될 소지가 있다. 예를 들어, Makefile의 일부에서 데이터 파일 명칭을 바꾸고 나서 다른 곳에 명칭을 바꾸는 것을 잊은 한다.

D.R.Y.(Don't Repeat Yourself)

대다수 프로그래밍 언어에서 프로그래머가 긴 연산과정 루틴을 간결하고 표현력있고 아름다운 코드로 기술할 수 있는 기능을 지원한다. R, 파이썬, 자바 언어에 사용자 정의 변수와 함수 기능이 유용한데 이유는 구체적인 모든 사항을 반복해서 작성할 필요가 없기 때문이다. 단지 한번만 작성하는 이러한 좋은 프로그래밍 습관이 D.R.Y.(Don't Repeat Yourself) 원칙으로 알려져 있다. 우리나라 말로 복붙, 삽질 하지말자 정도가 체감상 맞을 듯 싶다.

Makefile에서 반복되는 코드를 제거해 나가자. `results.txt` 규칙에서, 데이터 파일과 파일 저장소 아카이브 명칭이 중복된다:

```
results.txt : isles.dat abyss.dat last.dat
    python testzipf.py abyss.dat isles.dat last.dat > results.txt
```

먼저 문서저장소 아카이브 명칭을 살펴보면, 동작에 나온 문서저장소 명칭을 `$@`으로 교체할 수 있다:

```
results.txt : isles.dat abyss.dat last.dat
    python testzipf.py abyss.dat isles.dat last.dat > $@
```

`$@`은 Make 자동 변수(automatic variable)로, '현재 규칙에 대한 대상'을 의미한다. Make가 실행되면, Make가 해당 변수를 대상 명칭으로 교체한다.

동작에 나온 의존성을 `$~`으로 교체할 수 있다:

```
results.txt : isles.dat abyss.dat last.dat
    python testzipf.py $~ > $@
```

`$~`은 또 다른 자동변수로, '현재 규칙에 대한 모든 의존성'을 의미한다. 다시 한번, Make가 실행될 때, Make가 해당 변수를 의존성으로 교체한다.

텍스트 파일을 갱신하고, 위에서 작성한 규칙을 재실행한다:

```
$ touch books/*.txt
$ make results.txt
```

다음에 산출 결과가 나와 있다:

```
python countwords.py books/isles.txt isles.dat
python countwords.py books/abyss.txt abyss.dat
python countwords.py books/last.txt last.dat
python testzipf.py isles.dat abyss.dat last.dat > results.txt
```

50.1 의존성 갱신하기

지금 다음을 실행하면 어떤 일이 발생할까:

```
$ touch *.dat
$ make analysis.tar.gz
```

1. 아무 일도 없다.
2. 모든 파일이 다시 생성된다.
3. 단지 .dat 파일만 다시 생성된다.

4. 단지 results.txt만 다시 생성된다.

도전과제 해법 4. 단지 results.txt만 다시 생성된다.

*.dat에 대한 규칙은 실행되지 않는데 이유는 해당되는 .txt 파일 변경이 일어나지 않아서 그렇다.

다음을 실행하게 되면

```
$ touch books/*.txt
$ make results.txt
```

.dat 파일과 results.txt 파일이 다시 생성된 것을 확인할 수 있다.

위에서 살펴봤듯이, \$^은 '현재 규칙에 대한 모든 의존성'을 의미한다. results.txt 파일에 대해서는 잘 동작하는데, 이유는 해당 동작이 모든 의존성을 testzipf.py에 입력처럼 동일하게 처리하기 때문이다.

하지만, 일부 규칙에 대해, 첫째 의존성을 다르게 처리하고 싶을 수 있다. 예를 들어, .dat 파일에 대한 규칙은 첫째 의존성(만을) 사용해서 wordcount.py에 입력 파일로 적용코져 한다. 만약 추가적인 의존성을 추가하면(곧 그렇듯이), countwords.py에 입력 파일로 전달되면 안된다. 왜냐하면, 호출될 때 입력 파일만 호명되길 기대하기 때문이다.

Make는 이런 목적을 위한 자동변수가 있는데 \$<으로, '현재 규칙에 대한 첫째 의존성'을 의미한다.

50.2 자동변수 규칙 작성

자동 변수를 사용하도록 .dat 규칙을 다시 작성하시오.

자동 변수 \$@('현재 규칙에 대한 대상')와 \$< ('현재 규칙에 대한 첫째 의존성')을 사용하도록 .dat 규칙을 다시 작성하시오.

```
# Generate summary table.
results.txt : *.dat
    python testzipf.py $^ > $@

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

isles.dat : books/isles.txt
    python countwords.py books/isles.txt isles.dat

abyss.dat : books/abyss.txt
    python countwords.py books/abyss.txt abyss.dat

last.dat : books/last.txt
    python countwords.py books/last.txt last.dat
```

```
.PHONY : clean
clean :
    rm -f *.dat
    rm -f results.txt

자동변수 적용 해답

# Generate summary table.
results.txt : isles.dat abyss.dat last.dat
    python testzipf.py $^ > $@

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

isles.dat : books/isles.txt
    python countwords.py $< $@

abyss.dat : books/abyss.txt
    python countwords.py $< $@

last.dat : books/last.txt
    python countwords.py $< $@

.PHONY : clean
clean :
    rm -f *.dat
    rm -f results.txt
```

50.3 자동변수 요약

- Make 자동변수를 사용해서 Makefile에 중복을 제거한다.
- \$@을 사용해서 현재 규칙에 있는 대상을 참조한다.
- \$^을 사용해서 현재 규칙에 있는 의존성을 참조한다.
- \$<을 사용해서 현재 규칙에 있는 첫째 의존성을 참조한다.

Chapter 51

데이터와 코드 의존성

지금까지 작성한 Makefile은 다음과 같다:

```
# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

isles.dat : books/isles.txt
    python wordcount.py $< $@

abyss.dat : books/abyss.txt
    python wordcount.py $< $@

last.dat : books/last.txt
    python wordcount.py $< $@

# Generate archive file.
analysis.tar.gz : *.dat
    tar -czf $@ $^

.PHONY : clean
clean :
    rm -f *.dat
    rm -f analysis.tar.gz
```

데이터 파일은 텍스트 파일에 대한 제품이기도 하지만, 텍스트 파일을 처리하고 데이터 파일을 생성하는 countwords.py, 스크립트에 대한 제품이기도 하다. countwords.py 파일 수정(예를 들어 요약 데이터 신규 칼럼 추가 혹은 기존 요약결과 제거 등)은 출력결과를 .dat 파일변경에도 일조를 하게 된다. 따라서, touch 명령어를 사용해서 countwords.py 파일을 수정한 것처럼 하고 Make를 다시 실행시키자.

```
$ make dats
$ touch countwords.py
$ make dats
```

아무런 일도 발생하지 않았다! `countwords.py` 파일을 수정했지만, 데이터 파일을 갱신하지 않아서 `.dat` 파일 생성에 관여하는 규칙이 `countwords.py` 파일 의존성을 기록하지 못했다.

또한, `countwords.py` 파일을 데이터 파일에 대한 의존성으로 추가해야만 된다:

```
isles.dat : books/isles.txt countwords.py
```

```
    python countwords.py $< $@
```

```
abyss.dat : books/abyss.txt countwords.py
```

```
    python countwords.py $< $@
```

```
last.dat : books/last.txt countwords.py
```

```
    python countwords.py $< $@
```

`wordcount.py` 프로그램을 편집한 척하고, Make를 재실행하면,

```
$ touch wordcount.py
```

```
$ make dats
```

다음 결과를 얻게 된다:

```
python countwords.py books/isles.txt isles.dat
```

```
python countwords.py books/abyss.txt abyss.dat
```

```
python countwords.py books/last.txt last.dat
```

시운전 (Dry Run)

`make` 명령어를 실행할 때 `-n` 플래그를 사용하게 되면 실제 명령을 실행하지 않고 실행할 명령어를 보여준다.

```
$ touch countwords.py
```

```
$ make -n dats
```

`-n` 플래그 없이 화면에 동일한 출력결과를 보여주지만, 실제 명령은 실행되지 않는다. ‘dry-run’ 모드를 사용해서 실제 둘리지 건에 `Makefile`이 제대로 설정되었는지 사전 확인할 수 있다.

`countwords.py`와 `testzipf.py` 파일을 `.dat` 파일에 의존성으로 추가한 후, `results.txt`에 명기된 대상을 빌드하는데 관여된 의존성을 도식화했는데, `Makefile`에서 구현된 사항이 다음 그림에 나와 있다:

.txt 파일은 `wordcount.py` 파일에 의존성을 갖지 않는가?

`.txt` 파일은 입력 파일이며 어떤 의존성도 갖지 않는다. 입력파일이 `wordcount.py` 파일에 의존성을 만드려면, 거짓 의존성(**false dependency**) 도입이 필요하다.

직관적으로 `results.txt` 파일에 의존성을 `countwords.py` 파일에 추가해야 한다. `.dat` 파일을 다시 만드는데 최종표가 빌드되어야 하기 때문이다. 하지만 그럴필요가

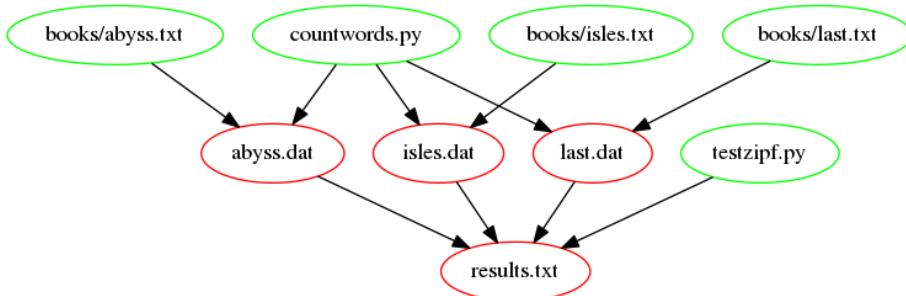


Figure 51.1: `countwords.py`와 `testzipf.py` 파일을 의존성으로 추가한 후에, `results.txt` 의존성

없다는 것이 밝혀졌다. `countwords.py` 파일을 갱신할 때 `results.txt` 파일에 생긴 일을 살펴보자.

```
$ touch countwords.py
$ make results.txt
```

상기 명령을 실행하면 다음을 얻게 된다.

```
python countwords.py books/abyss.txt abyss.dat
python countwords.py books/isles.txt isles.dat
python countwords.py books/last.txt last.dat
python testzipf.py abyss.dat isles.dat last.dat > results.txt
```

`results.txt` 파일 포함해서 전체 파이프라인이 촉발되어 실행되었다. 이 과정을 이해하기 위해 의존성 그래프에 따르면 `results.txt` 파일은 `.dat` 파일에 의존성을 갖는다. `countwords.py` 파일을 갱신하게 되면 `.dat` 파일 갱신을 촉발시킨다. 따라서, `make`가 `.dat` 파일 의존성이 `results.txt` 타겟 파일보다 신규 상태임을 인식하게 되어 `results.txt` 파일을 다시 만들어낸다. 이것이 `make`의 강력함을 보여주는 한 사례다: 파이프라인의 일부 파일이 갱신되면 적절한 후속 단계를 자동 실행시킨다.

51.1 입력파일 갱신

다음 명령을 실행시키게 되면 어떤 결과가 나오게 될까?

```
$ touch books/last.txt
$ make results.txt
```

1. `last.dat` 파일만 다시 생성된다.
2. 모든 `.dat` 확장자를 갖는 파일이 다시 생성된다.
3. `last.dat, results.txt` 파일만 다시 생성된다.
4. 모든 `.dat, results.txt` 파일만 다시 생성된다.

해답 3. `last.dat, results.txt` 파일만 다시 생성된다.

의존성 나무그래프를 따라가면 정답이 명확하게 이해된다.

51.2 results.txt 의존성 testzipf.py

results.txt 파일의 의존성에 testzipf.py 파일을 추가하면 어떻게 될까? 그리고 이유는 무엇일까?

해답

다음과 같이 results.txt 파일에 규칙을 추가하게 되면

```
results.txt : isles.dat abyss.dat last.dat testzipf.py
            python testzipf.py $^ > $@
```

testzipf.py 는 \$^의 일부가 되어 명령어는 사실 다음과 같이 된다.

```
python testzipf.py abyss.dat isles.dat last.dat testzipf.py > results.txt
```

testzipf.py 파일에서 .dat 파일처럼 스크립트를 파싱하게 되어 오류가 발생한다. 오류가 나온 것을 실제로 돌려 확인해보자.

```
$ make results.txt
```

다음과 같은 결과가 나오게 된다.

```
python testzipf.py abyss.dat isles.dat last.dat testzipf.py > results.txt
Traceback (most recent call last):
  File "testzipf.py", line 19, in <module>
    counts = load_word_counts(input_file)
  File "path/to/testzipf.py", line 39, in load_word_counts
    counts.append((fields[0], int(fields[1]), float(fields[2])))
IndexError: list index out of range
make: *** [results.txt] Error 1
```

results.txt 파일에 대한 의존성에 testzipf.py 스크립트를 반영해야 된다. 앞선 사례를 통해 \$^ 규칙을 사용할 수는 없다는 것이 확인되었다. 하지만, testzipf.py 파일을 첫번째 의존성으로 이동하고 나서 \$<을 사용해서 참조하면 된다. .dat 파일을 지칭하는데는 임시로 *.dat를 사용한다.(추후 더 좋은 해법을 다룰 것이다.)

```
results.txt : testzipf.py isles.dat abyss.dat last.dat
            python $< *.dat > $@
```

현재까지 다른 Makefile

```
# Generate summary table.
results.txt : testzipf.py isles.dat abyss.dat last.dat
            python $< *.dat > $@

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat
```

```
isles.dat : books/isles.txt countwords.py
    python countwords.py $< $@

abyss.dat : books/abyss.txt countwords.py
    python countwords.py $< $@

last.dat : books/last.txt countwords.py
    python countwords.py $< $@

.PHONY : clean
clean :
    rm -f *.dat
    rm -f results.txt
```


Chapter 52

패턴 규칙

Makefile에는 여전히 반복되는 콘텐츠가 있다. 텍스트 파일과 데이터 파일 명칭을 제외하고 각 .dat 파일에 대한 규칙은 동일한다. 이러한 규칙을 단일 **패턴 규칙(pattern rule)**으로 교체할 수 있는데, 패턴 규칙을 사용해서 books/ 디렉토리에 .txt 파일을 임의 .dat 파일로 빌드한다:

```
% .dat : books/%.txt countwords.py  
        python countwords.py $< $*.dat
```

%는 Make **와일드-카드(wild-card)**다. \$*은 특수 변수로, 특수변수가 스템(stem)¹을 규칙이 매칭하는 것으로 치환한다.

상기 규칙은 다음과 같이 해석할 수 있다: [something].dat 타겟을 갖는 파일을 빌드하는데 books/[that same something].txt 의존성을 갖는 파일을 찾아 countwords.py [the dependency] [the target] 명령을 실행한다.

만약 Make를 다시 실행하면,

```
$ make clean  
$ make dats
```

다음 결과를 얻게 된다:

```
python wordcount.py books/isles.txt isles.dat  
python wordcount.py books/abyss.txt abyss.dat  
python wordcount.py books/last.txt last.dat
```

이전과 마찬가지로 개별 .dat 타겟을 빌드하는데 Make를 여전히 사용할 수 있다. 어떤 스템이 매칭되더라도 신규 규칙은 여전히 정상동작한다.

Make 와일드-카드 사용하기

¹스템은 패턴규칙으로 매칭되는 타겟의 일부. 만약 file.dat가 타겟이고 패턴 규칙이 %.dat인 경우 \$* 스템은 file 이 된다.

Make % 와일드-카드는 대상과 해당 의존성에만 사용될 수 있다. 동작에는 사용될 수가 없다.

하지만, 동작에서 \$*을 사용할 수 있는데, 스템이 규칙이 매칭하는 것으로 치환한다.

Makefile은 이제 훨씬 더 짧아졌고, 깨끗해졌다:

```
# Generate summary table.
results.txt : testzipf.py isles.dat abyss.dat last.dat
    python $< *.dat > $@

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

%.dat : books/%.txt countwords.py
    python countwords.py $< $*.dat

.PHONY : clean
clean :
    rm -f *.dat
    rm -f results.txt
```

현재까지 다룬 Makefile

```
# Generate summary table.
results.txt : testzipf.py isles.dat abyss.dat last.dat
    python $< *.dat > $@

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

%.dat : books/%.txt countwords.py
    python countwords.py $< $*.dat

.PHONY : clean
clean :
    rm -f *.dat
    rm -f results.txt
```

패턴 규칙을 소개했고 dat 규칙에 \$*을 사용해서 사용법을 설명했다. 좀더 깔끔한 해법은 \$@을 사용해서 현재 규칙의 타겟을 명시하는 것이지만 그러면 \$*을 학습할 이유는 없을 것이다.

```
%.dat : books/%.txt countwords.py
    python countwords.py $< $@
```

52.1 패턴 규칙 주요점

- Make 와일드-카드 %를 대상과 의존성에 사용한다.
- Make 특수 변수 \$*을 동작에 사용한다.
- 규칙에는 Make 와일드-카드 사용을 회피한다.

Chapter 53

변수

지금까지의 노력에도 불구하고, Makefile에는 여전히 중복된 콘텐트, 즉 countwords.py 스크립트 명칭과 스크립트를 실행하는데 동원된 프로그램 python이 존재한다. 해당 스크립트의 이름을 바꾸면, 여러 곳에서 Makefile을 갱신해야만 된다.

스크립트 명칭을 보관하는 Make **변수(variable)** (Make 일부 버전에서는 **매크로(macro)**라고도 불림)를 소개한다:

```
COUNT_SRC=countwords.py
```

변수 할당(**assignment**)하는 예제가 위에 나와 있다 - COUNT_SRC 변수에 countwords.py 값이 할당되었다.

countwords.py는 스크립트로, 파이썬(python)에 전달되어 호출된다. 또 다른 변수를 도입해서 스크립트 실행을 표현한다:

```
LANGUAGE=python  
COUNT_EXE=$(LANGUAGE) $(COUNT_SRC)
```

Make가 실행될 때, \$(...)는 Make에게 변수를 해당 값으로 치환하도록 일러준다. 이것이 변수 참조(**reference**)다. 변수값을 사용하고자 하는 어떤 곳에서도, 이런 방식으로 작성하고 참조한다.

여기서는 COUNT_SRC 변수를 참조한다. Make로 하여금 COUNT_SRC 변수를 countwords.py 값으로 치환하게 한다. Make가 실행될 때, COUNT_EXE에 값으로 python countwords.py를 할당한다.

이런 방식으로 COUNT_EXE 변수를 정의하면 스크립트 실행방식을 쉽게 변경할 수 있게 된다 (예를 들어, 스크립트 구현 언어를 파이썬에서 R로 변경하게 되면).

변수 사용하기

Makefile를 갱신해서, %.dat 규칙이 COUNT_SRC 와 COUNT_EXE 변수를 참조하게 한다. 그리고 나서 ZIPF_SRC 와 ZIPF_EXE 변수명을 사용해서 testzipf.py

스크립트와 results.txt 규칙에 대해서도 동일하게 작업한다.

해답

```
LANGUAGE=python
COUNT_SRC=countwords.py
COUNT_EXE=$(LANGUAGE) $(COUNT_SRC)
ZIPF_SRC=testzipf.py
ZIPF_EXE=$(LANGUAGE) $(ZIPF_SRC)

# Generate summary table.
results.txt : $(ZIPF_SRC) isles.dat abyss.dat last.dat
$(ZIPF_EXE) *.dat > $@

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

%.dat : books/%.txt $(COUNT_SRC)
$(COUNT_EXE) $< $*.dat

.PHONY : clean
clean :
rm -f *.dat
rm -f results.txt
```

Makefile 상단에 변수를 위치시키게 되면, 찾고 변경하기 쉽게 한다는 의미가 된다. 대안으로, 변수 정보만 간직하는 Makefile을 새로 만들어 넣을 수 있다. config.mk 파일을 생성하자:

```
# Count words script.
LANGUAGE=python
COUNT_SRC=countwords.py
COUNT_EXE=$(LANGUAGE) $(COUNT_SRC)

# Test Zipf's rule
ZIPF_SRC=testzipf.py
ZIPF_EXE=$(LANGUAGE) $(ZIPF_SRC)
```

다음 명령어를 사용해서, config.mk 파일을 Makefile 내부로 Makefile을 가져올 수 있다:

```
include config.mk
```

Make를 재실행해서, 모든 것이 여전히 잘 동작하는지 살펴보자:

```
$ make clean
$ make dats
$ make results.txt
```

Makefile 구성정보와 작업을 수행하는 부분, 즉 규칙을 구분했다. 스크립트 명칭 혹은 실행방식을 변경하려면, Makefile에 있는 소스코드가 아니라, 단지 구성파일만 편집하면 된다. 이런 방식으로 코드를 구성과 결합시키지 않는 것이 좋은 프로그래밍 관례다. 더 모듈화 되고, 유연해지며, 재사용 가능한 코드가 되기 때문이다.

현재까지 다룬 Makefile

```
include config.mk

# Generate summary table.
results.txt : $(ZIPF_SRC) isles.dat abyss.dat last.dat
    $(ZIPF_EXE) *.dat > $@

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

%.dat : books/%.txt $(COUNT_SRC)
    $(COUNT_EXE) $< $*.dat

.PHONY : clean
clean :
    rm -f *.dat
    rm -f results.txt
```

새로 추가된 config 파일

```
# Count words script.
LANGUAGE=python
COUNT_SRC=countwords.py
COUNT_EXE=$(LANGUAGE) $(COUNT_SRC)

# Test Zipf's rule
ZIPF_SRC=testzipf.py
ZIPF_EXE=$(LANGUAGE) $(ZIPF_SRC)
```


Chapter 54

함수

현재 시점엔, Makefile은 다음과 같다:

```
include config.mk

# Generate summary table.
results.txt : $(ZIPF_SRC) isles.dat abyss.dat last.dat
$(ZIPF_EXE) *.dat > $@

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

%.dat : books/%.txt $(COUNT_SRC)
$(COUNT_EXE) $< $*.dat

.PHONY : clean
clean :
    rm -f *.dat
    rm -f results.txt
```

Make에는 많은 **함수(functions)**가 지원되어 더 복잡한 규칙을 작성할 수 있다. 한 사례가 wildcard다. wildcard는 특정 패턴과 매칭되는 파일 목록을 얻어와서, 변수에 저장할 수 있다. 예를 들어, 모든 텍스트 파일(.txt로 끝나는 파일) 목록을 불러와서 Makefile 시작부분에 변수에 불러온 값을 저장한다:

```
TXT_FILES=$(wildcard books/*.txt)
```

변수의 값을 보여주도록 .PHONY 대상과 규칙으로 추가한다:

```
.PHONY : variables
variables:
    @echo TXT_FILES: $(TXT_FILES)
```

(echo?)

Make는 동작을 실행할 때 동작을 화면에 출력한다. 동작 시작부에 @을 사용하면 Make로 하여금 동작을 출력하지 않게 한다. 그래서, echo 대신에 @echo를 사용함으로써, echo 실행 결과(변수값을 출력)를 볼 수 있지만, echo 명령어 자체는 볼 수 없게 된다.

Make를 실행하면:

```
$ make variables
```

다음 결과를 얻게 된다:

```
TXT_FILES: books/abyss.txt books/isles.txt books/last.txt books/sierra.txt
```

이제 *sierra.txt*도 포함된 것에 주목한다.

함수를 도입한 후, *analysis.tar.gz*에 명기된 대상을 빌드하는데 관여된 의존성을 도식화했는데, Makefile에서 구현된 사항이 다음 그림에 나와 있다:

patsubst ('pattern substitution', 패턴 치환)은 패턴, 대체 문자열, 명칭 목록을 순서대로 받는다; 목록에 나와 있는 패턴과 매칭되는 명칭은 대체 문자열로 치환된다. 다시, 결과는 변수에 저장한다. 예를 들어, 텍스트 파일 목록을 데이터 파일 목록(.dat로 끝나는 파일)으로 다시 작성하고 해당 결과를 변수에 저장한다:

```
DAT_FILES=$(patsubst books/%.txt, %.dat, $(TXT_FILES))
```

variables을 연장해서 DAT_FILES 값을 보여준다:

```
.PHONY : variables
variables:
    @echo TXT_FILES: $(TXT_FILES)
    @echo DAT_FILES: $(DAT_FILES)
```

Make를 실행하면,

```
$ make variables
```

다음 결과를 얻게 된다:

```
TXT_FILES: books/abyss.txt books/isles.txt books/last.txt books/sierra.txt
DAT_FILES: abyss.dat isles.dat last.dat sierra.dat
```

이제, *sierra.txt* 파일도 처리된다.

이것을 갖고, clean 과 dats 파일을 다시 작성한다:

```
.PHONY : clean
clean :
    rm -f $(DAT_FILES)
    rm -f analysis.tar.gz

.PHONY : dats
dats : $(DAT_FILES)
```

검사해 봅시다:

```
$ make clean
$ make dats
```

다음 결과를 얻게된다:

```
python countwords.py books/abyss.txt abyss.dat
python countwords.py books/isles.txt isles.dat
python countwords.py books/last.txt last.dat
python countwords.py books/sierra.txt sierra.dat
```

results.txt 도 다시 작성할 수 있다:

```
results.txt : $(ZIPF_SRC) $(DAT_FILES)
$(ZIPF_EXE) $(DAT_FILES) > $@
```

Make를 다시 실행하면:

```
$ make clean
$ make results.txt
```

다음 결과를 얻게 된다:

```
python countwords.py books/abyss.txt abyss.dat
python countwords.py books/isles.txt isles.dat
python countwords.py books/last.txt last.dat
python countwords.py books/sierra.txt sierra.dat
python testzipf.py last.dat isles.dat abyss.dat sierra.dat > results.txt
```

results.txt 파일 실행결과를 확인해보자.

```
$ cat results.txt
```

| Book | First | Second | Ratio |
|--------|-------|--------|-------|
| abyss | 4044 | 2807 | 1.44 |
| isles | 3822 | 2460 | 1.55 |
| last | 12244 | 5566 | 2.20 |
| sierra | 4242 | 2469 | 1.72 |

가장 빈도수가 높은 두 단어 출현비율이 지프의 법칙에서 예측했듯이 대략 2 근처다.

다음에 최종 Makefile이 나와 있다:

```
include config.mk

TXT_FILES=$(wildcard books/*.txt)
DAT_FILES=$(patsubst books/%.txt, %.dat, $(TXT_FILES))

# Generate summary table.
results.txt : $(ZIPF_SRC) $(DAT_FILES)
$(ZIPF_EXE) $(DAT_FILES) > $@
```

```

# Count words.
.PHONY : dats
dats : $(DAT_FILES)

%.dat : books/%.txt $(COUNT_SRC)
        $(COUNT_EXE) $< $@

.PHONY : clean
clean :
        rm -f $(DAT_FILES)
        rm -f results.txt

.PHONY : variables
variables:
        @echo TXT_FILES: $(TXT_FILES)
        @echo DAT_FILES: $(DAT_FILES)

config.mk 파일에 다음이 포함되어 있음을 기억하라:

# Count words script.
LANGUAGE=python
COUNT_SRC=countwords.py
COUNT_EXE=$(LANGUAGE) $(COUNT_SRC)

# Test Zipf's rule
ZIPF_SRC=testzipf.py
ZIPF_EXE=$(LANGUAGE) $(ZIPF_SRC)

```

Makefile 내부에 `results.txt` 타겟을 제작하는데 구현된 의존성을 다음 그래프가 보여주고 있다. 이를 통해 함수도 소개했다.

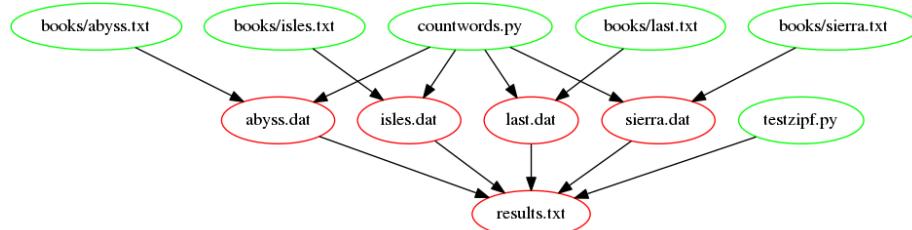


Figure 54.1: 함수를 도입한 후에 `results.txt` 의존성

54.1 함수 요약

- Make wildcard 함수를 사용해서 패턴과 매칭되는 파일 목록을 얻어온다.
- Make patsubst 함수를 사용해서 파일명을 다시 작성한다.

Chapter 55

문서화 Makefile

배수 내부에 실행되는 프로그램과 개발자가 작성한 프로그램 포함하여 내장 배수 명령어는 `--help` 플래그를 사용해서 프로그램과 명령어 사용법에 대한 정보를 전달하고 있다. 이런 연장선상에서 `help` 타겟을 Makefiles 내부에 작성해서 전달하는 것이 본인은 물론 다른 개발자에게도 도움이 된다. 이렇게 함으로써 주요 타겟에 대한 요약 정보와 동작방식에 대한 이해를 빠르게 높임으로써 꼭 필요하지 않는 경우 Makefile을 볼 필요는 없게 된다. `help` 타겟을 실행하게 되면 다음이 출력된다.

```
$ make help

results.txt : Generate Zipf summary table.
dats         : Count words in text files.
clean        : Remove auto-generated files.
```

그렇다면 구현하는 방법은 어떻게 될까? 다음과 같이 규칙을 작성하면 된다:

```
.PHONY : help
help :
    @echo "results.txt : Generate Zipf summary table."
    @echo "dats       : Count words in text files."
    @echo "clean      : Remove auto-generated files."
```

하지만 규칙을 매번 추가하거나 삭제, 혹은 규칙에 대한 설명을 변경할 때마다 이런 규칙도 수정해서 최신화시켜놔야 한다. 규칙 자체에 규칙에 대한 기술도 함께 유지하고 자동으로 추출하게 된다면 정말 멋진 일이 될 것이다.

배수 쉘이 여기서 우리를 구원해줄 수 있다. `sed` 명령어를 사용한다. `sed`는 stream editor 줄임말이다. `sed`는 텍스트를 읽어 필터링하고 필터된 결과를 텍스트로 저장한다.

따라서, 규칙에 대한 주석도 함께 작성해서 나중에 `sed`가 탐지할 수 있도록 표식을 남겨둔다. Make는 `#`을 주석으로 사용하기 때문에 `sed`가 탐지할 수 있도록 `##` 표식을

대신 사용한다. 예를 들어,

```
## results.txt : Generate Zipf summary table.
results.txt : $(ZIPF_SRC) $(DAT_FILES)
    $(ZIPF_EXE) $(DAT_FILES) > $@

## dats      : Count words in text files.
.PHONY : dats
dats : $(DAT_FILES)

%.dat : books/%.txt $(COUNT_SRC)
    $(COUNT_EXE) $< $@

## clean      : Remove auto-generated files.
.PHONY : clean
clean :
    rm -f $(DAT_FILES)
    rm -f results.txt

## variables   : Print variables.
.PHONY : variables
variables:
    @echo TXT_FILES: $(TXT_FILES)
    @echo DAT_FILES: $(DAT_FILES)
```

표식을 사용해서 자동으로 sed가 필터하는 주석과 다른 규칙이 기술하는 주석을 구별한다.

Makefile 파일에 sed를 적용한 help 타겟을 작성한다:

```
.PHONY : help
help : Makefile
    @sed -n 's/^##//p' $<
```

상기 규칙은 Makefile 자체에 의존한다. Makefile에 적힌 첫번째 의존성으로 sed를 실행하고 sed에게 명령하여 ##으로 시작되는 모든 행을 추출해서 sed가 출력하게 지정한다.

다음을 실행하게 되면

```
$ make help
```

다음을 얻게 된다.

```
results.txt : Generate Zipf summary table.
dats      : Count words in text files.
clean      : Remove auto-generated files.
variables   : Print variables.
```

타겟이나 규칙을 추가, 변경, 제거하게 되면 해당 규칙에 인접한 주석을 추가, 간선, 제거하는 것만 기억하면 된다. 해당 작업에 대한 주석으로 ## 관례만 존중하면 help

규칙이 자동으로 탐지하여 출력 도움말을 만들어낸다.

현재까지 다룬 Makefile

```
include config.mk

TXT_FILES=$(wildcard books/*.txt)
DAT_FILES=$(patsubst books/%.txt, %.dat, $(TXT_FILES))

## results.txt : Generate Zipf summary table.
results.txt : $(ZIPF_SRC) $(DAT_FILES)
    $(ZIPF_EXE) $(DAT_FILES) > $@

## dats      : Count words in text files.
.PHONY : dats
dats : $(DAT_FILES)

%.dat : books/%.txt $(COUNT_SRC)
    $(COUNT_EXE) $< $@

## clean      : Remove auto-generated files.
.PHONY : clean
clean :
    rm -f $(DAT_FILES)
    rm -f results.txt

## variables   : Print variables.
.PHONY : variables
variables:
    @echo TXT_FILES: $(TXT_FILES)
    @echo DAT_FILES: $(DAT_FILES)

.PHONY : help
help : Makefile
    @sed -n 's/^##//p' $<
```

config.mk 파일

```
# Count words script.
LANGUAGE=python
COUNT_SRC=countwords.py
COUNT_EXE=$(LANGUAGE) $(COUNT_SRC)

# Test Zipf's rule
ZIPF_SRC=testzipf.py
ZIPF_EXE=$(LANGUAGE) $(ZIPF_SRC)
```


Chapter 56

결론

Make 같은 자동 빌드 도구는 여러가지 방식으로 도움을 줄 수 있다. 반복되는 명령어를 자동화해서, 명령어를 수작업으로 실행할 때, 범하게 되는 오류위험을 줄여주고, 시간을 절약해서 도움이 된다.

생성할 파일이 어떤 방식으로 변경될 때, 자동적으로 생성되는 산출물(데이터 파일 혹은 그래프)만 다시 생성하게 함으로써 시간을 절약해준다.

대상, 의존성, 동작 개념을 통해, 코드, 스크립트, 도구, 구성(configuration), 원데이터, 파생된 데이터, 그래프, 논문 사이 의존성을 기록하는 문서화 형태로 역할을 수행한다.

56.1 PNG 생성하기

새로운 규칙을 추가하고, 기존 규칙을 갱신하고, 매크로를 새로 추가하여 다음 작업을 수행한다:

- plotcounts.py 프로그램을 사용해서 .dat 파일에서 .png 파일을 생성한다.
- 자동생성된 모든 파일(.dat, .jpg, results.txt)을 제거한다.

대다수 Makefile은 첫번째 all 타겟으로 디폴트 기본 포니 타겟을 정의한다. all 타겟을 이번 경우 .png 파일과 results.txt 파일을 빌드하는 작업을 수행한다. Makefile은 관례를 따라 all 타겟을 지원한다. 따라서, make results.txt 실행하는 대신 make all 혹은 더 줄여 make 명령어를 실행해야 된다. 기본디폴트 설정으로 Makefile에서 첫번째 타겟을 찾아 실행하게 만든다. 이번 경우 all 타겟이다.

Makefile

```
include config.mk  
  
TXT_FILES=$(wildcard books/*.txt)
```

```

DAT_FILES=$(patsubst books/%.txt, %.dat, $(TXT_FILES))
PNG_FILES=$(patsubst books/%.txt, %.png, $(TXT_FILES))

## all      : Generate Zipf summary table and plots of word counts.
.PHONY : all
all : results.txt $(PNG_FILES)

## results.txt : Generate Zipf summary table.
results.txt : $(ZIPF_SRC) $(DAT_FILES)
$(ZIPF_EXE) $(DAT_FILES) > $@

## dats      : Count words in text files.
.PHONY : dats
dats : $(DAT_FILES)

%.dat : books/%.txt $(COUNT_SRC)
$(COUNT_EXE) $< $@

## pngs      : Plot word counts.
.PHONY : pngs
pngs : $(PNG_FILES)

%.png : %.dat $(PLOT_SRC)
$(PLOT_EXE) $< $@

## clean     : Remove auto-generated files.
.PHONY : clean
clean :
    rm -f $(DAT_FILES)
    rm -f $(PNG_FILES)
    rm -f results.txt

## variables  : Print variables.
.PHONY : variables
variables:
    @echo TXT_FILES: $(TXT_FILES)
    @echo DAT_FILES: $(DAT_FILES)
    @echo PNG_FILES: $(PNG_FILES)

.PHONY : help
help : Makefile
    @sed -n 's/^##//p' $<

config.mk

# Count words script.
LANGUAGE=python

```

```

COUNT_SRC=countwords.py
COUNT_EXE=$(LANGUAGE) $(COUNT_SRC)

# Plot word counts script.
PLOT_SRC=plotcounts.py
PLOT_EXE=$(LANGUAGE) $(PLOT_SRC)

# Test Zipf's rule
ZIPF_SRC=testzipf.py
ZIPF_EXE=$(LANGUAGE) $(ZIPF_SRC)

```

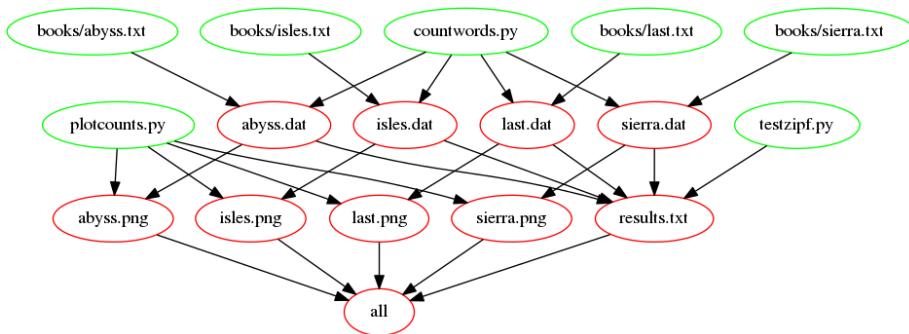


Figure 56.1: 이미지가 추가된 후 all 타겟 의존성

56.2 아카이브 생성

데이터, 코드, 결과가 모두 포함된 프로젝트를 담은 아카이브를 생성하면 유용하다. 많은 파일을 담아 패키지로 묶어 하나의 파일로 제작한 아카이브 파일(Archive file)은 다른 협업하시는 분과 쉽게 공유할 수 있고 다운로드 받기도 편하다. `Makefile` 자체 내부에 아카이브 생성 단계를 추가함으로써 프로젝트가 변경되더라도 아카이브 파일을 쉽게 갱신하여 최신화시킬 수 있다.

`Makefile` 파일을 편집해서 프로젝트 아카이브 파일을 생성한다. 규칙을 새로 추가하고, 변경해서, 새로운 변수도 추가해서 다음 작업을 수행하게 만든다.

- 프로젝트 디렉토리에 `zipf_analysis` 폴더를 새로 생성한다.
- 코드, 데이터, 그래프, 지프의 법칙 요약 표, `Makefile`, `config.mk` 모든 파일을 해당 디렉토리로 복사한다. `cp -r` 명령어를 사용해서 모든 파일과 디렉토리를 새로 생성한 `zipf_analysis` 디렉토리에 복사한다.

```
$ cp -r [files and directories to copy] zipf_analysis/
• 힌트: books 디렉토리에 변수를 새로 생성한다. zipf_analysis 디렉토리에 복사한다.
• zipf_analysis.tar.gz 아카이브를 생성한다. tar 배수 명령어를 다음과 같이 사용한다.
```

```
$ tar -czf zipf_analysis.tar.gz zipf_analysis
```

- all 타겟을 업데이트해서 zipf_analysis.tar.gz 파일을 생성한다.
- make clean이 호출되면 zipf_analysis.tar.gz 파일을 제거한다.
- make variables를 호출하게 되면 정의한 추가 변수 값을 출력한다.

Makefile 해답

```
include config.mk
```

```
TXT_DIR=books
TXT_FILES=$(wildcard $(TXT_DIR)/*.txt)
DAT_FILES=$(patsubst $(TXT_DIR)/%.txt, %.dat, $(TXT_FILES))
PNG_FILES=$(patsubst $(TXT_DIR)/%.txt, %.png, $(TXT_FILES))
RESULTS_FILE=results.txt
ZIPF_DIR=zipf_analysis
ZIPF_ARCHIVE=$(ZIPF_DIR).tar.gz

## all      : Generate archive of code, data, plots, summary table, Makefile, a
.PHONY : all
all : $(ZIPF_ARCHIVE)

$(ZIPF_ARCHIVE) : $(ZIPF_DIR)
    tar -czf $@ $<

$(ZIPF_DIR): Makefile config.mk $(RESULTS_FILE) \
    $(DAT_FILES) $(PNG_FILES) $(TXT_DIR) \
    $(COUNT_SRC) $(PLOT_SRC) $(ZIPF_SRC)
    mkdir -p $@
    cp -r $^ $@
    touch $@

## results.txt : Generate Zipf summary table.
$(RESULTS_FILE) : $(ZIPF_SRC) $(DAT_FILES)
    $(ZIPF_EXE) $(DAT_FILES) > $@

## dats      : Count words in text files.
.PHONY : dats
dats : $(DAT_FILES)

%.dat : $(TXT_DIR)/%.txt $(COUNT_SRC)
    $(COUNT_EXE) $< $@

## pngs      : Plot word counts.
.PHONY : pngs
pngs : $(PNG_FILES)
```

```

%.png : %.dat $(PLOT_SRC)
$(PLOT_EXE) $< $@

## clean      : Remove auto-generated files.
.PHONY : clean
clean :
    rm -f $(DAT_FILES)
    rm -f $(PNG_FILES)
    rm -f $(RESULTS_FILE)
    rm -rf $(ZIPF_DIR)
    rm -f $(ZIPF_ARCHIVE)

## variables   : Print variables.
.PHONY : variables
variables:
    @echo TXT_DIR: $(TXT_DIR)
    @echo TXT_FILES: $(TXT_FILES)
    @echo DAT_FILES: $(DAT_FILES)
    @echo PNG_FILES: $(PNG_FILES)
    @echo ZIPF_DIR: $(ZIPF_DIR)
    @echo ZIPF_ARCHIVE: $(ZIPF_ARCHIVE)

.PHONY : help
help : Makefile
@sed -n 's/^##//p' $<

config.mk 해답

# Count words script.
LANGUAGE=python
COUNT_SRC=countwords.py
COUNT_EXE=$(LANGUAGE) $(COUNT_SRC)

# Plot word counts script.
PLOT_SRC=plotcounts.py
PLOT_EXE=$(LANGUAGE) $(PLOT_SRC)

# Test Zipf's rule.
ZIPF_SRC=testzipf.py
ZIPF_EXE=$(LANGUAGE) $(ZIPF_SRC)

```

56.3 Makefile 아카이빙

아카이브 디렉토리에 대한 Makefile 규칙에 Makefile을 코드, 데이터, 그래프, 지프의 요약표에 추가될까?

해답

`countwords.py`, `plotcounts.py`, `testzipf.py` 코드 파일은 작업흐름의 개별 부분을 담고 있다. 코드 파일을 통해서 `.txt` 파일에서 `.dat` 파일을 생성해낸다. 그리고, `.dat` 파일에서 `.png` 파일과 `results.txt` 파일을 만들어낸다. 하지만 `Makefile`에는 코드파일, 원데이터, 파생데이터, 그래프를 비롯한 전체 작업흐름이 문서로 담겨있다. `config.mk` 파일은 `Makefile`에 대한 환경설정 정보도 담겨있어 아카이브로 저장되어야 한다.

56.4 아카이브 디렉토리 `touch`

코드, 데이터, 그래프, 요약표를 아카이브 디렉토리로 이동시킨 후에 아카이브 디렉토리에 `Makefile` 규칙을 왜 `touch`해야 할까?

해답

파일이 아카이브 디렉토리로 복사될 때 디렉토리 시간도장이 자동으로 갱신되지 않는다. 만약 코드, 데이터, 그래프, 요약표를 아카이브 디렉토리로 최신화되어 복사되면, `touch` 명령어로 아카이브 디렉토리 시간도장이 최신으로 갱신되어야 한다. 그래야만 규칙이 `zipf_analysis.tar.gz` 파일을 다시 실행시켜 제작시키는 것을 알게 된다. `touch`가 없다면 `zipf_analysis.tar.gz` 파일은 규칙이 처음 실행될 때 생성되고 아카이브 디렉토리 콘텐츠가 변경되더라도 후속 변경작업을 일어나지 않게 된다.

- Fox, John. 2005. “The R Commander: A Basic Statistics Graphical User Interface to R.” *Journal of Statistical Software* 14 (9): 1-42. <https://doi.org/10.18637/jss.v014.i09>.
- . 2016. *Using the r Commander: A Point-and-Click Interface for r*. Chapman: Hall/CRC.
- Fox, John, and Milan Bouchet-Valat. 2021. Rcmdr: R Commander. <https://socialsciences.mcmaster.ca/jfox/Misc/Rcmdr/>.
- Wilson, Greg. 2022. “Software Carpentry Web Site.” <http://software-carpentry.org>.