

Software Carpentry Version 5.2

차례

[Fork me on GitHub](#)



번역: 이광춘

저작 및 편집: Greg Wilson

- 소개
- 소프트웨어 카펜트리 팀
- 유닉스 쉘(Unix Shell)
- Git을 사용한 버전 관리
- 파일 프로그래밍
- 데이터베이스와 SQL 사용하기
- 유용한 것 몇가지
- 강사 교육자침
- 참고 정보
- 추천 도서
- 용어사전
- 규칙
- 라이센스

들어가며(introduction)

여기 꿈이 있다.

컴퓨터가 여구를 혁명적으로 바꾸었고, 혁명은 지금 시작단계다. 매일 전세계 과학자와 공학자들이 너무 크거나, 너무 작거나, 너무 빠르거나, 너무 비싸거나, 너무 위험하거나, 혹은 너무 어려워서 어느 방식이든지 연구하기 어려운 것들을 연구하기 위해서 컴퓨터를 사용한다.

지금 현실이 여기 있다.

매일 전세계 과학자들과 공학자들이 컴퓨터와 싸워하면서 시간을 낭비하고 있다. 얼마 걸리지 않을 작업이 몇 시간, 몇일이 걸리지만, 많은 것들은 결코 작동하지 않는다. 그리고 심지어 동작을 해도, 결과가 얼마나 신뢰성이 있는지에 대해서 많은 과학자들이 확신을 하지 못한다.

연구자들이 느끼는 대부분의 고통은 체계적으로 소프트웨어를 어떻게 개발하는지, 만약 프로그램이 잘 동작한다면 어떻게 작업결과를 전자우편을 통해서 전달하는 것을 제외하고 동료와 공유하는지, 혹은 지금까지 연구한 것을 어떻게 기록하는지 잘 모른다는 사실에 기인하다. 이러한 유감스러운 문제가 지속되는 이유는 다음 4 가지에 기인한다.

- 공간과 시간이 없다. (*No room, no time*) 모든 사람의 교육과정은 이미 꽉 차 있다. - 단순하게 다른 교육과정을 빼지 않고 컴퓨팅에 대해서 추가할 공간이 없다.
- 표준이 없다. (*No standards*) 검수자와 연구기금을 관리하는 기관에서 소프트웨어를 정확하게 작성한 것인지 확인하지 않고, 프로그램을 작성하는데 얼마나 걸리는지 혹은 임기까지 계산하지 않는다. 그래서 과학자들이 더 잘 할 동기가 없다.
- 장님이 장님을 인도한다. (*The blind leading the blind*) 선임 연구자는 본인 스스로 어떻게 하는지 모르는 것을 다음 세대 연구자에게 어떻게 하라고 가르칠 수 없다.
- 대형 메인프레임 컴퓨터 숭배 (*The cult of big iron*) 거의 모든 사람들이 사용하는 기본적인 기술보다는 정치가와 대학 총장이 첫 날 자랑하는 것에 관심과 자금이 대부분이 들어간다.

소프트웨어 공방(Software Carpentry)의 목적은 과학자와 공학자에게 어떻게 적은 시간으로 덜 수고스럽게 많은 작업을 할 수 있는지 보여준다. '10년 봄 이래로 100여번의 이를 워크샵을 통해서 4천명 이상의 학습자가 본 학습과정을 거쳐갔다. 다음에 어떻게 도움이 될 수 있는지 사례가 있다.

- 잘못된 파일을 덮어쓴 적이 있다면, 버전 관리(version control)를 어떻게 사용하는지 보여줄 것이다.
- 동일한 명령어를 반복해서 타이핑하는데 여러 시간을 소비한적이 있다면, 간단한 스크립트로 어떻게 작업을 자동화하는지 보여줄 것이다.
- 지난 주에 작성한 정말 동작하는 프로그램을 이해하는데 오후를 보낸적이 있다면, 코드를 잘게 쪼개서 읽고, 디버그하고, 기능을 개선하기 쉽게 모듈화 하는 것을 보여줄 것이다.

우리는 누구인가

소프트웨어 공방(Software Carpentry)은 공개 소프트웨어 프로젝트다. 강사는 자발적으로 참여한 자원봉사자이고, 모든 학습자료는 [크리에이티브 커먼즈 라이센스\(Creative Commons - Attribution License\)](#)로 자유로이 이용가능하다. 따라서, 원출처를 밝히기만 하면, 학습자료를 재사용하고, 다른 학습자료와 섞어서도 사용할 수 있다.

다른 공개된 자발적인 프로젝트와 마찬가지로, 소프트웨어 공방(Software Carpentry) 프로젝트는 확대되기 위해서 여러분의 도움이 필요하다. 만약 버그가 있다면, [GitHub 저장소](#)에 보고해주세요. 만약 워크샵을 개최하고자 한다면, [전자우편으로 연락주세요](#). 만약 교육을 하고자 한다면, [강사 훈련 과정](#)이 있습니다. 만약 학습과정이나 연습문제를 집필하고자 한다면, [전자우편으로 연락주세요](#).

좀 더 많은 정보가 필요하다면, [소프트웨어 공방\(Software Carpentry\) 웹사이트](#) 혹은 [모범사례 논문](#)을 읽거나, [가장 최근의 블로그 포스팅](#)을 참고하세요.

감사의 글

소프트웨어 공방(Software Carpentry)은 다음의 관대한 지원으로 가능하게 되었다.

- [Continuum Analytics](#)
- [Indiana University](#)

- Lawrence Berkeley National Laboratory
- Los Alamos National Laboratory
- MathWorks
- Michigan State University
- Microsoft
- MITACS
- The Mozilla Foundation
- The Python Software Foundation
- Queen Mary University of London
- Scimatic Software
- SciNet
- SHARCNET
- The Alfred P. Sloan Foundation
- The Space Telescope Science Institute
- The UK Meteorological Office
- The University of Alberta
- The University Consortium for Atmospheric Research
- The University of Toronto

이 과정의 첫번째 버전을 만들고 지도하는데 도움을 주신 Brent Gorda에게 특별한 감사의 말씀을 전한다.

이책은 ENIAC컴퓨터의 원조 프로그래머들에게 헌정한다.

Betty Jennings, Betty Snyder, Fran Bilas, Kay McNulty, Marlyn Wescoff, Ruth Lichterman

소프트웨어 카펜트리 팀(Our Team)

수년동안 소프트웨어 카펜트리(Software Carpentry)에 기여해주신 모든 분들께 감사의 말씀을 전합니다.

Nasser Mohieddin Abukhdeir	Laurent Gatto	Lex Nederbragt
Joshua Adelman	Molly Gibson	Victor Ng
Aron Ahmadia	Matt Gidden	Charlene Nielsen
Joshua Ainsley	Ivan Gonzalez	Danielle Nielsen
Carlos Anderson	Alexandre Gramfort	Randy Olson
Mario Antonioletti	Chris Gray	Geoff Oxberry
Feth Arezki	Julia Gustavsen	Aleksandra Pawlik
Dhavide Aruliah	Tommy Guy	Jason Pell
Pauline Barmby	Steven Haddock	Fernando Perez
Diego Barneche	Michael Hansen	Hans Petter Langtangen
Philipp Bayer	Ted Hart	Caitlyn Pickens
Trevor Bekolay	Trent Hauck	Bill Punch
Miguel Bernabeu	Elliott Hauser	Karthik Ram
Matt Billard	James Hetherington	David Rio Deiros
Sergi Blanco Cuaresma	Konrad Hinsen	Ariel Rokem
John Blischak	Steve Holden	Jorden Schossau
Darren Boss	Mark Holder	Anthony Scopatz
Azalee Bostroem	Preston Holmes	Michael Selik
Erik Bray	Alison Hoyt	Chang She
Eli Bressert	Katy Huff	Jeffrey Shelton
Matthew Brett	Damien Irving	Raniere Silva
Amy Brown	Paul Ivanov	Gavin Simpson
C. Titus Brown	Mike Jackson	Andrew Smith
Jennifer Bryan	David Jones	Joshua Ryan Smith
Orion Buske	Nick Jones	Jon Speicher
Rosangela Canino-Koning	Jessica Kerr	Adam Stark
Chris Cannam	David Ketcheson	Becky Stewart
Scott Chamberlain	W. Trevor King	Shoaib Sufi
Cliburn Chan	Justin Kitzes	Sarah Supp
Amanda Charbonneau	Christina Koch	Leszek Tarkowski
Shreyas Cholia	Steven Koenig	Tracy Teal
Adina Chuang Howe	Bernhard Konrad	Andy Terrel
Neil Chue Hong	David Koop	Matt Terry
Rodgers Cliff	Daniel Krasner	Samuel Thomson
Christophe Combelles	Karin Lagesen	Joan Touzet
Mike Conley	Jared Lander	Laura Tremblay-Boyer
Christophe Cossou	Ian Langmore ⁵	Will Trimble
Stefano Cozzini	Chris Lasher	Jacob Vanderplas
Karen Cranston	Doug Latornell	Gael Varoquaux
Stephen Crouch	Michelle Levesque	Nelle Varoquaux
Davor Cubranic	Phil Lies	Alex Viana
Emily Davenport	Nicolas Limare	Jens von der Linden

유닉스 쉘(Unix Shell)

유닉스 쉘(Unix Shell)은 대부분의 컴퓨터 사용자가 살아온 것보다 오래 동안 존재했다. 오래동안 생존한 이유는 사용자로 하여금 단지 키보드 몇번 쳐서 복잡한 작업을 할 수 있게 하는 강력한 도구이기 때문이다. 좀더 중요하게는 기존의 프로그램을 새로운 방식으로 조합해서 반복적인 작업을 자동화해서 동일한 작업을 반복적으로 하지 않게 만든다.

쉘(Shell) 소개

목표

- 쉘(shell)이 어떻게 키보드, 화면, 운영체제, 사용자 프로그램에 연관되는지 설명.
- 명령줄(CLI, command-line interface) 인터페이스가 화면 사용자 인터페이스 (GUI, graphic user interface) 대신에 언제, 왜 사용되어야 하는지 설명.

해양 생물학자 넬 니모(Nell Nemo) 박사가 방금전 6개월 [북태평양 소용돌이꼴조](#)사를 마치고 방금 귀환했다. [태평양 거대 쓰레기 지대](#)에서 젤리같은 해양생물을 표본주출했다. 총 합쳐서 거의 300개의 시료가 있고 다음 작업이 필요하다.

1. 소로 다른 300개 단백질의 상대적인 함유량을 측정하는 분석기계로 시료를 시험한다. 한 시료에 대한 기계 출력결과는 각 단백질에 대해서 한 줄의 파일형식으로 표현된다.
2. goostat으로 명명된 그녀의 관리자가 작성한 프로그램을 사용하여 각 단백질에 대한 통계량을 계산한다.
3. goodiff로 명명된 다른 대학원 학생중의 한명이 작성한 각 단백질의 통계량과 다른 단백질에 상응하는 통계량을 비교한다.
4. 작성. 그녀의 지도교수는 이번달말까지 이일을 정말 좋아해서 논문이 다음번 *Aquatic Goo Letters* 저널의 특별판에 게재되기를 희망한다.

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

나쁜 소식은 `goostat`, `goodiff`을 수작업으로 실행한다면, 파일이름 입력하고 “OK” 버튼을 45,150번 눌려야 된다는 사실이다 (`goostat` 300회 더하기 `goodiff` $300 \times 299/2$). 매번 30초씩 가정하면 2주 이상 소요될 것이다. 논문 마감일을 놓칠 수도 있지만, 이 모든 명령어를 올바르게 입력할 가능성은 거의 0에 가깝다.

다음 몇 수업은 대신에 그녀가 무엇을 해야하는지 탐색한다. 좀더 구체적으로 작업처리 중간에 반복되는 작업을 자동화하는 셸 명령어(command shell)를 어떻게 사용하는지 설명해서 논문을 쓰는 동안에 컴퓨터는 하루에 24시간 작업한다. 덤으로 중간 처리작업을 완성하면, 좀더 많은 데이터를 얻을 때마다 다시 재사용할 수 있다.

쉘이 무엇이고, 왜 사용할까요

상위 수준에서 컴퓨터는 네가지 일을 수행한다.

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

많은 방식으로 뇌-컴퓨터 연결, 음성 인터페이스를 포함한 마지막 일을 수행한다. 하지만, 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 다시 연결하는 것이다. 하지만, 중간에 1950년에서 1980년대 사이에 대부분의 사람들은 라인 프린터(line printer)를 사용했다. 이런 디바이스는 표준 키보드에 문자, 숫자, 특수부호의 입력과 출력만 허용해서 프로그래밍 언어와 인터페이스는 이러한 제약사항에서 설계됐다.

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 loop이다. 사용자가 명령어를 타이핑하고 엔터(enter)/반환(return)키를 입력하면, 컴퓨터가 일고, 실행하고, 결과를 출력한다. 그러면 사용자는 다른 명령을 타이핑하는 것을 로그 오프할 때까지 계속한다.

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.

쉘은 다른 것과 마찬가지로 프로그램이다. 조금 특별한 것은 자신이 연산을 수행하기 보다 다른 프로그램을 실행하는 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, 윈도우용 유닉스스러운 도구로 제공되는 패키지에도 적용된다.

Bash나 다른 쉘을 사용하는 것은 마우스 보다 프로그래밍 같은 느낌이 난다. 명령어는 간략해서 종종 2~3자리 수이고, 명령어는 종종 암호스럽고, 출력은 그래프같은 시각적인 것보다 텍스트 라인이다. 다른 한편으로는 쉘을 사용하여 좀 더 강력한 방식으로 존재하는 도구를 단지 몇 키보드 입력으로 조합해서 대용량의 데이터를 자동적으로 다룰 수 있는 파이프라인을 구축할 수 있다. 추가로 명령 라인은 종종 멀리 떨어진 컴퓨터와 상호작용하는 가장 쉬운 방법이다. 클러스트 컴퓨팅과 클라우드 컴퓨팅이 과학 데이터 클러칭(scientific data cruching)이 점점 대중화됨에 따라 원격 컴퓨터를 구동하는 것이 필수적인 기술이 되어가고 있다.

주요점

- 쉘은 프로그램으로 명령어를 읽고 다른 프로그램을 수행하는 것이 주요 목적이다.
- 쉘의 주요 장점은 키보드 입력 대비 높은 수행 비율(action-to-keystroke)로 반복적인 작업을 자동화하고 원격으로 네트워크에 연결된 컴퓨터에 접근할 수 있게 한다.
- 쉘의 주된 단점은 주로 텍스트로 되어 있고, 명령어와 수행이 수수께끼 같을 수 있다.

파일과 디렉토리

목표

- 파일과 디렉토리의 차이점과 다른점을 설명한다.
- 절대경로를 상대경로로 변환하고 반대로 상대경로를 절대경로로 변환한다.
- 특정 파일과 디렉토리를 확인할 수 있는 절대경로와 상대경로를 구성한다.
- 쉘의 읽기-실행-출력(read-run-print) 주기를 각 단계별로 설명한다.
- 명령 라인 호출에 실제 명령어, 플래그, 파일명을 확인한다.
- 템 완성기능을 시연하고 장점을 설명한다.

파일과 디렉토리를 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 조직한다.

파일과 디렉토리를 생성, 검사, 새이름, 삭제하는데 몇가지 명령어가 자주 사용된다. 명령어를 살펴보기 위해서 쉘 윈도우를 엽니다.

\$

달러 기호 (\$)는 프롬프트(prompt)로 쉘이 입력을 기다리는 것을 보여준다. 여러분의 쉘이 좀더 정교한 다른 것을 보여줄수도 있다.

`whoami` 명령어를 타이핑하고, 명령을 쉘에 보내기 위해서 엔터키(Enter Key, 종종 키보드에 따라 Return으로 표기도 됨)를 누릅니다. 명령어의 출력은 현재 사용자의 ID가 됩니다. 즉, 쉘이 생각하는 사용자가 누구인지를 보여줍니다.

\$ `whoami`

`nelle`

좀더 구체적으로, `whoami`를 타이핑할 때, 쉘은

1. `whoami` 프로그램을 찾고,

2. 프로그램을 실행하고,
3. 프로그램 실행 결과를 출력하고,
4. 새로운 프롬프트를 화면에 출력해서 더 많은 명령어를 받을 준비가 되어 있음을 알려줍니다.

다음으로 `pwd` 명령어를 실행하여 지금 어디에 있는지 알아봅시다. `pwd`는 “print working directory”의 첫 글짜를 땄습니다. 언제든지 현재 작업 디렉토리(current working directory)는 현재 디폴트 디렉토리가 된다. 즉, 명시적으로 다른 곳으로 지정하지 않았다면, 사용자가 명령어를 실행하는 디렉토리가 컴퓨터가 가정하는 디렉토리가 된다. 다음에서, 컴퓨터의 응답은 `/users/nelle`으로 넬(Nelle)의 홈 디렉토리(home directory)다.

```
$ pwd
```

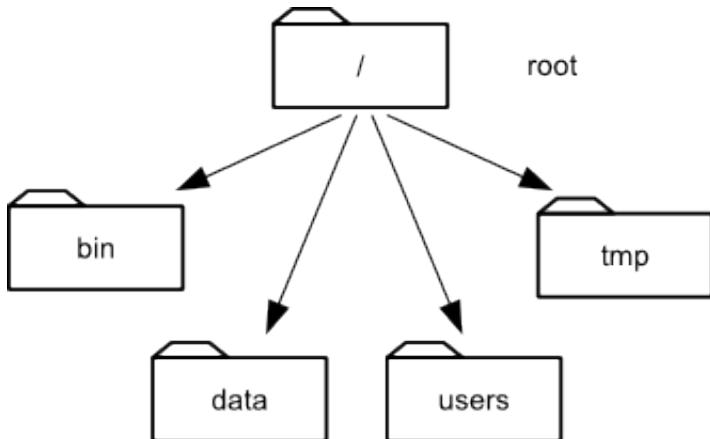
```
/users/nelle
```

알파벳 수프 (Alphabet Soup) 사용자가 누구인지를 파악하는데는 `whoami`이 사용되면, 사용자가 어디 있는지를 파악하는 명령어는 `wherami`이 되어야 한다. 왜 `pwd`가 대신에 사용될까? 일반적인 대답은 다음과 같다. 1970년대 초에 UNIX가 처음 개발되었을 때, 모든 키입력은 카운트되었고, 그 시절 장비는 느리고, 텔레타이프의 백스페이스는 너무나도 고생스러워서, 타이핑 오류 숫자를 줄이기 위해서 키입력의 숫자를 줄이는 것이 사용자 편의성(usability)의 진정한 승자가 되었다. 현실은 명령어가 종합계획 없이 전문어와 은어에 몰입된 사람들에 의해 서 유닉스에 하나씩 하나씩 추가되었다. 결과는 일관성이 없지만, 지금 우리는 뗄래야 뗄 수 없는 상태가 되었다.

“홈 디렉토리(home directory)”를 이해하기 위해서, 파일 시스템이 전체로 어떻게 구성되었는지 살펴보자. 최상단에 다른 모든 것을 담고 있는 루트 디렉토리(root directory)가 있다. 슬래쉬 / 문자로 나타내고, `/users/nelle`에서 맨 앞에 슬래쉬 이기도 하다.

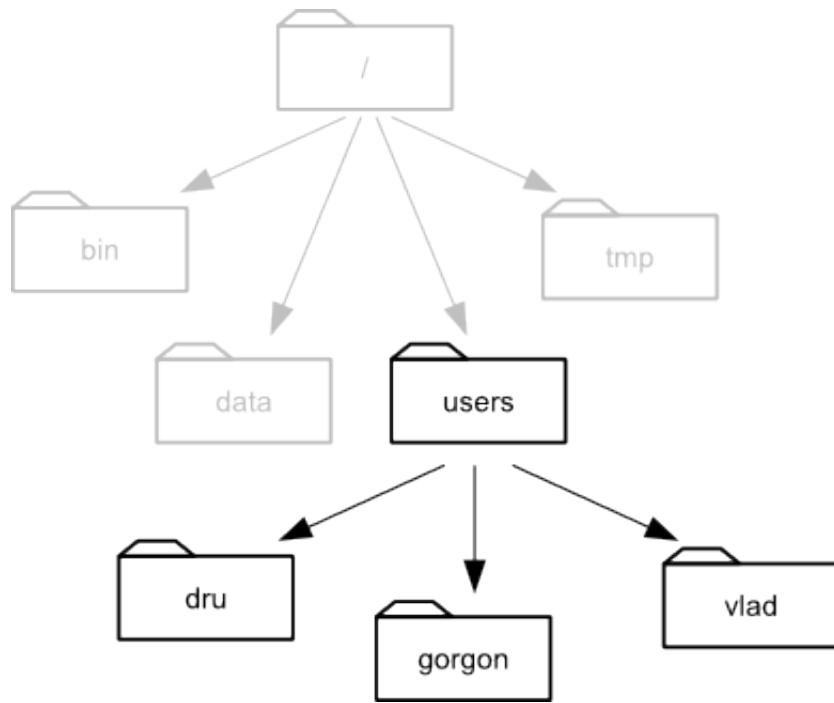
홈 디렉토리 안쪽에 몇 가지 다른 디렉토리가 있다. `bin` (몇몇 내장 프로그램이 저장된 디렉토리), `data` (여러가지 데이터 파일이 저장된 디렉토리), `users` (사용자의

개인 디렉토리가 위한 디렉토리), tmp (장기간 저장될 필요가 없는 임시 파일을 위한 디렉토리), 등등.



현재 작업 디렉토리 /users/nelle는 /users 내부에 저장되어 있는데 /users가 이름의 처음 부분이기 때문에 알 수 있다. 마찬가지로 /users는 루트 디렉토리 내부에 저장되어 있는데 이름이 /으로 시작되기 때문이다.

/users 하단에, 컴퓨터 계정의 각 사용자별 디렉토리를 볼 수 있다. 미이라 (Mummy) 파일은 /users/imhotep 디렉토리에 저장되어 있고, 늑대인가 (Wolfman)의 파일은 /users/larry 디렉토리에 저장되어 있고 /users/nelle 디렉토리에 nelle의 정보가 저장되어 있는데 이것이 왜 nelle이 디렉토리 이름의 마지막 부분인 이유다.



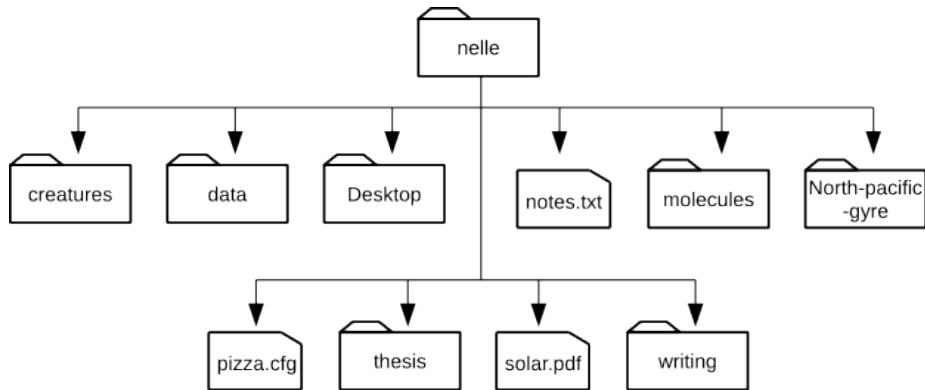
문자 / 에 두가지 의미가 있음을 주목한다. 파일 혹은 디렉토리 이름의 처음에 나타날 때는 루트 디렉토리를 의미한다. 이름 중간에 나타날 때는 단지 구분자임을 나타낸다.

Nelle의 홈 디렉토리에 무엇이 있는지 `ls` 명령어를 실행해서 살펴보자. `ls`는 “목록 보기(listing)”를 나타낸다.

```
$ ls
```

```

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



`ls`는 알파벳 순서로 깔끔하게 열로 정렬하여 현재 디렉토리의 파일과 디렉토리 이름을 출력한다. 플래그(flag) `-F`를 추가하여 출력을 좀 더 포괄적으로 생성할 수 있다. `ls`으로 하여금 디렉토리 이름 뒤에 `/`을 추가하게 한다.

```
$ ls -F
```

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

`/users/nelle` 디렉토리는 7개의 서브 디렉토리(sub-directories)를 담고 있다. 뒤에 슬래쉬를 갖지 않은 이름, 예를 들어 `notes.txt`, `pizza.cfg`, `solar.pdf`은 단순한 파일이다. `ls`과 `-F` 사이에 공백이 있음을 주목한다. 공백이 없으면 쉘은 존재하지 않는 `ls-F` 명령어를 실행한다고 생각한다.

이름에는 무엇이 있나요? Nelle의 파일 이름이 “무엇.무엇”으로 된 것을 알아챘을지 모르겠다. 이것은 단지 관례다. 파일 이름을 `mythesis` 혹은 원하는 무엇이든지 지을 수 있다. 하지만, 대부분의 사람들은 두 부분으로 구분된 이름을 사용하여 사람이나 프로그램이 다른 종류의 파일임을 구분하도록 돋는다. 이름의 두 번째 부분은 파일 확장자(filename extension)라고 불리며, 파일이 무슨 종류의 데이터를 담고 있는지 나타낸다. `.txt` 확장자는 텍스트 파일임을, `.pdf`는 PDF 문서임을, `.cfg`

확장자는 어떤 프로그램의 구성정보를 담고 있는 형상관리 파일임을 나타낸다.>

단지 관습이기는 하지만 중요하다. 파일은 바이트(byte)를 담고 있다. PDF 문서, 이미지, 등을 규칙에 따라 바이트를 해석하는 것은 우리와 우리의 프로그램에 맡겨졌다.

`whale.mp3`같은 고래 PNG 이미지의 이름이 고래 노래의 음성파일로 변환되는 마술은 없다. 설사 누군가 두번 클릭할 때, 운영체제가 음악 재생기로 열어 실행할 수는 있지만 작동은 하지 않을 것이다.

아제 `ls -F data`(즉, 명령어 `ls`와 인수(arguments) `-F`과 `data`)을 실행하여 Nelle 의 `data` 디렉토리에 무엇이 있는지 살펴보자. `ls` 명령어의 앞에 대쉬('—')가 없는 두번째 인수는 현재 디렉토리 이외의 목록을 보고자 한다고 것을 나타낸다.

```
$ ls -F data

amino-acids.txt    elements/      morse.txt
pdb/                 planets.txt   sunspot.txt
```

명령어 실행결과는 4개의 텍스트 파일과 두개의 하위 디렉토리가 있다는 것을 보여준다. 이렇게 계층적으로 조직하여 관리하는 것은 작업을 체계적으로 추적할 수 있게 도움을 준다. 홈 디렉토리에 수백의 파일을 놓는 것도 가능하다. 하지만, 마치 책상위에 수백장의 종이를 쌓는 것과 유사해서, 이런 전략은 본인을 파멸로 이끌 수도 있다.

그런데 `data` 디렉토리 이름을 적은 것에 주목하세요. 끝에 슬래쉬가 없습니다. `-F` 플래그를 사용해서 파일과 디렉토리를 구별했을 때 사용한 `ls` 명령어에 디렉토리 이름 뒤에 붙었습니다. 그리고 상대 경로(relative path)이기 때문에 슬래쉬로 시작도 하지 않았습니다. 즉, `ls`에게 파일의 루트에서 보다는 현재 위치에서 찾으라고 합니다.

매개 변수(Parameters) 대 인수(Arguments) [위키피디아](#)
([Wikipedia](#))에 따르면, 인수(argument) and 매개 변수(parameter)는 약간 다른 것을 의미한다. 하지만, 실무에서 대부분의 사람들은 상호

호환적으로 혹은 일관성 없이 사용한다. 여기서도 구별없이 사용할 것이다.

`ls -F /data`을 실행(앞에 슬래쉬를 가지고)하면, 다른 답을 얻게 되는데, 왜냐하면 `/data`은 절대 경로(absolute path)이기 때문이다.

```
$ ls -F /data  
  
access.log      backup/      hardware.cfg  
network.cfg
```

앞의 `/` 슬래쉬는 컴퓨터에게 파일시스템의 루트의 경로를 따르게 명령한다. 그래서 명령어를 실행할 때 어느 위치에 있는지 관계없이 항상 정확하게 한 디렉토리만을 참조한다.

현재 작업 디렉토리를 바꾸려고 하면 어떨까요? 작업 디렉토리를 변경하기 전에 `pwd` 명령어는 현재 `/users/nelle`에 있다고 보여주고 인수가 없는 `ls` 명령어는 디렉토리의 내용을 보여준다.

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

작업 디렉토리를 변경하기 위해서 `cd` 다음에 디렉토리 이름을 사용한다. `cd`는 “change directory”의 두문어다. 하지만 약간 오해의 소지가 있다. 명령어 자체가 디렉토리를 변경하지는 않고, 단지 사용자가 어느 디렉토리에 있는지에 대한 셸의 생각을 바꾸는 것이다.

```
$ cd data
```

`cd`는 어떤 것도 출력하지 않지만, `cd` 명령어 다음에 `pwd`를 실행하면, `/users/nelle/data` 디렉토리에 있음을 확인할 수 있다. `ls` 명령어를 실행하면, `/users/nelle/data` 디렉토리 내용을 볼 수 있다. 왜냐하면 이 디렉토리가 현재 사용자가 있는 디렉토리이기 때문이다.

```
$ pwd
```

```
/users/nelle/data
```

```
$ ls -F
```

```
amino-acids.txt    elements/      morse.txt  
pdb/              planets.txt    sunspot.txt
```

이제 디렉토리를 따라 아래로 어떻게 갈 수 있는지를 알게 되었다. 어떻게 상위 디렉토리로 갈 수 있을까요? 절대 경로를 사용할 수 있습니다.

```
$ cd /users/nelle
```

하지만, 상위 디렉토리로 가기 위해서는 `cd ..` 명령어를 사용하는 것이 거의 항상 더 간단하다.

```
$ pwd
```

```
/users/nelle/data
```

```
$ cd ..
```

`..`은 특수 디렉토리 이름으로 “이것을 포함하는 디렉토리(the directory containing this one)”를 의미한다. 좀더 간단하게 현재 디렉토리의 부모(parent)를 지칭한다. 당연하지만, `cd ..` 다음에 `pwd`를 실행하면, 다시 `/users/nelle`로 돌아온다.

```
$ pwd
```

```
/users/nelle
```

`ls` 명령어를 실행할 때 특수 디렉토리 `..`는 보통 보여지지 않는다. 만약 화면에 출력하고자 한다면, `ls` 다음에 `-a` 플래그를 줄 수 있다.

```
$ ls -F -a

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

`-a`은 “모두 보여주기(show all)”를 나타낸다. `ls` 명령어로 `..` 같은 `.`로 시작하는 디렉토리와 파일을 보여준다. `..`은 만약 `/users/nelle`에 위치하고 있다면, `/users`를 나타낸다. `.`은 “현재 작업 디렉토리”를 의미하는 또 다른 특수 디렉토리를 보여준다. 좀 중복스럽게 보이지만, 곧 어떻게 사용되는지 알게 될 것이다.

직교(Orthogonality) 특수 이름 `.`과 `..`는 `ls`에만 속하는 것 이 아니고 모든 프로그램에서 같은 방식으로 해석된다. 예를 들어, `/users/nelle/data` 디렉토리에 있는데 `ls ..` 명령어는 `/users/nelle`의 목록을 보여줄 것이다. 어떻게 조합되든지 상관없이 동일한 의미를 가지게 될 때, 프로그래머는 직교(orthogonal)한다고 부른다. 직교 시스템은 사람들이 배우기 훨씬 쉬운데 왜냐하면 기억해야 할 특수 사례와 예외가 더 적기 때문이다.

Nelle의 파일프라인: 파일 구성하기

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 명기한다. `conference-paper`와 `revised-results` 같은 이름을 사용하곤 했다. 하지만, 몇년이 지난 후에 이해하기 어렵다는 것을 발견했다. (마지막 지푸라기는 `revised-revised-results-3` 디렉토리를 자신이 생성한 것을 발견했을 때였다.)

Nelle은 월과 일에 0을 앞에 붙여 디렉토리를 “년-월-일(year-month-day)” 이름지었다. 왜냐하면 쉘은 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 ID 연구실 관례에 따라 표식을 붙였다. 시료의 장소, 시간, 깊이, 그리고 다른 특징을 기록하기 위해서 수집 기록에 사용한 것과 동일하다. 그래서 각 파일의 이름으로 사용하기로 결정했다. 분석기의 출력값은 텍스트 형식이기 때문에 NENE01729A.txt, NENE01812A.txt, ... 같이 확장자를 붙였다. 1520개 파일 모두 동일한 디렉토리에 저장되었다.

홈 디렉토리에서 Nelle은 다음 명령어 무슨 파일이 있는지 확인할 수 있다.

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

엄청나게 많은 타이핑이지만 쉘에게 많은 일을 시킬 수도 있다. 만약 다음과 같이 타이핑하고,

```
$ ls nor
```

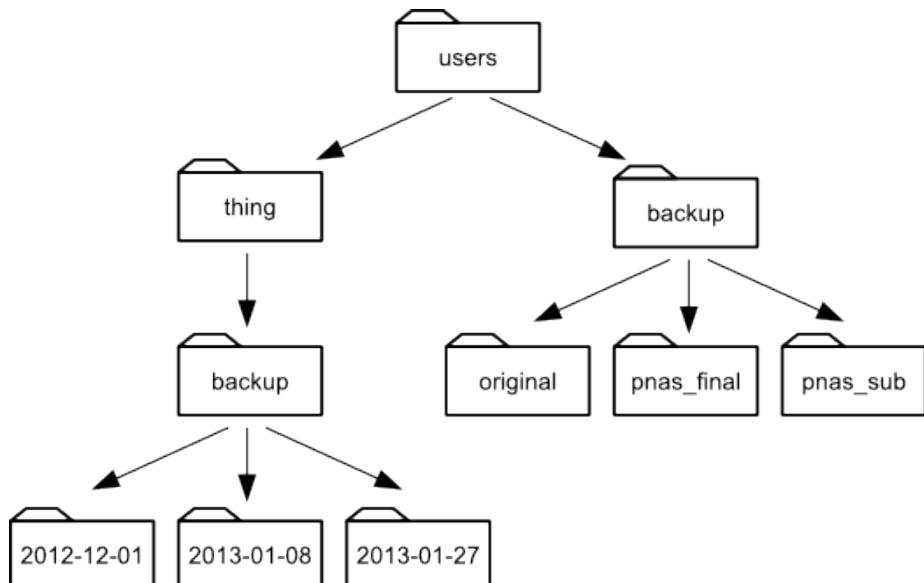
그리고 나서 템을 누르면, 자동적으로 쉘이 디렉토리 이름을 자동완성한다.

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

템을 다시 누르면, Bash가 명령문에 2012-07-03/을 추가하는데, 왜냐하면 유일한 가능한 완성조건이기 때문이다. 한번더 템을 누려면 아무것도 없다. 왜냐하면 1520 가지 경우의 수가 있기 때문이다. 템을 두번 누르면 모든 파일 목록을 가져온다. 이것을 템 완성(tab completion)이라고 부르고 앞으로도 다른 많은 툴에서도 많이 볼 것이다.

주요점

- 디스크의 정보를 관리를 담당하는 것은 파일 시스템이다.
- 정보는 파일에 저장되고, 파일은 디렉토리(폴더)에 저장된다.
- 디렉토리는 또한 다른 디렉토리에 저장될 수 있고, 디렉토리 트리를 구성하게 된다.
- `/` 자체는 전체 파일시스템의 루트 디렉토리다.
- 상대 경로는 현재 지점에서 시작하는 위치를 나타낸다.
- 절대 경로는 파일시스템의 루트에서 위치를 나타낸다.
- 경로의 디렉토리 이름은 유닉스에서는 `/`, 하지만 윈도우에서는 `\"`이다.
- `..`은 현재 디렉토리의 상위 디렉토리를 의미하고; `.` 자체는 “현재 디렉토리”를 의미한다.
- 대부분의 파일 이름은 파일명.확장자(something.extension)`이다. 확장자가 필수적이지 않고 특정한 것을 보증하지도 않지만, 파일의 데이터 형식을 나타내는 일반적으로 사용된다.
- 대부분의 명령어는 `-`로 시작하는 옵션(플래그)를 갖는다.



만약 `pwd` 명령어를 쳤을 때 화면에 `/users/thing`이 출력된다면, `ls .. /backup`은 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

만약 pwd 명령어를 쳤을 때 화면에 /users/thing이 출력되고, -r은 ls 명령어가 역순으로 화면에 출력하게 한다면, 무슨 명령어가 다음을 화면에 출력할까요?

pnas-sub/ pnas-final/ original/

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

디렉토리 이름없는 cd 명령어는 무엇을 수행할까요?

1. 아무 것도 하지 않는다.
2. 작업 디렉토리를 루트 /으로 변경한다.
3. 작업 디렉토리를 사용자의 홈 디렉토리로 변경한다.
4. 오류 메시지를 출력한다.

ls 명령어가 -s과 -h 을 인수로 사용되면 무슨 작업을 수행할까요?

파일과 디렉토리 생성

목표

- 주어진 도표에 맞는 계층적 디렉토리 구조 생성하기.
- 편집기 혹은 이미 만들어진 파일을 복사하거나 이름을 바꾸어서 상기 계층적 디렉토리에 파일을 생성하기.
- 명령 라인을 사용해서 디렉토리의 내용 화면에 출력하기.
- 특정 파일과 디렉토리 혹은 각각을 따로 삭제하기.

이제는 어떻게 파일과 디렉토리를 살펴보는지 알게되었지만, 어떻게 파일과 디렉토리를 먼저 생성할 수 있을까요? Nelle의 홈 디렉토리, `/users/nelle`,로 돌아가서 `ls -F` 명령어를 사용하여 무엇을 담고 있는지 살펴봅시다.

```
$ pwd
```

```
/users/nelle
```

```
$ ls -F
```

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

명령어 `mkdir thesis`을 사용하여 새 디렉토리 `thesis`를 생성합시다.(출력되는 것은 하나도 없습니다.)

```
$ mkdir thesis
```

이름에서 유출을 할 수도 하지 못할 수도 있지만, `mkdir`은 “make directory(디렉토리 생성하기)”를 의미한다. `thesis`는 상대 경로여서(즉, 앞선 슬래쉬가 없음) 새로운 디렉토리는 현재 작업 디렉토리 밑에 만들어진다.

```
$ ls -F
```

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

하지만 `thesis` 디렉토리에는 아직 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
```

어느 편집기가 좋을까요? nano가 텍스트 편집기다“라고 말할 때, 정말 ”텍스트“만 의미한다. 즉, 일반 문자 데이터만 작업할 수 있고, 테이블, 이미지, 혹은 다른 사람 친화적 미디어는 작업할 수 없다. 예제로 nano를 사용하는데 이유는 거의 누구나 훈련없이 사용할 수 있기 때문이다. 하지만 실제 작업에는 좀더 강력한 편집기를 사용하기 바란다. 유닉스 시스템 계열(맥 OS X, 리눅스)에서는 많은 프로그래머가 Emacs 혹은 Vim을 사용하는데 둘다 완전히 비직관적이고 심지어 유닉스 표준을 따른다. 혹은 그래픽 편집기로 Gedit를 사용한다. 윈도우에서는 Notepad++를 사용하는 것도 좋다.

무슨 편집기를 사용하든, 파일을 검색하고 저장하는 것을 앞 필요가 있다. 셀에서 편집기를 시작하면, (아마도) 현재 작업 디렉토리가 디폴트 위치가 된다. 컴퓨터 시작 메뉴에서 시작한다면, 대신에 데스크톱 혹은 문서 디렉토리에 파일을 저장하고 싶을지도 모른다. “다른이름으로 저장하기”로 다른 디렉토리로 이동하여 작업 디렉토리를 변경할 수 있다.

텍스트 몇 줄을 타이핑하고, 컨트롤+O (Control-O)를 눌러서 데이터를 디스크에 쓰면 저장된다.

GNU nano 2.0.6 File: draft.txt Modified
It's not "publish or perish" any more,
it's "share and thrive".

G Get Help **W** WriteOut **R** Read File **P** Prev Page **C** Cut Text **Cur Pos**
X Exit **J** Justify **W** Where Is **N** Next Page **U** UnCut Text **T** To Spell

파일이 저장되면, 컨트롤+X (Control-X)를 사용하여 편집기를 끝내고 쉘로 돌아간다. (유닉스 문서에서 ^A로 줄여서 “컨트롤+A(control-A)”를 표기한다.) nano 는 화면에 어떤 출력도 뿐여주지 않고 끝내지만, ls 명령어를 사용하여 draft.txt 파일이 생성된 것을 확인할 수 있다.

```
$ ls
```

```
draft.txt
```

rm draft.txt을 실행해서 깨끗이 정리합시다.

```
$ rm draft.txt
```

이 명령문은 파일을 제거한다. (rm은 “remove”를 줄였다.) ls를 다시 실행하면, 화면에 출력되는 것은 없고 파일이 사라진 것을 확인할 수 있다.

```
$ ls
```

삭제는 영원하다. 유닉스에는 휴지통이 없다. 파일을 삭제하면 파일 시스템의 관리대상에서 빠져서 디스트 저장공간이 다시 재사용되게 한다. 삭제된 파일을 찾아 되살리는 툴이 존재하지만, 어느 상황에서나 동작한다는 보장은 없다. 왜냐하면 컴퓨터가 파일이 저장된 공간을 바로 재사용할지 모르기 때문이다.

이번에는 파일을 다시 생성해서 cd ..를 사용하여 /users/nelle 상위 디렉토리로 이동해보자.

```
$ pwd
```

```
/users/nelle/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. 올바른 명령어는 `rmdir`이고 “remove directory(디렉토리 제거하기)”를 줄여서 표현한다. 하지만 이것도 동작하지 않는데 이유는 삭제하려는 디렉토리가 비어있지 않기 때문이다.

```
$ rmdir thesis  
rmdir: failed to remove `thesis': Directory not empty
```

이 작은 안전 기능이 많은 사람에게서 정말 많은 슬픔에서 구해줬다. 특히, 여러분이 타이핑에 초보라면 더욱 그렇다. `thesis` 디렉토리를 제거하려면, 먼저 `draft.txt` 파일을 삭제해야 한다.

```
$ rm thesis/draft.txt
```

The directory is now empty, so `rmdir` can delete it:

```
$ rmdir thesis
```

큰 힘에는 큰 책임이 따른다. (With Great Power Comes Great Responsibility) 디렉토리에 파일을 제거하고 디렉토리를 제거하는 방식은 지루하고 시간이 많이 걸린다. 대신에 `-r` 옵션을 가진 `rm` 명령어를 사용할 수 있다. `-r` 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 피해는 엄청나다.

다시한번 디렉토리와 파일을 생성하자. 이번에는 `thesis/draft.txt` 경로에서 `nano`를 실행함을 주목하자. 이전에는 `thesis` 디렉토리로 가서 `draft.txt`에 `nano`를 실행했다.

```
$ pwd  
/users/nelle  
  
$ mkdir thesis  
  
$ nano thesis/draft.txt  
$ ls thesis  
  
draft.txt
```

`draft.txt`는 특히 정보를 주는 이름이 아니어서 `mv`를 사용하여 파일 이름을 변경하자. `mv`는 “move”의 줄임말이다.

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

첫 번째 매개변수는 `mv` 명령어에게 이동하려는 대상을 두번째 매개변수는 어디로 이동되는지를 나타낸다. 이번 경우에는 `thesis/draft.txt`을 `thesis/quotes.txt`으로 이동한다. 이렇게 파일을 이동하는 것은 파일 이름을 바꾸는 것과 동일한 효과를 가진다. `ls` 명령어를 사용하여 `thesis` 디렉토리가 이제 `quotes.txt` 한 파일만을 가지고 있음을 확인할 수 있다.

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

일관되지는 않는데 `mv` 명령어는 디렉토리에도 쓸수 있다. 하지만, `mkdir` 명령어는 없다.

`quotes.txt` 파일을 현재 작업 디렉토리로 이동합시다. `mv`를 다시 사용한다. 하지만 이번에는 두번째 매개변수로 디렉토리 이름을 사용해서 파일이름을 바꾸지 않고 새로운 장소에 놓는다. (이것이 왜 명령어가 “move(이동)”으로 불리는 이유다.) 이번 경우에 사용하는 디렉토리 이름이 앞에서 언급한 특수 디렉토리 이름이다.

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

효과는 과거 있었던 디렉토리의 파일을 현재 작업 디렉토리로 옮기는 것이다. `ls` 명령어가 `thesis` 디렉토리가 비었음을 보여준다.

```
$ ls thesis
```

더 나아가, `ls` 명령어를 파일 이름 혹은 디렉토리 이름의 매개변수로 사용하면 그 해당 파일 혹은 디렉토리만 화면에 보여준다. 이것을 사용하여 `quotes.txt` 파일이 현재 작업 디렉토리에 있음을 볼 수 있다.

```
$ ls quotes.txt
```

```
quotes.txt
```

`cp` 명령어는 `mv` 명령어와 거의 동일하게 동작한다. 차이점은 이동하는 대신에 복사한다는 것이다. 매개변수로 두개의 경로를 가진 `ls` 명령어로 제대로 작업을 했는지 확인할 수 있다. 대부분의 유닉스 명령어와 마찬가지로 `ls` 명령어에 한번에 수천개의 경로가 주어질 수 있다.

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

```
quotes.txt thesis/quotations.txt
```

복사를 수행했는지 증명하기 위해서 현재 작업 디렉토리에서 `quotes.txt` 파일을 삭제하고 동일한 `ls` 명령어를 실행한다. 이번에는 현재 디렉토리에서 `quotes.txt` 파일을 찾을 수 없지만, 삭제하지 않은 `thesis` 폴더의 복사본은 찾아서 보여준다.

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

또다른 유용한 축약 텁 쉘은 경로의 시작 ~ (틸드,tilde) 문자를 “현재 사용자 홈 디렉토리”로 해석한다. 예를 들어, Nelle의 홈 디렉토리가 /home/nelle이면, ~/data은 /home/nelle/data과 동일한다. 경로에 첫번째 문자일 경우에만 동작해서 here/there/~/elsewhere은 /home/nelle/elsewhere이 아니다.

주요점

- 유닉스 문서는 '^A'이 “컨트롤+A(control-A)”를 의미한다.
- 쉘은 휴지통 개념이 없다. 무언가 삭제되면, 정말 삭제된다.
- Nano는 매우 간단한 텍스트 편집기다. 실제 작업에는 제발 다른 편집기를 사용하세요.

아래 보여진 일련의 명령문에 끝에 ls명령어의 출력값은 무엇일까요?

```
$ pwd  
/home/jamie/data  
$ ls  
proteins.dat  
$ mkdir recombine  
$ mv proteins.dat recombine  
$ cp recombine/proteins.dat ../proteins-saved.dat  
$ ls
```

다음이 주어졌다고 가정합니다.

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

무슨 명령어를 실행해야 아래 명령어를 실행했을 때 다음에 보여지는 출력을 생성 할까요?

```
$ ls  
analyzed raw  
$ ls analyzed  
fructose.dat sucrose.dat
```

다음과 같이 몇개의 파일 이름과 디렉토리 이름이 주어졌을 때 cp 명령어는 무엇을 수행할까요?

```
$ mkdir backup  
$ cp thesis/citations.txt thesis/quotations.txt backup
```

다음과 같이 세개 혹은 그 이상의 파일 이름이 주어졌을 때 cp는 무엇을 수행할까요?

```
$ ls -F  
intro.txt methods.txt survey.txt  
$ cp intro.txt methods.txt survey.txt
```

ls -R 명령어는 디렉토리의 내용을 재귀적으로 화면에 출력한다. 즉, 하위 디렉토리, 하위의 하위 디렉토리 등등 알파벳 순으로 계층적 수준으로 뿌려준다. ls -t 명령어는 마지막 변경사항의 시간 순으로 내용을 화면에 출력한다. 즉, 가장 최근에 변경된 파일 혹은 디렉토리를 먼저 정렬하여 화면에 뿌려준다. ls -R -t은 파일과 디렉토리를 어떤 순서로 화면에 보여줄까요?

파이프와 필터

목표

- 명령어의 출력결과를 파일로 다시 돌리기
- 키보드 입력 대신에 다시 보내기(redirection)를 사용하여 파일 처리하기
- 2단계 혹은 3단계를 가진 명령문 구성하기
- 프로그램 혹은 파이프라인에 처리할 입력을 주지 않았을 때 통상 발생하는 일을 설명하기
- “작은 조각, 느슨하게 결합하기(small pieces, loosely joined)”라는 유닉스 철학을 설명하기

몇가지 기초 명령어를 배웠기 때문에 마침내 쉘의 가장 강력한 기능을 살펴볼 수 있다. 새로운 방식으로 존재하는 프로그램을 쉽게 조합할 수 있게 합니다. 간단한 유기분자 설명을 하는 6개 파일을 담고 있는 `molecules`(분자)라는 디렉토리로 시작한다. `.pdb` 파일 확장자는 단백질 데이터 뱅크 (Protein Data Bank) 형식으로 분자의 각 원자의 형식과 위치를 표시하는 간단한 텍스트 형식을 나타낸다.

```
$ ls molecules
```

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

명령어 `cd`로 디렉토리로 가서 `wc *.pdb` 명령어를 실행한다. `wc` 명령어는 “word count”의 축약어로 파일의 라인 수, 단어수, 문자수를 카운트한다. `*.pdb`에서 `*`은 0 혹은 더 많은 일치하는 문자를 찾는다. 그래서 쉘은 `*.pdb`을 통해 `.pdb` 전체 리스트를 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
```

와일드 카드(Wildcards) *는 wildcard다. 와일드카드는 0 혹은 그 이상의 문자와 일치해서, *.pdb은 ethane.pdb, propane.pdb 등을 매치하는데 맨 앞에 'p'만 일치하기만 하면 되기 때문이다.

?도 또한 와일드카드지만 단지 단일 문자만 매칭한다. 이것이 의미하는 바는 p?.pdb은 pi.pdb 혹은 p5.pdb을 매칭하지만, propane.pdb은 매칭하지 않는다. 한번에 원하는 수만큼 와일드카드를 사용할 수 있다. 예를 들어, p*.p?*는 'p'로 시작하고 ':'과 'p', 그리고 최소 한자의 이상의 문자로 끝나는 임의의 문자열을 매칭한다고 표현할 수 있는데 '?'이 한 문자를 매칭해야하고 마지막 ':'은 끝에 임의의 문자숫자와 매칭할 수 있기 때문이다. 그래서 p*.p?*은 preferred.practice과 심지어 p.pi도 매칭한다(첫번째 ":"은 어떤 문자도 매칭할 수가 없음). 하지만 quality.practice은 매칭할 수 없는데 이유는 'p'로 시작하지 않고, preferred.p도 매칭할 수 없는데 'p' 다음에 최소 하나의 문자가 필요한데 없기 때문이다.

쉘이 와일드카드를 봤을 때, 요청된 명령문을 시작하기 전에 와일드카드를 확장하여 매칭할 파일 이름 목록을 생성한다. wc과 ls 명령어는 결코 와일드카드 문자를 보지 못하고 단지 와일드카드가 매칭하는 것만을 보게된다. 이것은 직교 설계(orthogonal design)의 또 다른 사례이다.

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 -c을 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 단계는 다음 명령을 실행하는 것이다.

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

> 은 쉘로 하여금 화면에 처리 결과를 뿌리는 대신에 파일로 되돌리기(redirect) 게 한다. 만약 파일이 존재하지 않으면 파일을 생성하고 파일이 존재하면 파일의 내용을 덮어쓰기 한다. (이것이 왜 화면에 출력결과가 없는 이유다. wc이 출력하는 모든 것은 lengths 파일에 대신 들어간다.) ls lengths을 통해 파일이 존재하는 것을 확인한다.

```
$ ls lengths
```

lengths

cat lengths을 사용해서 화면에 lengths의 내용을 보낼 수 있다. cat은 “concatenate”를 줄인 것이고 하나씩 하나씩 파일의 내용을 출력한다. 이번 사례에는 단지 하나의 파일이 있어서 cat는 단지 한 파일이 담고 있는 내용만을 보여준다.

```
$ cat lengths
```

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

sort 명령어를 사용해서 파일 내용을 정렬합니다. ‘-n’ 옵션을 사용해서 알파벳 대신에 숫자 방식으로 정렬할 것임을 표시한다. 이 명령어는 파일 자체를 변경하지 않고 대신에 정렬된 결과를 화면으로 보낸다.

```
$ sort -n lengths
```

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

> lengths을 사용해서 wc 실행결과를 lengths에 넣었듯이, 명령문 다음에 >; sorted-lengths을 넣음으로서 임시 파일이름인 sorted-lengths에 정렬된 목록 라인을 담을 수 있다. 이것을 실행한 다음에, 또 다른 head 명령어를 실행해서 sorted-lengths에서 첫 몇 행을 뽑아낼 수 있다.

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

```
9  methane.pdb
```

head에 -1 매개변수를 사용해서 파일의 첫번째 행만이 필요하다고 지정한다. -20은 처음 20개 행만을 지정한다. sorted-lengths가 가장 작은 것에서부터 큰 것으로 정렬된 파일 길이 정보를 담고 있어서, head의 출력 결과는 가장 짧은 행을 가진 파일이 되어야만 한다.

이것이 혼란스럽다고 생각한다면, 여러분은 정말 좋은 회사에 다니고 있는 것이다. wc, sort, head 명령어가 무엇을 수행하는지 이해해도, 중간에 산출되는 파일은 무엇이 진행되고 있는지 따라가기가 쉽지 않다. sort과 head를 함께 실행해서 이해하기 훨씬 쉽게 할 수 있다.

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

```
9  methane.pdb
```

두 명령문 사이의 수직 막대를 파이프(pipe)라고 부른다. 수직막대는 쉘에게 왼편의 명령문의 출력결과를 오른쪽 명령문의 입력값을 사용한다고 말을 전한다. 컴퓨터는 필요하면 임시 파일을 생성하거나 한 프로그램에서 주 기억장치의 다른

프로그램으로 데이터를 복사하거나 혹은 완전히 다른 작업을 수행한다. 사용자는 알 필요도 없고 관심을 가질 이유도 없다.

또 다른 파이프를 사용해서 `wc`의 출력결과를 `sort`에 바로 보내고 나서 다시 처리 결과를 `head`에 보낸다.

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

```
9 methane.pdb
```

이것이 정확하게 수학자가 $\sin(\pi x)^2$ 같은 중첩함수를 사용하는 것과 같다. $\sin(\pi x)^2$ 은 x 곱하기 π 의 사인 제곱과 같다. 우리의 경우는 `*.pdb`의 행수를 세어서 정렬의 첫 부분을 계산하는 것이다.

파이프를 생성할 때 뒤에서 실질적으로 일어나는 일은 다음과 같다. 컴퓨터가 프로그램을 실행할 때 프로그램의 소프트웨어와 현재 상태 정보를 담기 위해서 주기 억장치 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”)라고 불리는 디폴트 출력 채널이 있다.

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 임시적으로 키보드에 타이핑하는 무엇이든지 그 프로세스의 표준 입력으로 보내지고, 프로세스는 표준 출력에 무엇이든지 화면에 전송한다.

`wc -l *.pdb > lengths`을 실행할 때 여기 일어나는 것을 설명하면 다음과 같다. `wc` 프로그램을 실행할 새로운 프로세스를 생성하라고 컴퓨터에 말하면서 쉘은 시작한다. 파일이름을 매개변수로 제공했기 때문에 표준 입력 대신에 `wc`는 매개변수에서 입력값을 읽어온다. `>`을 사용해서 출력값을 파일에 되돌리는데 사용했기 때문에, 쉘은 프로세스의 표준 출력결과를 파일에 연결한다.

`wc -l *.pdb | sort -n`을 실행한다면, 쉘은 두 개의 프로세스를 생성한다. (파이프의 각 프로세스에 대해서 하나씩) 그래서 `wc`과 `sort`은 동시에 실행된다. `wc`의 표준 출력은 직접적으로 `sort`의 표준 입력으로 들어간다. `>`같은 되돌리기가 없기 때문에 `sort`의 출력은 화면으로 나가게 된다. `wc -l *.pdb | sort -n | head`

-1을 실행하면, 파일에서 wc에서 sort으로, sort에서 head를 통해 화면으로 나가게 되는 데이터 흐름을 가진 3개의 프로세스가 있다.

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

중요한 점은 표준입력에서 텍스트 행을 읽고 표준 출력에 텍스트 행을 쓰는 임의의 프로그램은 이런 방식으로 행동하는 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 프로그램의 힘을 배가할 수 있다.

입력 되돌리기 프로그램의 출력 결과를 되돌리기 위해서 >을 사용하는 것과 마찬가지로 <을 사용해서 입력을 되돌릴 수도 있다. 즉, 표준 입력 대신에 파일로 부터 읽을 수 있다. 예를 들어, 첫째 사례로, wc 는 무슨 파일을 여는지를 명령 라인의 매개변수에서 얻는다. 두번째 사례는, wc가 명령 라인 매개변수가 없다. 그래서 표준 입력에서 읽지만, 쉘에게 ammonia.pdb의 내용을 wc에 표준 입력으로 보내라고 했다.

Nelle의 파이프라인 Pipeline: 파일 확인하기

앞에서 설명한 것 처럼 Nelle은 분석기를 통해 시료를 시험해서 1520개 파일을 north-pacific-gyre/2012-07-03 디렉토리에 생성했다. 빠르게 진정성 확인하기 위해서 다음과 같이 타이핑한다.

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

결과는 다음과 같은 1520 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 -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 누군가 기계를 사용했고, 다시 돌려놓는 것을 깜빡 잊었다. 시료를 다시 시험하기 전에 파일중에 너무 큰 데이터가 있는지를 확인한다.

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

```
300 NENE02040A.txt  
300 NENE02040B.txt  
300 NENE02040Z.txt  
300 NENE02043A.txt  
300 NENE02043B.txt
```

숫자는 좋아 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 위해서 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 없다. 다른 방법으로 정보를 더 수집하기에는 너무 늦어서, 분석에서 두 파일을 제외하기로 했다. rm 명령어를 사용하여 삭제할 수 있지만, 향후에 깊이(depth)정보가 관련없는 다른 분석을 실시할 수도 있다. 그래서 와일드 카드 표현식 *[AB].txt을 사용하여 파일을 조심해서 선택하기로 한다. 언제나 그렇듯이, '*'는 임의 숫자의 문자를 매칭한다. [AB] 표현식은 'A'혹은 'B'를 매칭해서 Nelle이 가지고 있는 유효한 데이터 파일 모두를 매칭한다.

주요점

- 명령어 > 파일 (command > file)은 명령어의 출력을 파일로 되돌린다.
- 첫째_명령어 | 둘째_명령어(first | second)는 파이프라인이다. 첫째_명령어 출력은 둘째_명령어의 입력으로 사용된다.
- 쉘을 사용하는 가장 좋은 방법은 파이프를 사용해서 간단한 단일 목적 프로그램(필터)을 조합하는 것이다.

파일에 sort를 실행하면,

```
10  
2  
19  
22  
6
```

출력은 다음과 같다.

```
10  
19  
2  
22  
6
```

동일한 입력에 sort -n을 실행하면, 대신에 다음을 얻게된다.

```
2  
6  
10  
19  
22
```

인수 -n가 왜 이런 효과를 가지는지 설명하세요.

다음 명령문과

```
wc -l < mydata.dat
```

다음 명령문의 차이점은 무엇인지 설명하세요.

```
wc -l mydata.dat
```

명령문 uniq은 입력으로부터 인접한 중복된 행을 제거한다. 예를 들어, salmon.txt 파일이 다음을 포함한다면,

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

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 위해서 파이프로 무슨 다른 명령어를 조합할 수 있을까요?

`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 되돌리기를 마친 텍스트는 무엇일까요?

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

다음 명령문을 실행하면,

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

다음 출력결과를 만들어 낸다.

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

파일이 담고 있는 동물이 무엇인지를 알아내기 위해서 무슨 다른 명령어가 파이프 라인에 추가되어야 하나요? (동물 이름에 어떠한 중복도 없어야 합니다.)

루프(Loops)

목표

- 파일 집합에서 각 파일에 따로 따로 나누어서 하나 혹은 그 이상의 명령어를 적용하는 루프를 작성하기
- 루프가 실행되는 동안에 루프 변수가 취하는 값을 추적하기
- 변수 이름과 값의 차이에 대해 설명하기
- 공백과 어떤 문자부호 문자는 파일 이름에 사용되지 말아야 되는지 설명하기
- 무슨 명령어가 최근에 실행되었는지를 어떻게 확인하는지 시범으로 보여주기
- 명령어를 다시 타이핑하지 않고 최근에 실행된 명령어를 다시 실행하기

와일드카드와 템 자동완성은 타이핑을 (타이핑 실수를) 줄이는 두가지 방법이다. 또 다른 것은 쉘이 반복해서 어떤것을 수행하게 하는 것이다. `basilisk.dat`, `unicorn.dat` 등등으로 이름 붙여진 수백개의 게놈 데이터 파일이 있다고 가정하자. 이번 예제에서, 단지 두개의 예제 파일만 있는 `creatures` 디렉토리를 사용할 것이지만 원칙은 훨씬 더 많은 파일에 즉시 적용될 수 있다. 신규 파일이 도착할 때, 기존 파일을 `original-basilisk.dat`과 `original-unicorn.dat`으로 이름을 변경하고 싶다. 하지만 다음을 사용할 수는 없다.

```
$ mv *.dat original-*.dat
```

왜냐하면 두 파일 경우에도 전개가 다음과 같이 될 것이다.

```
$ mv basilisk.dat unicorn.dat
```

이것은 파일을 백업하지 않고, `basilisk.dat` 파일 무엇이든지 `unicorn.dat` 파일의 내용으로 교체할 것이다.

대신에 루프(loop)를 사용해서 리스트의 각 파일에 대해서 임의의 연산을 수행할 수 있다. 여기 간단한 예제로 교대로 각 파일의 첫 3줄만을 화면에 출력한다.

```
$ for filename in basilisk.dat unicorn.dat
> do
>     head -3 $filename
> done
```

```
COMMON NAME: basilisk
CLASSIFICATION: basiliscus vulgaris
UPDATED: 1745-05-02
COMMON NAME: unicorn
CLASSIFICATION: equus monoceros
UPDATED: 1738-11-24
```

쉘의 키워드 `for`를 보면, 쉘은 리스트의 각각에 대해서 명령문(혹은 명령문 그룹)을 반복할 것이라는 것을 안다. 이번 경우에는 리스트는 두 파일이름이다. 루프를 반복할 때마다, 현재 작업하고 있는 파일의 이름은 `filename`으로 불리는 변수(variable)에 할당된다. 루프 내부에 변수 이름 앞에 \$ 기호를 붙여 변수의 값을 얻는다. 즉, `$filename` 루프가 첫번째 돌 때 `basilisk.dat`이 되고, `unicorn.dat`이 두번째가 되고 계속 이어진다. 마지막으로 실제 실행되는 명령어는 오랜 친구인 `head`가 되어서, 루프는 교대로 각 데이터 파일의 첫 3줄을 출력한다.

프롬프트 따라가기 쉘 프롬프트가 \$에서 >으로 바뀌고 루프안에서 타이핑을 할 때 다시 계속된다. 두번째 프롬프트는 >, 완전한 명령문을 타이핑하는 것을 끝마치지 않았다는 것을 상기시키려고 다르다.

목적을 좀더 사람 독자에게 명확히 하기 위해서 루프의 변수를 `filename`로 했다. 쉘 자체는 변수가 어떻게 작명되든지 문제삼지 않는다. 만약 루프를 다음과 같이 작성하거나,

```
for x in basilisk.dat unicorn.dat
do
    head -3 $x
done
```

혹은

```
for temperature in basilisk.dat unicorn.dat
do
    head -3 $temperature
done
```

둘다 정확하게 동일하게 동작한다. 이렇게는 절대 하지 마세요. 사람이 프로그램을 이해할 수 있을 때만 프로그램은 유용해서, x같은 의미없는 이름이나, temperature 같은 오해를 줄 수 있는 이름은 프로그램을 읽는 사람이 생각하기에 프로그램이 수행해야 할 것을 프로그램이 수행하지 못할 가능성을 높인다.

다음에 좀더 복잡한 루프가 있다.

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

쉘이 *.dat을 전개해서 쉘이 처리할 파일 리스트로 생성한다. 그리고 나서 루프 몸통(loop body) 부분이 파일 각각에 대해서 두 명령어를 실행한다. 첫 명령어 echo는 명령 라인 매개변수를 표준 출력으로 화면에 뿐려준다. 예를 들어,

```
$ echo hello there
```

다음을 화면에 출력한다.

```
hello there
```

이 사례에서, 쉘이 파일 이름으로 \$filename을 전개했기 때문에, echo \$filename은 단지 파일의 이름만 화면에 출력한다. 다음과 같이 작성할 수 없는 것에 주목한다.

```
for filename in *.dat
```

```

do
$filename
head -100 $filename | tail -20
done

```

왜냐하면, \$filename이 basilisk.dat으로 전개될 때 루프의 처음에 쉘은 프로그램을 basilisk.dat을 실행하려고 한다. 마지막으로, head와 tail 조합은 무슨 파일이 처리되든지 81-100줄만 선택해서 화면에 뿐려준다.

파일, 디렉토리, 변수 등 이름에 공백 루프의 파일 이름 전개는 파일 이름에 공백을 사용하지 말아야 하는 또 다른 이유다. 데이터 파일이 다음과 같은 이름으로 되었다고 가정하자.

```

basilisk.dat
red dragon.dat
unicorn.dat

```

다음을 사용하여 파일을 처리하려고 한다면,

```

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

```

쉘은 *.dat을 전개해서 다음을 생성한다.

```
basilisk.dat red dragon.dat unicorn.dat
```

좀 오래된 Bash 혹은 대부분의 쉘과 마찬가지로, filename이 순차적으로 다음 값으로 할당될 것이다.

```

basilisk.dat
red
dragon.dat
unicorn.dat

```

이것이 문제다. `head`는 `red`과 `dragon.dat` 파일을 읽을 수가 없다. 왜냐하면 파일이 존재하지 않고 `red dragon.dat` 파일을 읽도록 할 수도 없다.

변수 사용을 인용부호(quoting) 처리해서 약간 더 강건하게 스크립트를 작성할 수 있다.

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

하지만, 파일명에 공백 혹은 다른 특수 문자의 사용을 피하는 것이 훨씬 더 간단하다.

원본 파일의 이름을 바꾸는 문제로 다시 돌아가서, 다음 루프를 사용해서 문제를 해결할 수 있다.

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

상기 루프는 `mv` 명령문을 각 파일이름에 실행한다. 처음에 `$filename`이 `basilisk.dat`로 전개될 때, 쉘은 다음을 실행한다.

```
mv basilisk.dat original-basilisk.dat
```

두번째에는 명령문이 다음과 같다.

```
mv unicorn.dat original-unicorn.dat
```

두번 측정, 한번 실행 루프는 많은 작업을 한번에 수행하는 방법이다. 혹은 만약 잘못된 작업을 수행한다면, 한번에 많은 실수를 저지른다. 루프가 수행하는 것을 확인하는 한 방법은 실제로 실행하는 대신에 실행할 명령어를 echo를 사용하여 메아리 치는 것이다. 예를 들어, 다음과 같이 파일 이름을 바꾸는 루프를 작성할 수 있다.

```
for filename in *.dat
do
    echo mv $filename original-$filename
done
```

mv을 실행하는 대신에, 루프가 echo을 실행해서 실제 명령어를 실행하지 않고 다음을 화면에 출력한다.

```
mv basilisk.dat original-basilisk.dat
mv unicorn.dat original-unicorn.dat
```

그리고 나서, 위쪽 화살표를 사용해서 루프를 다시 화면에 출력하고, 뒤쪽 화살표를 사용해서 echo 단어에 도달해서 삭제하고 실제 mv 명령어로 루프를 실행하기 위해서 “엔터(enter)”키를 누른다. 이 방법은 실패하지 않는 완전한 것은 아니지만, 루프가 어떻게 동작하고 있느지를 학습할 때, 어떤 일이 일어나고 있는지를 살펴볼 수 있는 간편한 방법이다.

Nelle의 파이프라인: 파일 처리하기

Nelle은 지금 데이터 파일을 처리할 준비가 되었다. 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 명심하세요.

```
$ cd north-pacific-gyre/2012-07-03
$ for datafile in *[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. “stat”을 각 입력 파일에 접두어로 두는 것은 간단해 보이는 작업을 수행하도록 루프를 변경한다.

```
$ for datafile in *[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
```

`goostats`을 아직 실행하지는 않았지만, 이제는 적합한 파일을 선택해서 올바른 출력 파일이름을 생성하는 것을 확신할 수 있다.

명령어를 반복적으로 타이핑하는 것은 귀찮은 일이지만 Nelle은 실수를 하는 것에 대해서 적정하고 있다. 그래서 루프를 다시 입력하는 대신에 위쪽 화살표를 누른다. 위쪽 화살표에 대응해서, 쉘은 한줄에 전체 루프를 다시 보여준다. (스크립트 부분을 구분하기 위해서 세미콜론을 사용)

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

왼쪽 화살표 키를 사용해서 Nelle은 백업하고 `echo` 명령어에서 `goostats`으로 변경한다.

```
$ for datafile in *[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. 잠시후에 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 편빛해서 다음과 같이 작성한다.

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

시작과 끝 쉘에 ^A, 콘트롤+A(Control-A)를 타이핑해서 라인의 처음으로 가고 ^E를 쳐서 라인의 끝으로 이동한다.

이번에 프로그램을 실행할 때, 매 5초간격으로 한줄을 출력한다.

NENE01729A.txt

NENE01729B.txt

NENE01736A.txt

...

1518 곱하기 5초를 60으로 나누면 작성한 스크립트는 약 2시간 정도 실행한다고 볼 수 있다. 마지막 점검으로 또다른 터미널 윈도우를 열어서, north-pacific-gyre/2012-07-03 디렉토리로 가서 cat stats-NENE01729B.txt 을 사용해서 출력 파일 중의 하나를 면밀히 조사한다. 출력결과가 좋아보여서 커피를 마시고 좀더 논문을 읽기로 한다.

역사를 아는 사람은 반복할 수 있다. 앞선 작업을 반복하는 또다른 방법은 history 명령어를 사용해서 실행된 마지막 수백개의 명령어의 리스트를 얻고나서, 이를 명령어 중의 하나를 반복하기 위해서 !123("123"은 명령 숫자로 교체된다.)을 사용하는 것이다. 예를 들어 Nelle이 다음과 같이 타이핑한다.

```
$ history | tail -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
```

그러면 단순히 !458을 타이핑함으로써 NENE01729B.txt 파일에 goostats를 다시 실행할 수 있다.

주요점

- `for`루프는 리스트에 있는 모든 것에 명령어를 한번씩 반복한다.
- 모든 `for` 루프는 현재 “것(thing)”을 참조하는 변수를 필요로 한다.
- `$name`을 사용해서 변수를 전개하라. (즉, 값을 얻어라.)
- 공백, 인용부호, 혹은 ‘*’, ‘?’같은 와일드카드 문자를 파일이름에 사용하지 마라. 변수 전개를 복잡하게 한다.
- 파일에 일관된 이름을 부여해서 루프를 돌릴 때 와일드카드 패턴으로 파일을 찾기 용이하게 하라.
- 명령어를 편집하고 반복하기 위해서, 위쪽 화살표를 사용해서 앞선 명령어를 스크롤하라.
- `history` 명령어를 사용해서 최근 명령어를 화면에 출력하고, `!number`를 사용해서 숫자로 명령을 반복하라.

`ls` 명령어가 초기 다음과 같이 화면에 출력한다고 가정하자.

```
fructose.dat      glucose.dat      sucrose.dat
```

다음 스크립트의 출력결과는 무엇인가요?

```
for datafile in *.dat
do
    ls *.dat
done
```

같은 디렉토리에서, 다음 루프의 효과는 무엇인가요?

```
for sugar in *.dat
do
    echo $sugar
    cat $sugar > xylose.dat
done
```

1. fructose.dat, glucose.dat, sucrose.dat을 출력하고, sucrose.dat을 복사해서 xylose.dat을 생성한다.
2. fructose.dat, glucose.dat, sucrose.dat을 출력하고, 모든 파일 3개를 합쳐서 xylose.dat을 생성한다.
3. fructose.dat, glucose.dat, sucrose.dat, xylose.dat을 출력하고, sucrose.dat을 복사해서 xylose.dat을 생성한다.
4. 위 어느 것도 아니다.

expr 명령어는 명령-라인 매개변수를 사용하여 간단한 연산을 한다.

```
$ expr 3 + 5
8
$ expr 30 / 5 - 2
4
```

다음이 주어진 상태에서 다음의 출력값은 무엇인가?

```
for left in 2 3
do
    for right in $left
    do
        expr $left + $right
    done
done
```

다음 루프가 무슨 작업을 하는지 말로 설명하세요.

```

for how in frog11 prcb redig
do
    $how -limit 0.01 NENE01729B.txt
done

```

쉘 스크립트

목표

- 고정된 파일 집합에 하나의 명령어 혹은 일련의 명령어를 실행하는 쉘 스크립트를 작성하기
- 명령 라인에서 쉘 스크립트 실행하기
- 명령 라인에서 사용자가 정의한 파일 집합에 동작하는 쉘 스크립트 작성하기
- 사용자가 작성한 쉘 스크립트를 포함하는 파이프라인 생성하기

마침내 무엇이 쉘을 그토록 강력한 프로그래밍 환경이 되도록 만드는지 볼 준비가 되었다. 자주 반복적으로 사용하는 명령어를 파일에 저장할 것이고, 단 하나의 명령어를 타이핑함으로써 나중에 이 모든 연산을 다시 실행할 수 있다. 역사적 이유로 파일에 저장된 명령어 꾸러미는 통상 쉘 스크립트(shell script)라고 부르지만 실수로 그렇게 부르는 것은 아니다. 실제로 작은 프로그램이다.

`molecules/` 디렉토리로 돌아가서 `middle.sh` 파일에 다음 행을 추가해서 시작해봅시다.

```

$ cd molecules
$ nano middle.sh

head -15 octane.pdb | tail -5

```

앞서 작성한 파이프에 변형으로 `octane.pdb` 파일에서 11-15 행을 선택한다. 기억할 것은 명령어로서 실행하지 않고, 명령어를 파일에 넣는다.

파일을 저장하면, 쉘로 하여금 파일이 담고 있는 명령어를 실행하게 할 수 있다. 지금 쉘은 `bash`이고, 다음과 같이 명령어를 실행한다.

```
$ 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
```

과연, 스크립트의 출력은 정확하게 파이프라인을 직접적으로 실행한 것과 동일하다.

텍스트 대 텍스트가 아닌 것 아무거나 종종 마이크로소프트 워드 혹은 리브르오피스 Writer 프로그램을 “텍스트 편집기”로 호출한다. 하지만, 프로그램에 대해서는 조금더 주의를 할 필요가 있다. 디폴트로, 마이크로소프트 워드는 .docx 파일을 사용해서 텍스트를 저장할 뿐만 아니라 글꼴, 제목, 등등의 형식 정보도 함께 저장한다. 이렇나 추가 정보는 문자로 저장되지 않아서 head 같은 도구에게는 무의미하다. head 같은 툴은 입력 파일이 문자, 숫자, 표준 컴퓨터 키보드의 특수 문자만을 포함하기를 바란다. 그러므로 프로그램을 편집할 때, 일반 텍스트 편집기를 사용하거나, 혹은 일반 텍스트로 파일을 저장하도록 주의한다.

만약 임의의 파일에서 행을 선택하고자 한다면 어떨까요? 파일 이름을 바꾸기 위해서 매번 middle.sh을 편집할 수 있지만, 단순히 명령어를 다시 타이핑하는 것보다 아마 오래 걸릴 것이다. 대신에 middle.sh을 편집해서 octane.pdb를 \$1으로 불리는 특수 변수로 변경하자.

```
$ cat middle.sh
```

```
head -20 $1 | tail -5
```

쉘 스크립트 내부에서, \$1은 “명령 라인에 첫 파일 이름(혹은 다른 매개변수)”을 의미한다. 스크립트를 다음과 같이 이제 실행할 수 있다.

```
$ 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 스크립트를 실행한다.

```
$ 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

하지만, 여전히 행의 범위를 조절할 때마다 `middle.sh`을 편집할 필요가 있다. 특수 변수 `$2`과 `$3`을 사용해서 이 문제를 해결하자.

```
$ cat middle.sh
```

```
head $2 $1 | tail $3
```

```
$ bash middle.sh pentane.pdb -20 -5
```

ATOM	14	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`을 읽는 다음 사람은 잠시 시간을 들여서 스크립트가 무엇을 수행하는지 알아내야 할지 모른다. 스크립트를 상단에 주석(comments)을 추가해서 좀 더 낫게 만들 수 있다.

```
$ cat middle.sh

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

주석은 #문자로 시작하고 행의 끝까지 간다. 컴퓨터는 주석을 무시하지만, 사람들
이 스크립트를 이행하고 사용하는데 정말 귀중한다.

만약 많은 파일을 단 하나의 파이프라인으로 처리하고자 한다면 어떨까? 예를 들어,
.pdb 파일을 길이 순으로 정렬하려면 다음과 같이 태이핑한다.

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

`wc -l`은 파일에 행수를 출력하고 `sort -n`은 숫자순으로 파일의 행수를 정렬한다.
파일에 담을 수 있지만, 현재 디렉토리에 .pdb 파일만을 정렬한다. 다른 종류의
파일의 정렬된 목록을 얻으려고 한다면, 스크립트에 이 모든 이름을 얻는 방법이
필요하다. \$1, \$2 등을 사용할 수 없는데 왜냐하면 얼마나 많은 파일이 있는지를
알 수 없기 때문이다. 대신에, 특수 변수 `$*`을 사용한다. `$*`은 “쉘 스크립트에 모든
명령-라인 매개변수”를 의미한다. 예제가 여기 있다.

```
$ cat sorted.sh

wc -l $* | sort -n

$ 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
```

왜 쉘 스크립트가 어떤 것도 하지 않을까? 스크립트가 파일 꾸러미를 처리하고 했지만, 어떠한 파일 이름도 주지 않는다면 무슨 일이 발생할까요? 예를 들어, 만약 다음과 같이 타이핑한다면 어떨까요?

```
$ bash sorted.sh
```

하지만 *.dat (혹은 다른 어떤 것)을 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 가정해서 단지 앉아서 사용자가 인터랙티브하게 어떤 데이터를 전달 하길 기다린다. 하지만, 밖에서는 사용자에게 보이는 것은 스크립트가 거기 앉아서 정지한 것처럼 보여 스크립트가 아무 일도 수행하지 않는 것처럼 보인다.

간단한 쉘 스크립트로 마치기 전에 두개가 더 있다. 다음과 같은 스크립트를 살펴보면,

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

아마도 스크립트가 무엇을 하는지 생각해 낼 수 있다. 다른 한편으로 다음 스크립트를 살펴보면,

```
# List files sorted by number of lines.  
wc -l $* | sort -n
```

사용자가 생각해 낼 필요가 없다. 상단의 주석이 무엇을 수행하는지 자동으로 말해준다. 한줄 혹은 두줄의 이와 같은 문서화는 앞으로 여러분 자신과 다른 사람이 여러분이 작성한 스크립트나 프로그램을 재사용하기 쉽게 한다. 주의할 점은 매번 스크립트를 변경할 때마다, 주석이 여전히 정확한지를 확인해야만 한다. 독자에게 잘못된 방향을 주는 설명은 아무것도 없는 것보다 더 나쁠 수 있다.

둘째로, 유용한 무언가를 수행하는 일련의 명령어를 방금 실행했다고 가정하자. 예를 들어 스크립트가 논문에 사용될 그래프를 생성했다. 필요하면 나중에 그래프를

다시 생성하고자 해서 파일에 명령어를 저장하고자 한다. 명령문을 다시 타이핑 (그리고 잠재적으로 잘못 타이핑할 수도 있다)하는 대신에 다음과 같이 할 수도 있다.

```
$ history | tail -4 > redo-figure-3.sh
```

redo-figure-3.sh 파일은 이제 다음을 담고 있다.

```
297 goostats -r NENE01729B.txt stats-NENE01729B.txt  
298 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
```

명령어의 일련 번호를 제거하기 위해서 편집기에서 한동안 작업한 후에 그림을 어떻게 생성하는지에 대한 정밀로 정확한 기록을 갖게 된다.

매긴 번호 지우기(Unnumbering) 앞선 명령에 붙은 일련번호를 제거하기 위해서 Nelle은 colrm (“column removal(열 제거)”의 줄임말)를 사용할 수 있다. 매개변수가 입력에서 제거할 문자 범위가 된다.

```
$ history | tail -5  
173 cd /tmp  
174 ls  
175 mkdir bakup  
176 mv bakup backup  
177 history | tail -5  
$ history | tail -5 | colrm 1 7  
cd /tmp  
ls  
mkdir bakup  
mv bakup backup  
history | tail -5  
history | tail -5 | colrm 1 7
```

실제로, 대부분의 사람들은 쉘 프롬프트에서 몇번 명령어를 실행해서 올바들게 수행되는지를 확인한 다음 재사용을 위해서 파일에 저장한다. 이런 유형의 작업은 데이터와 워크플로어에서 발견한 것을 `history`를 호출해서 재사용할 수 있게 하고 출력을 깔끔하게 하기 위해 약간의 편집을 하고 쉘 스크립트로 저장한다.

Nelle의 파이프라인: 스크립트 생성하기

지도교수의 즉석 코멘트는 Nelle의 파일을 처리할 때 `goostats`에 추가 몇개의 매개변수를 주어야 한다는 것을 깨닫게 했다. 수작업으로 모든 분석을 했다면 아마도 재난이었을 것이다. 하지만, 루프덕분에 다시 작업하는데는 몇시간이 소요될 것이다.

만약 무언가가 두번 수행될 필요가 있다면 아마도 세번 혹은 네번도 수행될 필요가 있다는 것을 경험으로 알고 있다. 편집기를 실행하고 다음과 같이 작성한다.

```
# Calculate reduced stats for data files at J = 100 c/bp.  
for datafile in $*  
do  
    echo $datafile  
    goostats -J 100 -r $datafile stats-$datafile  
done
```

(매개변수 `-J 100`과 `-r`은 지도교수가 사용해야 된다고 알려준 것이다.) `do-stats.sh` 이름으로 파일에 저장해서 다음과 같이 타이핑해서 첫번째 단계 분석을 다시 수행할 수 있다.

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

또한 다음과 같아도 할 수 있다.

```
$ bash do-stats.sh *[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.

```

# Calculate reduced stats for A and Site B data files at J = 100 c/bp.
for datafile in *[AB].txt
do
    echo $datafile
    goostats -J 100 -r $datafile stats-$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. 좀 더 모험적이라면, 스크립트를 변경해서 명령라인 매개변수를 검증해서 만약 어떠한 매개변수도 제공되지 않았다면 *[AB].txt 을 사용한다. 물론, 이런 접근법은 유연성과 복잡성 사이에 서로 대립되는 요소 사이의 균형, 즉 트레이드오프(trade-off)를 야기한다.

주요점

- 재사용을 위해서 통상 쉘 스크립트로 불리는 명령어를 파일에 저장한다.
- bash filename은 파일에 저장된 명령어를 실행한다.
- \$* 모든 쉘 스크립트의 명령-라인 매개변수를 말한다.
- \$1, \$2, 등등은 특정 명령-라인 매개변수를 말한다.
- 사용자가 무슨 파일을 처리할지 결정하게 하는 것이 좀더 유연하고 좀더 내장된 유닉스 명령어와 일관성이 있다.

Leah는 수백개의 데이터 파일이 있고, 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

```

임의의 파일이름을 명령-라인 매개변수로 갖는 `species.sh` 이름의 쉘 스크립트를 작성하라. `cut`, `sort`, `uniq`을 사용해서 각각의 파일별로 나오는 유일한 종의 목록을 화면에 출력하세요.

디렉토리 이름과 파일이름 확장자를 매개변수를 갖는 `longest.sh` 이름의 쉘 스크립트를 작성해서 그 디렉토리에 그 확장자를 가지는 파일 중에 가장 긴 줄을 가진 파일이름을 화면에 출력하세요. 예를 들어, 다음은

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

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

다음 명령어를 실행하면,

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

파일의 마지막 명령어는 `history` 명령어 자체다. 즉, 쉘은 `history`에 실질적으로 실행하기 전에 명령어 이력(log)을 추가한다. 사실, 쉘은 *항상** 명령어를 실행하기 전에 이력(log)에 명령어를 추가한다. 그렇게 하는 이유가 무엇이라고 생각합니까?

Joel의 data 디렉토리가 `fructose.dat`, `glucose.dat`, `sucrose.dat` 파일 세개를 담고 있다. 다음행을 담고 있는 스크립트로 `bash example.sh *.dat`로 실행할 때 `example.sh` 이름의 스크립트가 무엇을 수행하는지 설명하세요.

```
# Script 1
echo **

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

# Script 3
echo $*.dat
```

파일, 문자, 디렉토리 등 찾기

목표

- grep 명령어를 사용해서 간단한 패턴과 매칭되는 행을 텍스트 파일에서 선택한다.
- find 명령어를 사용해서 간단한 패턴과 매칭하는 파일을 찾는다.
- 한 명령어의 출력결과를 다른 명령어의 명령-라인 매개변수로 사용한다.
- “텍스트(text)”와 “바이너리(binary)” 의미와 많은 툴이 바이너리를 잘 다루지 못하는 이유를 설명한다.

검색에 대해서 어떻게 말하는지에 따라 사람의 나이를 유추할 수 있다. 젊은 사람은 동사로 “구글(Google)”을 사용하고 무뚝뚝한 나이든 유닉스 프로그래머는 “grep”을 사용한다. grep은 “global/regular expression/print”의 축약어로 초기 유닉스 편집기에 흔한 일련의 연산이다. 매우 유용한 명령-라인 프로그램의 이름이기도 하다.

grep은 패턴과 매칭되는 파일의 행을 찾아 화면에 뿐려준다. 예를 들어, *Salon* 잡지 1988년 경쟁부문에서 3개 하이쿠(haiku, 일본의 전통 단시)를 담고 있는 파일을 사용한다. 이 예제 파일은 “writing” 하위 디렉토리에서 작업을 할 것이다.

```
$ cd  
$ cd 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 Rothenberg](#)가 말했듯이, 디지털 정보는 어느 것이 먼저 오든 영원한 영속성을 가지거나 혹은 5년이다.

단어 “not”을 포함하지 않는 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. 무척이나 간단하다. 모든 글자와 숫자를 쓴 문자가 자신에 대해서 매칭을 한다. 찾고자 하는 이름이나 혹은 파일의 이름이 나온 후에, 출력값은 “not”을 포함하는 파일에 3개 행이 된다.

다른 패턴을 시도해 보자. 이번에는 “day”이다.

```
$ grep day haiku.txt
```

```
Yesterday it worked  
Today it is not working
```

이번에는 출력값은 “day”를 포함한 “Yesterday”, “Today” 단어를 포함하는 행이 된다. grep 명령어에 -w 옵션을 주면, 단어 경계로 매칭을 제한해서, “day” 단어만을 가진 행만이 화면에 출력된다.

```
$ grep -w day haiku.txt
```

이 경우에, 어떤 것도 없어서, grep 출력값은 비게된다.

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 확인한다.

다른 유닉스 명령어와 마찬가지로 옵션을 조합할 수 있다. 예를 들어, -i 옵션은 대소문자 구분없이 매칭하고, -v 옵션은 뒤집어서 매칭을 하여, 두 옵션을 사용하여 대소문자에 관계없이 매칭이 되지 않는 행만을 화면에 출력할 수 있다.

```
$ grep -i -v the haiku.txt
```

```
You bring fresh toner.
```

```
With searching comes loss
```

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

grep 명령어는 옵션이 많다. 옵션에 대해서 알기 위해서 man grep을 타이핑한다. man은 유닉스 “manual(매뉴얼)” 명령어다. 명령어에 대한 기술과 옵션을 출력하고 (만약 운이 좋다면) 어떻게 사용할 수 있는지 예제도 몇개 제공한다.

```
$ man grep
```

GREP(1)

NAME

grep, egrep, fgrep - print lines matching a pattern

SYNOPSIS

```
grep [OPTIONS] PATTERN [FILE...]
grep [OPTIONS] [-e PATTERN | -f FILE] [FILE...]
```

DESCRIPTION

grep searches the named input FILEs (or standard input if no files are named, or if a single minus (-) is given as file name) for lines containing a match to the given PATTERN. By default prints the matching lines.

...

OPTIONS

Generic Program Information

--help Print a usage message briefly summarizing these command-line options and the bug address, then exit.

-V, --version

Print the version number of grep to the standard output stream. This version number should be included in all bug reports (see below).

Matcher Selection

-E, --extended-regexp

Interpret PATTERN as an extended regular expression (ERE, see below). (-E is specified by POSIX.)

-F, --fixed-strings

Interpret PATTERN as a list of fixed strings, separated by newlines, any of which is to be matched. (-F is specified by POSIX.)

...

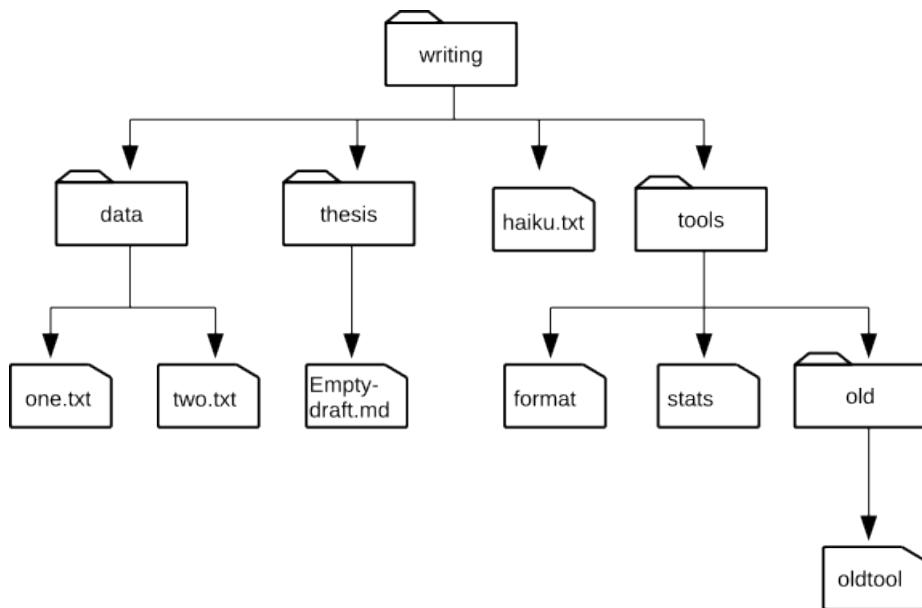
와일드카드(Wildcards) grep의 진정한 힘은 옵션에서 나오지 않고, 패턴이 와일드카드를 포함할 수 있다는데서 온다. (기술적 명칭은 정규 표현식(Regular Expressions)이고, “grep” 내부에 “re”가 축약어로

들어가 있다.) 정규 표현식은 복잡하기도 하고 강력하기도 하다. 복잡한 검색을 하고자 한다면, 웹사이트에서 수업을 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 옵션을 사용해서 인용부호 안에 패턴을 넣어서 쉘이 해석하는 것을 방지한다. (예를 들어, 패턴이 '*'을 포함한다면, grep을 실행하기 전에 쉘이 전개하려 할 것이다.) 패턴에서 '\^'은 행의 시작에 매칭을 고정 한다. ':'은 한 문자만 매칭하고(쉘의 '?'과 마찬가지로), 'o'는 실제 영문 'o'와 매칭한다.

grep이 파일에 행을 찾는 반면에, find 명령어는 파일 자체를 검색한다. 다시, find 명령어는 정말 옵션이 많다. 가장 간단한 것이 어떻게 동작하는지 보여주기 위해서, 다음에 보여지는 디렉토리 구조를 사용한다.



Nelle의 writing 디렉토리는 haiku.txt로 불리는 파일과, 4개의 하위 디렉토리를 포함한다. thesis 디렉토리는 슬프게고 비어있고, data 디렉토리는 one.txt

과 `two.txt`을 포함하고, `tools` 디렉토리는 `format`과 `stats` 프로그램을 포함하고 빈 디렉토리 `old`가 있다.

첫 명령어로, `find . -type d`을 실행하자. 항상 그렇듯이, . 자체가 의미하는 바는 현재 작업 디렉토리로 검색을 시작하는 디렉토리이기도 하다. `-type d`은 “디렉토리인 것들”을 의미한다. 과연, `find`의 출력은 .을 포함하는 상기 작은 디렉토리 나무 구조에서 다섯개의 디렉토리 이름이 된다.

```
$ find . -type d
```

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

`-type d`에서 `-type f`으로 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/two.txt
```

`find` 명령어는 자동적으로 하위 디렉토리, 또 하위 디렉토리의 하위 디렉토리로 가서 주어진 패턴과 매칭되는 모든 것을 찾는다. 이것을 원치 않는다면, `-maxdepth`를 사용해서 검색의 깊이를 제한할 수 있다.

```
$ find . -maxdepth 1 -type f
```

```
./haiku.txt
```

`-maxdepth`의 반대는 `-mindepth`로, `find`에게 단지 일정 깊이 혹은 아래 깊이만 검색해서 출력하게 한다. 그러므로, `-mindepth 2`는 2 혹은 그 이상의 수준 이하에서 모든 파일을 찾게된다.

```
$ find . -mindepth 2 -type f  
  
./data/one.txt  
./data/two.txt  
./tools/format  
./tools/stats
```

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 한 것을 수행하자. 단일 인용부호에 `*.txt`을 넣어서 쉘이 와일드카드 `*`을 전개하지 못하게 한다. 이런 방식으로 `find` 명령어는 실질적으로 `*.txt` 패턴을 얻고 `haiku.txt` 파일 이름으로 전개하지 않는다.

```
$ find . -name '*.*.txt'  
  
.data/one.txt  
.data/two.txt  
.haiku.txt
```

리스트(list) vs. 찾기(find) 올바른 옵션이 주어진 상태에서 비슷한 작업을 수행하도록 `ls`와 `find` 명령어를 만들 수 있다. 하지만, 정상 상태에서 `ls`는 가능한 모든 것을 목록으로 출력하는 반면에, `find`는 어떤 특성을 가진 것을 검색하고 보여준다는 점에서 차이가 있다.

앞에서 언급했듯이, 명령-라인(command-line)의 힘은 툴을 조합하는데 있다. 파일로 어떻게 조합하는지를 살펴봤고, 또 다른 기술을 살펴보자. 방금 보았듯이, `find . -name '*.txt'` 명령어는 현재 디렉토리 및 하위 디렉토리의 모든 텍스트 파일 목록을 보여준다. 어떻게 하면 `wc -l` 명령어와 조합해서 모든 파일의 행을 카운트할 수 있을까?

가장 간단한 방법은 `$()` 내부에 `find` 명령어를 위치하는 것이다.

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

```
11 ./haiku.txt  
300 ./data/two.txt  
70 ./data/one.txt  
381 total
```

쉘이 상기 명령어를 실행할 때, 처음 수행하는 것은 `$()` 내부를 실행하는 것이다. 그리고 나서 `$()` 표현식을 명령어의 출력 결과로 대체한다. `find`의 출력 결과가 3개의 파일 이름, 즉, `./data/one.txt`, `./data/two.txt`, `./haiku.txt`이여서, 쉘은 다음과 같이 명령문을 구성하게 된다.

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

상기 명령문이 사용자가 원하는 것이다. 이 표현식이 *과 ? 같은 와일드카드를 전개할 때 정확하게 쉘이 수행하는 것이다. 하지만 자신의 “와일드카드”로 사용자가 원하는 임의의 명령어를 사용하자.

`find`와 `grep`을 함께 사용하는 것은 일반적이다. `find`가 패턴을 매칭하는 파일을 찾고 `grep`이 또 다른 패턴과 매칭하는 파일 내부의 행을 찾는다. 예제로 다음에 현재의 부모 디렉토리에서 모든 `.pdb` 파일에 “FE” 문자열을 검색해서 철 원자를 포함하는 PDB파일을 찾을 수 있다.

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

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

바이너리 파일(Binary File) 텍스트 파일에 있는 것을 찾는데만 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 일어나지 않았고, 아마도 그러지 않을 것이다. 왜냐하면 지원할 너무나 많은 형식이 존재하기 때문이다.

두번째 선택사항은 데이터를 텍스트로 변환하거나 데이터에서 텍스트 같은 비트를 추출하는 것이다. 아마도 가장 흔한 접근 방법으로 정보를 추출하기 위해서 데이터 형식마다 하나의 툴만 개발하면 되기 때문이다. 한편으로, 이 접근법은 간단한 것을 쉽게 할 수 있게 한다. 부정적인 면으로, 복잡한 것은 일반적으로 불가능하다. 예를 들어, `grep`을 이용해 이미지 파일에서 X와 Y 크기를 추출하는 프로그램을 작성하기는 쉽다. 하지만, 공식을 담고 있는 엑셀 같은 스프레드시트 셀에서 값을 찾아내는 것을 어떻게 작성할까?

세번째 선택사항은 쉘과 텍스트 처리가 모두 한계를 가지고 있다는 것을 인지하고 대신에 파이썬 같은 프로그램 언어를 사용하는 것이다. 이러한 시점이 왔을 때 쉘에서 너무 고생하지 마세요. 파이썬을 포함한 많은 프로그래밍 언어가 많은 아이디어를 여기에서 가져왔다. 모방은 또한 칭찬의 가장 충심어린 형태이기도 하다.

결론

유닉스 쉘은 지금 사용하는 대부분의 사람보다 나이가 많다. 그토록 오랫동안 생존한 이유는 지금까지 만들어진 가장 생산성이 높은 프로그래밍 환경 중 하나 혹은 가장 생산성은 높기 환경이기 때문이다. 구문이 암호스러울 수도 있지만, 숙달한 사람은 다른 명령어를 대화하듯이 실험할 수 있고 나서 자신의 작업을 자동화하기 위해서 학습한 것을 사용한다. 그래피 사용자 인터페이스(GUI)가 처음에는 더 좋을 수 있지만, 여전히 두번째는 쉘이 최강이다. 화이트헤드(Alfred North Whitehead) 박사가 1911년 썼듯이 “문명은 생각없이 수행할 수 있는 중요한 작업의 수를 확장

함으써 발전한다.(Civilization advances by extending the number of important operations which we can perform without thinking about them.”)

주요점

- `find` 명령어를 사용해서 파일과 디렉토리를 찾고, `grep`을 사용해서 파일에 텍스트 패턴을 찾으세요.
- `$(command)` 명령어의 출력 결과를 우선 삽입한다.
- `man command` 주어진 명령어에 대해서 매뉴얼 페이지를 화면에 출력한다.*

다음 쉘 스크립트에 대해서 무슨 것을 수행하는지 짧은 설명문을 작성하세요.

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

`grep` 명령어의 `-v` 옵션은 패턴 매칭을 반전해서 패턴과 매칭하지 않는 행만 출력된다. 다음 명령어 중에서 어느 것이 `ose.dat`로 끝나는 (예로, `sucrose.dat` 혹은 `maltose.dat`), 하지만 `temp` 단어는 포함하지 않는 `/data` 디렉토리에 있는 모든 파일을 찾아낼까요?

1. `find /data -name '*.dat' | grep ose | grep -v temp`
2. `find /data -name ose.dat | grep -v temp`
3. `grep -v temp $(find /data -name '*ose.dat')`
4. 위 어떤 것도 아님.

깃을 사용한 버전 관리(Git)

버전관리 (Version Control)은 디저털 세상에서 연구실 수첩이다. 버전관리는 전문가가 자신이 한것과 다른 사람과 협업한 것을 기록하고 관리하기 위해서 사용하는 것이다. 모든 대형 소프트웨어 개발 프로젝트는 버전관리에 의존하며, 대부분의 프로그래머는 작은 일에도 또한 사용한다. 단지 소프트웨어만 그런 것이 아니다. 책 (본 프로젝트도 여기에 포함된다.), 논문, 작은 데이터세, 시간에 따라 변하고 공유될 필요가 있는 어느 것이나 버전관리 시스템에 저장될 수 있고 되어야 한다.

버전 관리 소개

Universal Missions 회사는 (Euphoric State University에서 분사한 항공 서비스 전문회사) 늑대인가(Wolfman)과 드라큘라(Dracula)를 고용해서 다음 행성 착륙선을 화성에 보낼 수 있는지를 조사하려고 한다. 늑대인간과 드라큘라 모두 동시에 계획에 대해서 작업을 할 수 있길 희망하지만, 과거에도 동일한 문제에 봉착한 경험이 있다. 순서대로 작업을 한다면, 각자는 서로가 작업이 끝날 때까지 한참을 기다려야 한다. 하지만, 자신만의 작업본을 가지고 변경사항을 가지고 주고 받고 한다면, 작업이 중간에 망설 될 수도 있고, 덮어쓰기도 될 수 있고, 중복될 수도 있다.

작업을 관리하기 위해서 적합한 해결책은 버전 관리(version control)를 사용하는 것이다. 버전관리가 전자우편을 주고 받는 것보다 나은데 이유는 다음과 같다.

- 버전 관리에 커밋(commit)된 어떠한 것도 잊어버지 않는다. 이것이 의미하는 바는 편집기의 원상태로 되돌리(undo) 같은 기능이다. 모든 옛 파일의 버전이 저장되어 있다. 그래서 정확하게 누가 무엇을 특정한 날에 작성했는지를 보거나 특정한 결과를 생성하기 위해서 무슨 버전의 프로그램을 사용했는지를 항상 살펴볼 수 있다.
- 버전 관리는 누가 무슨 변경을 언제 했는지를 기록한다. 그래서 나중에 다른 사람이 궁금한 점이 있다면, 누구에게 질문을 해야하는지 알 수 있다.
- 다른 사람이 변경한 사항을 간과해서 무심히 넘어가거나, 덮어쓰기고 지나가기가 불가능하지는 않지만, 어렵다. 버전 관리 시스템이 자신이 작업과 다른 사람의 작업에 충돌(conflict)이 발생할 때마다 자동적으로 통지를 준다.

도전 과제 위키피디아에 모든 변경사항과 저자 정보는 추적되고 기록된다. 여기에 가서 화성에 대한 글의 모든 변경이력을 확인할 수 있다. 일단 달 마지막 편집된 것을 찾아서 “prev” 링크를 클릭해서 변경사항을 확인한다.

Git라는 대중적인 공개 소프트웨어 버전 관리 시스템을 어떻게 사용하는지 수업 한다. Git는 다른 버전 관리 소프트웨어보다 좀더 복잡하지만, 폭넓게 사용되는데 이유는 시작하기 쉽고 GitHub라는 호스팅 사이트 때문이다. 어떠한 버전관리 시스템을 사용하든지, 학습에서 중요하는 점은 이해하기 어려운 명령어의 자세한 점보다는 각 버전 관리 시스템이 주안점을 두고 있는 워크플로우(workflow)다.

더 훌륭한 백업 방법

목표

- 컴퓨터마다 요구되는 초기화 및 설정단계, 그리고 저장소마다 필요한 것이 무엇인지 설명한다.
- 파일 한개 혹은 여러개의 파일에 대해서 변경-추가-커밋(modify-add-commit) 주기를 수행하고, 각 단계별로 정보가 어디에 저장되는지를 설명한다.
- Git 변경 번호를 확인하고 사용한다.
- 파일을 동일한 옛 버전 파일과 비교한다.
- 옛 버전 파일을 복원한다.
- Git 설정을 변경하여 특정 파일을 버전관리에서 제외하고 왜 제외하는 것이 때때로 유용한지 설명한다.

누군가가 무엇을 했는지, 언제 했는지를 기억하기 위해서 버전 관리를 어떻게 사용할 수 있는지 탐험을 시작할 것이다. 다른 사람과 협업을 하지 않더라도, 버전관리는 훨씬 더 낫다.

“Piled Higher and Deeper” by Jorge Cham, <http://www.phdcomics.com>

설정

처음 Git를 새로운 컴퓨터에 사용할 때, 몇 가지 설정이 필요하다. 다음은 Dracula 가 새로 구입한 노트북에 어떻게 설정하는지 보여준다.

```
$ git config --global user.name "Vlad Dracula"  
$ git config --global user.email "vlad@tran.sylvan.ia"  
$ git config --global color.ui "auto"  
$ git config --global core.editor "nano"
```

(드라큘라 정보 대신에 자신의 이름과 전자우편 주소를 사용하세요. 윈도우에서는 notepad 같은 실제 컴퓨터에서 작동하는 편집기를 선택했는지 확인하세요.)

Git 명령어는 `git verb` 형식으로 작성되고 `verb` 가 실제로 수행하고자 하는 명령이다. 상기의 경우 Git에게 다음을 명령한다.

- 자신의 이름과 전자우편 주소
- 출력 결과를 색깔을 넣어 표현한다.
- 주 사용 텍스트 편집기
- 설정 사항을 전역으로 한다. (즉, 모든 프로젝트에 적용)

상기 4개 명령어는 한번만 실행되면 된다. `--global` 플래그는 Git에게 해당 컴퓨터에서 실행되는 모든 프로젝트에 적용되도록 설정한다.

프록시(Proxy) 몇몇 네트워크에서 프록시를 사용할 필요가 있다. 만약 이런 경우라면, 프록시에 대해서 Git 설정을 다음과 같이 변경한다.

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

프록시를 사용하지 않기 위해서 다음을 사용한다.

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

저장소 생성하기

Git 설정이 완료되면, Git를 사용할 수 있다. 작업을 위해서 디렉토리를 생성하자.

```
$ mkdir planets  
$ cd planets
```

생성된 디렉토리를 Git 저장소(repository)로 만든다. 저장소는 Git가 파일의 옛 버전을 저장하는 장소다.

```
$ git init
```

`ls`를 사용해서 디렉토리의 내용을 살펴보면, 아무것도 변경된 것이 없는 것처럼 보인다.

```
$ ls
```

하지만, `-a` 플래그를 추가해서 모든 것이 보이도록 한다면, Git가 `.git`로 불리는 숨겨진 디렉토리를 생성한 것을 볼 수 있다.

```
$ ls -a
```

```
. .. .git
```

Git는 이 특별한 하위 디렉토리에 프로젝트에 대한 정보를 저장한다. 만약 `.git`를 삭제한다면, 프로젝트 이력을 모두 잃어버린다.

모든 것이 제대로 설정되었는지를 확인을 다음과 같이 프로젝트 상태를 Git 명령어로 수행한다.

```
$ git status
```

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

파일 변경사항 추적하기

전진기지로서 화성의 적합성에 관한 기록을 담고 있는 `mars.txt` 파일을 생성한다. (파일 편집을 위해서 `nano` 편집기를 사용한다. 원하면 무슨 편집기를 사용해도 된다. 특히, 앞에서 전역으로 설정한 `core.editor`가 될 필요도 없다.)

```
$ nano mars.txt
```

`mars.txt` 파일에 다음 텍스트를 타이핑한다.

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

`mars.txt` 파일은 이제 한 줄을 포함하고 있다.

```
$ ls
```

```
mars.txt
```

```
$ cat mars.txt
```

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

다시 한번 프로젝트의 상태를 확인하고자 하면, Git가 새로운 파일을 하나를 알아차렸다고 말하고 있다.

```
$ 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 add`를 사용해서 Git에게 추적관리하라고 작업명령을 낼 수 있다.

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

그리고 나서 올바르게 되었는지 확인한다.

```

$ git status

# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   mars.txt
#

```

Git는 `mars.txt` 파일을 추적할 것이라는 것을 알고 있지만, 아직 저장소에는 어떤 변경사항도 기록하지는 않았다. 이를 위해서 명령어 하나를 더 실행할 필요가 있다.

```

$ git commit -m "Starting to think about Mars"

[master (root-commit) f22b25e] Starting to think about Mars
 1 file changed, 1 insertion(+)
 create mode 100644 mars.txt

```

`git commit`을 실행할 때, Git는 `git add`를 사용해서 저장하려고 하는 모든 대상을 받아서 `.git` 디렉토리 내부에 영구 사본으로 저장한다. 이 영구 사본을 수정(revision)이라고 하고, 짧은 식별자는 `f22b25e`이다. (여러분의 수정번호의 짧은 확장자는 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, Git는 `nano`(혹은 시초에 설정한 다른 편집기)를 실행해서 좀더 긴 메시지를 작성할 수 있다.

아제 `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 보내다.

```
$ git log

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

        Starting to think about Mars
```

`git log`는 역 시간순으로 저장소의 모든 변경사항을 나열한다. 각 수정사항 목록은 전체 수정 식별자(앞서 `git commit` 명령어로 출력한 짧은 문자와 동일하게 시작), 수정한 사람, 언제 생성되었는지, 생성할 때 Git에 작성한 로그 메시지를 포함한다.

내 변경사항은 어디로 갔을까? 이 시점에서 `ls` 명령어로 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.)

파일 변경하기

이제 드라큘라가 파일에 정보를 좀더 추가했다고 가정하자. (다시 한번 `nano` 편집기로 편집하고 나서 `cat`으로 파일 내용을 살펴본다. 다른 편집기를 사용할 수도 있고, `cat`으로 파일 내용을 반드시 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 status`를 실행하면, 이미 알고 있는 파일이 변경되었다고 알려준다.

```
$ 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 사항(`git add`로 수행)을 저장한다고 말하지는 않았다. `git diff`를 사용해서 작업 내용을 두번 검증한다. `git diff`는 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 아리송한데 실제로 다른 파일이 주어졌을 때 파일 하나를 어떻게 재구성하는지를 말해주는 `patch`와 편집기 같은 도구를 위한 일련의 명령이기 때문이다.

1. 첫번째 행은 Git이 신규와 옛 버전 파일을 비교하는 유닉스 `diff` 명령어와 유사한 출력결과를 생성하게 한다.
2. 두번째 행은 정확하게 Git 파일 어느 수정(revisions)본을 비교하는지 알려준다. `df0654a`과 `315bf3a`은 수정을 위한 목적으로 중복되지 않게 컴퓨터가 생성한 표식이다.

- 나머지 행은 실제 차이가 나는 것과 어디 행에서 발생했는지 보여준다. 특히 첫번째 열의 + 기호는 어디서 행이 추가 되었는지 보여준다.

변경사항을 커밋(commit)하자.

```
$ git commit -m "Concerns about Mars's moons on my furry friend"

# 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")
```

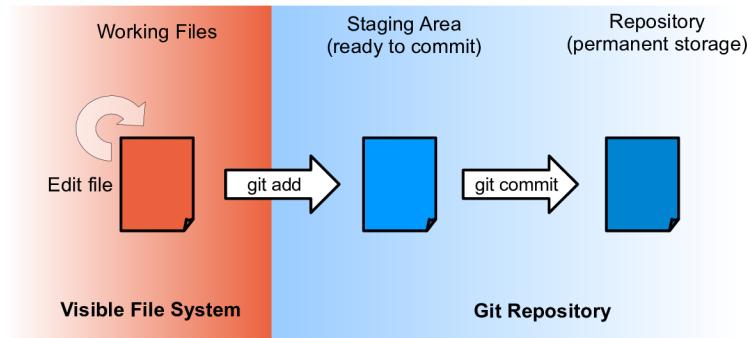
이럴 수가, git add을 먼저 하지 않아서 Git가 커밋을 할 수 없다. 고쳐봅시다.

```
$ git add mars.txt
$ git commit -m "Concerns about Mars's moons on my furry friend"

[master 34961b1] Concerns about Mars's moons on my furry friend
 1 file changed, 1 insertion(+)
```

실제로 무엇을 커밋하기 전에 커밋하고자하는 파일을 먼저 추가하라고 Git이 주장하는데, 왜냐하면 한번에 모든것을 커밋하지 싶지 않을지도 모르기 때문이다. 예를 들어, 작성하고 있는 논문에 지도교수 작업 몇몇 인용을 추가한다고 가정하자. 논문 중간에 인용되는 추가부분과 상응하는 참고 문헌 부분을 커밋하고는 싶지만 결론 부분을 커밋하고는 싶지 않다. (아직 결론은 완성되지 않았다.)

이런 부분을 고려해서, Git은 특별한 준비 영역(staging)이 있어서 현재 변경부분 (change set)을 추가는 했으나 아직 커밋하지 않는 것을 준비 영역에 기억하고 있다. git add를 하면 이 영역에 작업 중 해당하는 것을 놓고 git commit이 커밋으로 장기 저장소로 복사한다.



편집기의 파일을 준비 영역으로 그리고, 장기 저장소로 옮기는 것을 살펴보자. 먼저 파일에 또 하나의 행을 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. Git말할 수 있는 것은 영구히 저장되는 것과 현재 디렉토리에 작업하고 있는 것과 차이가 없다. 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 "Thoughts about the climate"

[master 005937f] Thoughts about the climate
 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

Thoughts about the climate

commit 34961b159c27df3b475cfe4415d94a6d1fcd064d

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

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

Concerns about Mars's moons on my furry friend

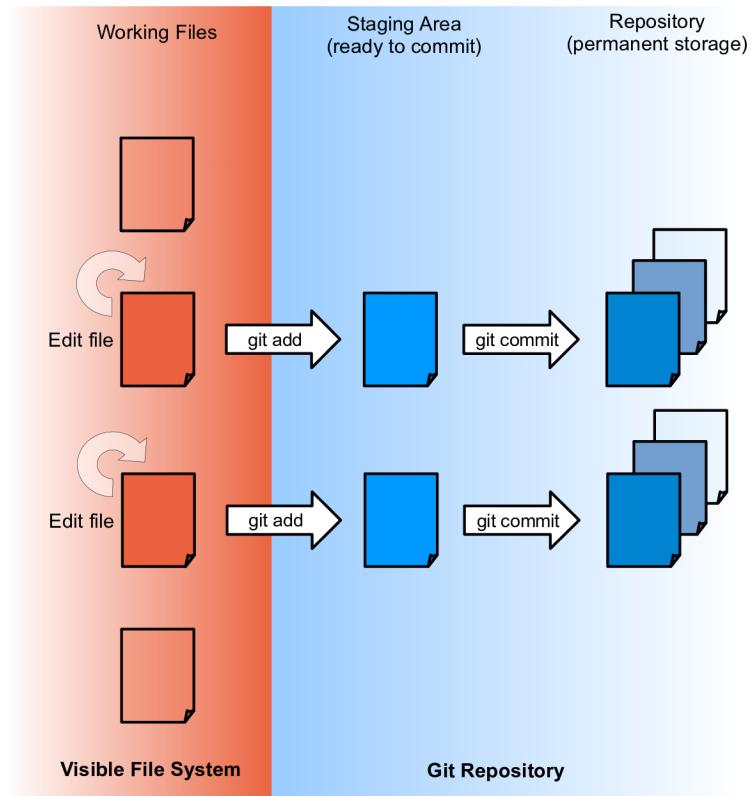
commit f22b25e3233b4645dabd0d81e651fe074bd8e73b

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

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

Starting to think about Mars

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



이력 탐험하기.

언제 변경한 것을 변경했는지를 알기 위해서는 `git diff`를 다시 사용한다. 하지만 오래된 버전은 `HEAD~1`, `HEAD~2`, 등등 표기를 사용하여 참조한다.

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

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
```

```
$ 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 +1,3 @@  
 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
```

이런 방식으로, 연쇄 수정 사슬을 만들 수 있다. 가장 최근 사슬의 끝값은 HEAD로 참조된다. ~ 표기법을 사용하여 앞선 수정을 참조할 수 있다. 그래서 HEAD~1(“head” 빼기 1로 읽는다.)은 “바로 앞선 수정”을 의미하고, HEAD~123은 지금 있는 위치에서 123번째 이전 수정으로 간다는 의미다.

수정된 것을 git log 명령어가 화면에 뿐려주는 숫자와 문자로 구성된 긴 문자열을 사용하여 참조할 수도 있다. 변경사항에 대한 중복되지 않는 ID로 “중복되지 않는(unique)”의 의미는 정말 유일하다는 의미다. 특정 컴퓨터에 있는 임의의 파일 집합에 모든 변경사항은 중복되지 않는 40 문자 식별자가 붙어있다. 첫번째 커밋은 ID로 f22b25e3233b4645dabd0d81e651fe074bd8e73b 이 주어졌다. 그래서 다음과 같이 시도하자.

```
$ git diff f22b25e3233b4645dabd0d81e651fe074bd8e73b mars.txt  
  
diff --git a/mars.txt b/mars.txt  
index df0654a..b36abfd 100644  
--- a/mars.txt  
+++ b/mars.txt  
@@ -1 +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
```

올바든 정답이지만, 난수 40 문자로 된 문자열을 타이핑하는 것은 매우 귀찮은 일이다. 그래서 Git 앞의 몇개 문자만으로 사용할 수 있게 했다.

```
$ git diff f22b25e mars.txt

diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +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
```

옛 버전 복구하기

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

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

We will need to manufacture our own oxygen
```

이제 `git status`를 통해서 파일이 변경되었다고 하지만, 변경사항은 아직 준비영역(Staging)에 옮겨지지 않은 것을 확인한다.

```
$ 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.

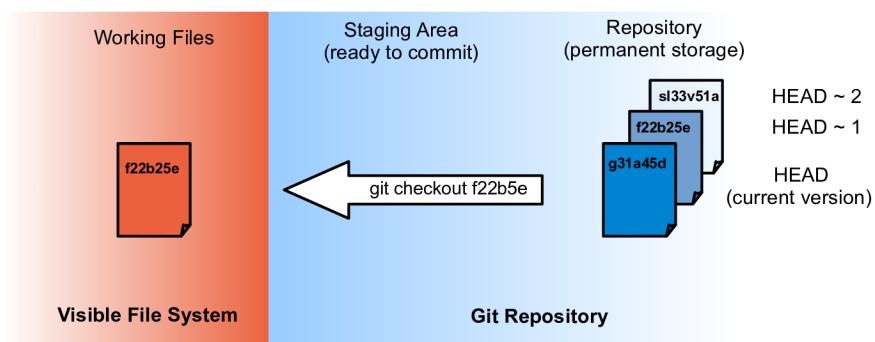
```
$ git checkout HEAD mars.txt  
$ cat mars.txt
```

```
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
```

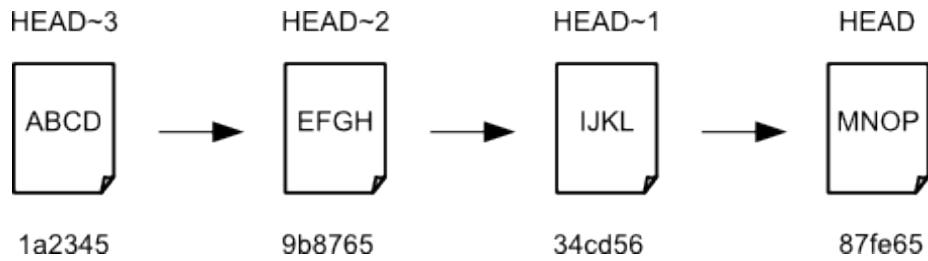
이름에서 유추할 수 있듯이, git checkout 명령어는 파일의 옛 버전을 확인한다. 즉, 되살린다. 이 경우 HEAD에 기록된 가장 최근에 저장된 파일 버전을 되살린다. 좀더 오래된 버전을 되살리고자 한다면, 대신에 수정 식별자를 사용한다.

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

실행 취소를 하는 변경을 하기 *전에** 저장소 상태를 확인하는 수정 번호를 사용해야 한다는 것을 기억하는 것은 중요하다. 흔한 실수는 제거하려는 변경하려고 사용한 커밋 수정 번호를 사용하는 것이다. 아래 예제에서는 수정 번호가 f22b25e 인 가장 최신 커밋(HEAD~1) 앞의 상태로 다시 돌아가고자 한다.



다음 도표는 파일 이력이 어떻게 보이는지 예시로 보여준다. (가장 최근에 커밋된 버전 HEAD에서 거슬러 올라간다.)



흔한 경우를 간단히 `git status` 출력 결과를 주의 깊이 읽게 되면, 힌트를 포함한 것을 알게 된다.

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

출력 결과가 말해주듯이, 버전 식별자 없는 `git checkout` 명령어가 HEAD에 저장된 상태로 파일을 되돌린다. 이중 대쉬 --가 명령어와 구분하기 위해서 파일 이름과 되살리려는 파일 이름을 구별하는데 필요하다. 이중 대쉬가 없다면, Git는 파일 이름을 수정 식별자로 사용하려고 한다.

파일이 하나씩 하나씩 옛 상태로 돌아간다는 사실이 사람들이 작업을 조직하는 방식에 변화를 주는 경향이 있다. 모든 것이 하나의 큰 문서로 되어있다면, 결론 부분에 후에 만든 변경없이 소개부분에 변경을 다시 되돌리기가 어렵다.(하지만 불가능하지는 않다.) 다른 한편으로 만약 소개부분과 결론부분이 다른 파일에 저장되어 있다면, 시간 앞뒤로 이동하기가 훨씬 쉽다.

파일 무시하기

만약 Git가 추적하기 원하지 않는 파일이 있다면 어떨까요? 편집기에서 자동 생성되는 백업파일 혹은 자료 분석 중에 생성되는 중간물 파일을 예가 된다. 몇개 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)
```

벽면 관리아래 파일을 놓는 것은 디스크 공간 낭비다. 더 좋은 않는 것은 파일을 모두 목록에 넣는 것은 실질적으로 중요한 변경사항을 놓치게 할 수도 있다. 그래서 Git에게 중요하지 않는 파일을 무시하게 하자.

.gitignore로 불리는 파일을 프로젝트 루트 디렉토리에 생성해서 무시할 것을 명기해서 수행한다.

```
$ nano .gitignore
$ cat .gitignore

*.dat
results/
```

상기 패턴은 .dat 확장자를 가지는 임의의 파일과 results 디렉토리에 있는 모든 것을 무시한다. (하지만, 이들 파일 중의 일부가 이미 추적되고 있다면, Git는 계속 추적한다.)

.gitignore 파일을 생성하자마자 git status 출력결과는 훨씬 깨끗하다.

```
$ 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` 파일이다. 우리는 이들 파일을 추적하여 관리하지 않는다고 생각할 수도 있지만, 우리와 저장소를 공유하고 있는 모든 사람은 우리가 추적관리하지 않는 동일한 것을 추적관리하고 싶지 않을 것이다. `.gitignore` 를 추가해서 커밋하자.

```
$ git add .gitignore  
$ git commit -m "Add the ignore file"  
$ git status  
  
# On branch master  
nothing to commit, working directory clean
```

보너스로, `.gitignore`는 실수로 원하지 않는 파일을 저장소에 추가하는 것을 피하게 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.  
fatal: no files added
```

만약 `.gitignore` 설정을 우선하여 파일을 추가하려면, `git add -f`를 사용해서 강제로 Git에 파일을 추가할 수 있다. 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

```

주요점

- 컴퓨터마다 사용자 이름, 전자우편, 편집기 그리고 다른 환경설정 사항을 git config로 지정한다.
- git init 명령어는 저장소를 초기화한다.
- git status 명령어는 저장소 상태를 보여준다.
- 파일은 사용자가 볼 수 있는 프로젝트 작업 디렉토리, 다음번 커밋이 구축되는 준비 영역(Staging), 그리고 스냅샷(snapshot)이 항상 기록되는 로컬 저장소에 저장될 수 있다.
- git add 명령어는 파일을 준비 영역에 놓는다.
- git commit 명령어는 준비 영역의 스냅샷을 로컬 저장소에 생성한다.
- 변경사항을 커밋할 때마다 항상 로그 메시지를 작성하라.
- git diff 명령어는 수정사항 사이의 차이를 화면에 출력한다.
- git checkout 명령어는 옛 버전 파일을 복구한다.
- .gitignore 파일은 Git가 무시하여 기록관리하지 않는 파일 정보를 담고 있다.

컴퓨터에 bio라는 신규 Git 저장소를 생성하세요. me.txt 파일에 3줄 자신의 약력을 작성하고 변경사항을 커밋하세요. 그리고 나서, 한줄을 변경하고 4번째 행을 추가해서 변경상태와 최초 상태의 차이를 화면에 출력하세요.

다음 일련의 명령어는 Git 저장소 내부에 또다른 Git 저장소를 생성하는 것이다.

```
cd          # return to home directory
mkdir alpha # make a new directory alpha
cd alpha   # go into alpha
git init    # make the alpha directory a Git repository
mkdir beta  # make a sub-directory alpha/beta
cd beta    # go into alpha/beta
git init    # make the beta sub-directory a Git repository
```

이렇게 하는 것이 왜 좋은 생각이 아닐까요?

협업(Collaboration)

목표

- 원격 저장소가 무엇인지 그리고 왜 원격 저장소가 유용한지 설명한다.
- 원격 저장소가 복제(clone)되었을 때 무엇이 발생하는지 설명한다.
- 원격 저장소에서 변경사항이 푸쉬(push)와 풀(pull)되었을 때 무엇이 발생하는지 설명한다.

버전 제어(version control)는 다른 사람과 협업할 때 진정 찾아온다. 우리는 이미 버전 제어를 위해서 필요한 컴퓨터의 거의 모든 것을 가졌다. 한가지 빠진 것은 한 저장소에서 다른 저장소로 변경사항을 복사하는 것이다.

Git 같은 시스템은 임의의 두 저장소 사이에 작업을 옮길 수 있게 한다. 하지만, 실무에서는 중앙 허브에 하나의 원본을 두고 다른 사람의 노트북이나 PC보다는 웹에 두고 사용하는 것이 가장 쉽다. 대부분의 프로그래머는 프로그램 마스터 원본을 [GitHub](#) 혹은 [BitBucket](#) 같은 호스팅 서비스에 두고 사용한다. 이번 학습의 마지막 부분에서 이러한 접근법의 장점과 단점을 살펴본다.

세상 사람들과 현재 프로젝트에서 변경한 사항을 공유하는 것에서 시작하자. GitHub에 로그인하고 우측 상단의 아이콘을 클릭해서 `planets`으로 신규 저장소를 생성하자.

The screenshot shows the GitHub homepage. At the top, there is a navigation bar with links for 'Explore', 'Gist', 'Blog', and 'Help'. On the right side of the header, there is a user profile for 'gvwilson' with a yellow diamond icon, followed by options to 'Create a new repo', 'Edit Your Profile', and other account settings. Below the header, there are two main sections: 'Popular repositories' and 'Repositories contributed to'. The 'Popular repositories' section lists 'sigcse2014-ipython-workshop' (2 stars) and 'swcarpentry/website' (31 stars). The 'Repositories contributed to' section lists 'swcarpentry/website' (31 stars). At the bottom of the page, there is a large button labeled 'Create a new repo'.

저장소 이름을 “planets”으로 만들고 “Create Repository”를 클릭한다.

The screenshot shows the 'Create a new repo' form. It has fields for 'Owner' (set to 'gvwilson') and 'Repository name' (set to 'planets'). Below these fields, a note says 'Great repository names are short and memorable. Need inspiration? How about massive-adventure.' There is a 'Description (optional)' field with a placeholder text area. Under 'Visibility', the 'Public' option is selected, with a note: 'Anyone can see this repository. You choose who can commit.' The 'Private' option is also available. Below visibility, there is a checkbox for 'Initialize this repository with a README', which is unchecked. A note next to it says 'This will allow you to git clone the repository immediately.' At the bottom of the form, there are buttons for 'Add .gitignore: None' and 'Add a license: None', and a large green 'Create repository' button.

저장소가 생성되자 마자, Git는 URL을 가진 페이지와 사용자의 로컬 저장소 환경 설정을 어떻게 하는지 정보를 화면에 출력한다.

PUBLIC  gwwilson / planets

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

 Set up in Desktop or   git@github.com:gwwilson/planets.git 

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

Create a new repository on the command line

```
touch README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:gwwilson/planets.git
git push -u origin master
```

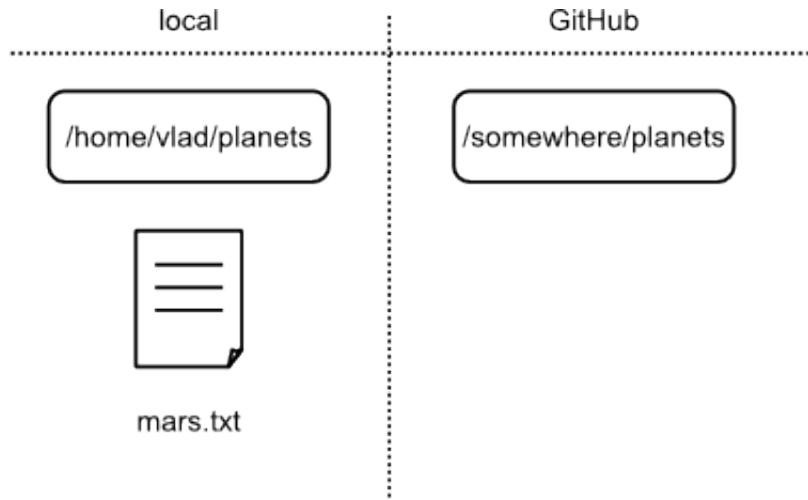
Push an existing repository from the command line

```
git remote add origin git@github.com:gwwilson/planets.git
git push -u origin master
```

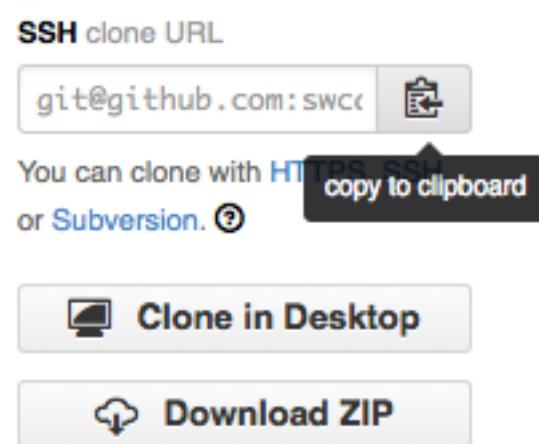
다음 명령어는 효과적으로 GitHub 서버에 다음을 수행한다.

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

현재 로컬 저장소는 여전히 `mars.txt` 파일의 초기 작업을 담고 있다. 하지만, GitHub의 원격 저장소는 아직 어떠한 파일도 담고 있지는 않다.



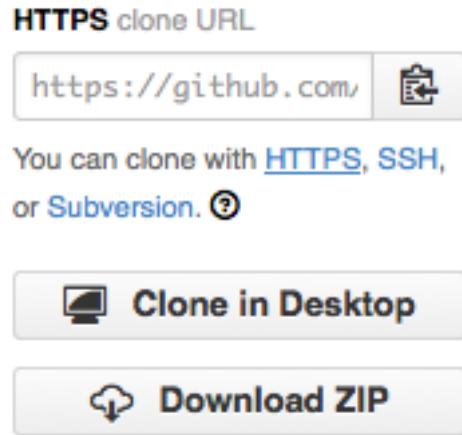
다음 단계는 두 저장소를 연결하는 것이다. 로컬 저장소에 GitHub 저장소를 원격(remote)으로 만들어서 두 저장소를 연결한다. GitHub에 저장소 홈페이지는 문자열을 포함하고 있어서 식별할 수 있다.



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 좀 더 안전하다. [GitHub](#), [Atlassian/BitBucket](#) 그리고 [GitLab](#) (this one has a screencast)에 훌륭한 지도서 중 하나를 따라하는 것도 좋다.

SSH에 대해서 좀 더 알고자 하면 작은 학습에 여러분을 초대한다.



웹 브라우저에서 URL을 복사하고 planets 저장소로 가서 다음 명령어를 실행 한다.

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

Vlad가 아니고 여러분의 저장소의 URL을 사용했는지 확인한다. 유일한 차이점은 vlad 대신에 여러분의 사용자아이름(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` 이름은 원격 저장소의 로컬 별명이다. 원한다면 다른 이름을 사용할 수도 있지만, `origin` 이름은 가장 일반적인 선택이다.

별명이 `origin`으로 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, 오류 메시지로 “Could not resolve hostname”으로 마지막 명령어가 실패할 가능성이 있다. 이것을 해결하기 위해서 프록시에 대한 사항을 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 전달한다.

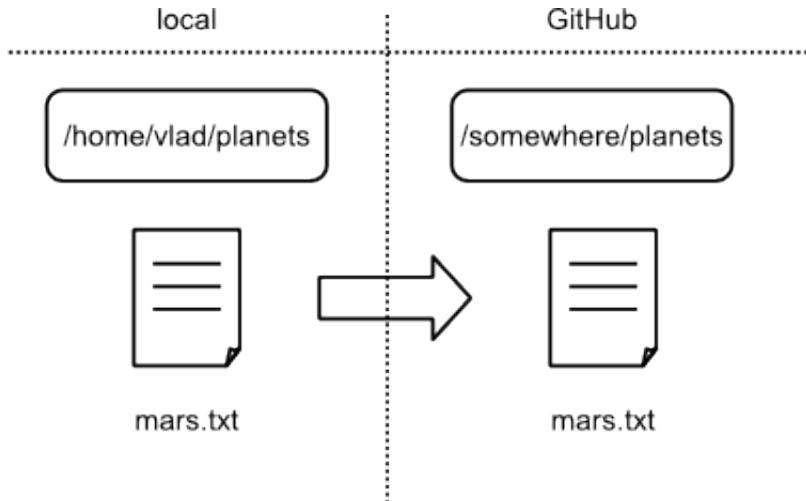
```
$ git config --global --unset http.proxy  
$ git config --global --unset https.proxy
```

비밀번호 관리자(password manager) 운영체제가 비밀번호 관리자로 설정이 되어있다면, 사용자이름(username)과 비밀번호(password)가 필요할 때, git push 명령어는 사용하려고 한다. 비밀번호 관리자를 사용하는 대신에 터미널에서 사용자이름과 비밀번호를 입력하려고 한다면, 다음과 같이 타이핑한다.

```
$ unset SSH_ASKPASS
```

디폴트로 설정하기 위해서는 ~/.bashrc 끝에 상기 명령어를 추가하면 된다.

이제 로컬 저장소와 원격 저장소는 다음과 같은 상태가 된다.



‘-u’ 플래그(flag) Git 문서에 git push과 함께 사용되는 -u 옵션을 볼 수 있다. 중급 학습에서 다루는 개념과 연관되어서 여기서는 넘어가도록 한다.

또한, 원격 저장소에서 로컬 저장소를 변경사항을 풀(pull)해서 가져올 수도 있다.

```
$ git pull origin master
```

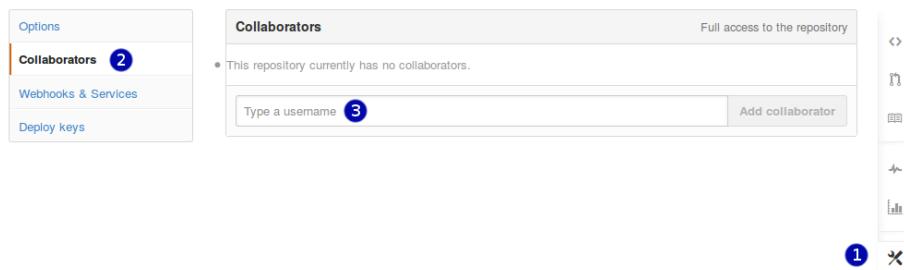
```
From https://github.com/vlad/planets
 * branch            master      -> FETCH_HEAD
 Already up-to-date.
```

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

다음 단계로, 함께 작업하는 것을 살펴보자. 협업을 위해서 사용할 GitHub의 저장소 하나를 고른다.

혼자 훈련하기 스스로 학습을 해왔다면, 두번째 터미널 윈도우를 열고 다른 디렉토리(예를 들어, `/tmp`)로 바꾸어 계속 진행할 수 있다. 이 두번째 윈도우가 다른 컴퓨터에서 작업하는 여러분의 협력자를 대표한다. GitHub에 접근권한을 다른 사람에게 줄 필요는 없다. 왜냐하면 두 윈도우 모두 여러분이기 때문이다.

사용되고 있는 저장소의 협력자가 다른 사람에게 접근할 수 있도록 권한을 줄 필요가 있다. GitHub에 오른쪽에 ‘setting’ 버튼을 클릭해서 협력자의 사용자이름을 입력한다.

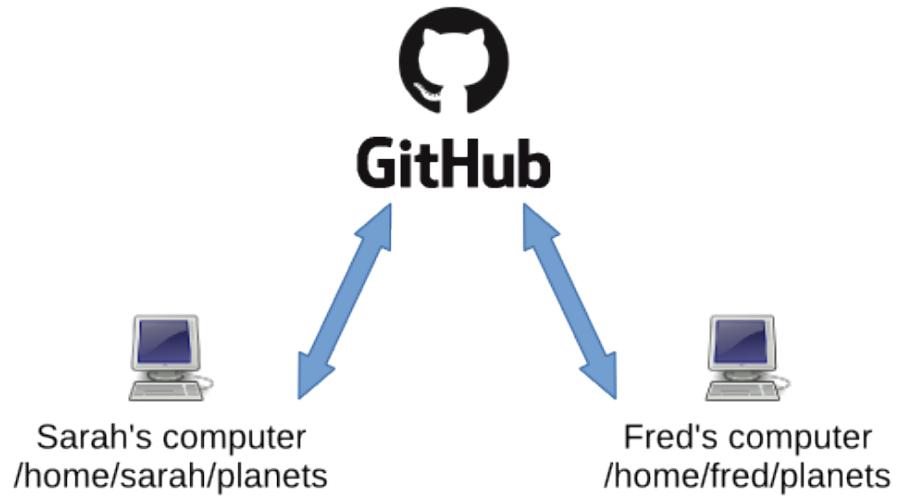


또 다른 협력자는 `cd` 명령어로 또 다른 디렉토리로 가서 (`ls` 명령어로 `planets` 폴더를 볼 수 없다.), 자신의 컴퓨터에 저장소 사본을 만든다.

```
$ git clone https://github.com/vlad/planets.git
```

‘vlad’를 여러분 협력자의 사용자이름(저장소를 소유하고 있는 사람)으로 바꾸세요.

`git clone` 명령어는 원격 저장소와 동일한 새로운 로컬 사본을 생성한다.



아제 새로운 협력자가 자신의 저장소에 변경사항을 만들 수 있다.

```
$ cd planets
$ nano pluto.txt
$ cat pluto.txt
```

It is so a planet!

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

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

그리고, 변경사항을 GitHub에 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 `origin`이라는 원격 저장소를 생성할 필요는 없다. 저장소를 복제(`clone`)할 때 사용한 이름을 Git이 자동적으로 사용해서 수행한다. (수작업으로 원격 설정을 할 때 앞에서 왜 `origin` 이름이 현명한 선택인 이유다.)

이제 우리의 컴퓨터에 GitHub 원본 저장소의 변경사항을 다운로드할 수 있다.

```
$ 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
```

주요점

- 로컬 Git 저장소는 하나 혹은 그 이상의 원격 저장소에 연결될 수 있다.
- SSH를 어떻게 설정하는지 배울 때까지 HTTPS 프로토콜을 사용해서 원격 저장소에 연결할 수 있다.
- `git push` 명령어는 로컬 저장소의 변경사항을 원격 저장소에 복사한다.
- `git pull` 명령어는 원격 저장소의 변경사항을 로컬 저장소에 복사한다.
- `git clone` 명령어는 원격 저장소를 복사해서 `origin`을 원격 저장소로자동으로 설정된 로컬 저장소를 생성한다.

GitHub 저장소를 생성하고, 복제하고, 파일을 추가하고, 파일 추가 변경사항을 GitHub에 푸쉬하고, 그리고 나서 GitHub 변경사항 타임스탬프(timestamp)를 살펴보자. GitHub는 시간을 어떻게 기록하고 왜 할까?

충돌(Conflicts)

목표

- 충돌이 무엇이고, 언제 생기는지를 설명한다.
- 병합(머지, merge)로부터 생기는 충돌을 해결한다.

사람들이 병렬로 작업을 할 수 있게 됨에 따라, 사람들이 누군가의 영역을 침범하게 된다. 혼자서 작업할 경우에도 이런 현상이 발생한다. 소프트웨어 개발을 노트북과 연구실의 서버에서 작업한다면, 각 작업본에 다른 변경사항을 만들 수 있다. 버전 제어(version control)가 중복 변경사항을 해결(resolve)할 수 있는 툴을 제공함으로서 이러한 충돌(conflicts)을 관리할 수 있게 한다.

충돌을 어떻게 해소할 수 있는지 확인하기 위해서, 먼저 파일을 생성하자. `mars.txt` 파일은 현재 두 협업하는 사람의 `planets` 저장소에서 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
```

협업하는 한 사람만의 작업본만 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 Sarah's copy
```

그리고 변경사항을 GitHub에 푸쉬하자.

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

```
[master 5ae9631] Adding 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 또 다른 협업하는 사람이 GitHub에서 갱신(update)하지 않고 변경사항을 작업파일에 만든다.

```
$ cd /tmp/planets  
$ 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
```

로컬의 변경사항을 커밋한다.

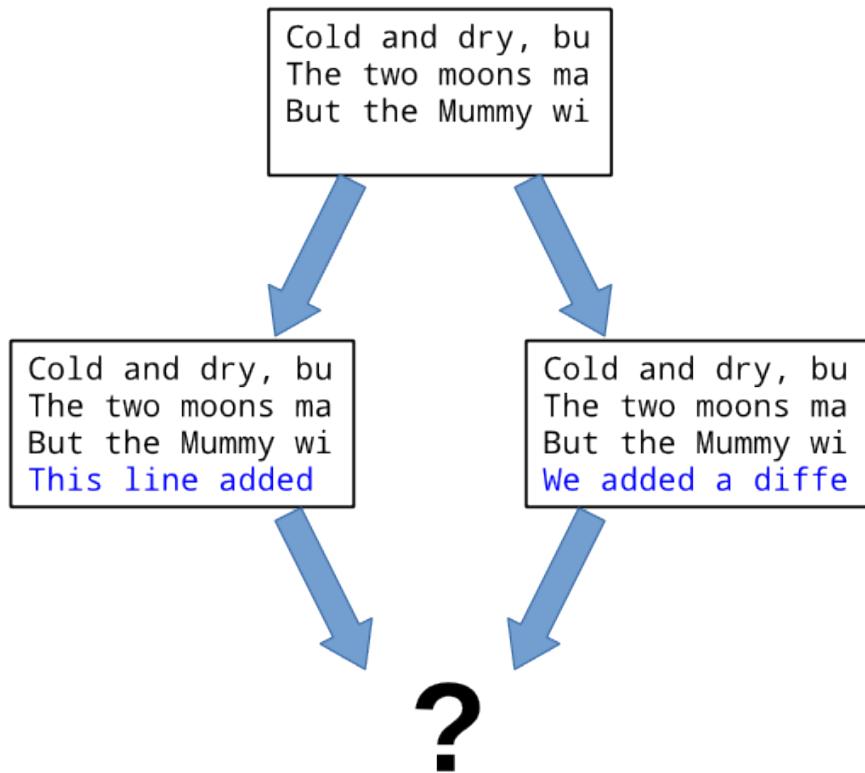
```
$ git add mars.txt  
$ git commit -m "Adding a line in my copy"
```

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

하지만 GitHub에는 푸쉬할 수 없다.

```
$ 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.
```



작업해서 변경한 사항이 다른 사람이 작업한 변경사항과 중첩되는 것을 Git이 탐지해서 앞에서 작업한 것을 뭉개지 않게 정지시킨다. 이제 해야될 일은 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 수행 결과 충돌이 있고, 해당 파일에 충돌되는 부분을 표시한다.

```
$ 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 Sarah's copy
>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d
```

<<<<< 다음에 HEAD에 있는 우리의 변경사항이 있다. Git이 자동으로 =====을 넣어서 충돌되는 변경사항 사이에 구분자로 넣고, >>>>>으로 GitHub에서 다운로드된 파일 내용의 마지막을 표시한다. (>>>>> 표시자 다음에 문자와 숫자로 구성된 문자열은 방금 다운로드한 수정 확장자다.)

파일을 편집해서 표시자/구분자를 제거하고 변경사항을 일치하는 것은 전적으로 우리에게 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 대체해서 파일이 다음과 같이 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 커밋하자.

```

$ 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 "Merging changes from GitHub"

[master 2abf2b1] Merging 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

```

Git가 병합하면서 수행한 것을 모두 추적하고 있어서, 수작업으로 다시 고칠 필요는 없다. 처음 변경사항을 만든 협력하는 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, Git가 다른 누군가 작업을 했다는 것을 알기 때문이다.

충돌되는 변경사항을 병합하는 버전 제어 기능으로 인해서 사용자가 프로그램이나 논문을 다중 파일로 쪼개서 작업하는 또 다른 이유다. 또 다른 좋은 점도 있다. 특정 파일에 반복되는 충돌이 있을 때마다, 버전 제어 시스템은 본직적으로 사용자에게 누가 무엇에 책임이 있는지, 작업을 다르게 나누는 방법을 찾을 수 있도록 명확하게 말해준다.

주요점

- 충돌은 두명 혹은 그 이상의 사람이 동시에 동일한 파일에 변경을 할 때 발생한다.
- 버전 제어 시스템은 사람들이 모르고 서로의 변경사항을 덮어쓰게 하지 못하게 한다. 대신에 충돌되는 부분을 눈에 부각시켜서 해소되어 일치할 수 있게 한다.

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

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

공개 과학(Open Science)

목표

- GNU 일반 공중 라이선스(GNU General Public License, GPL)과 대부분의 다른 공개 라이선스와 차이점을 설명한다.
- 크리에이티브 커먼즈 라이선스(Creative Commons Licence)와 조합될 수 있는 4개의 계약 조건을 설명한다.
- 프로젝트 저장소에 라이선스와 인용 정보를 올바르게 추가한다.
- 호스팅 코드와 데이터에 대한 선택사항과 각각의 장단점의 윤곽을 그린다.

“공개(open)”의 반대는 “폐쇄(closed)”가 아니다. “공개(open)”의 반대는 “망한(broken)” 것이다.

— John Wilbanks

공개 정보를 공유하는 것은 과학에서 이상적일지 모르지만, 현실은 좀 더 복잡하다. 현재, 보통 실무는 다음과 같다.

- 과학자가 데이터를 수집하고 학과에 가끔 백업되는 컴퓨터에 저장한다.
- 데이터를 분석하기 위해서 작은 프로그램을 작성하고 수정한다. (프로그램도 연구원의 로컬 노트북에 저장된다.)
- 분석 결과가 생성되자 마자, 작성해서 논문을 제출한다. 데이터를 논문에 포함할 수도 있다. (점점 많은 저널이 데이터를 요구한다.) 하지만, 아마도 프로그램 코드는 포함하지 않는다.
- 시간이 흐른다.

- 저널에서 검토(review) 결과를 연구원의 분야의 익명으로 소수의 사람들에게 서 받아 보낸다. 검토 결과를 충족하도록 논문을 수정한다. 수정하는 동안에 앞서 작성한 프로그램, 스크립트를 변경해서 다시 제출한다.
- 좀 더 많은 시간이 흐른다.
- 종국에 논문이 출판된다. 논문이 데이터 온라인 사본에 링크를 포함할 수도 있다. 하지만, 논문은 유료로 돈을 내야만 접근 가능한 장벽에 막혀있다. 개인 혹은 기관 접근 권한을 가진 사람만이 논문을 읽을 수 있다.

하지만, 점점 더 많은 과학자들에게는 프로세스가 다음과 같다.

- 과학자가 수집한 데이터가 수집되는 즉시, [figshare](#) 혹은 [Dryad](#) 같은 공개 접근 저장소에 저장된다. 그리고 DOI가 부여된다.
- 과학자가 작업물을 보관할 저장소를 GitHub에 생성한다.
- 분석작업을 수행함에 따라, 스크립트의 변경사항을 (아마도 몇몇 산출 결과도 포함해서) 저장소에 푸쉬한다. 논문을 위해서 저장소를 다목적으로 사용한다. 이 저장소가 다른 동료 과학자와 협업하는 허브가 된다.
- 논문 상태에 만족할 정도로 진행되면, [arXiv](#)와 다른 사전 프린트 서비스에 등록해서 다른 동료 과학자를 초대해서 피드백을 받는다.
- 피드백에 기초하여 저널에 논문을 마지막으로 제출하기 전에 몇번의 수정사항을 게시할 수도 있다.
- 출판된 논문은 사전출판, 코드, 그리고 데이터 저장소의 링크를 포함한다. 그렇게 함으로써 다른 과학자가 자신의 연구의 시작점으로 삼아서 연구를 쉽게 연결해서 할 수 있게 한다.

이러한 공개 연구 모델은 발견을 가속화한다. 연구 작업이 더 많이 공개될수록, 더 많이 인용되고 재사용된다. 하지만, 이런 방식으로 작업하고 연구하고자 하는 사람들은 실무에서 “공개(open)”이 정확하게 의미하는 것에 대해서 결정을 내릴 필요가 있다.

라이선싱(Licensing)

첫 질문이 라이선싱(licensing)이다. 광범위하게 말해서, 소프트웨어에 두 종류, 데이터와 출판에 6 종류의 공개 라이선스가 있다. 소프트웨어에 대해서, 한편으로는 [GNU 일반 공중 라이선스\(GNU General Public License, GPL\)](#) 계열과 다른 한편으로는 [MIT](#), [BSD](#) 같은 계열을 선택할 수 있다. 라이선스 모두 프로그램을 제한없이

공유 및 변경가능하다. 하지만, GPL은 전염성이 있는(infective) 특징이 있다. 코드의 수정된 버전을 배포하는 누구나 혹은 GPL 코드를 포함한 어느 것이든지 자신의 코드도 동일하게 자유로이 공개가능하게 만들어야 한다.

GPL 지지자는 자유로이 이용가능한 코드에서 혜택을 받은 사람이 또한 공동체에 다시 기여를 하도록 하는 것이 요건으로 필요하다고 주장한다. 반대론자는 이에 맞서 이러한 조건 없이도 많은 공개 소프트웨어 프로젝트가 오랜동안 성공적으로 진행되어 왔고, GPL이 오히려 다른 것과 코드를 조합하기 힘들게 한다고 반대주장을 한다. 결국 논란의 종국에 가장 중요한 것은 다음이다.

1. 모든 프로젝트는 라이선스가 무엇인지 명확하게 말하는 LICENSE 혹은 LICENSE.txt 같은 파일을 험 디렉토리에 포함한다.
2. 새로운 라이선스를 작성하기 보다 기존에 존재하는 라이선스를 사용한다.

두번째 점이 처음 것 만큼 중요하다. 대부분의 과학자는 법률가가 아니다. 일반인에게 분별가능한 표현이 의도하지 않은 차이와 결과를 가져올 수도 있다. [OSI](#), [Open Source Initiative](#)가 공개 소프트웨어 라이선스 목록을 관리하고, [tl;drLegal](#)에서 대부분의 라이선스를 일반적인 영어 표현으로 설명하는 정보를 얻는다.

데이터, 출판, 및 이와 동일한 것들, 통상 콘텐츠 라이선스라고 하는 것에 대해서 과학자들은 선택할 수 있는 좀더 많은 선택지가 있다. 좋은 소식은 [크리에이티브 커먼즈\(Creative Commons\)](#)라는 기관이 4개의 기본적인 제약사항을 조합해서 한 벌의 라이선스를 준비했다.

- 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.

상기 4개의 제약조건은 각각 “BY”, “ND”, “SA”, “NC”으로 축약해서 표현되어서, 예를 들어, “CC-BY-ND”는 저작자표시-변경금지 저작자를 밝히면 상업적 이용 포함하여 자유로이 이용이 가능하지만, 변경 없이 그대로 이용해야 합니다. 쉬운

영어로 6개의 CC 라이선스를 요약하고 전체 법조문에 대한 링크를 포함하는 [간략 정보](#)이 있고, 한국어로는 [CCKorea](#)에 정보가 있다.

이러한 범주에 맞지 않는 하나 더 중요한 라이선스가 있다. 과학자 그리고 다른 사람들이 간략하게 “PD”로 표기되는 사회적 공유재산(public domain) 형식을 선택할 수도 있다. 이러한 경우, 원 출처를 인용하지 않고 혹은 향후 재사용에 대한 제한없이 누구나 원하는 것을 할 수 있다. 다음 표는 6개의 크리에이티브 커먼즈 라이선스와 PD가 서로 어떻게 연결되는지를 보여준다.

표 2: Licenses that can be used for derivative work or adaptation

Original work	by	by-nc	by-nc-nd	by-nc-sa	by-nd	by-sa	pd
by	X	X	X	X	X	X	
by-nc		X	X	X			
by-nc-nd							
by-nc-sa				X			
by-nd							
by-sa						X	
pd	X	X	X	X	X	X	X

[Software Carpentry](#)는 가능하면 폭넓게 재사용될 수 있도록 수업자료에 대해서는 CC-BY, 코드에는 MIT 라이선스를 사용한다. 다시 한번, 가장 중요한 것은 프로젝트 루트 디렉토리에 LICENSE 파일로 명확하게 라이선스가 무엇인지 분명하게 말하는 것이다. 어떻게 프로젝트를 참조하는지 기술하는 CITATION 혹은 CITATION.txt 필요하면 파일을 포함할 수도 있다. Software Carpentry 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}
}
```

호스팅(Hosting)

저작물이나 작업을 공개하고자 하는 그룹에서 가지는 두번째 큰 질문은 코드와 데이터를 어디에 호스팅할지 정하는 것이다. 방법중의 하나는 연구실, 학과, 혹은 대학에 서버를 제공하여 계정관리와 백업 등을 관리하는 것이다. 주된 장점은 누가 무엇을 소유하는지 명확하다. 특히 민감한 정보(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.

또 다른 선택지는 도메인을 구입하고 ISP(Internet service provider)에 비용을 지불 한다. 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](#), [Google Code](#), 혹은 [SourceForge](#) 같은 공개 호스팅 서비스를 채용하는 것이다. 웹 인터페이스로 저장소를 생성하고, 메일링 리스트와 누가 무엇을 하는지 추적하는 방법 등을 제공한다. 규모의 경제와 네트워크 효과로 모두 이익을 볼 수 있다. 즉, 동일한 표준에 작은 많은 서비스를 실행하는 것보다 하나의 큰 서비스를 실행하는 것이 더 쉽고, 동일한 서비스를 사용한다면, 사람들이 협업하기도 더 쉽지만, 더 적은 비밀번호를 기억해야 되기 때문만은 아니다.

하지만, 이러한 모든 서비스는 사람들의 저작과 작업에 제한을 둔다. 특히, 대부분의 서비스는 사용자에게 선택권을 준다. 즉, 다른 사람과 작업을 공유한다면, 무료로

사용가능하지만, 만약 사적이용을 한다면, 비용을 지불할지도 모른다. 공유는 과학에서 유일하게 타당성 있는 선택사항처럼 보이지만, 많은 학교나 기관이 연구자가 이렇게 하도록 허가하지는 않는다. 왜냐하면 미래 특허 출원을 보호하거나, 단순히 새로운 것이 종종 무섭기 때문이다.

주요점

- 공개 과학 저작물은 패쇄된 것보다 더 유용하고, 더 많이 인용된다.
- GPL 소프트웨어를 채용하는 사람은 자신의 것도 공개해야한다. 다른 공개 소프트웨어 라이선스는 이런 제한을 두고 있지 않다.
- 크리에이티브 커먼즈 라이선스를 사용해서 저작자 표시, 2차 저작물의 생성, 공유 및 상업적 이용의 요구사항과 제한을 조합하고 매칭할 수 있다.
- 법률가 아닌 사람은 처음부터 라이선스를 작성하지 말아야 한다.
- 프로젝트를 대학 서버, 개인 도메인, 혹은 공개 저장소에 둘 수 있다.
- 지적 재산권과 민감 정보의 저장에 대한 규정은 코드와 데이터가 어디에 저장되든지 관계없이 적용된다.

여러분이 작성하고 있는 소프트웨어에 공개 소프트웨어 라이선스를 적용할 수 있는지 알아보세요. 여러분이 일방적으로 할 수 있나요? 혹은 여러분의 기관이나 조직의 다른 사람에게서 허락이 필요한가요? 만약 그렇다면 누굴까요?

공개 저장소에 여러분의 저작물을 호스팅할 수 있는지 알아보세요. 여러분이 일방적으로 할 수 있나요? 혹은 여러분의 기관이나 조직의 다른 사람에게서 허락이 필요한가요? 만약 그렇다면 누굴까요?

git 설치

목표

- Git 설치한다.
- 신규 소프트웨어 카펜트리 교육과정 작성
- 의존성(dependencies) 설치

Git 설치 및 Git 저장소 가져오기 Git가 설치되어 있지 않다면 리눅스 우분투 기준으로 다음 명령어를 사용하여 설치한다.

```
$ sudo apt-get update  
$ sudo apt-get install git
```

GitHub 저장소에서 사용자 컴퓨터로 클론하여 가져온다.

```
$ git clone https://github.com/swcarpentry/slideshows.git
```

소프트웨어 카펜트리 학습과목 생성

1. `data-cleanup` 저장소를 GitHub에 생성한다.
2. 템플릿 저장소를 학습과목과 동일한 이름으로 클론한다.

```
$ git clone -b gh-pages -o upstream https://github.com/swcarpentry/lesson-template.git data-
```

1. 다음 명령어를 이용하여 다운로드 받은 디렉토리로 이동한다.

```
$ cd data-cleanup
```

1. 다음 명령어를 사용하여 `origin`이라는 이름으로 GitHub 저장소를 추가한다.

```
$ git remote add origin https://github.com/statklee/data-cleanup
```

1. 작업을 진행한다.
2. 학습과목에 대한 HTML 페이지를 빌드한다. `pandoc`을 설치해야 한다. 기존 GitHub가 HTML 페이지를 빌드하는 것이 아니고 학습과목을 로컬 컴퓨터에서 빌드해서 GitHub에 푸쉬해한다.

```
$ make preview
```

1. 작업하여 빌드한 HTML 페이지를 GitHub 저장소 gh-pages 브랜치로 푸쉬 한다.

```
$ cd data-cleanup  
$ git add changed-files.md *.html  
$ git commit -m "Explanatory message"  
$ git push origin gh-pages
```

의존성(dependencies) 처음 리눅스를 설치했다면 아무것도 없기 때문에 가장 기본적인 make부터 설치한다. pandoc 설치를 위해서 파이썬 pip 도 설치한다. 마지막으로 소프트웨어 카펜트리 학습과정 작성을 위해서 pandocfilters도 설치한다.

1. make

```
$ sudo apt-get install make
```

1. Python pip

```
$ sudo apt-get install python-pip
```

1. pandoc

```
sudo apt-get install pandoc
```

1. pandocfilters Pandoc에 필터 작성을 도와주는 파이썬 모듈

```
sudo pip install pandocfilters
```

파이썬 프로그래밍(python)

프로그램을 어떻게 작성하는지 배우는 가장 좋은 방법은 의미있는 무언가를 작성하는 것이다. 그래서 이번 파이썬(python) 소개는 혼한 과학 작업(즉 데이터 분석)에 맞춰있다.

진정한 목적은 파이썬을 가르치는 것이 아니라 모든 프로그래밍에 달려있는 기본 개념을 전달한다. 파이썬을 사용해서 학습을 진행한다. 왜냐하면,

1. 예제로 어떤 언어든지 사용해야한다.
2. 무료이고, 문서화가 잘 되어 있고, 거의 모든 곳에서 실행된다.
3. 과학자들 사이에서 크고 (그리고 점증하는) 사용자 기반이 있다.
4. 경험에서 초보자가 대부분의 다른 언어들보다 익히기 쉽다는 것을 알고 있다.

하지만, 두 가지 가장 중요한 것은 무슨 언어에 상관없이 동료가 사용하는 것을 사용해서 작업 결과를 쉽게 공유할 수 있어야 하고, 언어를 잘 사용해야 한다.

환자 데이터(Patient Data) 분석

관절염에 새로운 치료법이 처방된 환자의 염증에 대한 연구를 진행하고 있고, 첫 12 데이터셋(Data Set)을 분석할 필요가 있다. 데이터셋은 CSV 형식(comma-separated values, 구분자가 콤마 값을 가진 파일 형식)으로 저장되어 있다. 각 행은 환자 한명의 정보로 구성되어 있고, 열은 일련의 날짜 정보를 나타낸다. 첫번째 파일의 첫 몇 행의 정보는 다음과 같다.

```
0,0,1,3,1,2,4,7,8,3,3,10,5,7,4,7,7,12,18,6,13,11,11,7,7,4,6,8,8,4,4,5,7,3,4,2,3,0,0  
0,1,2,1,2,1,3,2,2,6,10,11,5,9,4,4,7,16,8,6,18,4,12,5,12,7,11,5,11,3,3,5,4,4,5,5,1,1,0,1  
0,1,1,3,3,2,6,2,5,9,5,7,4,5,4,15,5,11,9,10,19,14,12,17,7,12,11,7,4,2,10,5,4,2,2,3,2,2,1,1  
0,0,2,0,4,2,2,1,6,7,10,7,9,13,8,8,15,10,10,7,17,4,4,7,6,15,6,4,9,11,3,5,6,3,3,4,2,3,2,1  
0,1,1,3,3,1,3,5,2,4,4,7,6,5,3,10,8,10,6,17,9,14,9,7,13,9,12,6,7,7,9,6,3,2,2,4,2,0,1,1
```

다음을 수행해야 된다.

- CSV 형식 데이터 파일을 주기억장치에 로딩/loading)한다.
- 모든 환자에 대해서 각 날짜별로 평균 염증을 계산한다.
- 결과값을 플롯(plot)한다.

상기 모든 것을 수행하기 위해서, 프로그래밍에 관해 약간 학습할 필요가 있다.

목표

- 라이브러리가 무엇인지와, 무슨 라이브러리가 사용되는지 설명한다.

- 파이썬 라이브러리를 로드(load)하고 라이브러리가 담고있는 것을 사용한다.
- 파일에서 테이블 형식 데이터를 프로그램으로 읽어온다.
- 값을 변수에 할당한다.
- 데이터에서 각각의 값과 일부를 선택한다.
- 배열 데이터에 연산을 수행한다.
- 단순 그래프를 화면에 출력한다.

데이터 로딩>Loading)

데이터 로딩>Loading)

단어는 유용하지만, 좀 더 유용한 것은 단어로 작성된 문장과 스토리다. 마찬가지로 많은 강력한 툴이 파이썬 같은 언어에 내장되어 있지만, 좀 더 생동감 있는 라이브러리(libraries)로 강력한 툴을 만들 수 있다.

염증 데이터를 로드하기 위해서, NumPy라는 라이브러리를 가져오기(import)를 할 필요가 있다. 일반적으로 숫자로 멋진 것을 하려고 하거나, 특히, 행렬을 가지고 있다면 NumPy 라이브러리를 사용해야 한다.

```
import numpy
```

라이브러리를 가져오는 것은 물품 보관함에서 연구실 장비를 꺼내서 벤치에 설치하는 것과 같다. 완료되면, 라이브러리에게 데이터를 읽어오라고 지시할 수 있다.

```
numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
array([[ 0.,  0.,  1., ...,  3.,  0.,  0.],
       [ 0.,  1.,  2., ...,  1.,  0.,  1.],
       [ 0.,  1.,  1., ...,  2.,  1.,  1.],
       ...,
       [ 0.,  1.,  1., ...,  1.,  1.,  1.],
       [ 0.,  0.,  0., ...,  0.,  2.,  0.],
       [ 0.,  0.,  1., ...,  1.,  1.,  0.]])
```

`numpy.loadtxt(...)` 표현식은 함수 호출(function call)로 파이썬에게 `numpy` 라이브러리에 속해 있는 `loadtxt` 함수를 실행하게 한다. 점 표기법(dotted notation)

은 파이썬에서 도처에서 사용되는데 `thing.component` 같은 어떤 것의 부분을 나타낸다.

`numpy.loadtxt`는 매개변수(parameters)가 두개 있다. 읽어 오려고 하는 파일이름과 라인에서 값을 구분하는 구분자(delimiter)다. 둘다 문자의 열, 짧게 줄여서 문자열(strings)이 되어야 한다. 그래서 인용부호 안에 넣는다.

타이핑을 마치고 쉬프트+엔터(Shift+Enter)를 누를 때, 노트북은 명령어를 실행한다. 함수 출력에 대해서 어떤 것도 지시한 것이 없기 때문에, 노트북은 출력 결과를 화면에 출력한다. 이 경우 출력결과는 방금 전에 로드한 데이터 자체다. 디폴트로, 단지 몇개의 행과 열만 보여진다. (커다란 배열을 화면에 출력할 때는 요소를 생략하기 위해서 ...를 사용하여 표기한다.) 공간을 절약하기 위해서 파이썬은 소수점 다음에 흥미로운 것이 없을 때는 숫자를 1.0 대신에 1.을 사용한다.

`numpy.loadtxt` 함수 호출은 파일을 읽어들이지만, 주기억장치에 데이터를 저장하지는 않는다. 저장하기 위해서는 변수(variable)를 배열에 할당(assign)할 필요가 있다. 변수는 `x`, `current_temperature`, `subject_id` 같은 값에 대한 이름이다. 파이썬 변수는 문자로 시작해야만 한다. 값을 변수에 =을 사용하여 간단하게 할당할 수 있다. 일례로, 한걸음 물러나서 테이블 데이터를 고려하는 대신에 가장 단순한 데이터 “집합(컬렉션,collection)”, 단일 값을 고려하자. 다음 라인은 변수에 값을 할당한다.

```
weight_kg = 55
```

변수가 값을 가지게 되면 출력할 수도 있다.

```
print weight_kg
```

```
55
```

변수로 산수를 할 수 있다.

```
print 'weight in pounds:', 2.2 * weight_kg
```

```
weight in pounds: 121.0
```

변수에 새로운 값을 할당해서 변수의 값도 변경할 수 있다.

```

weight_kg = 57.5
print 'weight in kilograms is now:', weight_kg

weight in kilograms is now: 57.5

```

상기 예제가 보여주듯이, 한번에 여러개를 콤마로 구분해서 출력할 수 있다.

만약 변수를 이름이 써진 포스트잇 같은 스티커 노트라고 가정한다면, 할당은 특정한 값에 스티커 노트를 붙이는 것과 같다.



이것이 의미하는 바는 한 개체에 값을 할당하는 것이 다른 변수의 값을 변경시키지는 않는다. 예를 들어, 변수에 개체의 무게를 파운드로 저장하자.

```

weight_lb = 2.2 * weight_kg
print 'weight in kilograms:', weight_kg, 'and in pounds:', weight_lb

weight in kilograms: 57.5 and in pounds: 126.5

```



그리고 나서 `weight_kg`를 변경하자.

```

weight_kg = 100.0
print 'weight in kilograms is now:', weight_kg, 'and weight in pounds is still:', weight_lb

weight in kilograms is now: 100.0 and weight in pounds is still: 126.5

```



`weight_lb` 변수는 값이 어디에서 왔는지 기억하지 않기 때문에, 자동적으로 `weight_kg`이 변경될 때 갱신되지 않는다. 엑셀같은 스프레드시트가 동작하는 방식과 이런 점이 다르다.

변수에 어떻게 값을 할당하듯이 동일한 구문을 사용하여 변수에 배열 값을 할당할 수도 있다. `read.csv`를 다시 실행하고 결과를 저장하자. `numpy.loadtxt`을 다시 시작하고 결과를 저장하자.

```
data = numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
```

상기 문장은 결과를 출력하지 않는데 할당은 어떤 것도 화면에 출력하지 않기 때문이다. 데이터가 주기억장치에 로딩되었는지 확인하고자 한다면, 변수의 값을 출력할 수 있다.

```
print data  
  
[[ 0.  0.  1. ...,  3.  0.  0.]  
 [ 0.  1.  2. ...,  1.  0.  1.]  
 [ 0.  1.  1. ...,  2.  1.  1.]  
 ...,  
 [ 0.  1.  1. ...,  1.  1.  1.]  
 [ 0.  0.  0. ...,  0.  2.  0.]  
 [ 0.  0.  1. ...,  1.  1.  0.]]
```

도전 과제

1. 도표를 그려서 다음 프로그램의 각 문장이 실행된 후에 무슨 변수가 무슨 값을 참조하는지 보이세요.

```
mass = 47.5  
age = 122  
mass = mass * 2.0  
age = age - 20
```

2. 다음 프로그램이 출력하는 것은 무엇인가요?

```
first, second = 'Grace', 'Hopper'  
third, fourth = second, first  
print third, fourth
```

데이터 능숙하게 다루기

데이터가 주기억장치에 올라갔기 때문에, 데이터를 가지고 무언가를 시작할 수 있다. 먼저 data가 무슨 형식(type)인지 확인해보자.

```
print type(data)  
  
<type 'numpy.ndarray'>
```

출력결과를 통해서 NumPy 라이브러리로 생성된 N-차원의 배열임을 확인할 수 있다. 데이터가 어떤 형태(shape)인지도 알 수 있다.

```
print data.shape  
  
(60, 40)
```

data가 60 행과 40 열로 구성된 것을 알 수 있다. `data.shape`은 `data`의 멤버(member)다. 즉, 더 커다란 값의 일부로 저장되어 있다. 라이브러리의 함수에 사용한 동일한 점표기법을 값의 멤버에 대해서 사용하는데 이유는 동일한 “부분과 전체” 관계를 가지기 때문이다.

행렬에서 값 하나를 얻으려고 한다면, 수학에서 하는 것과 같은 방식으로 꺽쇠 팔호내부에 인덱스(index)를 넣을 수 있다.

```
print 'first value in data:', data[0, 0]  
  
first value in data: 0.0  
  
print 'middle value in data:', data[30, 20]  
  
middle value in data: 13.0
```

`data[30, 20]` 표현식은 놀랍지 않지만, `data[0, 0]` 표현식은 여러분을 놀라게 할지 모른다. Fortran이나 Matlab 같은 프로그래밍 언어는 1에서 샘을 시작하는데 이유는 수천년동안 인류가 해왔던 동일한 방식이다. C++, Java, Perl, Python을 포함하는 C 언어 패밀리는 0에서 샘을 시작하는데 이유는 컴퓨터에게 더 간단하기

때문이다. 결과적으로, 파이썬에서 $M \times N$ 배열을 가지고 있다면, 인덱스는 첫번째 축으로 0에서 $M-1$, 두번째 축으로 0에서 $N-1$ 까지 간다. 익숙해지는데는 조금 시간이 걸리지만, 규칙을 기억하는 방식은 인덱스는 출발점에서 원하는 항목까지 도달하는데 얼마나 많은 단계를 거치는지 세는 것이다.

구석에서(In the Corner) 또한 여러분을 놀라게 하는 것은 파이썬이 배열을 출력할 때, 왼쪽 하단보다 왼쪽 상단 구석에 [0, 0] 인덱스를 가진 요소가 위치한다. 수학자가 행렬을 그리는 것과 일치하지만, 데카르트 좌표계(Cartesian coordinates)와는 다르다. 인덱스가 같은 이유로 (열, 행) 대신에 (행, 열)이여서, 데이터를 가지고 그래프를 그릴 때 혼동될 수 있다.

[30, 20] 같이 인덱스를 넣어서 배열의 요소값을 하나 선택할 수 있지만, 전체도 선택할 수 있다. 예를 들어, 다음과 같이 첫 환자 네명(행)에 대한 첫 10일치(열) 값을 선택할 수 있다.

```
print data[0:4, 0:10]

[[ 0.  0.  1.  3.  1.  2.  4.  7.  8.  3.]
 [ 0.  1.  2.  1.  2.  1.  3.  2.  2.  6.]
 [ 0.  1.  1.  3.  3.  2.  6.  2.  5.  9.]
 [ 0.  0.  2.  0.  4.  2.  2.  1.  6.  7.]]
```

슬라이스(slice) 0:4가 뜻하는 것은 “0번 인덱스에서 시작해서 4번 인덱스까지 가는데 인덱스 4는 포함하지 않는다.” 인덱스 몇번까지는 하지만 포함하지 않는다는 것에 익숙해지는데 시간이 걸리지만, 상한과 하한의 차이가 슬라이스에 있는 값의 갯수라는 것이 규칙이다. 슬라이스가 반드시 0번에서 시작할 필요는 없다.

```
print data[5:10, 0:10]

[[ 0.  0.  1.  2.  4.  2.  1.  6.  4.]
 [ 0.  0.  2.  2.  4.  2.  2.  5.  5.]
 [ 0.  0.  1.  2.  3.  1.  2.  3.  5.]
 [ 0.  0.  1.  2.  3.  1.  2.  3.  3.]]
```

```
[ 0.  0.  0.  3.  1.  5.  6.  5.  5.  8.]  
[ 0.  1.  1.  2.  1.  3.  5.  3.  5.  8.]]
```

슬라이스의 상한과 하한을 반드시 포함할 필요는 없다. 하한을 포함하지 않는다면, 파이썬은 디폴트 값으로 0을 사용한다. 만약 상한을 포함하지 않는다면, 슬라이스는 그 해당 축의 끝까지 간다. 만약 상한이나 하한을 포함하지 않는다면 (즉, ':'만 사용한다면), 슬라이스는 모든 것을 포함한다.

```
small = data[:3, 36:]  
print 'small is:'  
print small  
  
small is:  
[[ 2.  3.  0.  0.]  
 [ 1.  1.  0.  1.]  
 [ 2.  2.  1.  1.]]
```

배열은 자체로 일반적인 수학 연산을 배열값에 수행한다. 데이터를 가지고 할수 있는 가장 단순한 연산은 사칙연산(덧셈, 뺄셈, 곱셈, 나눗셈)이다. 배열에 사칙연산을 수행할 때, 배열의 각각의 요소에 대해서 연산을 수행한다.

```
doubledata = data * 2.0
```

그래서 상기 연산은 data에 해당 요소값 각각에 2를 곱한 값을 가지는 doubledata 배열을 생성한다.

```
print 'original:'  
print data[:3, 36:]  
print 'doubledata:'  
print doubledata[:3, 36:]  
  
original:  
[[ 2.  3.  0.  0.]  
 [ 1.  1.  0.  1.]  
 [ 2.  2.  1.  1.]]
```

```
doubledata:  
[[ 4.  6.  0.  0.]  
 [ 2.  2.  0.  2.]  
 [ 4.  4.  2.  2.]]
```

배열을 가지고 상기에서처럼 단일 값 각각에 연산을 수행하는 대신에 만약 동일한 크기와 형태를 가진 다른 배열이 있다면, 두 배열의 해당 요소에 연산을 수행할 수 있다.

```
tripledata = doubledata + data
```

따라서 상기 표현식은 `doubledata[0,0]`을 `data[0,0]`에 더해서 `tripledata[0,0]`이 되고, 배열의 다른 모든 요소에도 동일하게 계산된 결과 배열이 될 것이다.

```
print 'tripledata:'  
print tripledata[:3, 36:]  
  
tripledata:  
[[ 6.  9.  0.  0.]  
 [ 3.  3.  0.  3.]  
 [ 6.  6.  3.  3.]]
```

종종 데이터의 값을 더하고, 빼고, 곱하고, 나누는 것 이상 원한다. 배열을 통해서 배열의 값에 좀더 복잡한 연산을 수행할 수 있다. 예를 들어, 모든 환자에 대한 모든 날짜의 평균 염증값을 얻고자 한다면, 배열에 평균값을 단순히 요청하면된다.

```
print data.mean()
```

6.14875

`mean`은 배열의 메쏘드(method)다. 즉, 멤버 `shape`가 하는 동일한 방식으로 배열에 속한 함수다. 만약 변수가 명사라면, 메쏘드는 동사라고 볼 수 있고 문제의 것이 어떻게 수행하는지를 알고 있다. 왜 `data.shape`이 호출될 필요가 없고 (단지 사물이다.), `data.mean()`은 호출(액션이다.)되는지 설명하는 이유다. 또한,

`data.mean()`에 대해서 빈 괄호가 필요한 이유이기도 하다. 심지어 어떤 매개 변수도 전달할 필요가 없을 때, 괄호는 파이썬에게 가서 무언가 어떻게 하라고 하는 것이다.

NumPy 배열에는 많은 유용한 메쏘드가 있다.

```
print 'maximum inflammation:', data.max()
print 'minimum inflammation:', data.min()
print 'standard deviation:', data.std()

maximum inflammation: 20.0
minimum inflammation: 0.0
standard deviation: 4.61383319712
```

하지만, 데이터를 분석할 때, 종종 환자마다 최대값 혹은 일자별 평균값 같은 부분 통계량을 보고자 한다. 이를 수행하는 방법은 데이터 선택해서 임시 배열로 생성하고 계산을 수행하는 것이다.

```
patient_0 = data[0, :]
print 'maximum inflammation for patient 0:', patient_0.max()

maximum inflammation for patient 0: 18.0
```

실질적으로 변수의 행을 저장할 필요하는 없다. 대신에 선택한 것과 메쏘드 호출을 조합할 수 있다.

```
print 'maximum inflammation for patient 2:', data[2, :].max()

maximum inflammation for patient 2: 19.0
```

만약 모든 환자에 대한 최대 염증값 혹은 각 날짜별로 평균값이 필요하다면 어떨까? 다음 도표처럼, 축을 따라서 연산을 수행하고자 한다.

이러한 기능을 지원하기 위해서, 대부분의 배열 메쏘드는 작업하고자 하는 축을 지정할 수 있다. 만약 0번 축을 따라서 평균을 구하고자 한다면, 다음을 얻게 된다.

```
print data.mean(axis=0)
```

```
[ 0.          0.45         1.11666667   1.75         2.43333333   3.15
 3.8          3.88333333   5.23333333   5.51666667   5.95         5.9
 8.35         7.73333333   8.36666667   9.5          9.58333333
 10.63333333 11.56666667  12.35         13.25         11.96666667
 11.03333333 10.16666667  10.           8.66666667   9.15         7.25
 7.33333333  6.58333333   6.06666667   5.95         5.11666667   3.6
 3.3          3.56666667   2.48333333   1.5          1.13333333
 0.56666667]
```

빠른 점검 목적으로 배열에게 어떤 모양인지 확인할 수 있다.

```
print data.mean(axis=0).shape
```

```
(40,)
```

(40,) 결과값은 $N \times 1$ 벡터임을 보여줘서 모든 환자에 대해서 날짜별 평균 염증값을 알 수 있다. 만약 1번 축으로 평균값을 구한다면, 다음을 얻게 된다.

```
print data.mean(axis=1)
```

```
[ 5.45   5.425   6.1    5.9    5.55   6.225   5.975   6.65   6.625   6.525
 6.775  5.8     6.225  5.75   5.225  6.3     6.55    5.7     5.85    6.55
 5.775  5.825   6.175  6.1    5.8     6.425   6.05    6.025   6.175  6.55
 6.175  6.35    6.725  6.125  7.075  5.725   5.925   6.15    6.075   5.75
 5.975  5.725   6.3    5.9    6.75    5.925   7.225   6.15    5.95    6.275  5.7
 6.1     6.825   5.975  6.725  5.7     6.25    6.4     7.05    5.9 ]
```

상기 값은 모든 날짜에 대해서 환자별로 평균 염증값을 보여준다.

도전 과제 배열의 일부를 슬라이스(slice)라고 부른다. 문자열에 대해서도 또한 슬라이스를 취할 수 있다.

```
element = 'oxygen'
print 'first three characters:', element[0:3]
print 'last three characters:', element[3:6]
```

```
first three characters: oxy  
last three characters: gen
```

1. `element[:4]`의 값은 무엇인가? `element[4:]`의 값은 무엇인가? 혹은 `element[:]`의 값은 무엇인가?
2. `element[-1]`의 값은 무엇인가? `element[-2]`의 값은 무엇인가? 상기 해답이 주어졌을 때, `element[1:-1]`의 값이 무엇인지 설명하시오.
3. `element[3:3]`의 표현식은 빈 문자열(empty string)을 출력한다. 즉, 어떠한 문자도 포함하지 않은 문자열이다. 만약 `data`가 환자 데이터의 배열이라면, `data[3:3, 4:4]`은 무슨 값을 출력할까? `data[3:3, :]`은 무슨 값을 출력할까?

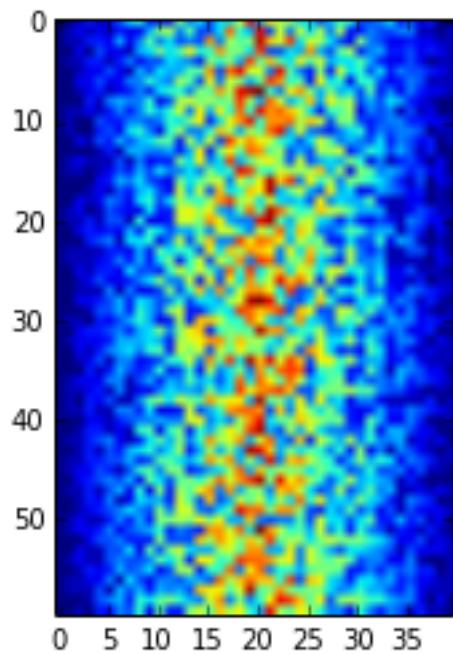
플로팅(Plotting)

수학자 Richard Hamming은 “컴퓨팅의 목적은 숫자가 아니라 직관(insight)다.” 그래서 직관을 키우는 가장 좋은 방법은 종종 데이터를 시각화하는 것이다. 시각화에 대한 온전한 강의를 펼칠만 하지만, 여기서는 파이썬 `matplotlib`의 기능 몇가지만 살펴본다. “공식적인” 플로팅 라이브러리는 없지만, `matplotlib`가 사실상의 표준이다. 먼저, IPython Notebook에게 별도의 윈도우에 떨어져서 보이기보다는 플롯이 인라인(inline)으로 즉시 화면에 출력되도록 설정한다.

```
%matplotlib inline
```

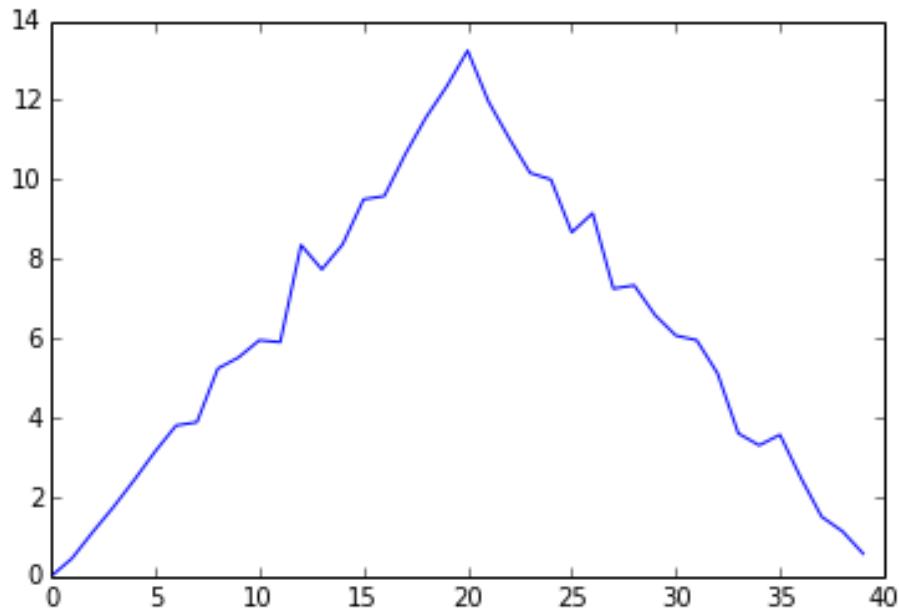
라인의 시작에 `%` 기호는 파이썬 문장(statement)보다는 notebook에 대한 명령어라는 신호를 준다. 다음으로, `matplotlib`에서 `pyplot` 모듈을 가져와서 함수 두개를 사용해서 자료에 대한 히트맵(heat map)을 생성하여 화면에 출력한다.

```
from matplotlib import pyplot  
pyplot.imshow(data)  
pyplot.show()
```



히트맵의 파란색 지역은 값이 낮은 반면에 붉은 지역은 높은 값을 나타낸다. 염증
값은 40일에 걸쳐 오르고 내린다. 시간에 따른 평균 염증값을 살펴보자.

```
ave_inflammation = data.mean(axis=0)
pyplot.plot(ave_inflammation)
pyplot.show()
```

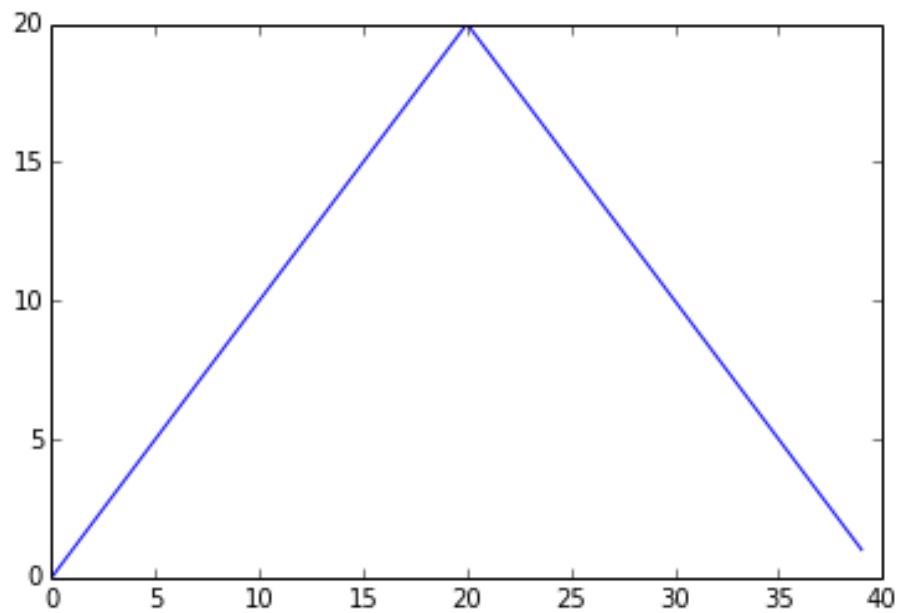


여기서, `ave_inflammation` 변수에 모든 환자에 대해서 날짜별로 평균 염증값을 저장하고, `pyplot`으로 평균 염증값의 선 그래프를 생성하여 화면에 출력한다. 결과는 대략 선형적으로 올라가고 내려가지만 의심스럽다. 다른 연구 결과에 기초하여 좀더 빠른 급격한 상승과 좀더 완만한 하강이 예측됐다. 또 다른 두개의 통계량(일자별로 최대 그리고 최소 염증값)을 살펴보자.

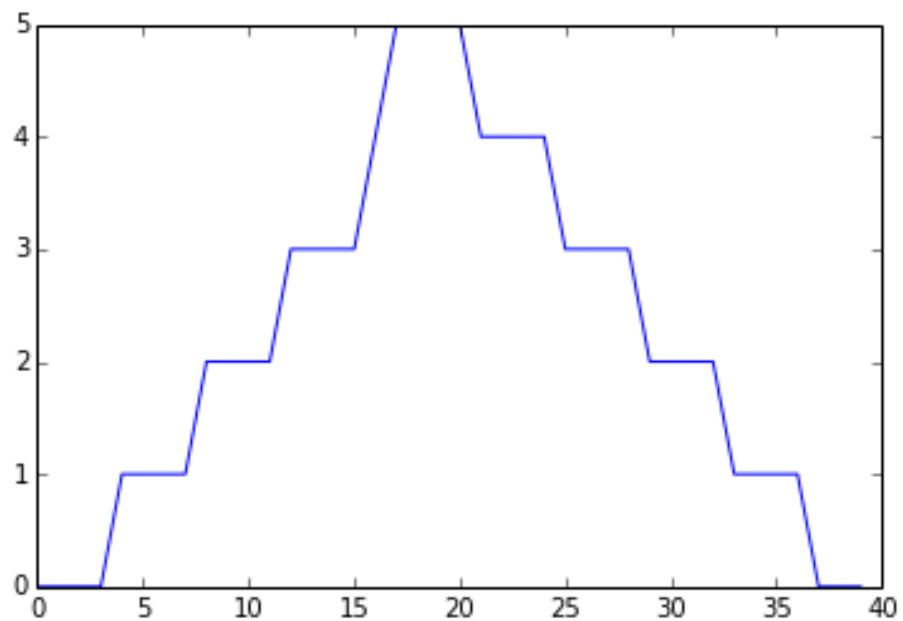
```
print 'maximum inflammation per day'
pyplot.plot(data.max(axis=0))
pyplot.show()
```

```
print 'minimum inflammation per day'
pyplot.plot(data.min(axis=0))
pyplot.show()
```

```
maximum inflammation per day
```



minimum inflammation per day



최대값은 완벽하게 매끄럽게 상승하고 하강하지만, 최소값은 계단 함수(Step function) 모양으로 보인다. 어느쪽의 결과도 그럴듯하지 못해서 계산에서 실수가 있거나 데이터에 뭔가 잘못되었다.

도전 과제

1. 왜 모든 플롯이 그래프 상단이 짤리나요? 왜 날짜별 최소 염증값의 플롯에 수직라인이 완벽하게 수직이 아니죠?
2. 모든 환자에 대해서 각 일자별로 염증 데이터의 표준편차를 보이는 그래프를 생성하세요.

요약하기

타이핑하는 양을 줄이기 위해서 라이브러리를 가져오기 할 때 에일리어스(alias, 별명)를 사용하는 것은 매우 일반적이다. numpy와 pyplot에 대해서 에일리어스를 사용해서 차례로 플롯 3개를 작성했다.

```
import numpy as np
from matplotlib import pyplot as plt

data = np.loadtxt(fname='inflammation-01.csv', delimiter=',')

plt.figure(figsize=(10.0, 3.0))

plt.subplot(1, 3, 1)
plt.ylabel('average')
plt.plot(data.mean(0))

plt.subplot(1, 3, 2)
plt.ylabel('max')
plt.plot(data.max(0))

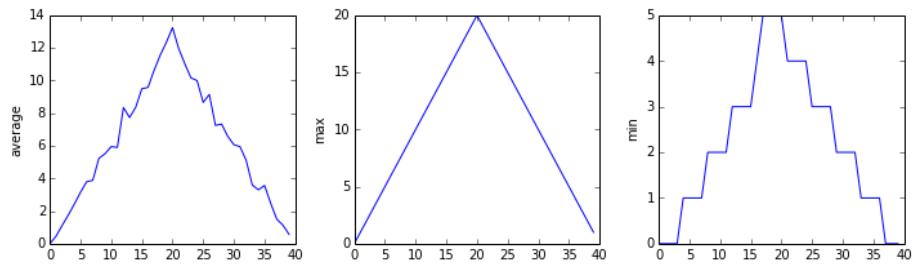
plt.subplot(1, 3, 3)
```

```

plt.ylabel('min')
plt.plot(data.min(0))

plt.tight_layout()
plt.show()

```



첫 두 라인은 대부분의 파이썬 프로그래머가 사용하는 에일리어스인 np과 plt으로 라이브러리를 다시 적재한다. loadtxt을 호출하여 데이터를 읽어들이고, 프로그램의 나머지는 플로팅 라이브러리에게 그래프가 얼마나 커야하는지, 3개의 하위 플롯을 생성한다고, 각각의 하위 플롯에 무엇을 그려넣을 것인지, 그리고 빠빠한 레이아웃으로 설정한다는 것을 담고있다. (만약 plt.tight_layout()에 맡겨놓으며, 그래프가 좀더 가깝게 압착하게 될 것이다.)

도전 과제

1. 프로그램을 변경하여 옆으로 배열되는 대신에 위에서 아래로 배열되도록 3 개의 플롯을 화면에 출력하세요.

주요점

- import libraryname을 사용하여 프로그램에 라이브러리를 가져온다.
- numpy 라이브러리로 사용하여 파이썬의 배열자료를 작업한다.
- 주기억장치 메모리에 값을 저장하기 위해서 variable = value을 사용하여 변수에 값을 할당한다.

- 값이 변수에 할당될 때마다 변수가 즉석에서 생성된다.
- `print something` 을 사용해서 `something` 의 값을 화면에 출력한다.
- `array.shape` 표현식은 배열의 모양을 반환한다.
- `array[x, y]` 을 사용하여 배열의 특정 한 요소를 선택한다.
- 배열 인덱스는 1이 아니고, 0에서 시작한다.
- `low:high` 을 사용해서 `low`에서 `high-1`까지 인덱스를 포함하는 슬라이스를 지정한다.
- 배열에서 동작하는 모든 슬라이싱과 인덱싱은 문자열에도 동일하게 동작한다.
- `# some kind of explanation` 을 사용하여 프로그램에 주석을 추가한다.
- 단순한 통계량을 계산하기 위해서 `array.mean()`, `array.max()`, `array.min()` 을 사용한다.
- `array.mean(axis=0)` 혹은 `array.mean(axis=1)` 을 사용하여 주어진 축을 따라서 통계량을 계산한다.
- `matplotlib`에서 `pyplot` 라이브러리를 사용해서 간단한 시각화 플롯을 생성한다.

다음 단계 지금까지 작업 결과를 통해 데이터 파일에 뭔가 잘못된 것이 있다는 것을 확인했다. 다른 11개 파일도 같은 방식으로 확인하고자 하지만, 동일한 명령어를 반복적으로 타이핑하는 것은 지루하고 오류에 쉽게 노출된다. 컴퓨터는 우리가 알고 있는 한 지루해 하지 않기 때문에, 단일 명령어로 전체 분석을 수행할 수 있는 방법을 생성하고 나서 각 파일에 대해서 각 단계를 어떻게 반복하는지를 해결해야 한다. 이런 작업이 다음 두 학습 주제다.

함수 생성

만약 분석할 데이터셋이 하나라면, 파일을 스프레드시트에 올려서 간단한 통계치를 구하고 그래프를 그리는 것이 아마도 훨씬 빠르다. 하지만, 확인할 파일이 12개이고 앞으로 더 늘어난다면 얘기는 달라진다. 이번 학습에서 함수를 어떻게 작성하는지 배워서 하나의 명령어로 몇 개 작업을 반복할 수 있다.

목표

- 매개변수(parameter)를 받는 함수를 작성한다.
- 함수에서 값을 반환한다.
- 함수를 디버깅하고 테스트한다.
- 콜 스택(call stack)이 무엇인지 설명하고 함수가 호출될 때 콜 스택에 변경사항을 추적한다.
- 함수 인자로 디폴트 값을 설정한다.
- 왜 프로그램을 작게 단일 목적 함수로 잘게 쪼개는지 설명한다.

함수 정의하기

화씨(Fahrenheit)에서 절대온도(Kelvin)로 온도를 변환하는 `fahr_to_kelvin` 함수를 정의하는 것부터 시작하자.

함수정의는 `def`로 시작하고 함수이름과 매개변수의 이름이 팔호 리스트로 다음에 위치한다. 함수 몸통부문(body)은 정의 아래에 통상 공백 4개로 들여쓰기 하는데 함수가 실행하는 문장이다.

함수를 호출할 때, 함수에 전달하는 값은 변수에 할당되어서 함수 내부에서 사용할 수 있다. 함수 내부에 리턴 문장(return statement))을 사용해서 요청하는 곳에 결과를 되돌린다.

작성한 함수를 실행하자. 본인이 작성한 함수를 호출하는 것은 다른 함수를 호출하는 것과 차이가 없다.

```
def fahr_to_kelvin(temp):  
    return ((temp - 32) * (5/9)) + 273.15  
  
print 'freezing point of water:', fahr_to_kelvin(32)  
print 'boiling point of water:', fahr_to_kelvin(212)  
  
freezing point of water: 273.15  
boiling point of water: 273.15
```

정의한 함수를 성공적으로 호출해서 반환한 값에 접근할 수 있다. 불행하게도 반환된 값이 올바른 것 같지 않다. 무엇이 잘못된 걸까?

함수 디버깅

디버깅은 올바르게 동작하지 않는 코드의 일부를 수정하는 것이다. 상기의 경우 `fahr_to_kelvin`가 잘못된 답을 주는 것을 알기 때문에 왜 그런지 이유를 파악해보자.

코드가 매우 큰 경우에 디버깅 과정을 도와주는 디버거(debugger)로 불리는 도구가 별도로 있다.

간단하고 짧은 함수여서 매개 변수를 골라, 함수를 작은 부분으로 쪼개서 각 부분에 값을 출력해서 디버그한다.

```
# We'll use temp = 212, the boiling point of water, which was incorrect
print "212 - 32:", 212 - 32
```

```
212 - 32: 180
```

```
print "(212 - 32) * (5/9):", (212 - 32) * (5/9)
```

```
(212 - 32) * (5/9): 0
```

문제가 $5/9$ 을 곱할 때 발생한다. $5/9$ 의 값이 0이기 때문이다.

```
5/9
```

```
0
```

컴퓨터는 숫자를 두 가지 중 한 가지로 저장한다. 정수(integers) 혹은 부동 소수점(floating-point numbers). 정수는 일반적으로 숫자를 세는 숫자이고, 부동 소수점은 실수와 실수의 소수부가 있다. 덧셈, 뺄셈, 곱셈은 정수나 부동 소수점이나 우리가 통상 예측하는 방식으로 동작하지만, 나눗셈은 다르게 동작한다. 정수를 다른 정수로 나누게 된다면, 나머지 없는 몫만 얻는다.

```
print '10/3 is:', 10/3
```

```
10/3 is: 3
```

다른 한편으로, 분모와 분자 둘중의 하나가 부동 소수점이라면, 컴퓨터는 부동 소수점 결과를 생성한다.

```
print '10.0/3 is:', 10.0/3
```

```
10.0/3 is: 3.33333333333
```

컴퓨터는 역사적인 이유로 이와 같이 동작한다. 정수 연산은 초기 컴퓨터에서 훨씬 빨랐고, 많은 상황에서 유용했다. 하지만, 여전히 혼란스럽다. 그래서 파이썬 3는 정수로 나눗셈을 할 때 부동 소수점 결과를 반환한다. 파이썬 2.7 버전을 사용하는 경우, 만약 $5/9$ 연산의 올바른 결과를 얻으려면, $5.0/9$, $5/9.0$ 혹은 다른 방식으로 연산을 작성해야 한다.

부동 소수점 결과를 얻는 다른 방식은 명시적으로 산출해야하는 결과를 컴퓨터에 지시한다. 숫자 중에 하나를 형변환(cast)한다.

```
print 'float(10)/3 is:', float(10)/3
```

```
float(10)/3 is: 3.33333333333
```

상기 방식의 장점은 변수와 함께 사용될 수 있다는 것이다. 직접 살펴보자.

```
a = 10
b = 3
print 'a/b is:', a/b
print 'float(a)/b is:', float(a)/b

a/b is: 3
float(a)/b is: 3.33333333333
```

a 혹은 b 를 다시 정의하는 것보다 더 쉽다는 것을 볼 수 있다.

지금까지 학습한 지식으로 `fahr_to_kelvin` 함수를 고쳐보자.

```

def fahr_to_kelvin(temp):
    return ((temp - 32) * (5.0/9.0)) + 273.15

print 'freezing point of water:', fahr_to_kelvin(32)
print 'boiling point of water:', fahr_to_kelvin(212)

freezing point of water: 273.15
boiling point of water: 373.15

```

잘 동작한다!

함수 조합하기

화씨온도를 절대온도로 어떻게 변환하는지 봤기 때문에 절대온도를 섭씨온도로 바꾸는 것은 쉽다.

```

def kelvin_to_celsius(temp):
    return temp - 273.15

print 'absolute zero in Celsius:', kelvin_to_celsius(0.0)

absolute zero in Celsius: -273.15

```

화씨온도에서 섭씨온도로 변환하는 것은 어떤가요? 공식을 적을 수도 있지만, 그럴 필요가 없다. 이미 작성한 두개의 함수를 조합(compose)할 수 있다.

```

def fahr_to_celsius(temp):
    temp_k = fahr_to_kelvin(temp)
    result = kelvin_to_celsius(temp_k)
    return result

print 'freezing point of water in Celsius:', fahr_to_celsius(32.0)

freezing point of water in Celsius: 0.0

```

어떻게 좀더 커다란 프로그램이 만들어지는지 첫번째 맛을 봤다. 기본 연산을 정의하고 원하는 효과를 얻기 위해서 이를 조합한다. 실제 함수는 상기 보여진 것보다 더 크다. 일반적으로 대략 6줄에서 20~30줄 정도 한다. 하지만 이보다 함수가 더 길거나 함수를 읽는 사람이 어떻게 동작하는지 이해할수 없는 것은 곤란하다.

도전 과제

1. 두 개의 문자열을 “더하는 것”은 사슬처럼 잇게 한다(concatenate). 즉, 'a' + 'b'는 'ab'이 된다. 두 개의 매개변수 `original`과 `wrapper`를 받아 `original`의 처음과 끝에 `wrapper`를 씌워 새로운 문자열을 반환하는 함수 `fence`를 작성하세요.

```
print fence('name', '*')
*name*
```

2. 만약 변수 `s`가 문자열이면, `s[0]`는 첫번째 문자이고, `s[-1]`은 마지막 문자가 된다. 입력문자열의 처음과 끝 문자로만 구성된 문자열을 반환하는 함수 `outer`를 작성하세요.

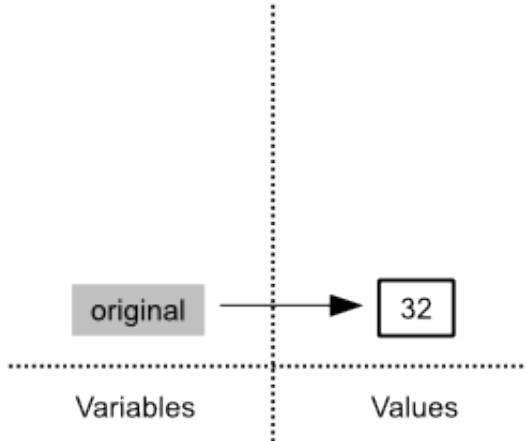
```
print outer('helium')
hm
```

콜 스택(Call Stack)

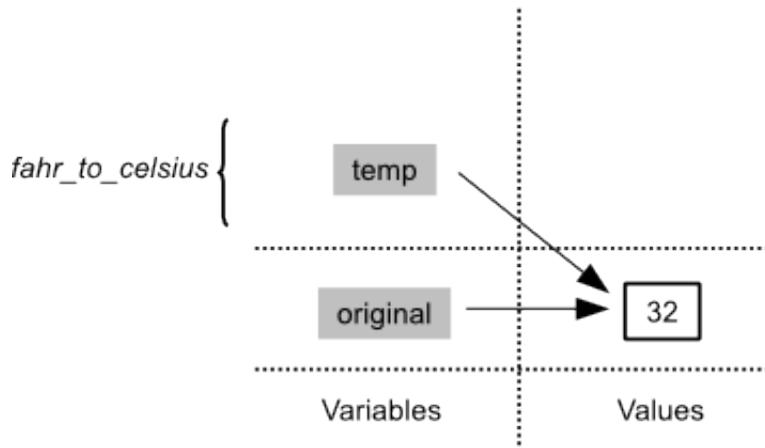
`fahr_to_celsius(32.0)`을 호출할 때 무엇이 생기는지 좀더 자세히 살펴보자. 좀 더 명확하기 하기 위해서, 변수에 초기값을 32.0으로 설정하고 결과를 `final`에 저장해서 출발해봅시다.

```
original = 32.0
final = fahr_to_celsius(original)
```

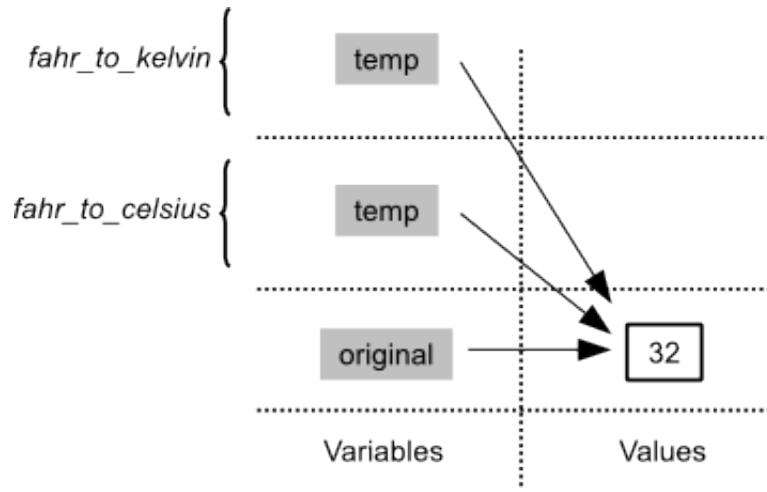
다음 다이어그램은 첫번째 행이 실행된 다음에 메모리가 어떻게 되지는 보여준다.



함수 `fahr_to_celsius`을 호출할 때, R은 변수 `temp`를 바로 생성하지는 않는다. 대신에 스택 프레임(stack frame)을 생성해서 `fahr_to_kelvin` 함수가 정의한 변수를 추적한다. 초기에 스택은 `temp` 값만을 가지고 있다.

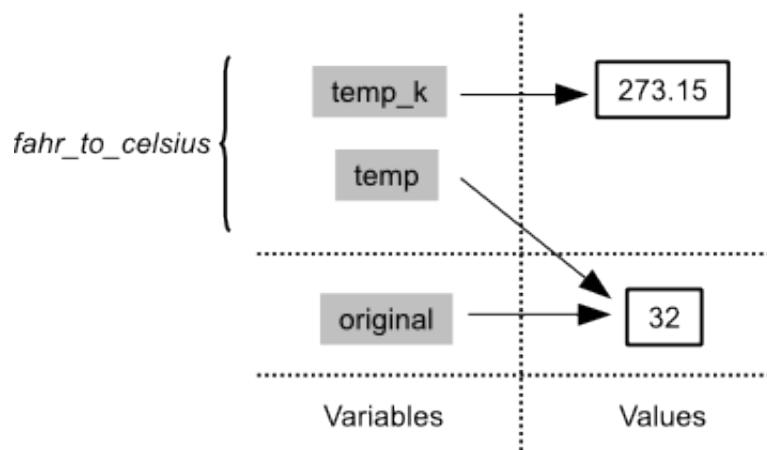


`fahr_to_celsius` 함수 내부에 `fahr_to_kelvin` 함수를 호출할 때, 파이썬은 또 다른 스택 프레임을 생성해서 `fahr_to_kelvin`의 변수를 저장한다.

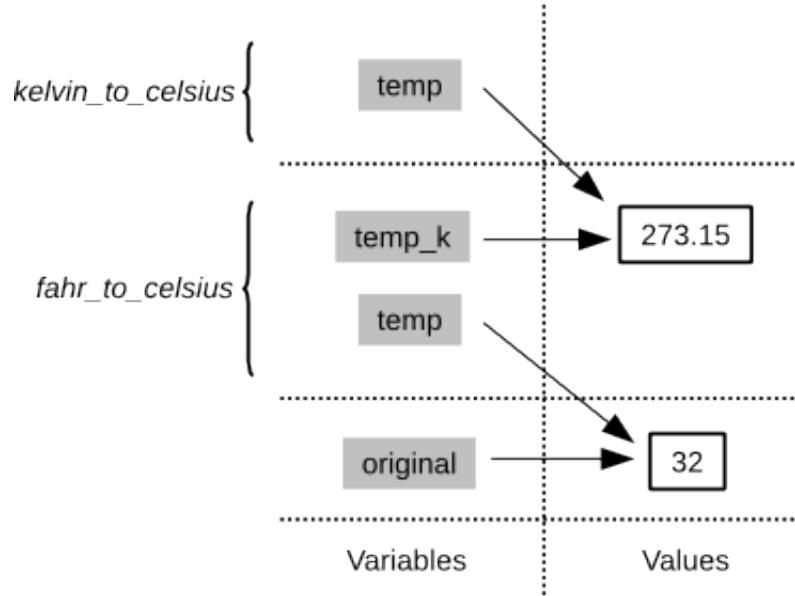


이제 `temp`로 불리는 동작하는 두개의 변수가 있다. 하나는 `fahr_to_celsius` 함수의 매개변수이고, 다른 하나는 `fahr_to_kelvin` 함수의 매개변수다. 프로그램의 같은 부분에 동일한 이름을 가진 변수 두개가 있는 것이 애매모호해서, 파이썬(그리고 다른 최신 프로그래밍 언어)은 각 함수 호출에 대해서 새로운 스택 프레임을 생성해서 다른 함수에서 정의된 변수와 구별되게 함수의 변수를 보관한다.

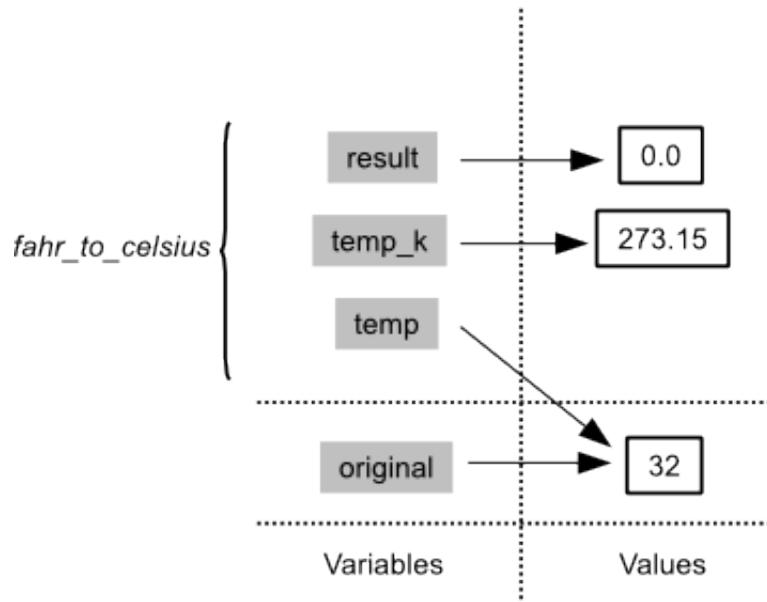
`fahr_to_kelvin` 함수 호출이 값을 반환할 때, 파이썬은 `fahr_to_kelvin` 함수의 스택 프레임을 사용한 후 버리고 절대 온도 정보를 보관하기 위해서 `fahr_to_celsius`에 대한 스택 프레임에 새로운 변수를 생성한다.



그리고 나서 `kelvin_to_celsius`을 호출하는데 함수의 변수를 저장할 스택 프레임을 생성한다는 의미다.

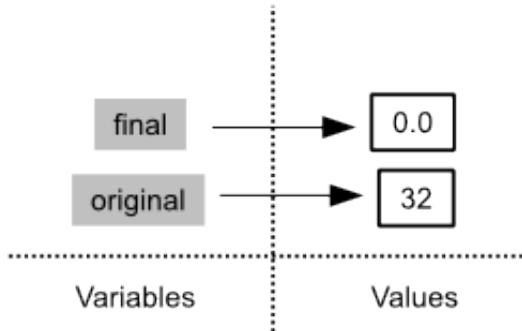


다시 한번, 파이썬은 `kelvin_to_celsius` 함수가 수행완료될 때 스택 프레임을 폐기한다. 그리고 `fahr_to_celsius` 함수를 위해 스택 프레임에 `result` 변수를 생성한다.



마지막으로, `fahr_to_celsius` 함수 수행이 완료될 때, 파이썬은 자신의 스택 프레임을 폐기하고 초기 시작한 스택 프레임에 있는 신규 변수 `final`에 결과값을

넣는다.



이 마지막 스택 프레임은 항상 존재해서 작성한 코드 중에 함수 외부에서 정의한 변수를 간직한다. 간직하지 않는 것은 다양한 스택 프레임에 있었던 변수다. 만약 함수가 수행 종료된 후에 `temp` 값을 얻고자 한다면, 파이썬은 그런 것은 없다고 회답한다.

```
print 'final value of temp after all function calls:', temp
```

```
-----  
NameError Traceback (most recent call last)  
<ipython-input-12-ffd9b4dbd5f1> in <module>()  
----> 1 print 'final value of temp after all function calls:', temp
```

```
NameError: name 'temp' is not defined
```

```
final value of temp after all function calls:
```

왜 이 모든 어려움으로 갈까요? 배열의 최대값과 최소값의 차이를 계산하는 `span`이라는 함수가 다음에 있다.

```
import numpy  
  
def span(a):  
    diff = a.max() - a.min()  
    return diff
```

```
data = numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
print 'span of data', span(data)

 20.0
```

`span` 함수는 값을 `diff` 변수에 할당함을 주목하세요. 염증 데이터 정보를 담고 있는 동일한 이름의 변수(`diff`)를 매우 사용할 수도 있다.

```
diff = numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
print 'span of data:', span(diff)

: 20.0
```

함수 호출 뒤에 변수 `diff`가 값 20.0을 갖게 되는 것을 기대하지 않는다. 그래서 `diff` 이름은 프로그램 메인 몸체부문에서 하는 것처럼 `span` 내부에 정의된 동일한 변수를 참조할 수 없다. 이 경우에 메인 프로그램에 `diff`와 다른 이름을 아마도 선택할 수 있지만, 변수의 값이 변경되는 경우마다 무슨 변수명이 사용되었는지를 살펴보기 위해 호출하는 NumPy의 모든 코드 행을 읽고 싶지는 않다.

여기서 기본적인 아이디어는 캡슐화(encapsulation)이고, 정확하고 이해하기 쉬운 프로그램을 작성하는 열쇠다. 함수가 하는 일은 몇개의 작업을 하나로 변환하는 것이어서 무언가를 하고자 할 때마다 수십개에서 수백개의 문장을 수행하는 대신에 단 하나의 함수 호출을 생각할 수 있다. 함수가 서로에게 간섭하지 않는다면 이 방식은 동작한다. 만약 서로 간섭하게 되면 다시 한번 세부사항에 주의를 기울여야 하고 급격하게 단기 기억에 과부하를 주게된다.

도전 과제

- 이전에 `fence`와 `outer` 함수를 작성했다. 다음을 실행할 때 콜 스택(call stack)이 어떻게 변하는지 다이어그램을 그려보세요.

```
print outer(fence('carbon', '+'))
```

테스팅과 문서화

함수에 명령어들을 넣어서 재사용할 수 있자마자, 작성한 함수가 제대로 동작하는지를 테스트할 필요가 있다. 어떻게 수행하는지 알아보기 위해서, 특정한 값 주위에 데이터를 중앙에 위치하게 하는 함수를 작성하자.

```
def center(data, desired):
    return (data - data.mean()) + desired
```

작성한 함수를 실제 데이터에 테스트할 수도 있으나, 값이 무엇이 되어야하는지 모르기 때문에 결과가 맞는지 구분하기가 어렵다. 대신에, NumPy를 사용해서 0으로 구성된 행렬을 생성하고 3 주위가 중심이 되게 하자.

```
z = numpy.zeros((2,2))
print center(z, 3)
```

```
[[ 3.  3.]
 [ 3.  3.]]
```

맞는 것처럼 보여서 실제 데이터에 중심(center)을 잡도록 하자

```
data = numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
print center(data, 0)

[[ -6.14875 -6.14875 -5.14875 ... , -3.14875 -6.14875 -6.14875]
 [-6.14875 -5.14875 -4.14875 ... , -5.14875 -6.14875 -5.14875]
 [-6.14875 -5.14875 -5.14875 ... , -4.14875 -5.14875 -5.14875]
 ... ,
 [-6.14875 -5.14875 -5.14875 ... , -5.14875 -5.14875 -5.14875]
 [-6.14875 -6.14875 -6.14875 ... , -6.14875 -4.14875 -6.14875]
 [-6.14875 -6.14875 -5.14875 ... , -5.14875 -5.14875 -6.14875]]
```

상기 출력으로부터 결과가 맞는지 구분하기 어렵다. 하지만, 확인을 할 수 있는 몇가지 테스트가 있다.

```

print 'original min, mean, and max are:', data.min(), data.mean(), data.max()
centered = center(data, 0)
print 'min, mean, and and max of centered data are:', centered.min(), centered.mean(), centered.max()

original min, mean, and max are: 0.0 6.14875 20.0
min, mean, and and max of centered data are: -6.14875 -3.49054118942e-15 13.85125

```

거의 맞는 것처럼 보인다. 원 평균은 약 6.1이였다. 그래서 0에서 하한은 약 -6.1이다. 중앙으로 위치한 데이터의 평균은 0은 아니지만, 무척 가깝다. 도전 과제에서 왜 그렇게 되었는지 탐색할 것이다. 좀 더 나아가 표준편차가 바뀌었는지 확인하자.

```

print 'std dev before and after:', data.std(), centered.std()

std dev before and after: 4.61383319712 4.61383319712

```

두 값이 동일하다. 하지만 6번째 소수점에서 차이가 있다면 알아채지 못할 것이다. 대신에 다음을 수행하자.

```

print 'difference in standard deviations before and after:', data.std() - centered.std()

difference in standard deviations before and after: -3.5527136788e-15

```

다시 차이가 매우 작다. 여전히 함수가 잘못될 가능성은 여전히 있다. 하지만, 분석으로 되돌릴 정도는 아닐 듯 하다. 하지만, 한 가지 더 작업이 있다. 후에 작성한 함수가 무엇을 위한 것이고 어떻게 사용하는지에 대해서 우리 자신에게 되새김되도록 함수의 문서화(documentation)를 위해서 문서를 작성해야 한다.

소프트웨어에 문서를 넣는 일반적인 방법은 다음과 같은 주석(comments)을 추가하는 것이다.

```

# center(data, desired): return a new array containing the original data centered around the mean
def center(data, desired):
    return (data - data.mean()) + desired

```

하지만, 더 나은 방법이 있다. 만약 함수의 첫번째 것이 변수에 할당되지 않은 문자열이라면, 그 문자열은 문서로서 함수에 덧붙여진다.

```
def center(data, desired):
    '''Return a new array containing the original data centered around the desired value.'''
    return (data - data.mean()) + desired
```

상기 것이 더 좋은데, 파이썬의 내장 도움말 시스템에 함수의 문서를 보여줄 수 있다.

```
help(center)
```

```
Help on function center in module __main__:
```

```
center(data, desired)
    Return a new array containing the original data centered around the desired value.
```

이와 같은 문자열을 docstring이라고 한다. 주석을 작성할 때 인용부호를 3번 사용할 필요는 없다. 하지만, 만약 인용부호를 3번 사용한다면, 여러줄에 걸쳐서 도움말을 쪼개 작성할 수 있다.

```
def center(data, desired):
    '''Return a new array containing the original data centered around the desired value.
    Example: center([1, 2, 3], 0) => [-1, 0, 1]'''
    return (data - data.mean()) + desired
```

```
help(center)
```

```
Help on function center in module __main__:
```

```
center(data, desired)
    Return a new array containing the original data centered around the desired value.
    Example: center([1, 2, 3], 0) => [-1, 0, 1]
```

도전 과제

1. `analyze` 함수를 작성해서 매개변수로 파일 이름을 받아서 [앞선 학습](#) 결과 (시간에 따른 염증의 평균값, 최소값, 최대값)를 그래프로 화면에 출력하도록 하세요. `analyze("inflammation-01.csv")` 결과는 이미 보여진 그래프를 생성해야 하지만, `analyze("inflammation-02.csv")` 결과는 두번째 데이터셋에 상응하는 그래프를 생성해야 한다. `docstring`으로 함수를 문서화하도록 확인하세요.
2. `rescale`함수를 작성해서 입력값으로 벡터를 받고, 0.0에서 1.0사이의 범위로 조정되게 해당 벡터를 반환하게 하세요. 만약 L 과 H 이 원 벡터이 하한과 상한이라면, v 의 치환값은 $(v-L) / (H-L)$ 이 되어야 한다. `docstring`으로 함수를 문서화하도록 확인하세요.
3. `help(numpy.arange)`과 `help(numpy.linspace)` 도움말 명령어를 실행해서 어떻게 함수를 사용하여 균등한 간격을 가진 값을 생성하는지 살펴보세요. 그리고 나서 이들 값을 사용하여 `rescale` 함수가 정상적으로 동작하는지 시험하세요.

초기 설정(Default) 정의

두 가지 방식으로 함수에 매개변수를 넘겨줬다. `span(data)`처럼 직접적으로, `numpy.loadtxt(fname='something.csv', delimiter=',')`과 같이 이름으로 넘겨졌다. 사실 매개변수를 `fname=` 없이 `loadtxt` 함수에 파일이름을 넘길 수 있다.

```
numpy.loadtxt('inflammation-01.csv', delimiter=',')  
  
array([[ 0.,  0.,  1., ...,  3.,  0.,  0.],  
       [ 0.,  1.,  2., ...,  1.,  0.,  1.],  
       [ 0.,  1.,  1., ...,  2.,  1.,  1.],  
       ...,  
       [ 0.,  1.,  1., ...,  1.,  1.,  1.],  
       [ 0.,  0.,  0., ...,  0.,  2.,  0.],  
       [ 0.,  0.,  1., ...,  1.,  1.,  0.]])
```

하지만, 여전히 `delimiter=`을 언급할 필요가 있다.

```
numpy.loadtxt('inflammation-01.csv', ',')  
  
-----  
TypeError Traceback (most recent call last)  
<ipython-input-26-e3bc6cf4fd6a> in <module>()  
----> 1 numpy.loadtxt('inflammation-01.csv', ',')  
  
/Users/gwilson/anaconda/lib/python2.7/site-packages/numpy/lib/npyio.pyc in loadtxt(fname, dt  
    775     try:  
    776         # Make sure we're dealing with a proper dtype  
--> 777         dtype = np.dtype(dtype)  
    778         defconv = _getconv(dtype)  
    779  
  
TypeError: data type "," not understood
```

무엇이 진행되는지 이해하고 작성한 함수를 사용하기 좀 더 쉽게 하기 위해서, 다음과 같이 `center` 함수를 다시 정의하자.

```
def center(data, desired=0.0):  
    '''Return a new array containing the original data centered around the desired value (0  
    Example: center([1, 2, 3], 0) => [-1, 0, 1]'''  
    return (data - data.mean()) + desired
```

중요 변경사항은 두번째 인자가 이제 `desired` 대신에 `desired = 0.0`이 되었다.
인자 두개로 가진 함수를 호출하면, 전과 동일한 방식으로 동작한다.

```
test_data = numpy.zeros((2, 2))  
print center(test_data, 3)  
  
[[ 3.  3.]  
 [ 3.  3.]]
```

하지만, 단지 하나의 인자로 함수를 호출할 수도 있다. 이 경우에 `desired`는 자동적으로 초기 설정값(default value) 0.0 이 할당된다.

```
more_data = 5 + numpy.zeros((2, 2))
print 'data before centering:', more_data
print 'centered data:', center(more_data)

data before centering: [[ 5.  5.]
 [ 5.  5.]]
centered data: [[ 0.  0.]
 [ 0.  0.]]
```

매우 편리하다. 만약 한 방식으로 동작하는 함수를 원하지만 때때로 다르게 동작할 필요가 있다면, 필요할 때만 매개변수를 넘기게 하고 정상적인 경우는 좀더 쉽게 하려고 초기 설정값을 넣는다.

다음 예제는 어떻게 파이썬이 값을 인자에 매칭하는지 보여준다.

```
def display(a=1, b=2, c=3):
    print 'a:', a, 'b:', b, 'c:', c

    print 'no parameters:'
display()
print 'one parameter:'
display(55)
print 'two parameters:'
display(55, 66)

no parameters:
a: 1 b: 2 c: 3
one parameter:
a: 55 b: 2 c: 3
two parameters:
a: 55 b: 66 c: 3
```

예제가 보여주듯이, 인자는 왼쪽에서 오른쪽으로 매칭된다. 그리고 명시적으로 값이 주어지지 않는 것은 초기 설정값을 갖는다. 인자를 넘길 때 값에 이름을 줌으로써 이런 행동을 오버라이드 override(override)할 수 있다.

```
print 'only setting the value of c'  
display(c=77)
```

```
only setting the value of c  
a: 1 b: 2 c: 77
```

상기 내용을 가지고, `numpy.loadtxt` 함수의 도움말을 살펴 보자.

```
help(numpy.loadtxt)
```

```
Help on function loadtxt in module numpy.lib.npyio:
```

```
loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None)  
Load data from a text file.
```

```
Each row in the text file must have the same number of values.
```

Parameters

`fname` : file or str

```
File, filename, or generator to read. If the filename extension is  
``.gz`` or ``.bz2``, the file is first decompressed. Note that  
generators should return byte strings for Python 3k.
```

`dtype` : data-type, optional

```
Data-type of the resulting array; default: float. If this is a  
record data-type, the resulting array will be 1-dimensional, and  
each row will be interpreted as an element of the array. In this  
case, the number of columns used must match the number of fields in  
the data-type.
```

`comments` : str, optional

```
The character used to indicate the start of a comment;
```

```
    default: '#'.

delimiter : str, optional
    The string used to separate values. By default, this is any
    whitespace.

converters : dict, optional
    A dictionary mapping column number to a function that will convert
    that column to a float. E.g., if column 0 is a date string:
    ``converters = {0: datestr2num}``. Converters can also be used to
    provide a default value for missing data (but see also `genfromtxt`):
    ``converters = {3: lambda s: float(s.strip() or 0)}``. Default: None.

skiprows : int, optional
    Skip the first `skiprows` lines; default: 0.

usecols : sequence, optional
    Which columns to read, with 0 being the first. For example,
    ``usecols = (1,4,5)` ` will extract the 2nd, 5th and 6th columns.
    The default, None, results in all columns being read.

unpack : bool, optional
    If True, the returned array is transposed, so that arguments may be
    unpacked using ``x, y, z = loadtxt(...)` `. When used with a record
    data-type, arrays are returned for each field. Default is False.

ndmin : int, optional
    The returned array will have at least `ndmin` dimensions.
    Otherwise mono-dimensional axes will be squeezed.
    Legal values: 0 (default), 1 or 2.
    .. versionadded:: 1.6.0
```

Returns

out : ndarray

Data read from the text file.

See Also

`load`, `fromstring`, `fromregex`

```
genfromtxt : Load data with missing values handled as specified.  
scipy.io.loadmat : reads MATLAB data files
```

Notes

This function aims to be a fast reader for simply formatted files. The `genfromtxt` function provides more sophisticated handling of, e.g., lines with missing values.

Examples

```
>>> from StringIO import StringIO # StringIO behaves like a file object  
>>> c = StringIO("0 1\n2 3")  
>>> np.loadtxt(c)  
array([[ 0.,  1.],  
       [ 2.,  3.]])  
  
>>> d = StringIO("M 21 72\nF 35 58")  
>>> np.loadtxt(d, dtype={'names': ('gender', 'age', 'weight'),  
...                      'formats': ('S1', 'i4', 'f4')})  
array([('M', 21, 72.0), ('F', 35, 58.0)],  
      dtype=[('gender', '|S1'), ('age', '<i4'), ('weight', '<f4')])  
  
>>> c = StringIO("1,0,2\n3,0,4")  
>>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)  
>>> x  
array([ 1.,  3.])  
>>> y  
array([ 2.,  4.])
```

많은 정보가 있지만, 가장 중요하는 부분은 처음 몇줄이다.

```
loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skiprows=0,  
        unpack=False, ndmin=0)
```

`loadtxt`는 하나의 매개변수 `fname`만 초기 설정을 갖지 않고, 다른 6개는 초기 설정을 가지는 것을 보여준다. 만약 함수를 다음과 같이 호출한다면,

```
numpy.loadtxt('inflammation-01.csv', ',', ',')
```

파일 이름이 사용자가 원하는 위치인 `fname`에 할당되지만, 구분자 문자열 `' , '`은 `delimiter` 대신에 `dtype`에 할당된다. 왜냐하면, `dtype`이 매개변수 리스트에 두 번째 위치에 있기 때문이다. 이러한 이유로 파일이름에 대해서 `fname=`을 명기할 필요는 없지만, 두번째 매개변수에 대해서는 `delimiter=`을 명기해야 한다.

도전 과제

1. `rescale` 함수를 재작성해서 초기 설정으로 데이터가 0.0에서 1.0 사이에 놓게 하세요. 하지만 호출자가 원한다면, 하한과 상한을 지정할 수 있게하세요. 옆 사람과 구현한 것을 비교하세요. 두 함수가 항상 같은 방식으로 동작하나요?

주요점

- `def name(...params...)`을 사용해서 함수를 작성하라.
- 함수의 몸통부분은 들여쓰기 한다.
- `name(...values...)`을 사용해서 함수를 호출하라.
- 숫자는 정수 혹은 부동 소수점으로 저장된다.
- 정수 나눗셈은 결과값은 몽만 산출하고 실수의 소수부는 산출하지 않는다.
- 함수가 매번 호출될 때마다, 신규 스택 프레임이 콜 스택(call stack)에 생성되어 인자와 로컬 변수를 가진한다.
- 파이썬은 상위 수준에서 변수를 찾기 전에 현재 스택 프레임에서 변수를 찾는다.
- 무언가의 도움말을 보기 위해서 `help(thing)`을 사용한다.
- 함수에 도움말을 제공하려면 함수에 `docstring`을 넣어라.

- 매개변수 목록에 `name = value`을 사용해서 함수를 정의할 때, 인자에 대해서 초기 설정값을 명기하라.
- 매개변수는 이름, 위치, 혹은 생략하고 넘길 수 있다. 생략하는 경우 초기설정값이 사용된다.

다음 단계 `analyze` 함수를 가지고 하나의 데이터셋을 시각화할 수 있다. 다음과 같이 12개 데이터넷 모두를 탐색하려고 사용할 수 있다.

```
analyze('inflammation-01.csv')
analyze('inflammation-02.csv')
...
analyze('inflammation-12.csv')
```

하지만, 12개 파일이름을 모두 정확하게 타이핑할 가능성은 크지 않다. 그리고 만약 백개 파일이 더 있다면 더 난감하게 될 것이다. 필요한 것은 파이썬이 각 파일에 대해서 한번씩 무엇을 수행하게 하는 것이다. 이것이 다음 학습주제다.

다수의 데이터셋 분석하기

하나의 데이터셋에 대해서 일별 염증율의 최소값, 평균값, 최대값의 그래프를 생성하는 함수 `analyze`를 만들었다.

```
%matplotlib inline

import numpy as np
from matplotlib import pyplot as plt

def analyze(filename):
    data = np.loadtxt(fname=filename, delimiter=',')
```

```

plt.figure(figsize=(10.0, 3.0))

plt.subplot(1, 3, 1)
plt.ylabel('average')
plt.plot(data.mean(0))

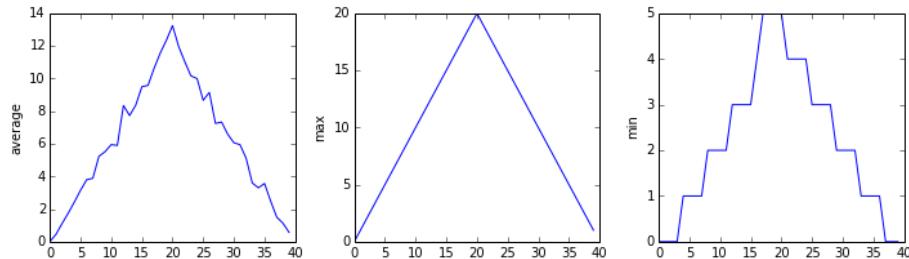
plt.subplot(1, 3, 2)
plt.ylabel('max')
plt.plot(data.max(0))

plt.subplot(1, 3, 3)
plt.ylabel('min')
plt.plot(data.min(0))

plt.tight_layout()
plt.show()

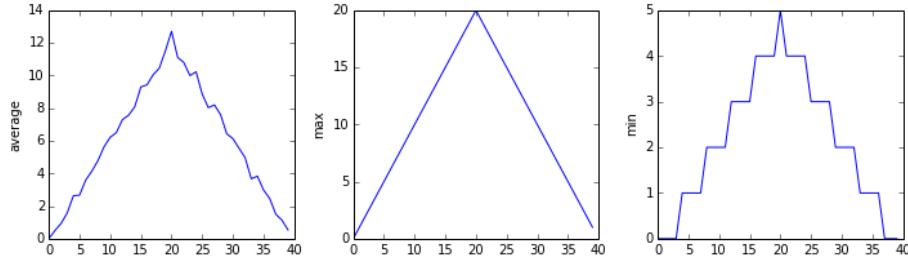
```

```
analyze('inflammation-01.csv')
```



작성한 함수를 사용해서 다른 데이터셋도 하나씩 하나씩 분석할 수 있다.

```
analyze('inflammation-02.csv')
```



하지만, 지금 당장 12개의 데이터셋이 있고 앞으로 더많은 데이터가 있을 것이다. 하나의 문장으로 모든 데이터에 대해서 그래프를 생성하고자 한다. 이를 위해서 어떻게 반복하는지를 컴퓨터를 학습시켜야 한다.

목표

- for 루프가 무엇을 수행하는지 설명한다.
- 올바르게 for 루프를 작성해서 간단한 계산을 반복한다.
- 루프가 실행될 때, 루프 변수의 변경을 추적한다.
- for 루프로 다른 변수가 생성될 때, 다른 변수의 변경사항도 추적한다.
- 리스트(list)가 무엇인지 설명한다.
- 간단한 값의 리스트를 생성하고 인덱스한다.
- 라이브러리 함수를 사용해서 간단한 패턴이 매칭되는 파일 목록을 얻는다.
- for 루프를 사용해서 다수의 파일을 처리한다.

For 루프

문장에 각 단어를 출력하고자 한다고 가정하자. 한 방법은 6개의 `print` 문을 사용한다.

```
def print_characters(element):
    print element[0]
    print element[1]
    print element[2]
    print element[3]
```

```
print_characters('lead')
```

```
l  
e  
a  
d
```

하지만, 두개의 이유로 좋지 못한 접근법이다.

1. 확장성이 좋지 않다. 만약 수백개 문자로 구성된 문자열의 문자를 화면에 출력한다면, 단순하게 타이핑하는게 더 좋을 것이다.
2. 강건하지가 못하다. 만약 조금 더 긴 벡터를 주면 데이터의 일부분만 화면에 출력한다. 만약 조금 짧은 입력을 준다면, 오류를 생성하는데 이유는 존재하지 않는 문자에 대해서 요청했기 때문이다.

```
print_characters('tin')
```

```
-----  
IndexError                                     Traceback (most recent call last)  
<ipython-input-13-5bc7311e0bf3> in <module>()  
----> 1 print_characters('tin')
```

```
<ipython-input-12-11460561ea56> in print_characters(element)  
      3     print element[1]  
      4     print element[2]  
----> 5     print element[3]  
      6  
      7 print_characters('lead')
```

```
IndexError: string index out of range
```

```
t  
i  
n
```

좀더 좋은 접근법이 다음에 있다.

```
def print_characters(element):
    for char in element:
        print char

print_characters('lead')
```

좀더 짧고 더 강건하다. 100개 문자열의 각 문자를 화면에 출력하는 것보다 확실히 짧다.

```
print_characters('oxygen')
```

`print_characters` 함수의 개선된 버전은 연산을 반복하기 위해서 `for` 루프(for loop)를 사용한다. 이 경우에 각각에 대해서 한번만 출력한다. 루프의 일반적인 형태는 다음과 같다.

```
for variable in collection:
    do things with variable
```

원하는 이름으로 루프 변수(loop variable)를 할 수 있으나, 루프가 시작하는 라인의 끝에 콜론(:)이 있어야 하고 루프의 몸통 부문은 들여쓰기를 해야한다.

반복적으로 변수를 생성하는 또 다른 루프가 있다.

```
length = 0
for vowel in 'aeiou':
    length = length + 1
print 'There are', length, 'vowels'
```

상기 작은 프로그램의 실행을 단계별로 추적할 가치가 있다. 'aeiou'에 5개 문자가 있어서, 3번 라인의 문장은 5번 실행된다. 첫번째 루프를 돌 때, `length`는 0(첫번째 라인에 할당된 값)이고, `vowel`은 'a'다. `length`의 이전 값에 1을 더해서 1이 되고, 새로운 값을 참조하여 `length` 값을 생성한다. 다음번에는 `vowel`이 'e'가되고 `length`는 1이 되고, `length`을 생성하여 2가 된다. 3회 더 생성한 후에

`length`는 5가 된다. 파이썬이 처리할 더 이상의 값이 '`'aeiou'`'에 남아있지 않아서, 루프는 종료되고, 4번째 라인의 `print`문이 최종값을 출력한다.

루프 변수는 루프의 상태를 기록하기 위해 사용되는 단지 변수에 불과하다. 루프가 종료된 후에도 여전히 존재한다. 또한 루프 변수와 마찬가지로 앞에서 정의한 변수를 재사용할 수 있다.

```
letter = 'z'  
for letter in 'abc':  
    print letter  
print 'after the loop, letter is', letter
```

문자열의 길이를 찾는 것은 일반적인 연산이라 파이썬은 `len`으로 불리는 내장 함수가 있음을 주목하세요.

```
print len('aeiou')
```

`len` 함수는 여러분이 스스로 작성하는 다른 어떤 함수보다도 훨씬 빠르고 두줄 루프보다 훨씬 읽기 쉽다. `len` 함수를 사용해서 아직 만나보지 못한 많은 다른 것의 길이도 알 수 있다. 그런 이유로 사용할 수 있다면 `len` 함수를 항상 사용해야 한다.

도전 과제

- 파이썬에는 숫자 리스트를 생성하는 `range` 내장 함수가 있다. `range(3)`는 `[0, 1, 2]`, `range(2, 5)`는 `[2, 3, 4]` 숫자 리스트를 생성한다. `range`를 사용하여 `,N개` 숫자를 출력하는 함수를 작성하세요.

```
print_N(3)  
1  
2  
3
```

- 누승(Exponentiation)은 파이썬의 내장함수다.

```
print 5**3  
125
```

동일한 결과를 계산하는 `pow` 함수도 있다. 동일한 결과를 계산하기 위해서 루프를 사용하는 `expo` 함수를 작성하세요.

3. 벡터 값의 합을 계산하는 `total` 함수를 작성하세요. R에는 내장함수 `sum`이 있어서 동일한 계산을 대신할 수 있지만, 이번 도전과제에는 사용하지 마세요. 문자열을 입력으로 받는 `rev` 함수를 작성해서 역순으로된 문자를 가지는 새로운 문자열을 생성하게 하세요.

```
print rev('Newton')  
notweN
```

항상 그렇지만, docstring을 반드시 포함하도록 하세요.

리스트(Lists)

`for` 루프가 연산을 많이 하는 방법이듯이, 리스트는 많은 값을 저장하는 방식이다. NumPy 배열과는 다르게, 리스트는 파일 언어에 내장되었다. 꺪쇄 팔호에 값을 넣어서 리스트를 생성한다.

```
odds = [1, 3, 5, 7]  
print 'odds are:', odds
```

인덱스를 사용해서 리스트에서 각각의 요소값을 선택한다.

```
print 'first and last:', odds[0], odds[-1]
```

리스트에 루프를 돌리게 되면, 루프 변수가 루프를 돌 때 한번에 요소값에 할당된다.

```
for number in odds:  
    print number
```

리스트와 문자열 사이의 중요한 차이점이 있다. 리스트의 값은 변경할 수 있지만, 문자열의 문자는 변경할 수 없다. 예를 들어,

```

names = ['Newton', 'Darwing', 'Turing'] # typo in Darwin's name
print 'names is originally:', names
names[1] = 'Darwin' # correct the name
print 'final value of names:', names

```

상기 리스트는 동작하지만,

```

name = 'Bell'
name[0] = 'b'

```

문자열에서는 동작하지 않는다.

변-변-변-변경(Ch-Ch-Ch-Changes) 변경될 수 있는 데이터는 변경가능(mutable)하다고 하는 반면에 변경할 수 없는 데이터는 변경불가능(immutable)하다고 한다. 문자열과 마찬가지로 숫자도 변경불가능하다. 숫자 0 가 값 1을 갖도록 하거나 그 반대로 만들 수 있는 방법은 없다. (최소한 파이썬은 아니지만, 사용자가 이와 같은 것을 할 수 있게 하는 언어가 있기는 하지만, 예측컨대 혼동스러운 결과를 줄 수 있다.) 반대로 리스트와 배열은 변경가능해서, 생성된 뒤에도 변경할 수 있다. 데이터를 변경하는 프로그램은 변경하지 못하는 프로그램보다 이해하기 더 어려울 수 있는데 이유는 최종 값이 무엇인지를 알아내기 위해서 코드의 많은 행을 읽고 머릿속으로 요약해야 하기 때문이다. 반대로, 작은 변경사항이 생길 때마다 원본과 거의 동일한 복사본을 생성하는 대신에 데이터를 변경하는 프로그램은 훨씬 더 효율적이다.

요소(element)에 할당하는 것과는 별도로 리스트의 내용을 변경하는 방법은 많이 있다.

```

odds.append(11)
print 'odds after adding a value:', odds

del odds[0]
print 'odds after removing the first element:', odds

```

```
odds.reverse()  
print 'odds after reversing:', odds
```

도전 과제

1. `total`이라는 함수를 작성해서 리스트의 모든 값의 합을 계산하도록 하세요.
(파이썬이 `sum`이라는 내장함수가 있지만 이번 도전과제로 사용하지 마세요.)

다수 파일 처리하기

이제 모든 데이터 파일의 처리에 필요한 거의 모든 것을 갖추었다. 빠진 단 한가지는 다소 유쾌하지 않은 이름을 가진 라이브러리다.

```
import glob
```

`glob` 라이브러리는 동일하게 `glob`이라고 불리는 단 한개의 함수가 포함되어 있다. 파일 이름을 패턴과 매칭하여 파일을 찾아내는 함수다. 패턴을 문자열로 전달하는데 와일드 카드 문자 *은 0 혹은 그 이상의 문자를 매칭하는 반면에 ?은 임의의 한 문자만 매칭한다. 지금까지 생성한 모든 IPython Notebooks의 이름을 얻는데 상기 와일드 카드 문자를 사용한다.

```
print glob.glob('*.ipynb')
```

```
['01-numpy.ipynb', '02-func.ipynb', '03-loop.ipynb', '04-cond.ipynb', '05-defensive.ipynb',
```

혹은, 모든 csv 파일 목록을 출력하기 위해서 와일드 카드 문자를 사용한다.

```
print glob.glob('*.csv')
```

```
['inflammation-01.csv', 'inflammation-02.csv', 'inflammation-03.csv', 'inflammation-04.csv',
```

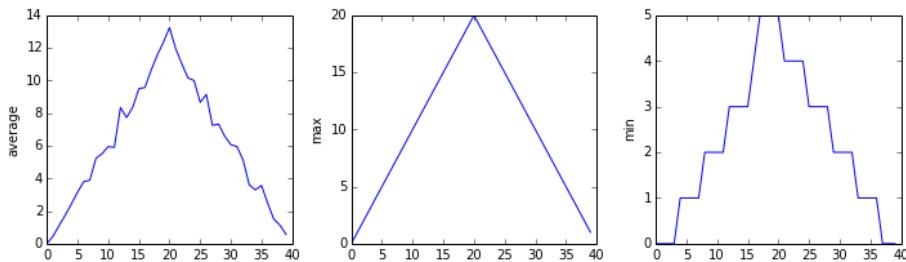
상기 예제가 보여주듯이, `glob.glob`의 결과는 문자열 리스트로 각 파일이름에 대해서 뭔가를 하기 위해서 루프를 반복해야 한다는 것이다. 지금 사례의 경우, 작업하려는 것은 `analyze` 함수다. 리스트의 첫 3개 파일을 분석해서 시험해보자.

```

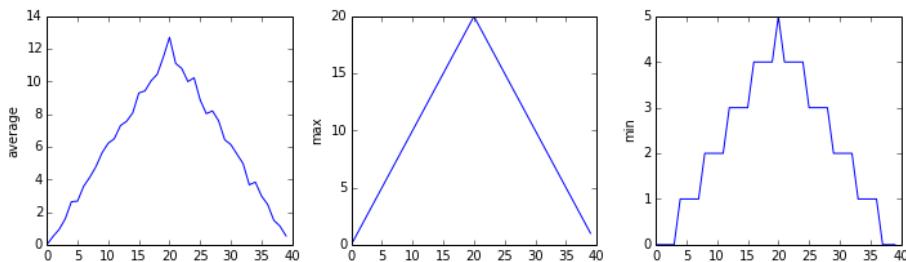
filenames = glob.glob('*.csv')
filenames = filenames[0:3]
for f in filenames:
    print f
    analyze(f)

```

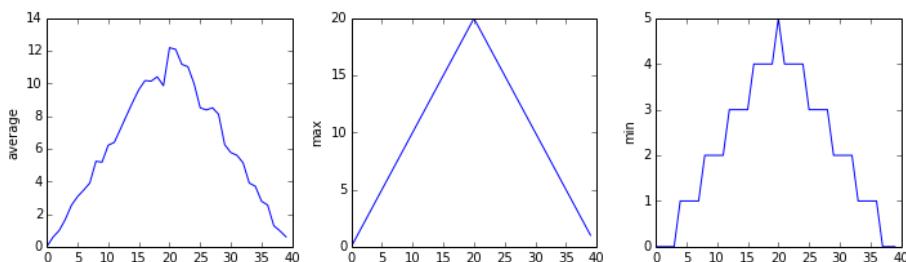
inflammation-01.csv



inflammation-02.csv



inflammation-03.csv



물론, 데이터셋의 최대값은 처음과 동일한 경사를 보여주고, 최소값은 동일한 계단구조를 보여준다.

도전 과제

- 파일 이름 패턴을 인자로만 받는 `analyze_all` 함수를 작성해서 파일 이름과 패턴이 매칭되는 파일에 대해서 `analyze`를 실행하세요.

주요점

- `for variable in collection`을 사용해서 한번에 하나씩 요소를 처리하세요.
- for 루프의 몸통 부문은 들여쓰기를 해야한다.
- 다른 값을 담고 있는 것의 길이를 알기 위해서 `len(thing)`을 사용하세요.
- `[value1, value2, value3, ...]` 은 리스트를 생성한다.
- 문자열과 배열과 동일한 방식으로 리스트를 인덱싱하고 슬라이싱한다.
- 리스트는 변경가능하다. 즉, 값이 변경될 수 있다.
- 문자열은 변경가능하지 않다. 즉, 문자열의 문자는 변경될 수 없다.
- `glob.glob(pattern)`을 사용해서 파일 이름과 패턴이 매칭되는 파일 목록을 생성하세요.
- 와일드 카드 `*`을 사용하여 패턴에서 0 혹은 그 이상 문자를 매칭하고, `?`은 임의의 한 문자만 매칭하는데 사용한다.

다음 단계 이제 원래 문제를 해결했다. 단일 명령문으로 임의 데이터 파일을 분석할 수 있다. 좀더 중요한 것은 프로그래밍에서 가장 중요한 두가지 아이디어를 경험했다.

- 함수를 사용해서 코드를 재사용하기 쉽게하고 이해하기 쉽게 했다.
- 리스트와 배열을 사용해서 관련된 값을 저장했고, 루프를 사용하여 데이터에 대한 연산을 반복했다.

한가지 더 소개할 큰 아이디어 있다. 그리고 나서 다시 돌아가서 첫번째 데이터셋을 표현하는데 사용된 것과 같은 히트맵(heat map)을 생성한다.

조건 선택

앞선 학습에서 어떻게 데이터를 다루고, 함수를 정의하고, 반복하는지를 배웠다. 하지만, 지금까지 작성한 프로그램은 무슨 데이터가 주어지든지 관계없이 항상 동일한 것을 수행한다. 프로그램이 다루는 값에 기반하여 선택하도록 만들고 싶다. 프로그램이 무슨 결정을 하는지 이해를 돋기 위해서, 어떻게 컴퓨터가 이미지(image)를 다루는지 살펴보면서 시작하자.

목표

- 색상 블록에서 간단한 “이미지(image)”를 생성한다.
- RGB 모델이 어떻게 색상을 표현하는지 설명한다.
- 튜플(tuple)과 리스트(list)의 차이점과 유사점을 설명한다.
- `if`, `elif`, `else` 분기를 포함하는 조건문을 작성한다.
- `and`와 `or`를 포함하는 표현식을 올바르게 평가한다.
- 중첩 루프와 조건문을 포함하는 코드를 올바르게 작성하고 해석한다.
- 자주 변경되는 코드를 함수에 넣어 작성하는 장점을 설명한다.

이미지 그리드 (Image Grids)

`ipythonblocks`으로 불리는 라이브러리를 사용하여 간단한 히트 맵(heat map)을 생성해보자. 첫번째 단계는 자신만의 “이미지(image)”를 직접 생성하는 것이다.

```
from ipythonblocks import ImageGrid
```

앞에서 살펴본 `import` 문장과는 다르게, 전체 `ipythonblocks` 라이브러리를 로드하지 않는다. 대신에, 라이브러리에서 `ImageGrid`만 로드한다. 왜냐하면 지금 당장 필요한 유일한 것이기 때문이다.

`ImageGrid`가 로드되면, 매우 단순한 색깔 격자(grid)를 생성하는데 사용할 수 있다.

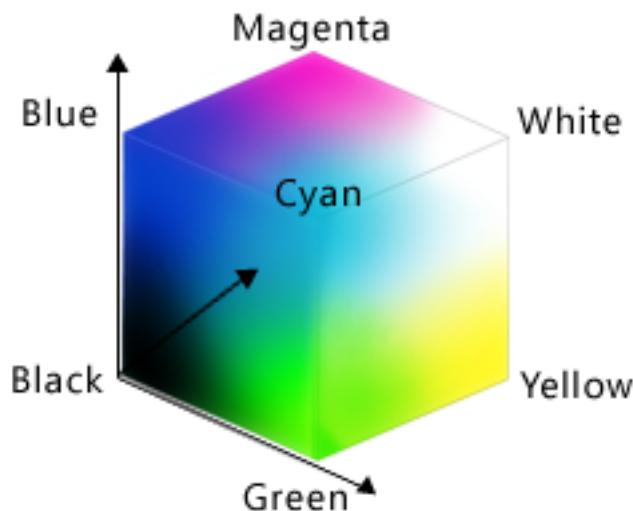
```
grid = ImageGrid(5, 3)
grid.show()
```

NumPy 배열처럼, `ImageGrid`는 몇몇 속성(property)이 있어서 관련 정보를 저장할 수 있다.

```
print 'grid width:', grid.width
print 'grid height:', grid.height
print 'grid lines on:', grid.lines_on

grid width: 5
grid height: 3
grid lines on: True
```

격자를 가지고 할 수 있는 명백한 것은 셀안에 색깔이지만, 이를 위해서 먼저 어떻게 컴퓨터가 색깔을 표현하는지 알 필요가 있다. 가장 일반적인 방식은 RGB 방식이고, RGB는 Red, Green, Blue를 나타낸다. RGB는 가색 모델(additive color model)이다. 모든 색은 적색, 녹색, 청색의 세기를 일부 조합한 것이다. 이 세가지 색을 직육면체의 각 축으로 생각할 수 있다.



RGB 색깔은 다중 값의 예이다. 직교 좌표계처럼 몇개의 부분으로 구성된다. 파이썬에서 다중 값을 표현하는 방식은 튜플(tuple)을 사용하는 것인데, 리스트에서 사용한 꺼쇠 괄호 대신에 괄호를 사용하여 표현한다.

```
position = (12.3, 45.6)
print 'position is:', position
color = (10, 20, 30)
print 'color is:', color

position is: (12.3, 45.6)
color is: (10, 20, 30)
```

리스트와 배열에서와 동일한 방식으로 인덱스를 사용하여 튜플에서 요소를 선택한다.

```
print 'first element of color is:', color[0]

first element of color is: 10
```

하지만, 리스트와 배열과 달리 튜플은 생성된 후에는 변경할 수 없다. 전문 용어로 튜플은 변경불가능(immutable)하다.

```
color[0] = 40
print 'first element of color after change:', color[0]
```

```
TypeError                                     Traceback (most recent call last)
<ipython-input-11-9c3dd30a4e52> in <module>()
----> 1 color[0] = 40
      2 print 'first element of color after change:', color[0]
```

```
TypeError: 'tuple' object does not support item assignment
```

튜플로 RGB 색깔을 표현한다면, 적색, 녹색, 청색 구성요소는 0 ~ 255 사이의 값을 취한다. 상한이 약간 이상하게 보일지 모르지만, 8-비트 바이트 (즉, $2^8 - 1$)로 표현할

수 있는 가장 큰 숫자다. 이 방식은 거의 대부분의 사람의 눈을 속일만큼 충분한 색채의 농담을 주면서 컴퓨터로 색깔을 다루기 쉽게 한다.

실제로 몇몇 RGB 색깔이 어떻게 보이는지 살펴보자.

```
row = ImageGrid(8, 1)
row[0, 0] = (0, 0, 0)    # no color => black
row[1, 0] = (255, 255, 255) # all colors => white
row[2, 0] = (255, 0, 0) # all red
row[3, 0] = (0, 255, 0) # all green
row[4, 0] = (0, 0, 255) # all blue
row[5, 0] = (255, 255, 0) # red and green
row[6, 0] = (255, 0, 255) # red and blue
row[7, 0] = (0, 255, 255) # green and blue
row.show()
```

(0,255,0)같은 단순한 색깔은 약간의 실습으로 쉽게 해독할수 있지만, (214,90,127)은 무슨 색깔일까? 사용자를 돋기 위해서, ipythonblocks 라이브러리는 show_color 함수를 제공한다.

```
from ipythonblocks import show_color
show_color(214, 90, 127)
```

또한 표준 색상표도 제공한다.

```
from ipythonblocks import colors
c = ImageGrid(3, 2)
c[0, 0] = colors['Fuchsia']
c[0, 1] = colors['Salmon']
c[1, 0] = colors['Orchid']
c[1, 1] = colors['Lavender']
c[2, 0] = colors['LimeGreen']
c[2, 1] = colors['HotPink']
```

```
c.show()
```

도전 과제

1. 아래 코드에서 _____ 을 채워 어두운 청색에서 검정색으로 색깔이 변하는 막대를 만드세요.

```
bar = ImageGrid(10, 1)
for x in range(10):
    bar[x, 0] = (0, 0, _____)
bar.show()
```

2. 왜 컴퓨터는 원색으로 적색, 녹색, 청색을 사용할까요?

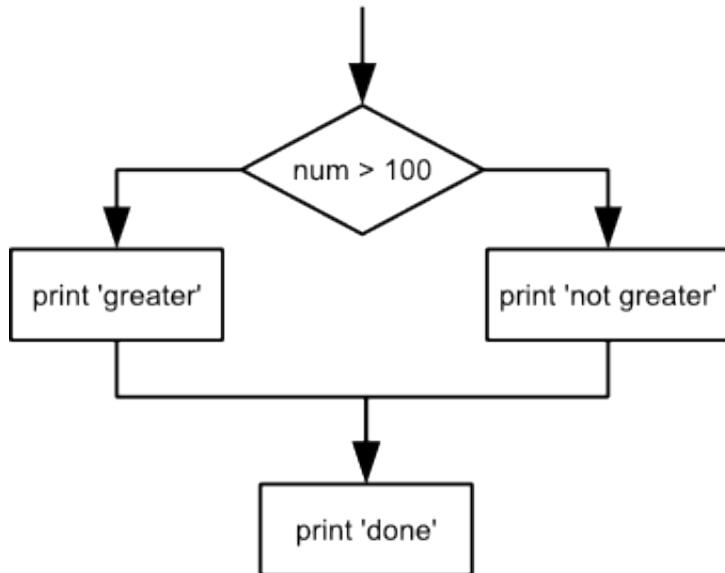
조건문 (Conditionals)

작성한 히트 맵을 생성하기 위해서 필요한 또 다른 사항은 데이터 값에 따라 색깔을 선택하는 방식이다. 파이썬이 이를 위해서 제공하는 도구는 조건문(conditional statement)이라고 불리고, 다음과 같다.

```
num = 37
if num > 100:
    print 'greater'
else:
    print 'not greater'
print 'done'

not greater
done
```

코드의 두번째 줄이 `if`문을 사용하여 파이썬에게 사용자가 선택을 하고 싶다고 전 한다. 만약 다음 시험(test)가 참(true)이면, `if`문의 몸통부분(바로 아래 들여쓰기한 행들)이 실행된다. 만약 시험이 거짓(false)이면, `else`의 몸통부분이 대신 실행된다. 오직 하나와 다른 것만이 실행된다.



조건문이 반드시 `else`를 포함할 필요는 없다. 만약 `else`가 없는 상태에서 시험(test)가 거짓이라면 파이썬은 아무것도 수행하지 않는다.

```
num = 53
print 'before conditional...'
if num > 100:
    print '53 is greater than 100'
print '...after conditional'

before conditional...
...after conditional
```

또한, 두가지 이상의 선택사항이 있다면, `elif`("else if")를 사용하여 몇개의 시험을 연쇄적으로 할 수 있다. 이렇게 하는 것이 숫자부호를 반환하는 함수 작성을 단순하게 한다.

```
def sign(num):
```

```

if num > 0:
    return 1
elif num == 0:
    return 0
else:
    return -1

print 'sign of -3:', sign(-3)

sign of -3: -1

```

상기 코드에서 주목할 중요한 사항은 동치 시험을 위해 한개의 등호 기호 대신에 등호 기호(==) 두개를 사용한 것이다. 왜냐하면 하나의 등호 기호는 할당을 의미한다. 이러한 관습은 C 언어에서 전해졌고, 많은 다른 프로그래밍 언어가 동일한 방식으로 동작하기 때문에, 익숙해질 필요는 있다.

`and` 와 `or`를 사용해서 시험을 조합할 수도 있다. `and`는 양쪽이 모두 참일 때만 참이다.

```

if (1 > 0) and (-1 > 0):
    print 'both parts are true'
else:
    print 'one part is not true'

one part is not true

```

반면에 `or`는 양쪽 중 하나만 참이면 참이다.

```

if (1 < 0) or ('left' < 'right'):
    print 'at least one test is true'

at least one test is true

```

이 경우, 둘 중에 하나는 둘중 하나 혹은 둘다를 의미한다. 하지만, 이쪽이나 저쪽 혹은 둘다 아니다를 의미하지는 않는다.

도전 과제

- True와 False가 파이썬에서 참과 거짓으로 사용되는 유일한 값은 아니다.

사실 임의의 값도 if 혹은 elif에 사용될 수 있다. 다음의 코드를 읽고 실행한 후에, 어느 값이 참이고, 어느 값이 거짓인지에 대한 규칙을 설명하세요. (만약 조건문의 몸통 부문이 하나의 문장으로 표현된다면, if와 같은 행에 코드를 작성할 수 있음을 주목하세요.)

```
if '': print 'empty string is true'  
if 'word': print 'word is true'  
if []: print 'empty list is true'  
if [1, 2, 3]: print 'non-empty list is true'  
if 0: print 'zero is true'  
if 1: print 'one is true'
```

- 만약 첫번째 매개변수가 두번째 매개변수의 10% 내에 들면 True를 반환하고 그렇지 않으면 False를 반환하는 near 함수를 작성하세요. 여러분이 작성한 코드와 동료의 코드와 비교하세요. 모든 가능한 숫자 쌍에 대해서 동일한 결과를 반환하나요?

중첩 (Nesting)

인식해야 될 또 다른 사항은 if 문이 함수와 쉽게 조합되듯이 루프와도 조합될 수 있다는 것이다. 예를 들어, 리스트의 양수를 더하고자 한다면 다음과 같이 코드를 작성한다.

```
numbers = [-5, 3, 2, -1, 9, 6]  
total = 0  
for n in numbers:  
    if n >= 0:  
        total = total + n  
print 'sum of positive values:', total  
  
sum of positive values: 20
```

동등하게 하나의 루프 안에서 양수의 합과 음수의 합도 계산할 수 있다.

```

pos_total = 0
neg_total = 0
for n in numbers:
    if n >= 0:
        pos_total = pos_total + n
    else:
        neg_total = neg_total + n
print 'negative and positive sums are:', neg_total, pos_total

negative and positive sums are: -6 20

```

심지어 루프 안에 루프를 하나 더 놓을 수도 있다.

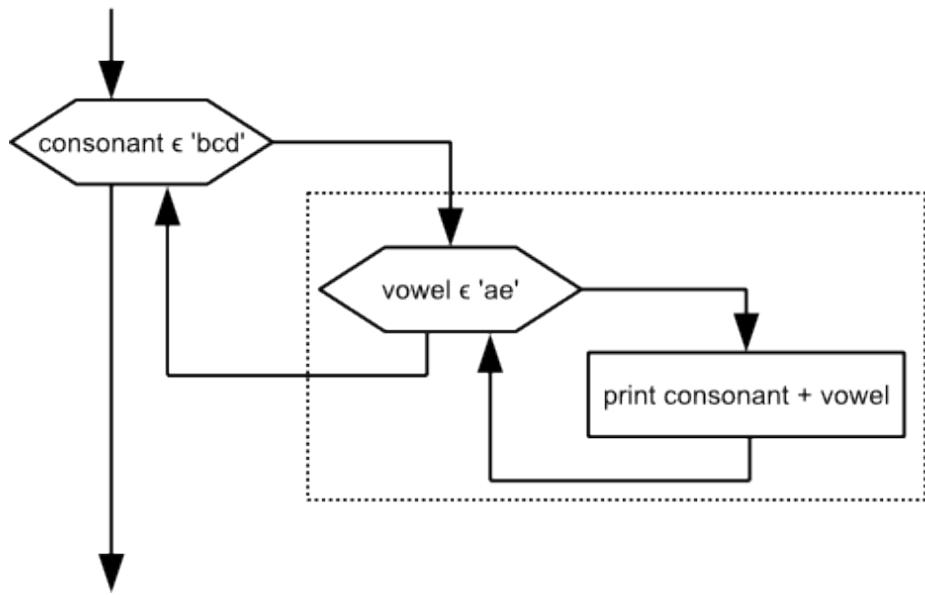
```

for consonant in 'bcd':
    for vowel in 'ae':
        print consonant + vowel

ba
be
ca
ce
da
de

```

다음 다이어그램에서 볼 수 있듯이, 외부 루프(outer loop)가 한번 실행될 때마다 내부 루프(inner loop)는 처음부터 끝까지 실행된다.



이미지에 패턴을 생성하기 위해서 중첩과 조건문을 조합할 수 있다.

```

square = ImageGrid(5, 5)
for x in range(square.width):
    for y in range(square.height):
        if x < y:
            square[x, y] = colors['Fuchsia']
        elif x == y:
            square[x, y] = colors['Olive']
        else:
            square[x, y] = colors['SlateGray']
square.show()

```

처음으로 직접 작성한 데이터 시각화 사례다. 색상이 x가 y 보다 작거나, 같거나, 큰 것을 보여준다.

도전 과제

1. 상기 코드 루프의 중첩을 변경(X축 루프를 Y축 루프로 감싸는 것)하는 것이 최종 이미지를 바꾸나요? 왜 그런가요? 아니면 왜 그렇지 않나요?
2. 파이썬 (그리고 C 언어 계열의 다른 언어)은 다음과 같이 동작하는 인-플레이스 연산자(in-place operators)를 제공한다.

```
x = 1 # original value
x += 1 # add one to x, assigning result back to x
x *= 3 # multiply x by 3
print x
6
```

인-플레이스 연산자를 사용하여 리스트의 양수와 음수를 합하는 코드를 다시 작성하세요. 이렇게 작성된 코드가 처음 작성한 코드보다 가독성이 더 좋나요? 아니면 더 떨어지나요?

히트 맵 (Heat Map) 생성하기

마지막 단계는 데이터를 사용자가 볼 수 있도록 바꾸는 것이다. 선행 학습과 마찬가지로, 첫번째 단계는 데이터를 주기억장치(메모리)에 올리는 것이다.

```
import numpy as np
data = np.loadtxt(fname='inflammation-01.csv', delimiter=',')
print 'data shape:', data.shape

data shape: (60, 40)
```

두번째 단계는 데이터와 동일한 크기를 가지는 이미지 그리드를 생성하는 것이다.

```
width, height = data.shape  
heatmap = ImageGrid(width, height)
```

(상기 코드의 첫번째 라인은 깔끔한 기교를 이용한 것이다. 즉, 튜플의 값을 풀어서 튜플의 항목 수 만큼 변수에 할당한다.

세번째 단계는 히트 맵에 셀을 어떻게 색칠을 할 것인지 결정한다. 간략화 해서, 적색, 녹색, 청색을 주요 색으로 사용한다. 데이터셋의 평균과 비교하여 색칠을 한다. 코드가 다음에 있다.

```
for x in range(width):  
    for y in range(height):  
        if data[x, y] < data.mean():  
            heatmap[x, y] = colors['Red']  
        elif data[x, y] == data.mean():  
            heatmap[x, y] = colors['Green']  
        else:  
            heatmap[x, y] = colors['Blue']  
heatmap.show()
```

상기 히트맵이 의도한 것처럼 보이지만, 코드와 이미지 모두 흥물스럽게 보인다.

1. 작은 노트북 화면에 전체를 한번에 보여주기에는 너무 크다.
2. 첫 히트 맵은 시간이 X 축을 따라야 하는데 시간이 Y축을 따른 것으로 보인다.
3. 청색 대비 적색이 눈에는 꽤 어려워 보인다.
4. 히트 맵은 단지 두가지 색만 보여주는데 이유는 정수 측정값의 어떤 것도 소수점 평균과 동일한 값을 가지지 않기 때문이다.

5. 루프를 매번 반복할 때마다 `data`의 평균을 한번 혹은 두번 계산한다. 이것이 의미하는 바는 데이터 셋이 40×60 인 경우 2400번 동일한 연산을 수행해야 한다.

어떻게 프로그램을 더 낫게 만들 수 있는지 다음을 살펴보자.

1. 각 블록 크기 초기값을 설정하기 위해서 `ImageGrid`에 선택 옵션으로 `block_size`를 준다.
2. 격자(그리드)를 생성하기 전에 데이터를 전치(transpose)한다.
3. 더 좋은 색을 선택한다. (저자는 개인적으로 오아키드(orchid, 난초), 자홍색(fuchsia, 푸크시아), 강한 분홍색(hot pink)를 선호한다.)
4. 값을 정확하게 평균과 같은지 확인하는 대신에 평균에 가까운지를 확인한다.
5. 루프를 시작하기 전에 평균을 한번만 계산하고, 그 값을 여러번 사용한다.

변경된 코드는 다음과 같다.

```
flipped = data.transpose()
width, height = flipped.shape
heatmap = ImageGrid(width, height, block_size=5)
center = flipped.mean()
for x in range(width):
    for y in range(height):
        if flipped[x, y] < (0.8 * center):
            heatmap[x, y] = colors['Orchid']
        elif flipped[x, y] > (1.2 * center):
            heatmap[x, y] = colors['HotPink']
        else:
            heatmap[x, y] = colors['Fuchsia']
heatmap.show()
```

야간 더 좋아졌지만, 색상 대비가 충분하지 못하다. 자홍색(fuchsia) 셀이 충분히 많지 않다. 자홍색을 좀더 색칠하기 위해서 평균의 범위를 넓히는게 필요해 보인다.

세번째로 루프를 다시 작성한다. 하지만 범위와 색을 좀더 쉽게 실험할 수 있도록 함수에 코드를 넣는 것이 올바른 방향이다.

```
def make_heatmap(values, low_color, mid_color, high_color, low_band, high_band, block_size):
    '''Make a 3-colored heatmap from a 2D array of data.'''
    width, height = values.shape
    result = ImageGrid(width, height, block_size=block_size)
    center = values.mean()
    for x in range(width):
```

```
for y in range(height):
    if values[x, y] < low_band * center:
        result[x, y] = low_color
    elif values[x, y] > high_band * center:
        result[x, y] = high_color
    else:
        result[x, y] = mid_color
return result
```

함수를 시험하기 위해서, 방금전에 사용한 설정으로 실행한다.

```
h = make_heatmap(flipped, colors['Orchid'], colors['Fuchsia'], colors['HotPink'], 0.8, 1.2,
h.show()
```

잘 동작하는 것처럼 보인다. 그래서 범위를 좀더 넓히고, 좀더 극적인 색을 사용하자.

```
h = make_heatmap(flipped, colors['Gray'], colors['YellowGreen'], colors['SpringGreen'], 0.5)
h.show()
```

마지막으로 출판하기 전에 좀더 실험을 하고자 할지 모르지만 함수를 작성하는 것이 실험을 쉽게 한다. 매개변수에 초기설정값(default value)를 주어서 다시 함수를 수정하는 것이 실험을 더 쉽게 한다. 선정된 색을 변경하는 것보다 좀더 자주 범위의 하한과 상한을 변경하기 때문에 앞쪽에 매개변수를 위치시키자.

```
def make_heatmap(values,
                  low_band=0.5, high_band=1.5,
                  low_color=colors['Gray'], mid_color=colors['YellowGreen'], high_color=colors['Red'],
                  block_size=5):
    '''Make a 3-colored heatmap from a 2D array of data.
    Default color scheme is gray to green.'''
    width, height = values.shape
    result = ImageGrid(width, height, block_size=block_size)
    center = values.mean()
    for x in range(width):
        for y in range(height):
            if values[x, y] < low_band * center:
                result[x, y] = low_color
            elif values[x, y] > high_band * center:
                result[x, y] = high_color
            else:
                result[x, y] = mid_color
    return result
```

초기 설정값이 추가되면, 함수의 첫줄이 너무 길어서 화면에 편안하게 맞춰지지 못한다. 코드가 길어서 화면의 오른쪽 끝에 맞닥드릴 때마다 개행을 하는 것보다 코드의 가독성을 위해서 매개변수를 논리적 그룹으로 묶어 구분한다.

다시 한번, 앞에서 사용한 것과 동일한 값으로 코드를 다시 실행하여 시험한다.
(매개변수의 순서를 변경했기 때문에 다른 순서로 매개변수를 넘긴다.)

```
h = make_heatmap(flipped, 0.5, 1.5, colors['Gray'], colors['YellowGreen'], colors['SpringGre
```

시각화되는 데이터를 제외하고 모든 것을 남겨놓을 수 있다. 혹은 데이터와 범위를 제공하고 초기 설정 색과 블록 크기를 재사용한다.

```
h = make_heatmap(flipped, 0.4, 1.6)
h.show()
```

이제 키보드 몇번으로 데이터를 탐색적으로 볼 수 있다. 즉, 프로그래밍이 아닌 과학에 집중할 수 있다.

도전 과제

1. 히트 맵 함수 외곽에 데이터를 왜 전치하나요? 왜 함수가 전치를 할수 없게 했나요?
2. 히트 맵 함수는 왜 즉시 화면에 출력하는 대신에 그리드를 반환하여 사용할까요? 이런 결정이 여러분이 생각하기에 좋은 혹은 나쁜 디자인 선택이라고 보나요?
3. 다음 코드의 전반적인 효과가 무엇인지 설명하세요.

~~~ temp = left left = right right = temp ~~~

상기 코드를 다음과 비교하세요.

~~~ left, right = right, left ~~~

두개 코드가 동일한 것을 수행하나요? 어느 코드가 가독성이 낫다고 보십니까?

주요점

- 색칠된 블록으로 단순한 “이미지”를 생성할 때 `ipythonblocks` 라이브러리에서 `ImageGrid`를 사용하세요.
- 3가지 색상(적색, 녹색, 청색)과 각 색상은 0..255 범위의 정수를 갖도록 설정하세요.
- 조건문 시작은 `if condition`문을, 부가 테스트는 `elif condition`문을, 그리고 디폴트는 `else`문을 사용하세요.
- 조건문의 분기 몸통부문은 들여쓰기 해야됩니다.
- 동치를 시험은 `==`을 사용하세요.
- `X and Y`은 `X`와 `Y`가 모두 참이여야만 참입니다.
- `X or Y`은 `X`와 `Y` 둘중하나가 참이거나 모두 참이여야만 참입니다.

- 0, 빈 문자열, 그리고 빈 리스트는 거짓(false)으로 다른 모든 숫자, 문자열, 리스트는 참(true)으로 간주된다.
- 다차원 데이터의 연산은 중첩 루프를 사용한다.
- 자주 바뀌는 매개변수를 함수에 넣고, 프로그램의 행동을 사용자 정의에 맞추도록 다른 매개변수로 호출하라.

다음 단계 최종 히트 맵 함수는 17줄이다. 이것이 의미하는 것은 만약 각 코드 라인이 95%의 가능성으로 맞다면, 전체 함수가 올바르게 동작할 가능성은 41%에 불과하다. 좀더 학습을 진행하기 전에 코드가 프로그래머가 동작하길 기대하는 것을 수행하는지 어떻게 시험하는지 배울 것이다. 이것이 다음 학습의 주제이다.

방어적 프로그래밍 (Defensive Programming)

앞선 학습에서 프로그래밍의 기본적인 도구를 소개했다. 변수와 리스트, 파일 입출력(I/O), 루프, 조건문, 그리고 함수. 아직 수행하지 않은 것은 프로그램이 정답을 얻었는지를 어떻게 보여주고 프로그램을 변경하고 수정하면서 여전히 정답을 얻고 있는지를 보여주는 것이다.

이를 달성하기 위해서, 다음이 필요하다.

- 자신의 연산을 확인하는 프로그램을 작성한다.
- 널리 사용되는 함수에 대한 테스트를 작성하고 수행한다.
- “정답”이라는 것이 실제로 무엇을 의미하는지 알고 있어야 한다.

좋은 소식은 이런 것들을 수행하는 것은 프로그래밍의 속도를 늦추지 않고 가속화한다. 실제 목공에서 나무를 자르기 전에 주의깊게 측정해서 절약되는 시간이 측정하는데 걸리는 시간보다 훨씬 크다.

목표

- 가정 설정문(assertion)이 무엇인지 설명한다.
- 프로그램의 상태를 올바르게 확인하는 가정 설정문(assertion)을 프로그램에 추가한다.
- 함수에 전제조건과 사후조건 가정 설정문(assertion)에 올바르게 추가한다.
- 테스트 주도 개발(test-driven development)가 무엇인지 설명하고, 새로운 함수를 생성할 때 사용한다.
- 왜 변수를 초기화하는데 임의의 상수보다 실제 데이터를 사용하는지 설명한다.
- 체계적으로 오류를 포함하는 코드를 디버그한다.

가정 설정문 (Assertions)

프로그램에서 정답을 얻는 첫번째 단계는 실수는 일어난다고 가정하고 이에 대비하여 방지하는 것이다. 이것을 방어적 프로그래밍(defensive programming)이라고 부르고, 가장 일반적인 방식은 코드에 가정 설정문.assertions)을 추가해서 실행시에 점검한다. 가정 설정문은 단순하게 프로그램의 특정 지점에서 항상 참이어야 하는 문장이다. 파이썬이 가정 설정문을 만나게 될 때, 가정 설정문의 조건을 확인한다. 만약 참이면, 파이썬은 아무것도 하지 않는다. 하지만 거짓이면, 파이썬은 즉시 프로그램을 정지시키고 마련된 오류 메시지를 출력한다. 예를 들어, 루프가 양수가 아닌 값을 마주치자 마자 바로 이 코드 부분이 정지한다.

```
numbers = [1.5, 2.3, 0.7, -0.001, 4.4]
total = 0.0
for n in numbers:
    assert n >= 0.0, 'Data should only contain positive values'
    total += n
print 'total is:', total
```

```
AssertionError                                                 Traceback (most recent call last)
<ipython-input-19-33d87ea29ae4> in <module>()
      2 total = 0.0
      3 for n in numbers:
```

```

----> 4     assert n >= 0.0, 'Data should only contain positive values'
      5     total += n
      6 print 'total is:', total

AssertionError: Data should only contain positive values

```

파이어 폭스 웹브라우저 같은 프로그램은 가정 설정문(assertion)으로 가득차 있다. 코드의 10-20%는 다른 80-90%의 코드가 올바르게 동작하는지 확인하기 위해서 존재한다. 대체로 가정 설정문은 다음 3개 범주안에 들어간다.

- 올바르게 동작하기 위해서 함수의 시작점에서 참이여 되는 것은 사전 조건(precondition)이다.
- 함수가 끝날 때 참을 보증하는 것이 사후 조건(postcondition)이다.
- 부분 코드 내부 특정한 지점에서 항상 참이어야 하는 것이 불변식(invariant)이다.

예를 들어, 4개의 좌표 (x_0 , y_0 , x_1 , y_1)로 구성된 튜플을 사용하여 직사각형을 표현한다고 가정하자. 연산을 수행하기 위해서, 정사각형을 정규화해서 원점과 가장 긴 축을 따라 1.0 단위를 가진다. 함수가 정규화를 하지만 입력값이 올바른 형식인지 결과가 의미가 있는지 점검한다.

```

def normalize_rectangle(rect):
    '''Normalizes a rectangle so that it is at the origin and 1.0 units long on its longest
    assert len(rect) == 4, 'Rectangles must contain 4 coordinates'
    x0, y0, x1, y1 = rect
    assert x0 < x1, 'Invalid X coordinates'
    assert y0 < y1, 'Invalid Y coordinates'

    dx = x1 - x0
    dy = y1 - y0
    if dx > dy:
        scaled = float(dx) / dy
        upper_x, upper_y = 1.0, scaled
    else:
        scaled = float(dx) / dy

```

```

        upper_x, upper_y = scaled, 1.0

    assert 0 < upper_x <= 1.0, 'Calculated upper X coordinate invalid'
    assert 0 < upper_y <= 1.0, 'Calculated upper Y coordinate invalid'

    return (0, 0, upper_x, upper_y)

```

주석을 제외한 2, 4, 5번 행의 사전 조건은 잘못된 입력을 잡아낸다.

```

print normalize_rectangle( (0.0, 1.0, 2.0) ) # missing the fourth coordinate
-----
AssertionError                               Traceback (most recent call last)
<ipython-input-21-3a97b1dcab70> in <module>()
----> 1 print normalize_rectangle( (0.0, 1.0, 2.0) ) # missing the fourth coordinate

<ipython-input-20-408dc39f3915> in normalize_rectangle(rect)
      1 def normalize_rectangle(rect):
      2     '''Normalizes a rectangle so that it is at the origin and 1.0 units long on its
----> 3     assert len(rect) == 4, 'Rectangles must contain 4 coordinates'
      4     x0, y0, x1, y1 = rect
      5     assert x0 < x1, 'Invalid X coordinates'

AssertionError: Rectangles must contain 4 coordinates

print normalize_rectangle( (4.0, 2.0, 1.0, 5.0) ) # X axis inverted
-----
AssertionError                               Traceback (most recent call last)
<ipython-input-22-f05ae7878a45> in <module>()
----> 1 print normalize_rectangle( (4.0, 2.0, 1.0, 5.0) ) # X axis inverted

<ipython-input-20-408dc39f3915> in normalize_rectangle(rect)
      3     assert len(rect) == 4, 'Rectangles must contain 4 coordinates'
      4     x0, y0, x1, y1 = rect

```

```
----> 5     assert x0 < x1, 'Invalid X coordinates'
       6     assert y0 < y1, 'Invalid Y coordinates'
       7
```

```
AssertionError: Invalid X coordinates
```

사후 조건은 계산 결과가 올바르지 않을 때 신호를 줌으로써 버그를 잡도록 도와준다. 예를 들어, 너비보다 더 큰 직사각형을 정규화한다면 모든 것이 OK 처럼보인다.

```
print normalize_rectangle( (0.0, 0.0, 1.0, 5.0) )

(0, 0, 0.2, 1.0)
```

하지만, 높이보다 더 넓은 정사각형을 정규화한다면 사전 선언문이 자동으로 실행된다.

```
print normalize_rectangle( (0.0, 0.0, 5.0, 1.0) )

-----
AssertionError                               Traceback (most recent call last)
<ipython-input-24-5f0ef7954aeb> in <module>()
----> 1 print normalize_rectangle( (0.0, 0.0, 5.0, 1.0) )

<ipython-input-20-408dc39f3915> in normalize_rectangle(rect)
      16
      17     assert 0 < upper_x <= 1.0, 'Calculated upper X coordinate invalid'
---> 18     assert 0 < upper_y <= 1.0, 'Calculated upper Y coordinate invalid'
      19
      20     return (0, 0, upper_x, upper_y)

AssertionError: Calculated upper Y coordinate invalid
```

작성한 함수를 다시 읽게되면, 10번째 행이 dx가 dy로 나누어지기 보다 dy가 dx로 나누어져야 한다. (Ctrl+M 그리고 L을 타이핑해서 행번호를 화면에 출력할

수 있다.) 만약 함수의 끝에 가정 설정문을 생략한다면, 유효한 답변으로 올바른 모양을 가진 무언가를 생성하고 반환해야 할 것이지만 하지는 그렇게 하지는 않는다. 버그를 탐지하고 디버깅하는 것은 거의 항상 가정 설정문을 작성하는 것보다 장기적으로 더 많은 시간이 걸린다.

하지만 가정 설정문이 오류를 잡아내는 것만 하는 것은 아니고 사람들로 하여금 프로그램을 이해하는데 도움도 준다. 각각의 가정 설정문은 프로그램을 읽는 사람에게 코드가 동작하는 것과 프로그램을 이해하는 것이 매칭되는지 확인할 수 있는 기회의 장을 제공하기도 한다.

대부분의 좋은 프로그래머는 코드에 가정 설정문을 추가할 때 두 가지 규칙을 따른다. 하나는 "[미리 실패하고, 자주 실패하라\(fail early, fail often\)](#)"는 것이다. 오류가 발생하는 시간과 장소와 인지하는 시점과 거리가 크면 클수록, 오류를 디버그하기가 더욱 어렵다. 그래서 좋은 코드는 가능한 이른 시점에 오류와 실수를 잡아낸다.

두 번째 규칙은 "[버그를 가정 설정문과 테스트로 변환하라\(turn bugs into assertions or tests\)](#)"는 것이다. 만약 코드 일부분에 실수를 하게된다면, 근처에서 다른 실수를하거나 다음번에 코드를 변경할 때 동일한 혹은 관련된 실수를 저지를 가능성이 높다. 여러분이 회귀(regressed)가 되지 않도록 (즉, 이전 문제를 다시 발생하지 않도록) 가정 설정문을 작성하는 것은 장기적으로 엄청난 시간을 절약할 수 있고 미래의 자신을 포함하여 까다로운 코드를 읽는 사람에게 경고를 주는데도 도움이 된다.

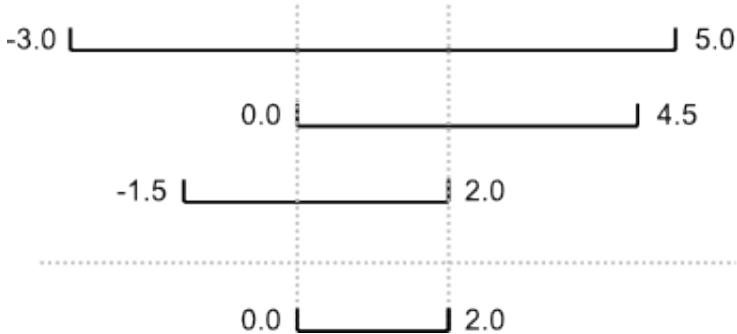
도전 과제

1. 리스트 숫자의 평균을 계산하는 함수 `average`를 작성한다고 가정하자. 사전 조건과 사후 조건으로 함수 `average`에 대해 무엇을 작성할까요? 여러분이 작성한 것과 주위 동료의 것과 비교하세요. 여러분의 테스트를 통과했으나 동료의 테스트는 통과하지 못한 혹은 반대 경우의 함수를 생각할 수 있나요?
2. 코드에 사전 설정문이 확인하는 것이 무엇인지 일상적인 말로 설명하세요. 그리고, 각각에 대해서 사전 설정문이 실패하게 되는 입력값의 예를 주세요.

```
~~~python def running(values): assert len(values) > 0 result = [values[0]] for v in values[1:]: assert result[-1] >= 0 result.append(result[-1] + v) assert result[-1] >= result[0] return result ~~~
```

테스트 주도 개발 (Test-Driven Development)

가정 설정문은 프로그램의 특정한 지점에서 무엇인가 참인지 확인하는데 도움이 된다. 다음 단계는 코드 일부분의 전반적인 동작을 확인하는 것이다. 즉, 특정한 입력값이 주어졌을 때, 올바른 출력값을 만들어 내는지 확인한다. 예를 들어, 두개 혹은 그 이상의 시계열이 중첩되는지 발견할 필요가 있다고 가정하자. 각 시계열의 범위는 숫자 짹으로 표현되고 시작과 끝을 표현하는 시간 간격이 있다. 출력값은 모든 시간을 포함하는 가장 큰 범위다.



대부분의 초보 프로그래머는 상기 문제를 다음과 같이 푼다.

1. `range_overlap` 함수를 작성한다.
2. 두개 혹은 3개의 입력값에 대해서 함수를 인터랙티브하게 호출한다.
3. 만약 함수가 잘못된 대답을 준다면, 함수의 잘못된 것을 고치고 다시 테스트를 시행한다.

명확하게 이 방식은 동작하지만 더 좋은 방식이 있다. (수천명의 과학자가 지금 이와 같이 작업을 하고 있다.)

1. 각 테스트에 대해서 짧은 함수를 작성한다.
2. 상기 테스트를 통한 `range_overlap` 함수를 작성한다.
3. 만약 함수 `range_overlap`가 잘못된 대답을 준다면, 함수의 잘못된 것을 고치고 다시 테스트를 시행한다.

함수를 작성하기 전에 테스트를 작성하는 것을 테스트 주도 개발(test-driven development) (TDD)라고 한다. TDD 지지하시는 분들은 이 방식이 더 빠르게 더 좋은 코드를 만들어낸다고 믿고 있다. 왜냐하면,

- 만약 테스트 대상 코드를 작성한 후에 테스트를 작성하게 된다면, 확증 편향(confirmation bias)에 빠지기 쉽다. 즉, 무의식적으로 오류를 발견하기봐 작성한 코드가 옳다는 것을 증명하기 위한 테스트를 작성한다.
- 테스트를 작성하는 것은 프로그래머가 함수가 실질적으로 무엇을 수행해야 하는지에 대해 파악하는데 도움을 준다.

`range_overlap`에 대한 3개의 테스트 함수가 있다.

```
assert range_overlap([ (0.0, 1.0) ]) == (0.0, 1.0)
assert range_overlap([ (2.0, 3.0), (2.0, 4.0) ]) == (2.0, 3.0)
assert range_overlap([ (0.0, 1.0), (0.0, 2.0), (-1.0, 1.0) ]) == (0.0, 1.0)

-----
AssertionError                                     Traceback (most recent call last)
<ipython-input-25-d8be150fbef6> in <module>()
      1 assert range_overlap([ (0.0, 1.0) ]) == (0.0, 1.0)
----> 2 assert range_overlap([ (2.0, 3.0), (2.0, 4.0) ]) == (2.0, 3.0)
      3 assert range_overlap([ (0.0, 1.0), (0.0, 2.0), (-1.0, 1.0) ]) == (0.0, 1.0)

AssertionError:
```

오류는 사실 안심을 준다. 아직 `range_overlap`을 작성하지는 않아서 만약 테스트가 통과된다면, 누군가 함수를 작성했고, 우연히 여러분이 함수를 사용한다는 표시다.

테스트를 작성하는 것에 대한 보너스로 암묵적으로 입력과 출력이 무엇인지 정의 한다는 것이다. 쌍으로 구성된 여러 리스트를 입력받아 하나의 리스트로 출력하는 것이다.

하지만 중요한 것이 빠졌다. 범위가 전혀 중첩되지 않는 경우에 대해서 어떠한 테스트도 준비하지 않았다.

```
assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == ???
```

상기 경우에 `range_overlap`은 무엇을 해야할까? 오류 메시지 실패로 종료, 중첩되지 않는다는 신호로 (0.0, 0.0) 같은 값을 출력, 혹은 다른 어떤 것을 수행.

함수를 실제로 구현할 때 여러 경우의 수 중에서 하나를 작성한다. 이슈가 있다는 것을 알아차리기 이전에 감정적으로 무언가 작성하는데 투자를 일으키기 전에 먼저 테스트를 작성하는 것은 무엇이 가장 최선이지 파악하는데 도움을 준다.

다음 사례의 경우는 어떻게 처리할까?

```
assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == ???
```

끝점을 맞대고 있는 두 부분은 중첩된 것인가 아닌가? 수학자는 대체로 “예 맞습니다”라고 하지만, 공학자는 대체로 “아닙니다”라고 말한다. 최선의 답은 “프로그램의 나머지 부분에서 무엇이든지 가장 유용한 것이 될 것이다”. 하지만, 다시 한번 `range_overlap`의 실제 구현은 무언가 수행하는 것이고, 구현이 무엇이든지 관계없이 전혀 중첩되는 것이 없을 때 수행되는 것과 일관성이 있어야 한다.

시계열 차트에서 X축으로 함수는 반환하는 범위를 사용하려고 계획하고 있기 때문에, 다음과 같이 결정한다.

1. 모든 중첩은 0 이 아닌 너비를 가져야 한다.
2. 중첩되는 것이 없을 때, 특수값 `None` 을 반환한다.

`None`은 파이썬에 내장되어져 있고, “여기에 아무것도 없어요(none here)”를 의미한다. 다른 언어는 종종 상응하는 값으로 `null` 혹은 `nil` 이라고 한다. 상기 결정 사항을 가지고, 마지막 두 테스트 작성은 마칠 수 있다.

```
assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == None
assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == None
```

```
AssertionError                                                 Traceback (most recent call last)
<ipython-input-26-d877ef460ba2> in <module>()
----> 1 assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == None
      2 assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == None
```

`AssertionError:`

다시, 함수를 작성하지 않아서 오류가 생겼다. 하지만 이제 함수를 작성할 준비가 되었다.

```

def range_overlap(ranges):
    '''Return common overlap among a set of [low, high] ranges.'''
    lowest = 0.0
    highest = 1.0
    for (low, high) in ranges:
        lowest = max(lowest, low)
        highest = min(highest, high)
    return (lowest, highest)

```

(잠시 시간을 가지고 왜 `lowest`에 `max`를 사용하고, `highest`에 `min`을 사용하는지 생각해보자. 테스트를 다시 실행하고자 하지만, 3개의 다른 셀에 여기저기 분산되어 있다 테스트를 좀더 쉽게 실행하도록 함수에 테스트 케이스를 모두 모아 놓자.

```

def test_range_overlap():
    assert range_overlap([(0.0, 1.0), (5.0, 6.0)]) == None
    assert range_overlap([(0.0, 1.0), (1.0, 2.0)]) == None
    assert range_overlap([(0.0, 1.0)]) == (0.0, 1.0)
    assert range_overlap([(2.0, 3.0), (2.0, 4.0)]) == (2.0, 3.0)
    assert range_overlap([(0.0, 1.0), (0.0, 2.0), (-1.0, 1.0)]) == (0.0, 1.0)

```

하나의 함수 호출로 `range_overlap`을 이제는 테스트할 수 있다.

```
test_range_overlap()
```

```

-----
AssertionError                                     Traceback (most recent call last)
<ipython-input-29-cf9215c96457> in <module>()
----> 1 test_range_overlap()

<ipython-input-28-5d4cd6fd41d9> in test_range_overlap()
      1 def test_range_overlap():
----> 2     assert range_overlap([(0.0, 1.0), (5.0, 6.0)]) == None
      3     assert range_overlap([(0.0, 1.0), (1.0, 2.0)]) == None
      4     assert range_overlap([(0.0, 1.0)]) == (0.0, 1.0)
      5     assert range_overlap([(2.0, 3.0), (2.0, 4.0)]) == (2.0, 3.0)

```

`AssertionError`:

`None`을 만들어내야 하는 첫번째 테스트는 실패해서, 작성한 함수에 뭔가 잘못된 것을 알게된다. 알지 못하는 것은 다른 4개의 테스트가 통과되었는지 실패했는지다. 왜냐하면 파이썬은 첫번째 오류를 탐지하지마자 프로그램을 정지한다. 여전히 약간의 정보가 없는 것보다는 낫다. 만약 그 입력값으로 함수의 동작을 추적하면 초기값을 입력값에 관계없이 `lowest`를 0.0 `highest`를 1.0으로 각각 초기 설정함을 알게된다. 이것은 또 다른 중요한 프로그래밍 규칙("항상 데이터로 초기화하라 ([always initialize from data](#))")을 위반하게 된다. 함수 `range_overlap`을 고치는 것은 연습으로 남겨둔다.

도전 과제

1. `range_overlap`을 고치시오. 변경을 한 후에 `test_range_overlap`을 다시 실행하세요.

디버깅 (Debugging)

테스팅을 통해서 문제를 발견하게 되면, 다음 단계는 문제를 고치는 것이다. 많은 초보자는 올바른 답을 만드는 것처럼 보일 때까지 대체로 랜덤(random)하게 변경을 해서 문제를 해결한다. 하지만 이러한 접근법은 매우 비효율적이고 결과는 대체로 테스팅하고 있는 경우에 대해서만 적합하다. 프로그래머가 좀더 경험이 많을수록, 좀더 체계적으로 디버그한다. 대부분의 경험있는 프로그래머는 다음에 설명된 규칙과 변형을 따른다.

프로그램이 수행하게 되어 있는 것을 이해한다. 무언가를 디버깅하는 첫번째 단계는 [프로그램이 수행하게 되어 있는 것을 이해](#)(know what it's supposed to do)하는 것이다. “작성한 프로그램이 동작을 하지 않는다”는 충분하지 않다. 문제를 진단하고 고치기 위해서는 올바른 결과와 잘못된 결과를 구별할 필요가 있다. 만약 실패한 케이스에 대해서 테스트 케이스를 작성한다면 - 즉 이것을 입력으로 가정

설정한다면, 함수는 저것을 결과로 산출한다 - 그러면 디버깅을 시작할 준비가 되었다. 만약 이렇게 할수 없다면, 무엇인가 고칠 때 어떻게 고칠지에 대해서 파악할 필요가 있다.

하지만, 과학적 소프트웨어에 대해서 테스트 케이스를 작성하는 것은 상용 응용프로그램에 대한 테스트 케이스를 작성하는 것보다 종종 더 힘들다. 왜냐하면, 만약 과학적 소프트웨어 코드의 결과가 무엇이 되어야 하는지 알고 있다면, 소프트웨어를 실행하지 않아도 된다. 그래서 결과를 작성하고 다음 프로그램으로 옮겨간다. 실무에서 과학자들은 다음을 수행하는 경향이 있다.

1. 단순화된 데이터로 테스트한다. 실제 데이터셋에 통계분석을 수행하기 전에, 하나의 레코드에 대해서, 두개의 동일한 레코드에 대해서, 두개의 레코드인데 한 단계가 차이가 나는, 혹은 수작업으로 정답을 계산할 수 있는 레코드에 대해서 통계를 계산한다.
2. 단순화된 케이스를 테스트한다. 만약 프로그램이 매우 빨리 회전하는 작은 방울의 과냉각된 헬륨에 자기장 소용돌이를 모의실험하려고 한다면, 첫번째 테스트는 회전하지 않는 작은 방울 헬륨이여야 하고, 어떤 외부 전자기장에 영향을 받지 말아야 한다. 마찬가지로, 만약 어떤 종에 대한 기후변화의 효과를 살펴보려고 한다면, 첫번째 테스트는 온도, 습도, 그리고 다른 요소를 상수로 고정하여야 한다.
3. 절대적인(*oracle*) 것과 비교한다. 테스트 오라클(test oracle)은 새로운 프로그램의 결과와 비교할 수 있는 무엇이다. 즉, 실험 데이터, 결과를 신뢰할 수 있는 이전 프로그램, 혹은 심지어 전문가도 될 수 있다. 만약 테스트 오라클이 있다면, 특별한 케이스에 대해서 출력 결과를 저장해서 프로그램을 다시 실행하지 않고 원하는 만큼 자주 새로운 결과값과 비교한다.
4. 보전 법칙을 확인하라. 질량, 에너지, 그리고 기타 양적 정보는 물리 시스템에서 보존된다. 프로그램에서도 또한 보존되어야 한다. 마찬가지로, 만약 환자 데이터를 분석한다면, 레코드 숫자는 같은 수가 유지되거나 다음 분석으로 옮겨가게 되면 줄어든다. (왜냐하면 결측값을 가진 레코드나 아웃라이어를 버려버리기 때문이다.) 만약, 파이프라인을 따라서 옮겨가다가 갑자기 “새로운” 환자가 값자기 나타난다면, 아마도 무언가 잘못되고 있다는 신호다.
5. 시각화하라. 데이터 분석자는 종종 간단한 시각화를 사용하여 수행하고 있는 과학과 코드의 정합성에 대해서 점검한다. (파이썬 학습의 [도입 학습](#)과 마

찬가지로). 하지만, 이 방법은 디버깅에 대해서 최후의 수단이 되어야 한다. 왜냐하면, 자동적으로 두개의 시각화 결과를 비교하는 것은 매우 어렵다.

매번 실패하게 만들기 (Make It Fail Every Time) 실패할 때만 무언가 디버그할 수 있다. 그래서 두번째 단계는 항상 매번 실패하게 만드는 테스트 케이스를 찾는 것이다. “매번(every time)”이 중요한데, 이유는 간헐적인 문제를 디버깅하는 것보다 더 좌절을 주는 것이 없기 때문이다. 만약 한번의 실패를 만들기 위해서 12번 함수를 호출해야 한다면, 실패가 실제로 일어났을 때로 스크롤하여 실패를 찾는 것이 확률적으로 높다.

이것과 관련해서, 코드가 “연결되어 있는지(plugged-in)” 확인하는 것이 중요하다. 즉, 실제로 우리가 생각하기에 문제인 것을 다룬다. 모든 프로그래머는 버그를 쫓아서 몇시간을 보내는데 단지 잘못된 데이터나 잘못된 환경설정 매개변수에 코드를 호출하거나 완전히 잘못된 소프트웨어 버전을 사용한 것을 깨닫기 위해서다. 이와 같은 실수는 특히 피곤할 때, 좌절했을 때, 마감시한에 임박했을 때 발생할 듯 하다. 이런 이유로 밤 늦게 혹은 밤새도록 코딩을 하는 것은 거의 가치없어서 지양해야된다.

빨리 실패하게 만들기 (Make It Fail Fast) 만약 버그가 표면에 나오는데 20분 걸린다면, 한시간에 3회 실험을 할 수 있다. 이것이 더 많은 시간에 더 적은 데이터를 갖는다는 것을 의미하지 말아야 한다. 프로그램이 실패하기를 기다리면서 다른 것에 의해서 산만하게 더 될 듯하다. 이것은 프로그래머가 문제에 사용하는 시간의 집중도가 떨어진다는 것을 의미한다. 그러므로 **빨리 실패하게 만드는 것(Make It Fail Fast)**이 매우 중요하다.

프로그램을 시간내에 빨리 실패하게 만드는 것 뿐만 아니라 공간적인 측면에서 프로그램을 빨리 실패하게 만들고 싶다. 즉, 가능하면 적은 코드 지역에 실패를 국지화하고자 한다.

1. 원인과 결과 사이의 간격이 작으면 작을수록, 연결점을 발견하기는 더욱 쉽다. 그래서 많은 프로그래머는 버그를 찾기 위해서 분리 정복 전략(divide and conquer strategy)를 사용한다. 만약 함수의 출력이 잘못된다면, 중간에 있는 것이 OK 인지 점검하고 나서 앞쪽 혹은 뒤쪽에 점검하고 이를 반복한다.

2. N개는 $N^{2/2}$ 다른 방식으로 상호작용한다. 그래서 테스트 부분으로 실행되지 않는 모든 코드 라인은 특별히 걱정할 필요가 없는 것 이상을 의미한다.

이유를 가지고 한번에 하나씩 변경하라. 임의의 코드 덩어리를 교체하는 것은 좋은 일을 하지 못할 것 같다. (결국, 처음에 잘못되면, 아마도 두번째 세번째도 잘못될 것이다.) 그래서 좋은 프로그래머는 **이유를 가지고 한번에 하나씩 변경** (*change one thing at a time, for a reason*) 한다. 좋은 프로그래머는 좀더 정보를 수집(루프의 순서를 변경한다면 버그가 여전히 남아있을까?)하거나 고친 부분을 테스트(처리하기 전에 데이터를 정렬함으로써 버그를 없앨 수 있을까?) 한다.

아무리 작을더라도 매번 변경을 할때마다, 즉시 테스트를 다시 돌려야한다. 왜냐하면 한번에 변경한 것이 더 많으면 많을수록, 무엇이 무엇에 (N^2 상호작용) 대해서 책임이 있는지 알아내기가 더더욱 힘들다.

그리고 모든 테스트를 다시 실행하게 된다면, 코드를 수정한 절반이상은 버그를 생성 혹은 재생성하게된다. 그래서 모든 테스트를 다시 실행을 통해서 회귀(*regressed*) 했는지 즉, 이전 문제를 다시 발생하게 했는지도 알 수 있다.

작업한 것을 기록하라. 훌륭한 과학자는 작업한 것을 기록한다. 그래서 작업한 것을 다시 재생성할 수 있고 동일한 실험을 반복하거나, 결과가 신통지 못한 것을 다시 실행하는데 시간을 낭비하지 않는다. 마찬가지로, 디버깅도 **작업한 것을 기록** (*keep track of what we've done*)하고 어떻게 잘 동작했는지도 기록할 때 가장 잘 된다. 만약 여러분이 다음과 같은 질문을 한다면, 왼쪽에서 오른쪽으로 코드 라인 흘수가 시스템 충돌을 일으켰는지? 오른쪽에서 왼쪽이 충돌을 일으켰는지? 코드 라인 짹수를 사용하다가 발생했는지?, 컴퓨터에서 잠시 떨어져서, 숨을 깊이 들이 마시고, 좀더 체계적으로 일을 시작해야하는 시간이다.

시간이 흘러 도움을 요청할 때 기록은 특히 유용하다. 명확하게 했던 것을 설명할 때 사람들은 좀더 귀를 기울여 듣는다. 그리고 사람들이 필요로 하는 유용한 정보를 좀더 잘 전달할 수 있다.

버전 제어 (version control) 재방문 버전 제어 (version control)는 종종 디버깅 동안 이전 특정 상태로 소프트웨어를 다시 원복하는데 사용된다. 그리고 버그와 연관된 코드 최근 변경을 사항을 탐색하는데도 사용된다. 특히, 대부분의 버전 제어 시스템은 `blame` 명령어가 있어서 특정한 코드 라인에 누가 마지막에 변경을 했는지도 확인할 수 있다.

겸손하라 (Be Humble) 그리고, 도움을 말하라. 만약 10분내로 버그를 발견할 수 없다면, [겸손\(be humble\)](#)하게 도움을 요청하라. 문제를 크게 설명하는 것만으로도 종종 도움이 된다. 왜냐하면, 생각하는 것을 듣는 것만으로도 일관되지 못한 것과 숨겨진 가정을 발견하는데 도움이 된다.

도움을 요청하는 것은 또한 확증 편향(confirmation bias)을 줄여준다. 만약 복잡한 프로그램을 작성하는데 한시간을 썼다면, 잘 동작하길 원해서 동작하지 않는 이유를 찾기보다는 왜 동작해야 하는지에 대해서 계속 본인 스스로에게 최면을 건다. 코드에 감정적으로 투자를 하지 않은 사람은 좀더 객관적일 수 있다. 이것이 왜 외부 사람이 간과한 간단한 실수를 종종 탐지하는 이유다.

겸손의 일부는 실수로부터 배우는 것이다. 프로그래머는 동일한 것을 반복해서 잘못하는 경향이 있다. 작업하는 언어와 라이브러리를 이해하지 못하거나, 프로그램이 어떻게 동작하는지에 대한 모델이 잘못된 것도 이유다. 어느 경우에나 왜 오류가 발생했는지 메모를 해두어 다음번에 점검하는 것이 실수를 다시 하지 않게 돌리는 빠른 방법이다.

그리고 이러한 방법이 장기적으로 좀더 생산적으로 여러분을 만든다. 외국 속담에 ["한주의 노력은 한 시간의 생각을 절약해준다\(A week of hard work can sometimes save you an hour of thought\)](#)(는 말이 있다. 어떤 유형의 실수를 피하도록, 코드를 모듈화하고, 테스트할 수 있는 덩어리로 만들고, 모든 가정(혹은 실수)을 가정 설정문(assertion)으로 만든다면, 더 많이 만들지는 못하지만 동작하는 프로그램을 만드는데 더 적은 시간이 걸릴 것이다.

주요점

- 방어적으로 프로그램하라. 즉, 오류가 발생한다고 가정하고 오류가 발생할 때 오류를 탐지하도록 코드를 작성하다.

- 프로그램에 가정 설정문을 넣어서 프로그램이 실행될 때 상태를 점검하게 하라. 그리고 프로그램을 읽는 사람이 작성한 프로그램이 어떻게 동작을 하는 것인지 이해할 수 있도록 도움을 줘라.
- 사전 조건을 사용해서 함수 입력값을 사용해도 안전한지 점검하라.
- 사후 조건을 사용해서 함수 출력값을 사용해도 안전한지 점검하라.
- 코드를 작성하기 전에 테스트를 작성해서 정확하게 코드가 무엇을 수행해야 되는지 결정하도록 하라.
- 코드를 디버그하기 전에 코드가 무엇을 수행해야하는지 파악하라.
- 매번 실패하게 만들어라.
- 빨리 실패하게 만들어라.
- 이유를 가지고 한번에 하나씩 변경하라.
- 작업한 것을 기록하라.
- 겸손하라.

다음 단계 IPython Notebook으로 파일 코드를 작성하고 테스트하는 기초를 학습했다. 학습할 필요가 있는 마지막 것은 파이프라인과 쉘 스크립트에서 사용할 수 있는 명령-라인 프로그램을 어떻게 작성하는 것이다. 그렇게 함으로써 다른 사람이 작업한 것을 여러분의 도구와 통합할 수 있다. 이것이 다음 학습의 주제이며 마지막 학습이다.

명령-라인 프로그램 (Command-Line Programs)

IPython Notebook과 다른 인터랙티브 도구는 데이터를 탐색하고 프로토타입 코드를 작성하는데는 훌륭하지만 조만간 파이프라인에서 프로그램을 사용하거나 수천개의 파일을 처리하는데 쉘 스크립트를 실행할 것이다. 이를 위해서 작성한 프로그램이 다른 유닉스 명령-라인 도구와 함께 동작하도록 만들 필요가 있다. 예를 들어, 데이터 셋을 읽고 환자당 평균 염증값을 출력하는 프로그램을 만들고 싶다.

```
$ python readings.py --mean inflammation-01.csv
```

5.45

```
5.425  
6.1  
...  
6.4  
7.05  
5.9
```

하지만, 첫 4번째 라인의 최소값을 보고자 할지 모른다.

```
$ head -4 inflammation-01.csv | python readings.py --min
```

혹은 여러개 파일을 순서대로 하나씩 최대 염증값을 보고자 할지 모른다.

```
$ python readings.py --max inflammation-*.csv
```

전반적인 요구사항은 다음과 같다.

1. 만약 파일 이름이 명령 라인에 주어지지 않는다면, 표준 입력(standard input)에서 데이터를 읽는다.
2. 만약 하나 혹은 그 이상의 파일이름이 주어진다면, 데이터를 파일이름에서 읽고 각 파일에 대해서 별도로 통계자료를 보고한다.
3. 무슨 통계치를 출력할 것인지 결정하기 위해서 `--min`, `--mean`, `--max` 옵션 플래그를 사용한다.

상기 요구사항을 만족하는 프로그램을 작성하기 위해서, 프로그램에서 어떻게 명령-라인 인자를 다루는지 그리고 어떻게 표준 입력을 받는지 파악할 필요가 있다. 이러한 질문을 다음에서 순차적으로 다룬다.

목표

- 명령-라인 인자 값을 프로그램에 사용한다.
- 명령-라인 프로그램에 옵션 플래그와 파일을 별도로 다룬다.
- 프로그램에서 표준 입력값을 데이터에서 읽어서 파이프라인에서 사용될 수 있게 한다.

명령-라인 인자 (Command-Line Arguments)

여러분이 선택한 텍스트 편집기를 사용하여, 텍스트 파일에 다음을 저장하세요.

```
!cat sys-version.py
```

```
import sys
print 'version is', sys.version
```

첫번째 행은 “system”을 간략하게 줄인 sys라는 라이브러리를 가져온다. 가져온 라이브러리는 sys.version 같은 값을 정의하는데 Python 버전이 무엇인지 기술한다. IPython Notebook 내부에서 다음과 같이 스크립트를 실행할 수 있다.

```
%run sys-version.py
```

```
version is 2.7.5 |Anaconda 1.8.0 (x86_64)| (default, Oct 24 2013, 07:02:20)
[GCC 4.0.1 (Apple Inc. build 5493)]
```

혹은 다음과 같아도 가능하다.

```
!ipython sys-version.py
```

```
version is 2.7.5 |Anaconda 1.8.0 (x86_64)| (default, Oct 24 2013, 07:02:20)
[GCC 4.0.1 (Apple Inc. build 5493)]
```

첫번째 방법(%run)은 .py 파일에 담긴 프로그램을 실행하는데 IPython Notebook에 있는 특수 명령어를 사용한다. 두번째 방법이 좀더 일반적이다. 느낌표(!)가 Notebook에 쉘 명령어를 실행한다고 지시한다. 그래서 실행하는 명령어는 스크립트 이름과 ipython이 된다.

좀더 흥미로운 것을 수행하는 또 다른 스크립트가 다음에 있다.

```
!cat argv-list.py
```

```
import sys
print 'sys.argv is', sys.argv
```

이상한 이름 argv는 “argument values”(인자값)을 줄여 표현한 것이다. 파이썬이 프로그램을 실행할 때마다, 명령 라인에 주어진 모든 값을 받아서 sys.argv 리스트에 넣는다. 그렇게 해서 프로그램이 인자값이 무엇인지를 판단할 수 있다. 만약 어떤 인자도 없이 프로그램을 실행한다면,

```
!ipython argv-list.py
```

```
sys.argv is ['/Users/gwilson/s/bc/python/novice(argv-list.py)']
```

리스트의 유일한 것은 스트립트의 전체 경로정보가 되고 항상 sys.argv[0]을 차지한다. 하지만, 만약 몇개의 인자를 넣어 실행한다면,

```
!ipython argv-list.py first second third
```

```
sys.argv is ['/Users/gwilson/s/bc/python/novice(argv-list.py)', 'first', 'second', 'third']
```

그러면 파이썬은 각각의 인자를 마술같은 리스트에 추가한다.

지금까지 학습한 것을 가지고, 단독 데이터 파일에 환자마다 평균값을 출력하는 readings.py를 작성해 보자. 첫번째 단계는 구현에 대한 윤곽을 잡는 함수와 실제 동작하는 함수에 대한 자리를 잡는 코드를 작성한다. 함수 이름을 원하는 무엇이든지 정할 수 있지만, 관례로 함수는 통상 main으로 부른다.

```
!cat readings-01.py
```

```
import sys
import numpy as np

def main():
    script = sys.argv[0]
    filename = sys.argv[1]
    data = np.loadtxt(filename, delimiter=',')
    for m in data.mean(axis=1):
        print m
```

이 함수는 스크립트 이름을 `sys.argv[0]`에서 얻는데 이유는 그곳이 항상 이름이 놓여지는 장소이기 때문이다. 처리할 파일 이름은 `sys.argv[1]`에서 얻는다. 다음에 간단한 테스트가 있다.

```
%run readings-01.py inflammation-01.csv
```

어떠한 출력도 없는데 이유는 함수를 정의했지만, 실질적으로 호출을 하지 않았기 때문이다. `main`에 호출을 추가하자.

```
!cat readings-02.py
```

```
import sys
import numpy as np

def main():
    script = sys.argv[0]
    filename = sys.argv[1]
    data = np.loadtxt(filename, delimiter=',')
    for m in data.mean(axis=1):
        print m

main()
```

그리고 실행하자.

```
%run readings-02.py inflammation-01.csv
```

```
5.45
5.425
6.1
5.9
5.55
6.225
5.975
6.65
```

6.625
6.525
6.775
5.8
6.225
5.75
5.225
6.3
6.55
5.7
5.85
6.55
5.775
5.825
6.175
6.1
5.8
6.425
6.05
6.025
6.175
6.55
6.175
6.35
6.725
6.125
7.075
5.725
5.925
6.15
6.075
5.75
5.975
5.725

6.3
5.9
6.75
5.925
7.225
6.15
5.95
6.275
5.7
6.1
6.825
5.975
6.725
5.7
6.25
6.4
7.05
5.9

올바른 방법 만약 작성중인 프로그램이 복잡한 매개변수나 복수의 파일이름을 가진다면, `sys.argv`를 직접적으로 다루지 말아야 한다. 대신에 파이썬 `argparse` 라이브러리를 사용한다. `argparse` 라이브러리는 체계적으로 일반적인 경우를 처리하고, 또한 사용자를 위해서 프로그래머가 실용적인 오류 메시지를 제공하기 쉽게 만들었다.

도전 과제

1. 덧셈과 뺄셈을 수행하는 명령-라인 프로그램을 작성하세요.

```
$ python arith.py 1 + 2  
3
```

```
$ python arith.py 3 - 4  
-1
```

만약 프로그램에 *을 사용해서 곱셈을 추가하려고 한다면 무슨 잘못이 있을까요?

2. [03-loop.ipynb](#)에서 소개된 glob 모듈을 사용해서, 특정 확장자를 가진 파일을 현재 디렉토리에서 출력하는 ls 의 간단한 버전을 작성하세요.

```
$ python my_ls.py py  
left.py  
right.py  
zero.py
```

다수 파일 처리하기

다음 단계는 프로그램에게 파일 다수를 어떻게 처리하는지 가르치는 것이다. 파일 당 60줄의 출력결과는 페이지를 넘기며 살펴보기에는 많은 불량이여서 3개의 작은 파일로 시작한다. 작은 파일 각각은 두 환자에 대한 3일치 데이터가 있다

```
!ls small-*.csv  
  
small-01.csv small-02.csv small-03.csv
```

```
!cat small-01.csv
```

```
0,0,1  
0,1,2
```

```
%run readings-02.py small-01.csv
```

```
0.333333333333333
```

```
1.0
```

작은 파일을 입력값으로 사용하는 것은 좀더 쉽게 결과를 확인할 수 있게 한다. 예를 들어, 프로그램이 각 행마다 올바르게 평균을 계산하는지 살펴볼 수 있다. 반면에 전에는 정말 믿음으로만 가지고 있었다. 이것은 또 다른 프로그래밍 규칙이다.
"간단한 것을 먼저 시험하라(test the simple things first)"

작성한 프로그램이 각각의 파일을 개별로 처리하기 원해서 각 파일 이름마다 한번씩 실행되는 루프가 필요하다. 명령 라인에 파일 이름을 지정한다면, 파일 이름은 `sys.argv`에 저장되지만, 주의가 필요하다. `sys.argv[0]`는 항상 파일이름이 아니고 스크립트 이름이다. 작성한 프로그램이 임의 갯수의 파일에 대해서 실행될 수 있기 때문에 알수 없는 갯수의 파일이름을 처리할 필요가 있다.

해결책은 `sys.argv[1:]` 내용에 루프를 돌리는 것이다. '1'은 파일이 1번 위치에서 슬라이스를 시작해서 프로그램 이름이 포함되지 않도록 한다. 상한을 비워두었기 때문에 슬라이스 인덱스가 리스트의 끝까지 가서 모든 파일 이름이 포함된다. 다음에 수정된 프로그램이 있다.

```
!cat readings-03.py

import sys
import numpy as np

def main():
    script = sys.argv[0]
    for filename in sys.argv[1:]:
        data = np.loadtxt(filename, delimiter=',')
        for m in data.mean(axis=1):
            print m

main()
```

그리고 실행 결과가 다음에 있다.

```
%run readings-03.py small-01.csv small-02.csv
```

```
0.333333333333
1.0
```

13.6666666667

11.0

Note: 이 지점에서, 스크립트 버전 3개(readings-01.py, readings-02.py, readings-03.py)를 생성했다. 실무에서는 이렇게 하지는 않을 것이다. 대신에 readings.R 파일만 보관하고 기능향상 작업을 할 때마다 버전 관리 시스템에 커밋한다. 하지만, 교육 목적으로 나란히 연속된 버전이

도전 과제

1. check.R 프로그램을 작성해서 인자로 하나 혹은 그 이상의 엔터 데이터 파일 이름을 가지고 모든 파일이 동일한 행과 열을 가지는지 검증하게 하세요. 프로그램을 시험하는 가장 최선의 방법은 무엇인가요?

명령어-라인 플래그(Command-Line Flags) 처리하기

다음 단계는 프로그램이 --min, --mean, --max 옵션 플래그에 관심을 두게 한다. 플래그는 항상 파일 이름 앞에 위치해서 다음과 같이 수행할 수 있다.

```
!cat readings-04.py

import sys
import numpy as np

def main():
    script = sys.argv[0]
    action = sys.argv[1]
    filenames = sys.argv[2:]

    for f in filenames:
        data = np.loadtxt(f, delimiter=',')

    if action == '--min':
```

```

        values = data.min(axis=1)
    elif action == '--mean':
        values = data.mean(axis=1)
    elif action == '--max':
        values = data.max(axis=1)

    for m in values:
        print m

main()

```

작성한 것이 잘 동작한다.

```
%run readings-04.py --max small-01.csv
```

```

1.0
2.0

```

하지만, 몇가지 잘못된 것이 있다.

1. main 함수가 너무 커서 편안하게 읽기가 쉽지 않다.
2. action 인자가 인정된 3개의 플래그 중에 하나가 아니라면, 프로그램을 각각의 파일 로딩/loading)하지만 아무것도 수행하기 않는다. 왜냐하면, 조건을 매칭하는 곳에서 어느 분기에도 해당되지 않기 때문이다. 이와 같이 침묵하는 실패(Silent failures)가 항상 디버그하기가 어렵다.

새로 작성한 버전은 각 파일의 처리를 루프에서 빼내서 처리하는 자신만의 함수를 만들었다. 처리를 수행하기 전에 action이 사전에 정의된 플래그중의 하나인지를 검사해서 프로그램이 빨리 종료한다.

```
!cat readings-05.py
```

```

import sys
import numpy as np

```

```

def main():
    script = sys.argv[0]
    action = sys.argv[1]
    filenames = sys.argv[2:]
    assert action in ['--min', '--mean', '--max'], \
        'Action is not one of --min, --mean, or --max: ' + action
    for f in filenames:
        process(f, action)

def process(filename, action):
    data = np.loadtxt(filename, delimiter=',')

    if action == '--min':
        values = data.min(axis=1)
    elif action == '--mean':
        values = data.mean(axis=1)
    elif action == '--max':
        values = data.max(axis=1)

    for m in values:
        print m

main()

```

상기 프로그램은 앞서 작성한 프로그램보다 더 길다. 하지만, 좀더 완전히 이해하기 쉬운 8줄과 12줄 프로그램 덩어리로 쪼갰다.

파이썬은 [argparse](#) 라이브러리가 있어서 복잡한 명령어-라인 플래그를 처리하는데 도움이 된다. 이번 학습에서는 [argparse](#) 라이브러리를 다루지 않을 것이다. 하지만, 좀더 자세한 사항은 파이썬 공식 문서의 일부인 Tshepang Lekhonkhobe 의 [Argparse tutorial](#)을 참고바란다.

도전 과제

1. 상기 프로그래을 다시 작성해서 `--min`, `--mean`, `--max` 대신에 `-n`, `-m`, `-x`을 각각 사용하게 하세요. 코드가 가독성이 좋습니까? 프로그램이 더 이해하기 좋습니까?
2. 이와는 별도로, 프로그램을 변경해서 만약 어떤 행동(action)이 명기되지 않거나 혹은 잘못된 동작이 주어지면, 어떻게 사용되어야 하는지 설명하는 메시지를 출력하게 하세요.
3. 이와는 별도로, 프로그램을 수정해서 만약 어떤 행동(action)도 명기되지 않으면, 데이터의 평균을 화면에 출력하게 코드를 작성하세요.

표준 입력 (Standard Input) 처리하기

프로그램이 다음으로 할 작업은 파일 이름이 주어지지 않았다면 표준 입력에서 데이터를 읽는 것이다. 파일 이름을 파일프라인에 넣고 입력값으로 되돌려 사용하는 것이 예이다. 또 다른 스크립트를 작성해서 실험을 해보자.

```
!cat count-stdin.py

import sys

count = 0
for line in sys.stdin:
    count += 1

print count, 'lines in standard input'
```

상기 작은 프로그램은 자동으로 프로그램의 표준 입력에 연결되는 `sys.stdin`으로 불리는 특수 “파일”에서 라인(행)을 읽어들인다. 파일을 별도로 열 필요는 없다. 즉, 프로그램이 시작할 때 파일과 운영시스템이 처리해준다. 하지만 정규 파일에서 할 수 있었던 거의 모든 것을 할 수 있게 한다. 마치 정규 명령어-라인 프로그램인 것처럼 유닉스 쉘에서 실행을 시도해 보자.

```
!ipython count-stdin.py < small-01.csv

2 lines in standard input
```

%run을 사용해서 실행하면 어떨까요?

```
%run count-stdin.py < small-01.csv
```

```
0 lines in standard input
```

결과에서 알 수 있듯이, %run 명령어는 파일 되돌리기(redirection)을 이해하지 못한다. 단지 쉘의 무엇으로 이해한다.

흔한 실수는 다음과 같이 표준입력에서 읽어서 무언가 실행하려고 하는 것이다.

```
!ipython count_stdin.py small-01.csv
```

즉, 표준입력에서 파일로 되돌리는 문자(<)를 생략한 것이다. 이 경우에 표준 입력에는 아무 것도 없어서 프로그램은 누군가 키보드로 무엇인가를 입력하기를 루프 시작에서 기다리기만 한다. 사람이 할 수 있는 것이 아무것도 없기 때문에 작성한 프로그램은 수령에 빠진 듯 동작할 수 없게 된다. 노트북 Kernel 메뉴에서 Interrupt 옵션을 사용해서 정지시켜야 한다.

프로그램을 다시 작성해서 만약 어떤 파일이름도 제공된게 없다면 sys.stdin에서 데이터를 로딩한다. 운좋게도, numpy.loadtxt는 파일이름 혹은 첫번째 매개변수로 열린 파일을 처리할 수 있다. 그래서 실질적으로 process를 변경할 필요는 없다. main을 약간 수정한다.

```
def main():
    script = sys.argv[0]
    action = sys.argv[1]
    filenames = sys.argv[2:]
    assert action in ['--min', '--mean', '--max'], \
        'Action is not one of --min, --mean, or --max: ' + action
    if len(filenames) == 0:
        process(sys.stdin, action)
    else:
        for f in filenames:
            process(f, action)
```

작성한 프로그램을 시도해 보자. (잠시 후에 왜 head로 출력결과를 보냈는지 파악하게 된다.)

```
!ipython readings-06.py --mean < small-01.csv | head -10

[TerminalIPythonApp] CRITICAL | Bad config encountered during initialization:
[TerminalIPythonApp] CRITICAL | Unrecognized flag: '--mean'
=====
IPython
=====

Tools for Interactive Computing in Python
=====

A Python shell with automatic history (input and output), dynamic object
introspection, easier configuration, command completion, access to the
system shell and more. IPython can also be embedded in running programs.
```

이럴 수가 있나: 데이터의 행별로 평균값 대신에 IPython 도움말이 왜 나왔을까? IPython이 실행하는 프로그램에 대한 것과 명령-라인 인자에 대한 것을 분간이 어렵다. 의미를 명확히 하기 위하고 둘을 구분하기 위해서 --(이중 대휘)를 사용한다.

```
!ipython readings-06.py -- --mean < small-01.csv

0.333333333333
1.0
```

더 나아졌다. 이제 완료했다. 프로그램이 처음 기획했던 모든 것을 수행한다.

도전 과제

1. line-count.py 프로그램을 작성해서 유닉스 wc 명령어처럼 동작하게하세요.

- 만약 어떤 파일이름도 주어지지 않는다면, 표준 입력에 행 숫자만을 보고한다.
- 만약 하나 혹은 그 이상의 파일이름이 주어지면, 각 파일의 행 숫자와 전체 행 숫자를 보고한다.

주요점

- `sys` 라이브러리는 파이썬 프로그램과 프로그램이 실행되는 시스템을 연결 한다.
- `sys.argv` 리스트는 프로그램이 실행되는데 필요한 명령-라인 인자를 담고 있다.
- 침묵하는 실패(Silent failures)를 피한다.
- “파일” `sys.stdin`을 사용해서 프로그램의 표준 입력에 연결한다.
- “파일” `sys.stdout`을 사용해서 프로그램의 표준 출력에 연결한다.

데이터베이스와 SQL 사용하기

거의 모든 사람이 스프레드시트(spreadsheet) 사용했고, 거의 모든 사람이 종국에는 한계에 맞닥뜨렸다. 데이터셋이 더욱 복잡할수록, 데이터를 걸려내고, 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.

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.

목표

- 다음 장에서 사용될 예제 데이터베이스를 구축한다.
- 데이터베이스가 이용 가능한지를 점검하고 어느 테이블이 있는지도 확인한다.

설치

인터랙티브하게 다음 학습을 수행하기 위해서는 [설치 방법](#)에 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
```

github에서 “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>)에 위치한 SQL 파일(“gen-survey-database.sql”)을 다운로드 한다.

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 [cURL](#)을 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 -O”로 대체하세요.]

```
mom@durga:~/swc/sql$ wget https://raw.githubusercontent.com/swcarpentry/bc/master/novice/sq
```

상기 명령문으로 Wget은 HTTP 요청을 생성해서 github 저장소의 “gen-survey-database.sql” 원 파일만 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  
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 103.245.222.133  
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|103.245.222.133|:443...  
HTTP request sent, awaiting response... 200 OK  
Length: 3297 (3.2K) [text/plain]  
Saving to: 'gen-survey-database.sql'
```

```
100% [=====] 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.sql”에 저장된 지시방법에 따라서 데이터를 채워넣는다.

명령-라인 터미널에서 SQLite3 프로그램을 호출하기 위해서, 다음 명령문을 실행한다.

```
sqlite3 survey.db < gen-survey-database.sql
```

SQLite DB 연결 및 설치 테스트

생성된 데이터베이스에 연결하기 위해서, 데이터베이스를 생성한 디렉토리 안에서 SQLite를 시작한다. 그래서 “~/swc/sql” 디렉토리에서 다음과 같이 타이핑한다.

```
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 “sqlite”로 명령-라인 프롬프트로 다음과 같이 변경되어 표시된다.

```
/novice/sql$ sqlite3 survey.db
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
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.

```
.tables
```

그리고 “.table”의 출력결과는 다음과 같다.

```
sqlite> .tables
Person    Site    Survey    Visited
```

이제, 설치를 완료해서 다음 학습으로 진행할 수 있다. 현재 명령-라인 SQLite 세션에서 다음 연습을 수행할 수 있다. IPython의 마술같은 방법 (%로 시작하는 각 명령문의 첫 행)은 IPython notebook에서만 동작하기 때문에, 터미널에서 SQLite를 사용하는 동안 생략할 수 있다. 만약 IPython notebook 사용을 선호하면 SQLite를 끝낼 수 있다.

SQLite3 DB 명령-라인 인터페이스(CLI)를 어떻게 빠져나올까요?

SQLite3를 빠져나오기 위해서, 다음과 같이 타이핑한다.

```
sqlite> .quit
```

SQLite3 CLI 대신에 IPython notebook을 어떻게 사용할까요?

만약 예제를 따라가는데 IPython notebook 사용을 선호한다면, IPython이 로컬 컴퓨터에 설치되었는지 점검하라. 만약 설치되어 있지 않다면, [설치 방법](#)을 따르세요. 만약 IPython이 이미 로컬 컴퓨터에 설치되어 있다면 notebook을 열기 위해서 작업 폴더 “~/swc/sql” 내부에서 “ipython notebook”을 타이핑하세요.

```
~/swc/sql$ ipython notebook
```

상기 명령어가 IPython 커널을 구동해서 디폴트 브라우저에 인터랙티브 노트북을 화면에 표시해서 학습하면서 편집할 수 있게 한다. 다음 학습에 보여지는 명령문을 IPython notebook에서 수행할 수 있다. 훈련이 종료되면 변경사항을 간직하기 위해서 노트북 저장을 기억하세요.

데이터 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.

목표

- 테이블, 레코드, 필드의 차이점을 설명한다.
- 데이터베이스와 데이터베이스 관리자의 차이를 설명한다.
- 단독 테이블에서 특정 필드에 있는 모든 값을 선택하는 질의문(쿼리, query)를 작성한다.

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.

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 | | |
| 837 | MSK-4 | 1932-01-14 | |
| 844 | DR-1 | 1932-03-22 | |

46244점에

3개 항목 (`Visited` 테이블에서 1개, `Survey` 테이블에서 2개)은 붉은색으로 표기한 것을 주목하라. 왜냐하면 어떠한 값도 담고 있지 않아서 그렇다.

b측값(missing)을 다시 다룰 것이다. 지금으로서는 과학자의 이름을 화면에 표시하는 SQL을 작성하자. SQL `select` 문을 사용해서 원하는 칼럼이름과 원하는 테이블이름을 준다. 쿼리와 결과는 다음과 같다.

```
%load_ext sqlitemagic  
  
%%sqlite survey.db  
select family, personal from Person;
```

| | |
|----------|-----------|
| Dyer | William |
| Pabodie | Frank |
| Lake | Anderson |
| Roerich | Valentina |
| Danforth | Frank |

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. (case insensitive)

```
%%sqlite survey.db  
SeLeCt FaMiLy, PeRsOnAl FrOm PeRsOn;
```

| | |
|----------|-----------|
| Dyer | William |
| Pabodie | Frank |
| Lake | Anderson |
| Roerich | Valentina |
| Danforth | Frank |

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 방식으로 제어할 수 있다. 예를 들어, 쿼리를 다음과 같이 작성해서 칼럼을 교환할 수 있다.

```
%%sqlite survey.db  
select personal, family from Person;
```

| | |
|-----------|----------|
| William | Dyer |
| Frank | Pabodie |
| Anderson | Lake |
| Valentina | Roerich |
| Frank | Danforth |

혹은 심지어 칼럼을 반복할 수도 있다.

```
%%sqlite survey.db  
select ident, ident, ident from Person;
```

| | | |
|----------|----------|----------|
| dyer | dyer | dyer |
| pb | pb | pb |
| lake | lake | lake |
| roe | roe | roe |
| danforth | danforth | danforth |

손쉬운 방법으로, *을 사용해서 테이블의 모든 칼럼을 선택할 수도 있다.

```
%%sqlite survey.db  
select * from Person;
```

| | | |
|----------|-----------|----------|
| dyer | William | Dyer |
| pb | Frank | Pabodie |
| lake | Anderson | Lake |
| roe | Valentina | Roerich |
| danforth | Frank | Danforth |

도전 과제

1. Site 테이블에서 사이트 이름만 선택하는 쿼리를 작성하세요.
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?

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.

정렬하고 중복 제거하기

목표

- 특정 순서로 결과를 표시하는 쿼리를 작성한다.
- 데이터에서 중복값을 제거하는 쿼리를 작성한다.

데이터는 종종 임여가 있어서, 쿼리도 종종 과잉 정보를 반환한다. 예를 들어, survey 테이블에서 측정된 수량 정보를 선택하면, 다음을 얻게된다.

```
%load_ext sqlitemagic

%%sqlite survey.db
select quant from Survey;
```

```
rad
sal
rad
sal
rad
sal
temp
rad
sal
temp
rad
temp
sal
rad
sal
temp
sal
rad
```

```
sal  
sal  
rad
```

결과를 좀더 읽을 수 있게 만들기 위해서 쿼리에 `distinct` 키워드를 추가해서 중복된 출력을 제거한다.

```
%%sqlite survey.db  
select distinct quant from Survey;
```

```
rad  
sal  
temp
```

하나 이상의 칼럼(예를 들어 survey 사이트 ID와 측정된 수량)을 선택한다면, 별개로 구별된 값의 쌍이 반환된다.

```
%%sqlite survey.db  
select distinct taken, quant from Survey;
```

```
619  rad  
619  sal  
622  rad  
622  sal  
734  rad  
734  sal  
734  temp  
735  rad  
735  sal  
735  temp  
751  rad  
751  temp  
751  sal
```

```
752    rad  
752    sal  
752    temp  
837    rad  
837    sal  
844    rad
```

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.

도전 과제

1. Site 테이블에서 별개로 구별되는 날짜를 선택하는 쿼리를 작성하세요.

앞서 언급했듯이, 데이터베이스 레코드는 특별한 순서로 저장되지 않는다. 이것의 의미하는 바는 쿼리 결과가 반드시 정렬되어 있지 않다는 것이다. 설사 정렬이 되어 있더라도, 종종 다른 방식으로 정렬하고 싶을 것이다. 예를 들어 과학자의 이름 대신에 프로젝트 이름으로 정렬할 수도 있다. SQL에서 쿼리에 `order by` 절을 추가해서 간단하게 구현할 수 있다.

```
%%sqlite survey.db  
select * from Person order by ident;
```

| | | |
|----------|-----------|----------|
| danforth | Frank | Danforth |
| dyer | William | Dyer |
| lake | Anderson | Lake |
| pb | Frank | Pabodie |
| roe | Valentina | Roerich |

디폴트로, 결과는 오름차순으로 정렬되어야 한다. (즉, 가장 적은 값에서 가장 큰 값 순으로 정렬된다.) `desc` (“descending”)를 사용해서 역순으로도 정렬할 수 있다.

```
%%sqlite survey.db  
select * from person order by ident desc;
```

| | | |
|----------|-----------|----------|
| roe | Valentina | Roerich |
| pb | Frank | Pabodie |
| lake | Anderson | Lake |
| dyer | William | Dyer |
| danforth | Frank | Danforth |

(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.

```
%%sqlite survey.db  
select taken, person from Survey order by taken asc, person desc;
```

| | |
|-----|------|
| 619 | dyer |
| 619 | dyer |
| 622 | dyer |
| 622 | dyer |
| 734 | pb |
| 734 | pb |
| 734 | lake |
| 735 | pb |
| 735 | None |
| 735 | None |
| 751 | pb |
| 751 | pb |
| 751 | lake |
| 752 | roe |
| 752 | lake |

```
752    lake
752    lake
837    roe
837    lake
837    lake
844    roe
```

만약 중복을 제거한다면 이해하기가 더 쉽다.

```
%%sqlite survey.db
select distinct taken, person from Survey order by taken asc, person desc;
```

```
619    dyer
622    dyer
734    pb
734    lake
735    pb
735    None
751    pb
751    lake
752    roe
752    lake
837    roe
837    lake
844    roe
```

도전 과제

1. Visited 테이블에서 별개로 구별되는 날짜를 반환하는 쿼리를 작성하세요.
2. 성(family name)으로 정렬된 Person 테이블에 과학자의 성명 전부를 화면에 출력하는 쿼리를 작성하세요.

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.

필터링 (Filtering)

목표

- 사용자가 정의한 조건을 만족하는 레코드를 선택하는 쿼리를 작성한다.
- 쿼리 절이 실행되는 순서를 설명한다.

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.

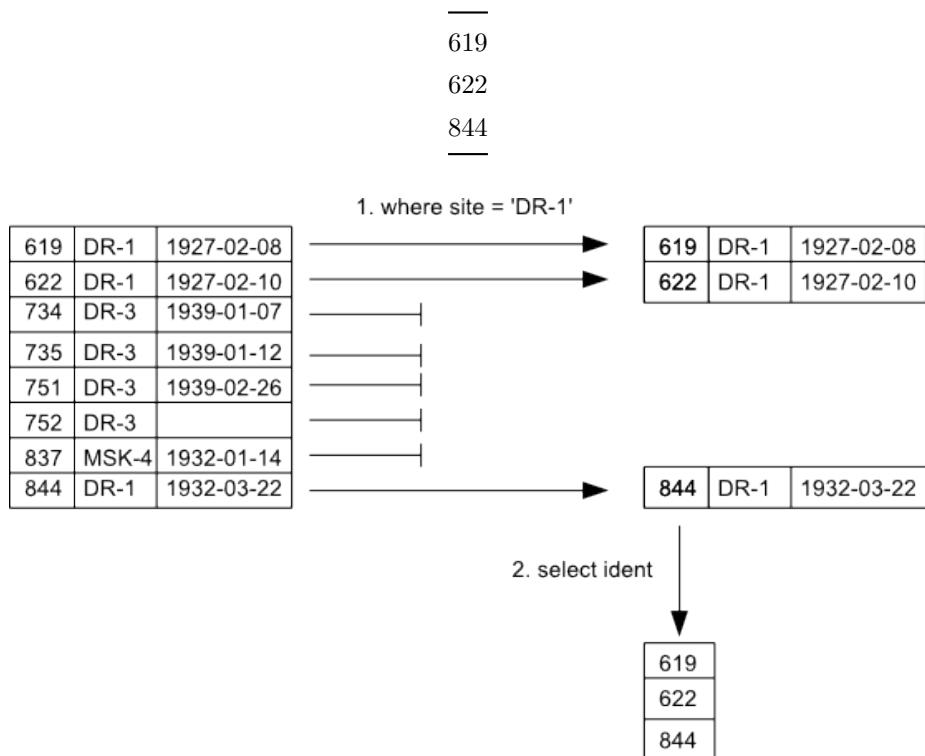
```
%load_ext sqlitemagic  
  
%%sqlite survey.db  
select * from Visited where site='DR-1';
```

| | | |
|-----|------|------------|
| 619 | DR-1 | 1927-02-08 |
| 622 | DR-1 | 1927-02-10 |
| 844 | DR-1 | 1932-03-22 |

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.

```
%%sqlite survey.db
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.

```
%%sqlite survey.db
select * from Visited where (site='DR-1') and (dated>='1930-00-00');
```

844 DR-1 1932-03-22

(각 테스트 주위의 팔호는 엄밀히 말해 필요하지는 않지만 쿼리를 좀더 읽기 쉽게 한다.)

대부분의 데이터베이스 관리자는 날짜에 대한 특별한 데이터 형식을 가진다. 사실 많이 있지만 두가지 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, 그럴수도 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.

```
%%sqlite survey.db
select * from Survey where person='lake' or person='roe';
```

| | | | |
|-----|------|------|-------|
| 734 | lake | sal | 0.05 |
| 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 | 11.25 |

다른 방식으로, `in`을 사용하여 특정 집합에 값이 있는지 확인할 수 있다.

```
%%sqlite survey.db  
select * from Survey where person in ('lake', 'roe');
```

| | | | |
|-----|------|------|-------|
| 734 | lake | sal | 0.05 |
| 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 | 11.25 |

`and`과 `or`을 조합할 수는 있지만, 어느 연산자가 먼저 수행되는지 주의할 필요가 있다. 만약 팔호를 사용하지 않는다면, 다음을 얻게 된다.

```
%%sqlite survey.db  
select * from Survey where quant='sal' and person='lake' or person='roe';
```

| | | | |
|-----|------|-----|-------|
| 734 | lake | sal | 0.05 |
| 751 | lake | sal | 0.1 |
| 752 | lake | sal | 0.09 |
| 752 | roe | sal | 41.6 |
| 837 | lake | sal | 0.21 |
| 837 | roe | sal | 22.5 |
| 844 | roe | rad | 11.25 |

상기 결과는 Lake가 측정한 염분량과 Roerich가 측정한 임의 측정값이다. 대신에 아마도 다음과 같은 결과를 얻고자 했을 것이다.

```
%%sqlite survey.db
select * from Survey where quant='sal' and (person='lake' or person='roe');
```

| | | | |
|-----|------|-----|------|
| 734 | lake | sal | 0.05 |
| 751 | lake | sal | 0.1 |
| 752 | lake | sal | 0.09 |
| 752 | roe | sal | 41.6 |
| 837 | lake | sal | 0.21 |
| 837 | roe | sal | 22.5 |

마지막으로 `distinct`와 `where`를 사용하여 두번째 수준의 필터링을 한다.

```
%%sqlite survey.db
select distinct person, quant from Survey where person='lake' or person='roe';
```

| | |
|------|------|
| lake | sal |
| lake | rad |
| lake | temp |
| roe | sal |
| roe | rad |

하지만, 기억하라. `distinct`는 처리될 때 선택된 칼럼에 표시되는 값에만 적용되고 전체 행에는 적용되지 않는다.

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 데이터

베이스에 쿼리를 작업하지 말고, 1만명 샘플을 뽑아 쿼리를 돌리거나 작은 프로그램을 작성해서 랜덤으로 혹은 그럴듯한 1만명 레코드를 생성해서 사용한다.

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.
- 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.

주요점

- `where`를 사용해서 불 조건(Boolean conditions)에 따라 레코드를 필터링한다.
- 필터링이 전체 레코드에 적용되어서, 조건을 실제로 표시되지 않는 필드에 사용할 수 있다.

새로운 값 계산하기

목표

- 각 선택된 레코드에 대해 새로운 값을 계산하는 쿼리를 작성한다.

주의깊이 탐험 기록을 다시 정독한 뒤에, 탐험대가 보고한 방사선 측정치가 5% 만큼 상향되어 수정될 필요가 있다는 것을 깨달았다. 저장된 데이터를 변형하기보다는 쿼리의 일부분으로서 즉석에서 계산을 수행할 수 있다.

```
%load_ext sqlitemagic  
  
%%sqlite survey.db  
select 1.05 * reading from Survey where quant='rad';
```

10.311

8.19

8.8305

7.581

4.5675

2.2995

1.533

11²³8₂₅

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 띠게된다.) 예를 들어, 온도 측정치를 화씨에서 섭씨로 소수점 아래 두자리에서 반올림하여 변환할 수 있다.

```
%%sqlite survey.db
select taken, round(5*(reading-32)/9, 2) from Survey where quant='temp';
```

| | |
|-----|--------|
| 734 | -29.72 |
| 735 | -32.22 |
| 751 | -28.06 |
| 752 | -26.67 |

다른 필드의 값을 조합할 수도 있다. 예를 들어, 문자열 접합 연산자 (string concatenation operator, `||`)를 사용한다.

```
%%sqlite survey.db
select personal || ' ' || family from Person;
```

| |
|-------------------|
| William Dyer |
| Frank Pabodie |
| Anderson Lake |
| Valentina Roerich |
| Frank Danforth |

`first`와 `last` 대신에 필드 이름으로 `personal`과 `family`을 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. |

| 성명 전부(Full Name) | 알파벳 순서 | 이유 |
|---------------------------------|----------------|--|
| 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 있는지에 따라 달라짐 |
| Elizabeth Alexandra May Windsor | Elizabeth | 군주가 통치하는 이름에 따라 알파벳순으로 정렬 |
| Thomas a Beckett | Thomas | 시성된(canonized) 이름에 따라 성인이름 사용 |

분명하게, 심지어 두부분 “personal”과 “family”으로 나누는 것도 충분
하지 않다.

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.

```
%%sqlite survey.db
select * from Person where ident='dyer' union select * from Person where ident='roe';
```

| | | |
|------|-----------|---------|
| dyer | William | Dyer |
| roe | Valentina | Roerich |

union을 사용하여 앞선 도전과제에서 기술되어 수정된 Roerich가 측정한, Roerich
만 측정한 염도 측정치의 통합 리스트를 생성하세요. 출력결과는 다음과 같아야
한다.

| | |
|-----|-------|
| 619 | 0.13 |
| 622 | 0.09 |
| 734 | 0.05 |
| 751 | 0.1 |
| 752 | 0.09 |
| 752 | 0.416 |
| 837 | 0.21 |
| 837 | 0.225 |

- Visited 테이블에 사이트 식별자는 '-'으로 구분되는 두 부분으로 구성되어 있다.

```
%%sqlite survey.db
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 0 을 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.)

주요점

- SQL은 쿼리의 일부로서 레코드의 값을 사용한 계산을 수행한다.

결측 데이터 (Missing Data)

목표

- 데이터베이스가 어떻게 결측 정보를 표현하는지 설명한다.
- 결측 정보를 다룰 때, 3개 값을 가진 로직(three-valued logic) 데이터베이스 사용을 설명한다.
- 결측 정보를 올바르게 처리하는 쿼리를 작성한다.

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(nothing 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.

```
%load_ext sqlitemagic
```

```
%%sqlite survey.db
select * from Visited;
```

| | | |
|-----|-------|------------|
| 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 | None |
| 837 | MSK-4 | 1932-01-14 |
| 844 | DR-1 | 1932-03-22 |

`Null` 다른 값과는 다르게 동작한다. 만약 1930년 이전 레코드를 선택한다면,

```
%%sqlite survey.db
select * from Visited where dated<'1930-00-00';
```

| | | |
|-----|------|------------|
| 619 | DR-1 | 1927-02-08 |
| 622 | DR-1 | 1927-02-10 |

결과 2개를 얻게 되고, 만약 1930년 동안 혹은 이후 레코드를 선택한다면,

```
%%sqlite survey.db
select * from Visited where dated>='1930-00-00';
```

| | | |
|-----|-------|------------|
| 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 |

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 값 을 알지 못한다면, 비교도 참인지 거짓인지 알수가 없다. 데이터베이스는 “알 수 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.

```
%%sqlite survey.db
select * from Visited where dated=NULL;
```

```
%%sqlite survey.db
```

```
select * from Visited where dated!=NULL;
```

null 인지 아닌지를 점검하기 위해서, 특별한 테스트 `is null`을 사용해야 한다.

```
%%sqlite survey.db  
select * from Visited where dated is NULL;
```

| | | |
|-----|------|------|
| 752 | DR-3 | None |
|-----|------|------|

혹은, 역으로는 `is not null`을 사용한다.

```
%%sqlite survey.db  
select * from Visited where dated is not NULL;
```

| | | |
|-----|-------|------------|
| 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 |

null 값은 나타나는 곳마다 두통을 일으킨다. 예를 들어, Dyer가 측정하지 않은 모든 염분 정보를 찾는다고 가정하자. 다음과 같이 쿼리를 작성하는 것은 당연하다.

```
%%sqlite survey.db  
select * from Survey where quant='sal' and person!='lake';
```

| | | | |
|-----|------|-----|------|
| 619 | dyer | sal | 0.13 |
| 622 | dyer | sal | 0.09 |
| 752 | roe | sal | 41.6 |
| 837 | roe | sal | 22.5 |

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.

```
%%sqlite survey.db
select * from Survey where quant='sal' and (person != 'lake' or person is null);
```

| | | | |
|-----|------|-----|------|
| 619 | dyer | sal | 0.13 |
| 622 | dyer | sal | 0.09 |
| 735 | None | sal | 0.06 |
| 752 | roe | sal | 41.6 |
| 837 | roe | sal | 22.5 |

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.

도전 과제

1. 날짜가 알려지지 않은 (즉 `null`) 항목은 빼고, 날짜 순으로 `Visited` 테이블에 있는 레코드를 정렬한 쿼리를 작성하세요.
2. 다음 쿼리가 무슨 결과를 할까요?

```
select * from Visited where dated in ('1927-02-08', null);
```

상기 쿼리가 실질적으로 무엇을 생기게 할까요?

3. 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?

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.

집합(Aggregation)

목표

- "집합(aggregation)을 정의하고 사용예를 제시한다.
- 집합하는 값을 계산하는 쿼리를 작성한다.
- 집합을 수행하는 쿼리의 실행을 추적한다.
- 결측 데이터가 어떻게 집합되는 동안에 다뤄지는지 설명한다.

이제 데이터의 평균과 범위를 계산하고자 한다. Visited 테이블에서 모든 날짜 정보를 어떻게 선택하는지 알고 있다.

```
%load_ext sqlitemagic  
  
%%sqlite survey.db  
select dated from Visited;
```

| |
|------------|
| 1927-02-08 |
| 1927-02-10 |
| 1939-01-07 |
| 1930-01-12 |
| 1930-02-26 |
| None |

1932-01-14

1932-03-22

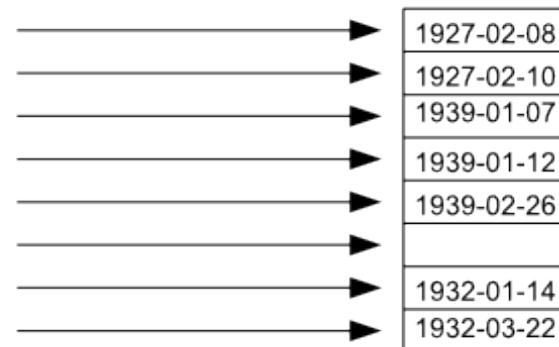
하지만 조합하기 위해서는 `min` 혹은 `max` 같은 집합 함수(aggregation function)를 사용해야만 한다. 각 함수는 입력으로 레코드 집합을 받고 출력으로 단일 레코드를 만든다.

```
%%sqlite survey.db  
select min(dated) from Visited;
```

1927-02-08

1. select date

| | | |
|-----|-------|------------|
| 619 | DR-1 | 1927-02-08 |
| 622 | DR-1 | 1927-02-10 |
| 734 | DR-3 | 1939-01-07 |
| 735 | DR-3 | 1939-01-12 |
| 751 | DR-3 | 1939-02-26 |
| 752 | DR-3 | |
| 837 | MSK-4 | 1932-01-14 |
| 844 | DR-1 | 1932-03-22 |



2. min

1927-02-08

```
%%sqlite survey.db  
select max(dated) from Visited;
```

1939-01-07

`min`과 `max`은 SQL에 내장된 단지 두개의 집합 함수다. 다른 세개는 `avg`, `count`, `sum`이 있다.

```
%%sqlite survey.db
select avg(reading) from Survey where quant='sal';
```

7.20333333333

```
%%sqlite survey.db
select count(reading) from Survey where quant='sal';
```

—
9
—

```
%%sqlite survey.db
select sum(reading) from Survey where quant='sal';
```

64.83

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.

```
%%sqlite survey.db
select min(reading), max(reading) from Survey where quant='sal' and reading<=1.0;
```

0.05 0.21

출력결과가 놀라움을 줄 수도 있지만, 원 결과값과 집합 결과를 조합할 수도 있다.

```
%%sqlite survey.db
select person, count(*) from Survey where quant='sal' and reading<=1.0;
```

| | |
|------|---|
| lake | 7 |
|------|---|

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.

또 다른 중요한 사실은 집합할 어떠한 값도 없을 때, 집합 결과는 0 혹은 다른 임의의 값 보다 “알지 못한다(don't know)”가 된다.

```
%%sqlite survey.db
select person, max(reading), sum(reading) from Survey where quant='missing';
```

| | | |
|------|------|------|
| None | None | None |
|------|------|------|

집합 함수의 마지막 중요한 한가지 기능은 매우 유용한 방식으로 나머지 SQL과는 일관되지 않는다는 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.

```
%%sqlite survey.db
select min(dated) from Visited;
```

| |
|------------|
| 1927-02-08 |
|------------|

명시적으로 항상 다음과 같이 필터하는 쿼리를 작성할 필요가 없다.

```
%%sqlite survey.db
```

```
select min(dated) from Visited where dated is not null;
```

1927-02-08

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.

```
%%sqlite survey.db
select person, count(reading), round(avg(reading), 2)
from Survey
where quant='rad';
```

roe 8 6.56

왜냐하면 데이터베이스 관리자가 각 과학자별로 구분된 집합하기 보다는 임의의 한 명의 과학자 이름만 선택하기 때문이다. 단지 5명의 과학자만 있기 때문에, 다음과 같은 형식의 5개 쿼리를 작성할 수 있다.

```
%%sqlite survey.db
select person, count(reading), round(avg(reading), 2)
from Survey
where quant='rad'
and person='dyer';
```

dyer 2 8.81

하지만, 이러한 접근법은 성가시고, 만약 50명 혹은 500명의 과학자를 가진 데이터 세트을 분석한다면, 모든 쿼리를 올바르게 작성할 가능성은 작다.

필요한 것은 데이터베이스 관리자가 `group by` 절을 사용해서 각 과학자별로 시간을 집합하도록 지시하는 것이다.

```
%%sqlite survey.db
select person, count(reading), round(avg(reading), 2)
from Survey
where quant='rad'
group by person;
```

| | | |
|------|---|-------|
| dyer | 2 | 8.81 |
| lake | 2 | 1.82 |
| pb | 3 | 6.66 |
| roe | 1 | 11.25 |

`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.

```
%%sqlite survey.db
select person, quant, count(reading), round(avg(reading), 2)
from Survey
group by person, quant;
```

| | | | |
|------|------|---|-------|
| None | sal | 1 | 0.06 |
| None | temp | 1 | -26.0 |
| dyer | rad | 2 | 8.81 |
| dyer | sal | 2 | 0.11 |
| lake | rad | 2 | 1.82 |
| lake | sal | 4 | 0.11 |
| lake | temp | 1 | -16.0 |
| pb | rad | 3 | 6.66 |

| | | | |
|-----|------|---|-------|
| pb | temp | 2 | -20.0 |
| roe | rad | 1 | 11.25 |
| roe | sal | 2 | 32.05 |

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.

```
%%sqlite survey.db
select person, quant, count(reading), round(avg(reading), 2)
from Survey
where person is not null
group by person, quant
order by person, quant;
```

| | | | |
|------|------|---|-------|
| dyer | rad | 2 | 8.81 |
| dyer | sal | 2 | 0.11 |
| lake | rad | 2 | 1.82 |
| lake | sal | 4 | 0.11 |
| lake | temp | 1 | -16.0 |
| pb | rad | 3 | 6.66 |
| pb | temp | 2 | -20.0 |
| roe | rad | 1 | 11.25 |
| roe | sal | 2 | 32.05 |

좀더 면밀하게 살펴보면, 이 쿼리는,

1. `Survey` 테이블에서 `person` 필드가 `null`이 아닌 레코드를 선택한다.
2. 상기 레코드를 부분집합으로 그룹지어서 각 부분집합의 `person`과 `quant`의 값을 같다.
3. 먼저 `person`으로 부분집합을 정렬하고나서 `quant`로 각 하위 그룹내에서도 정렬한다.

4. 각 부분집합의 레코드 숫자를 세고, 각각 reading 평균을 계산하고, 각각 person과 quant 값을 선택한다. (모두 동등하기 때문에 어느 것인지는 문제 가 되지 않는다.)

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, 5.0 으로 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 구 분자가 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?

주요점

- 집합 함수는 많은 값을 조합해서 하나의 새로운 값을 만든다.
- 집합 함수는 null 값을 무시한다.
- 필터링 다음에 집합이 일어난다.

데이터 조합하기 (Combining Data)

목표

- 두 테이블을 조인(join)하는 쿼리 연산을 설명한다.
- 의미있는 값의 조합만 포함하기 위해서 조인문을 포함하는 쿼리 결과를 어떻게 제한하는지 설명한다.
- 동일한 키를 갖는 테이블을 조인하는 쿼리를 작성한다.
- 기본키(primary key)과 외래키(foreign key)가 무엇인지 그리고 왜 유용한지 설명한다.
- 원자값(atomic value)이 무엇이고, 왜 데이터베이스 필드는 원자값만 포함해야 하는지 설명한다.

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.

```
%load_ext sqlitemagic  
  
%%sqlite survey.db  
select * from Site join Visited;
```

| | | | | | |
|------|--------|---------|-----|------|------------|
| DR-1 | -49.85 | -128.57 | 619 | DR-1 | 1927-02-08 |
| DR-1 | -49.85 | -128.57 | 622 | DR-1 | 1927-02-10 |
| DR-1 | -49.85 | -128.57 | 734 | DR-3 | 1939-01-07 |
| DR-1 | -49.85 | -128.57 | 735 | DR-3 | 1930-01-12 |
| DR-1 | -49.85 | -128.57 | 751 | DR-3 | 1930-02-26 |
| DR-1 | -49.85 | -128.57 | 752 | DR-3 | None |

| | | | | | |
|-------|--------|---------|-----|-------|------------|
| DR-1 | -49.85 | -128.57 | 837 | MSK-4 | 1932-01-14 |
| DR-1 | -49.85 | -128.57 | 844 | DR-1 | 1932-03-22 |
| DR-3 | -47.15 | -126.72 | 619 | DR-1 | 1927-02-08 |
| DR-3 | -47.15 | -126.72 | 622 | DR-1 | 1927-02-10 |
| DR-3 | -47.15 | -126.72 | 734 | DR-3 | 1939-01-07 |
| DR-3 | -47.15 | -126.72 | 735 | DR-3 | 1930-01-12 |
| DR-3 | -47.15 | -126.72 | 751 | DR-3 | 1930-02-26 |
| DR-3 | -47.15 | -126.72 | 752 | DR-3 | None |
| DR-3 | -47.15 | -126.72 | 837 | MSK-4 | 1932-01-14 |
| DR-3 | -47.15 | -126.72 | 844 | DR-1 | 1932-03-22 |
| MSK-4 | -48.87 | -123.4 | 619 | DR-1 | 1927-02-08 |
| MSK-4 | -48.87 | -123.4 | 622 | DR-1 | 1927-02-10 |
| MSK-4 | -48.87 | -123.4 | 734 | DR-3 | 1939-01-07 |
| MSK-4 | -48.87 | -123.4 | 735 | DR-3 | 1930-01-12 |
| MSK-4 | -48.87 | -123.4 | 751 | DR-3 | 1930-02-26 |
| MSK-4 | -48.87 | -123.4 | 752 | DR-3 | None |
| MSK-4 | -48.87 | -123.4 | 837 | MSK-4 | 1932-01-14 |
| MSK-4 | -48.87 | -123.4 | 844 | DR-1 | 1932-03-22 |

`join`은 두 테이블을 외적(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.

```
%%sqlite survey.db
select * from Site join Visited on Site.name=Visited.site;
```

| | | | | | |
|------|--------|---------|-----|------|------------|
| DR-1 | -49.85 | -128.57 | 619 | DR-1 | 1927-02-08 |
| DR-1 | -49.85 | -128.57 | 622 | DR-1 | 1927-02-10 |

| | | | | | |
|-------|--------|---------|-----|-------|------------|
| DR-1 | -49.85 | -128.57 | 844 | DR-1 | 1932-03-22 |
| DR-3 | -47.15 | -126.72 | 734 | DR-3 | 1939-01-07 |
| DR-3 | -47.15 | -126.72 | 735 | DR-3 | 1930-01-12 |
| DR-3 | -47.15 | -126.72 | 751 | DR-3 | 1930-02-26 |
| DR-3 | -47.15 | -126.72 | 752 | DR-3 | None |
| MSK-4 | -48.87 | -123.4 | 837 | MSK-4 | 1932-01-14 |

`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 추가하자 마자 데이터베이스 관리자는 두 다른 사이트에 관한 조합된 정보는 사용한 뒤에 버려버리고, 원하는 레코드만 남겨둔다.

조인 결과에 필드이름을 명기하기 위해서 `table.field`를 사용한 것에 주목하세요. 이렇게 하는 이유는 테이블이 동일한 이름을 가질 수 있고 어느 필드를 언급하는지 좀더 구체성을 뛸 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.

```
%%sqlite survey.db
select Site.lat, Site.long, Visited.dated
from   Site join Visited
on     Site.name=Visited.site;
```

| | | |
|--------|---------|------------|
| -49.85 | -128.57 | 1927-02-08 |
| -49.85 | -128.57 | 1927-02-10 |
| -49.85 | -128.57 | 1932-03-22 |
| -47.15 | -126.72 | None |
| -47.15 | -126.72 | 1930-01-12 |
| -47.15 | -126.72 | 1930-02-26 |
| -47.15 | -126.72 | 1939-01-07 |
| -48.87 | -123.4 | 1932-01-14 |

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.

```
%%sqlite survey.db
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;
```

| | | | | |
|--------|---------|------------|------|-------|
| -49.85 | -128.57 | 1927-02-08 | rad | 9.82 |
| -49.85 | -128.57 | 1927-02-08 | sal | 0.13 |
| -49.85 | -128.57 | 1927-02-10 | rad | 7.8 |
| -49.85 | -128.57 | 1927-02-10 | sal | 0.09 |
| -47.15 | -126.72 | 1939-01-07 | rad | 8.41 |
| -47.15 | -126.72 | 1939-01-07 | sal | 0.05 |
| -47.15 | -126.72 | 1939-01-07 | temp | -21.5 |
| -47.15 | -126.72 | 1930-01-12 | rad | 7.22 |
| -47.15 | -126.72 | 1930-01-12 | sal | 0.06 |
| -47.15 | -126.72 | 1930-01-12 | temp | -26.0 |
| -47.15 | -126.72 | 1930-02-26 | rad | 4.35 |
| -47.15 | -126.72 | 1930-02-26 | sal | 0.1 |
| -47.15 | -126.72 | 1930-02-26 | temp | -18.5 |
| -48.87 | -123.4 | 1932-01-14 | rad | 1.46 |
| -48.87 | -123.4 | 1932-01-14 | sal | 0.21 |
| -48.87 | -123.4 | 1932-01-14 | sal | 22.5 |
| -49.85 | -128.57 | 1932-03-22 | rad | 11.25 |

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, 외래키는 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.

대부분의 데이터베이스 디자이너는 모든 테이블은 잘 정의된 기본키가 있어야 된다고 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.

```
%%sqlite survey.db
select rowid, * from Person;
```

| | | | |
|---|----------|-----------|----------|
| 1 | dyer | William | Dyer |
| 2 | pb | Frank | Pabodie |
| 3 | lake | Anderson | Lake |
| 4 | roe | Valentina | Roerich |
| 5 | danforth | Frank | Danforth |

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, 스프레드쉬트와 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)

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';
```

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 기본키가 되는 필드값(혹은 필드의 집합)이다.
- 테이블사이에 기본키와 외래키를 매칭해서 의미없는 레코드의 조합을 제거 할 수 있다.
- 조인을 좀더 단순하고 효율적으로 만들기 위해서 키(key)는 원자값(atomic value)이 되어야 한다.

데이터 생성과 변형(Creating and Modifying Data)

목표

- 테이블을 생성하는 쿼리를 작성한다.
- 레코드를 삽입, 변형, 삭제하는 쿼리를 작성한다.

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 4 개를 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 객체” |

대부분의 데이터베이스는 불(boolean)과 날짜/시간 값도 지원한다. SQLite는 불 값을 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 사용하는야에 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 `where` 절과 `delete` 문을 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.

도전 과제

1. `Survey.person`의 `null`인 모든 사용자를 문자열 '`unknown`'으로 대체하는 SQL문을 작성하세요.
2. 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.

3. 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.)

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.

데이터베이스로 프로그래밍

목표

- SQL 쿼리를 실행하는 짧은 프로그램을 작성한다.
- SQL 쿼리를 포함하는 프로그램의 실행을 추적한다.
- 왜 대부분의 데이터베이스 응용프로그램이 SQL 보다 다른 범용 언어로 작성되는지 설명한다.

마무리하면서, 파일 같은 범용 프로그래밍 언어에서 데이터베이스를 어떻게 접근하는지 살펴보자. 다른 언어도 거의 같은 모델을 사용한다. 라이브러리와 함수 이름이 다를지 모르지만, 개념은 동일한다.

`survey.db`라는 이름의 파일에 저장된 SQLite 데이터베이스에서 위도와 경도를 선택하는 짧은 파일 프로그램이 다음에 있다.

```
import sqlite3
connection = sqlite3.connect("survey.db")
cursor = connection.cursor()
cursor.execute("select site.lat, site.long from site;")
results = cursor.fetchall()
for r in results:
    print r
cursor.close()
connection.close()

(-49.85, -128.57)
(-47.15, -126.72)
(-48.87, -123.4)
```

`sqlite3` 라이브러리를 가져오는 것부터 프로그램이 시작한다. 만약 MySQL, DB2 혹은 다른 데이터베이스에 접속한다면, 다른 라이브러리를 가져올 것이지만, 동일한 기능을 제공한다. 그래서 만약 다른 이 데이터베이스에서 저 데이터베이스로 바꾼다면 프로그램의 나머지 부분은 변경할 필요(적어도 그렇게 많지는 않다.)가 있다.

2번째 행이 데이터베이스에 연결을 설정한다. SQLite를 사용하기 때문에, 명시하는데 필요한 전부는 데이터베이스 파일 이름이다. 다른 데이터베이스 시스템은

사용자명과 비밀번호를 또한 제공하도록 요구할지도 모른다. 3번째 행은 연결을 이용하여 커서(cursor)를 생성한다. 편집기의 커서처럼, 커서의 역할은 데이터베이스에 어느 위치에 있는지 추적하는 것이다.

4번째 행에서 커서를 사용해서 사용자를 대신해서 데이터베이스에 쿼리 실행 요청을 한다. 쿼리는 SQL로 작성되고 문자열로 cursor.execute에 전달된다. SQL이 제대로 작성되어 있는지 확실히 하는 것이 사용자의 몫이다. 만약 제대로 작성되어 있지 않거나 실행될 때 뭔가 잘못되었다면, 데이터베이스는 오류를 보고한다.

5번째 행에 cursor.fetchall 호출에 응답하여 데이터베이스가 쿼리 결과를 반환한다. 결과는 결과집합에 각 레코드마다 하나의 항목을 가진 리스트다. 만약 리스트(6번째 행)를 루프 반복을 돌려서 리스트 항목(7번째 행)을 출력하면, 각각은 각 필드에 하나의 요소를 가진 튜플(tuple)인 것을 알 수 있다.

마지막으로, 8번째와 9번째 행은 커서와 데이터베이스 연결을 종료한다. 왜냐하면 데이터베이스는 한번에 열수 있는 제한된 숫자의 연결만 유지할 수 있기 때문이다. 하지만, 연결을 설정하는 것은 시간이 소요되어서, 단지 백만분의 수초 후에 다시 연결을 하고 또 다른 작업을 하려는 연결을 하고, 작업을 하고 나서 연결을 종료하는 것은 하지 말아야 한다.

실제 응용프로그램에서 쿼리는 사용자가 제공하는 값에 달려있다. 예를 들어, 다음 함수는 사용자의 ID를 매개변수로 받아서 이름을 반환한다.

Queries in real applications will often depend on values provided by users. For example, this function takes a user's ID as a parameter and returns their name:

```
def get_name(database_file, person_ident):
    query = "select personal || ' ' || family from Person where ident='" + person_ident + "'"

    connection = sqlite3.connect(database_file)
    cursor = connection.cursor()
    cursor.execute(query)
    results = cursor.fetchall()
    cursor.close()
    connection.close()

    return results[0][0]
```

```
print "full name for dyer:", get_name('survey.db', 'dyer')

full name for dyer: William Dyer
```

함수의 첫번째 행에 문자열 결합을 사용해서 사용자가 넘겨준 사용자 ID를 포함하는 쿼리를 완성한다. 단순하게 보일지 모르지만, 만약 누군가 다음 문자열을 입력값으로 준다면 무슨일이 일어날까?

```
dyer'; drop table Survey; select '
```

프로젝트 이름 뒤에는 쓰레기(garbage)처럼 보이지만, 매우 주의깊게 고른 쓰레기다. 만약 이 문자열을 쿼리에 삽입하면, 결과는 다음과 같다.

```
select personal || ' ' || family from Person where ident='dyer'; drop table Survey; select '
```

만약 쿼리를 실행하게 된다면, 데이터베이스에 있는 테이블 중의 하나를 삭제한다.

이것을 SQL 주입 공격(SQL injection attack)이라고 부른다. SQL 주입공격은 수년에 걸쳐서 수천개의 프로그램을 공격하는데 사용되었다. 특히, 많은 웹사이트가 먼저 사려깊게 입력값을 점검하지 않고 사용자에게서 데이터를 입력받는 값을 쿼리로 바로 입력한다.

악의를 가진 사용자가 다양한 많은 방식으로 쿼리에 명령어를 몰래 밀어넣으려고 한다. 이러한 위협을 다루는 가장 안전한 방식은 인용부호 같은 문자를 대체 상응값으로 대체하는 것이다. 그렇게 해서 안전하게 문자열 내부에 사용자가 입력한 무엇이든지 넣을 수 있다. 문자열로 문장을 작성하는 대신에 준비된 문장(prepared statement)를 사용해서 작업할 수 있다. 만약에 준비된 문장을 사용한다면, 예제 프로그램은 다음과 같다.

```
def get_name(database_file, person_ident):
    query = "select personal || ' ' || family from Person where ident=?;""

    connection = sqlite3.connect(database_file)
    cursor = connection.cursor()
    cursor.execute(query, [person_ident])
    results = cursor.fetchall()
```

```

cursor.close()
connection.close()

return results[0][0]

print "full name for dyer:", get_name('survey.db', 'dyer')

full name for dyer: William Dyer

```

주요 변경사항은 쿼리 문자열과 `execute` 호출에 있다. 쿼리 자체 형식을 만드는 대신에 쿼리 템플릿에 값을 삽입하고자 하는 곳에 물음표를 넣는다. `execute`를 호출할 때, 쿼리의 물음표 숫자만큼의 값을 담고 있는 리스트를 제공한다. 라이브러리는 입력값을 순서대로 물음표와 매칭하고 특수 문자를 별도 상응값으로 번역해서 안전하게 사용하게 된다.

도전 과제

1. 10.0에서 25.0 사이의 100,000개 난수를 가지는 레코드를 가지고, `reading`으로 불리는 단일 필드를 가지고, `Pressure`라는 단일 테이블을 가지고, `original.db`이라는 이름을 가지는 신규 데이터베이스를 파일에 생성하는 파이썬 프로그램을 작성하세요.
2. `original.db`과 동일한 구조를 가지는 `backup.db`으로 불리는 새로운 데이터베이스를 생성하는 파이썬 프로그램을 작성하세요. `backup.db`는 `original.db`에서 `backup.db`로 20.0보다 큰 모든 값을 복사한 값을 담고 있다. 어느 것이 더 빠른가요? 쿼리의 값을 필터링하는 것 혹은 주기억장치에 모든 것을 읽어드리고 파이썬에서 필터링하는 것 중에서 선택하세요.

주요점

- 일반적으로 범용 언어로 데이터베이스 응용프로그램을 작성하고 SQL 쿼리를 프로그램에 내장한다.

- 데이터베이스에 접속하기 위해서 프로그램은 접속하려는 데이터베이스 관리자에 특정된 라이브러리를 사용해야 한다.
- 프로그램은 하나 혹은 그 이상의 연결을 단일 데이터베이스에 열고, 각각에 대해서 활성화된 하나 혹은 그 이상의 커서를 가진다.
- 프로그램은 쿼리 결과를 배치모드로 혹은 한번에 모두 읽어들인다.

유용한 것 몇가지(extras)

학습 흐름에 자연스럽게 녹아들 수지 못한 몇가지 유용한 것이 있다. 여기에 그들 중 몇몇을 모았다.

Git 브랜치(Branching)

다음이 지금 상태 정보다.

```
$ git log
```

```
commit 005937fbe2a98fb83f0ade869025dc2636b4dad5
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:14:07 2013 -0400
```

```
Thoughts about the climate
```

```
commit 34961b159c27df3b475cf4415d94a6d1fcd064d
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:07:21 2013 -0400
```

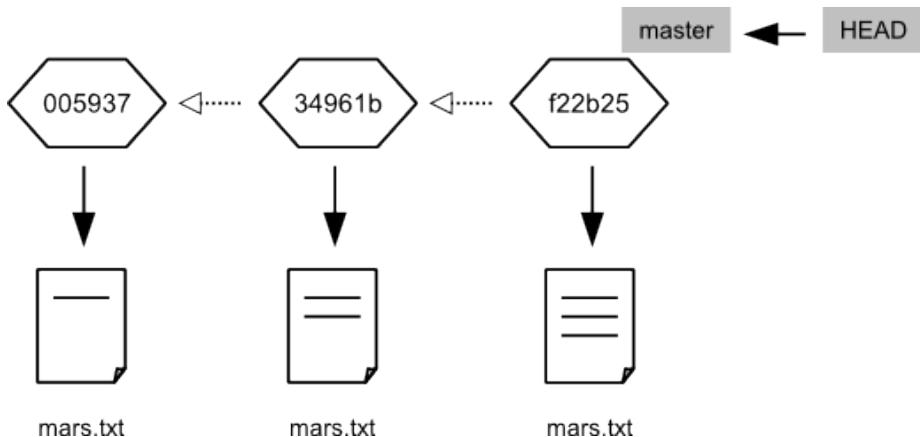
```
Concerns about Mars's moons on my furry friend
```

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

Starting to think about Mars

```
$ 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
```

상기 저장소 정보를 다음과 같이 도식화할 수 있다. (master라는 상자가 왜 있는지 곧 알게된다.)



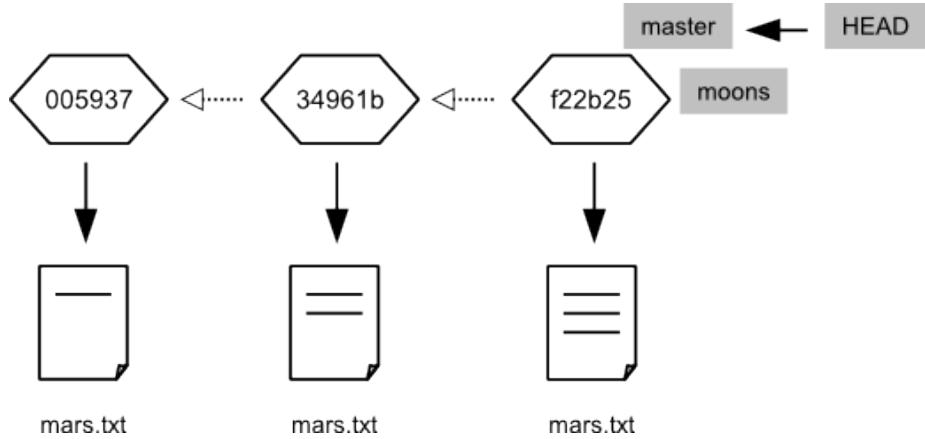
다음 명령어를 실행하자.

```
$ git branch moons
```

아무것도 하지 않는 것처럼 보인다. 하지만, 시스템 뒤에서는 `moons`이라는 브랜치 (branch, 분기)를 생성했다.

```
$ git branch
```

```
* master
  moons
```



Git는 지금부터 이력에 두개 복마크를 유지관리한다. 저장소를 설치할 때 자동적으로 생성되는 `master`, 방금 전에 생성한 `moons`이다.

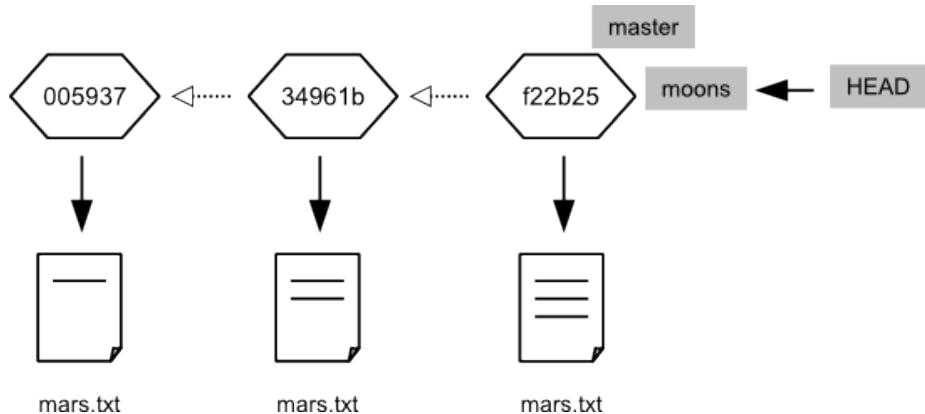
지금은 복마크 두개 모두 같은 수정기록(revision)을 지정하고 있지만 변경할 수 있다. `moons` 브랜치를 활성화하자.

```
$ git checkout moons
```

```
Switched to branch 'moons'
```

```
$ git branch
```

```
  master
* moons
```



파일은 같아 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
```

파일이 동일하기 때문에 파일에 라인을 하나 추가하자.

```
$ echo "Maybe we should put the base on one of the moons instead?" >> mars.txt
```

그리고 나서 완전히 새로운 파일을 추가하자.

```
$ echo "Phobos is larger than Deimos" > moons.txt  
$ ls
```

```
mars.txt      moons.txt
```

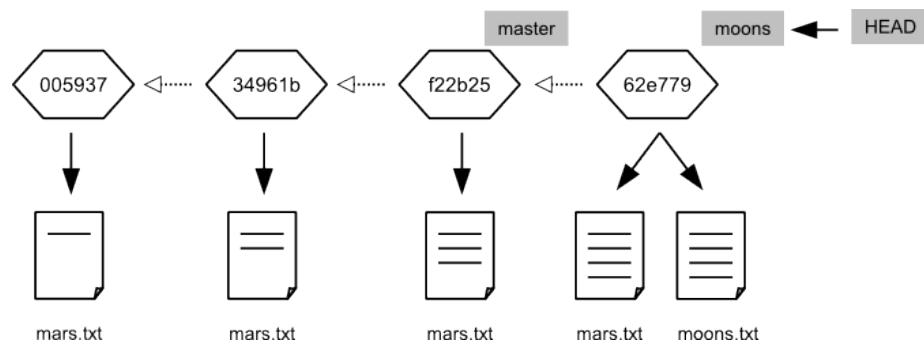
이제 Git를 통해서 변경된 파일이 하나 있고 신규 파일이 하나 더 있음을 확인할 수 있다.

```
$ git status  
  
# On branch moons  
# 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  
#  
# Untracked files:  
#   (use "git add <file>..." to include in what will be committed)  
#  
#       moons.txt  
no changes added to commit (use "git add" and/or "git commit -a")
```

상기 변경사항을 추가하고 커밋(commit)하자. (`git commit`에 `-A` 플래그 옵션은 “모든 것을 추가”를 의미한다.)

```
$ git add -A  
$ git status  
  
# On branch moons  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       modified:   mars.txt  
#       new file:   moons.txt  
#  
  
$ git commit -m "Thinking about the moons"  
  
[moons 62e7791] Thinking about the moons  
2 files changed, 2 insertions(+)  
create mode 100644 moons.txt
```

저장소가 다음에 보여진 상태로 바뀌었다.



`moons` 브랜치는 방금 전에 변경사항을 기록하고 전진했다. 하지만, `master`는 여전히 이전 상태 그대로다. `master` 브랜치로 다시 돌아가면,

```
$ git checkout master
```

변경사항이 사라진 것처럼 보인다.

```

$ ls

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

```

변경사항은 여전히 저장소에 있다— 다만, master가 현재 가리키고 있는 수정기록(revision)에는 없다. 본질적으로, 분기하기 전에 원본과 이력을 공유하는 병렬 타임라인을 생성했다.

이러한 점을 좀더 보여주기 위해서 master 브랜치에 약간의 변경을 만들자.

```

$ echo "Should we go with a classical name like Ares Base?" > names.txt
$ git status

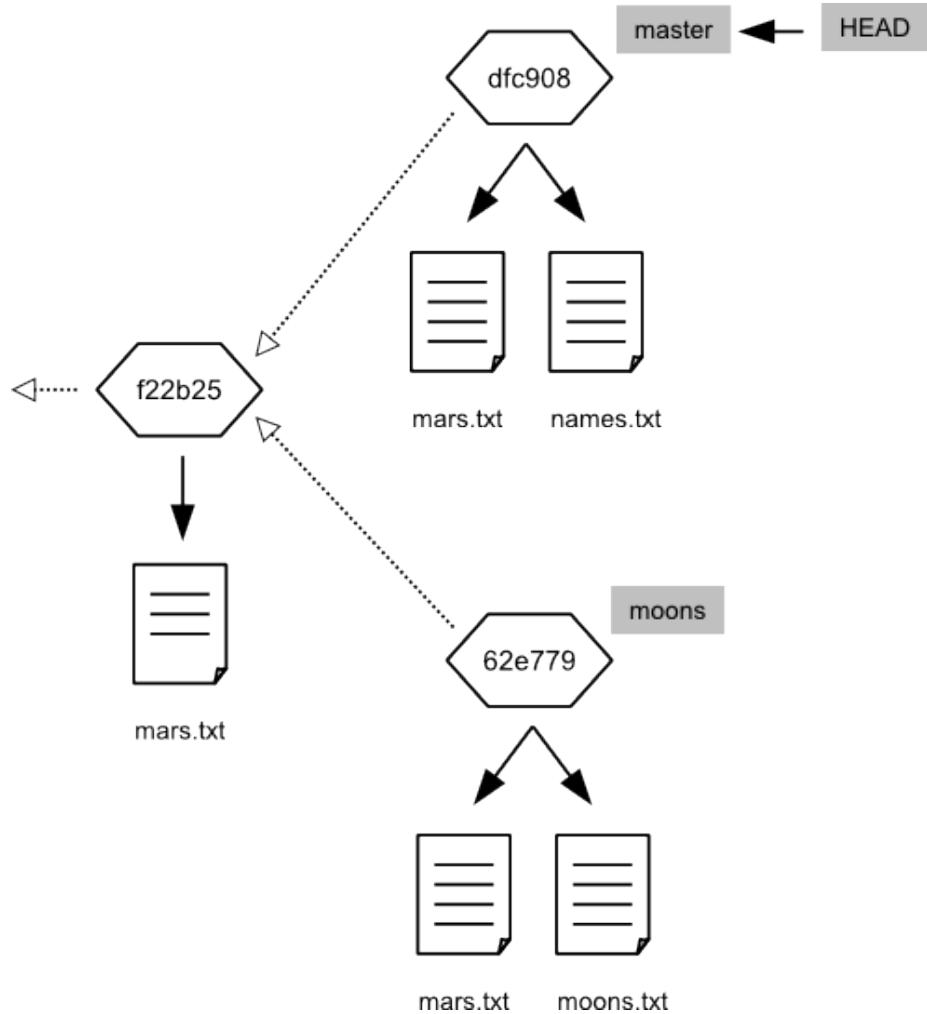
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       names.txt
nothing added to commit but untracked files present (use "git add" to track)

$ git add names.txt
$ git commit -m "We will need a cool name for our secret base"

[master dfcf908] We will need a cool name for our secret base
 1 file changed, 1 insertion(+)
 create mode 100644 names.txt

```

현재 저장소의 상태는 다음과 같다.



master와 moons 모두 원본 동일한 상태에서 출발하여 이동해갔지만, 다른 방식으로 옮겨갔다. 두 브랜치 모두 무한하게 독립적 상태를 계속 유지해 나갈 수 있다. 하지만, 어느 시점에는 아마도 변경 사항을 병합(merge)할 것이다. 지금 병합을 수행해보자.

```
$ git branch
* master
  moons

$ git merge moons
```

`git merge` 명령어를 실행하면, Git이 편집기를 열어서 작업하는 것에 대한 로그 항목을 적을 수 있게 한다. 초기 편집기 세션은 다음을 포함한다.

```
Merge branch 'moons'
```

```
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

만약 무언가 잘못된 것을 인식하고 병합을 하지 않기로 결정한다면, 파일에 모든 것을 삭제해야한다—Git이 빈 로그 메시지를 “진행하지 마세요”로 해석한다. 그렇지 않다면, #로 주석처리 되지 않는 모든 것이 로그에 저장된다. 이 경우에, 기본 로그 메시지와 함께 로그에 저장된다. 파일을 저장하고 편집기를 나오게 되면, Git은 다음을 화면에 표시한다.

```
Merge made by the 'recursive' strategy.

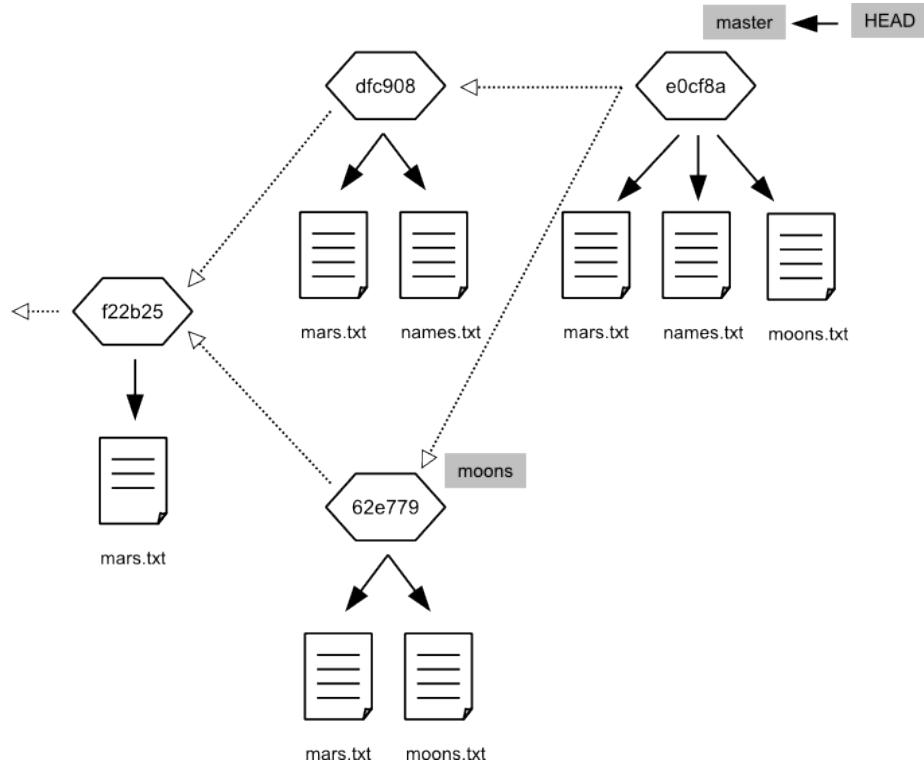
mars.txt | 1 +
moons.txt | 1 +
2 files changed, 2 insertions(+)
create mode 100644 moons.txt
```

아제 한곳에 모든 변경사항이 있다.

```
$ ls

mars.txt      moons.txt      names.txt
```

그리고, 저장소는 다음과 같다.



다음 Git 명령어를 이용하여 저장소 이력을 그림으로 표현한다.

We can ask Git to draw a diagram of the repository's history with this command:

```
$ git log --oneline --topo-order --graph
```

```
* e0cf8ab Merge branch 'moons'  
|\  
| * 62e7791 Thinking about the moons  
* | dfcf908 We will need a cool name for our secret base  
|/  
* 005937f Thoughts about the climate  
* 34961b1 Concerns about Mars's moons on my furry friend  
* f22b25e Starting to think about Mars
```

상기 ASCII 그림은 작은 일련의 변화에는 좋지만, 조금 크고 복잡한 것에 대해서는 문자 대신에 선을 이용하여 그래프를 그리는 GUI를 사용하는 것이 훨씬 낫다.

분기(브랜칭, branching)가 대부분의 초보자에게는 불필요한 복잡하다는 인상을 준다. 특히 저자가 한명인 프로젝트인 경우에는 더욱 그렇다. 결국, 만약 프록젝트에 변화를 주고자 한다면, 평행 우주를 생성해서 얻는 것은 무엇인가?

답변은 분기는 한번에 하나의 작업에 집중하기 쉽게 도와준다. 만약 파일 입출력(I/O)모듈이 항상 복소수 형태로 저장하면 작업이 훨씬 쉬워지는 것을 알아차렸을 때, 공간 상관관계(spatial correlation)를 계산하는 함수를 다시 작성하는 중간에 있다고 가정하자. 대부분의 사람은 공간 상관관계 변경사항을 따로 놓고, 파일 입출력(I/O)를 변경하고 나서, (잘 되면) 원래 작업으로 돌아온다.

이러한 접근법의 문제점은 파일 입출력(I/O)을 재작성하는 도중에 오류 처리를 다시 작성해야 한다는 것을 깨달을 때 조차도, 작업하는 것을 기억해야 한다는 것이다. 복잡하게 위로 쌓여진 작업으로 끝나는 것이 일반적이고, 다시 올바르게 정돈하기는 더욱 어렵다. 분기는 안전한 장소에 작업한 것을 두고, 새로운 문제를 해결하고 나서, 다시 선행한 작업을 다시 수행하게 한다.

실무에서, 대부분의 개발자는 절대 `master` 브랜치에서 직접 수정하지 않는다. 대신에 변경하고자 하는 모든 수정사항에 대해서 `master` 브랜치에서 새로운 브랜치를 생성하고 나서, 작업이 완료되었을 때, `master`와 작업한 브랜치를 병합한다. 앞선 가상의 사례로 돌아가서, 다음과 같이 작업한다.

1. 변경 사항에 대해서 `better-spatial-correlation` 같은 브랜치를 생성한다.
2. `Master`로 다시 돌아가서 상기 변경 사항에 대해서 `file-input-produces-complex-values`라는 또 다른 브랜치를 생성한다.
3. `file-input-produces-complex-values` 브랜치를 `master`와 병합한다.
4. `master`를 `better-spatial-correlation` 브랜치와 병합한다.
5. 공간 상관관계 함수 작업을 마무리 하고, 모든 것을 다시 `master`와 병합한다.

그리고, 만약 작업 중간에, 지도교수가 대학의 새로운 스타일 지침에 따라 그래프 그리는 루틴을 변경하라고 지시하면, 간단하게 `master`로 다시 전환해서, 적당한 브랜치를 생성하고, 변경사항을 기록하고, 원하는 그래프를 그리고, 병합을 하기 전까지 변경사항을 브랜치에 저장한다.

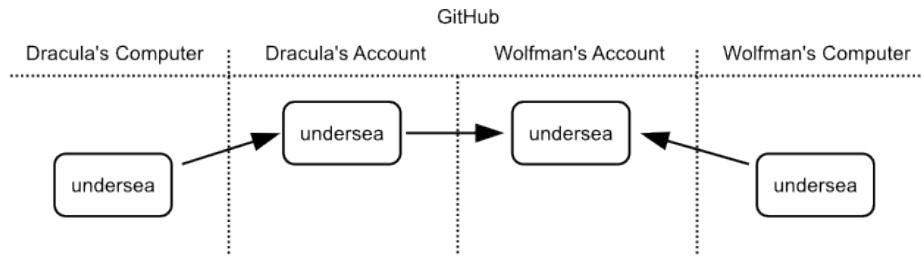
저장소 포킹(Forking)

Git 학습에서 살펴본 모든 사람이 단일 저장소에 푸쉬(push)하고 풀(pull)하는 모델은 완벽하게 사용성이 높다. 하지만, 이 모델은 여러분이 저장소에 쓰기 권한을 가진 경우에만 동작한다. 때때로, 누군가의 저장소에 기여를 하고 싶지만, 변경사항을 푸쉬할 수 없는 경우가 있다. 대신에, Github에 목표로 하는 저장소의 사본을 생성하고, 복사한 저장소에 일단 변경사항을 푸쉬하고, 원 저작자가 검토 후에 여러분의 변경사항이 승인되어 원래 저장소에 변경되도록 요청한다.

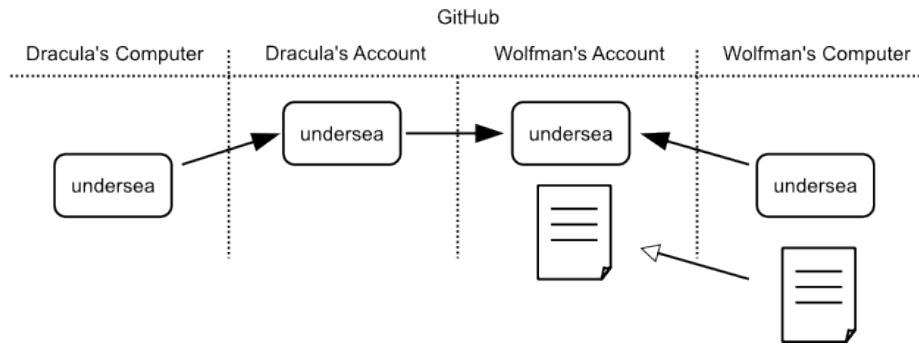
늑대인간이 드라큘라의 Github 프로젝트에 변경을 하고 싶다고 가정하자. 새로운 프로젝트를 생성하는 대신에, 늑대인간이 드라큘라의 프로젝트를 포크(forks)한다. 즉, GitHub에 복제(클론, clone)한다. GitHub 웹 인터페이스를 사용해서 이를 수행한다.



그리고 나서 늑대인간은 자신만의 데스크탑 컴퓨팅에 사본을 생성하려고 드라큘라의 것이 아닌 자신의 GitHub 저장소를 복제(clones)한다.



이제 늑대인간은 자신의 컴퓨터에 변경사항을 저장할 수 있다. 프로젝트에 수정사항을 만들어 로컬 저장소에 커밋(commit)한다. 그리고 나서, `git push`를 사용해서 수정사항을 GitHub에 복사한다.

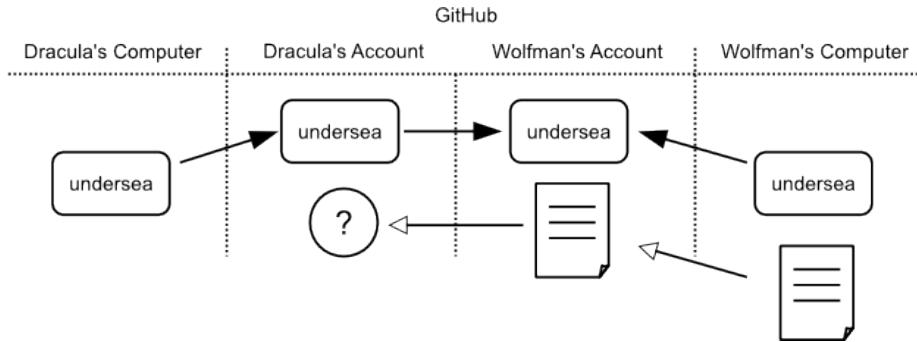


GitHub에 올라오게 되자마자, 새로운 잠재적 공동 협력자와 변경사항이 공유된다. 늑대인간은 자신의 포크를 공유하거나 다른 사람들이 변경된 늑대인간의 저장소를 포크할 수 있다. 늑대인간은 드라큘라와 변경사항을 공유할 수도 있다. 논문을 게시하기 전에 늑대인간의 논문을 리뷰(review)하는 것과 마찬가지로, 드라큘라는 변경사항을 보고 피드백을 주거나 변경사항을 승인하여 자신의 저장소와 병합할 수도 있다.

마찬가지로, 늑대인간은 드라큘라가 만든 변경사항을 리뷰하고 자신의 프로젝트 포크에 병합할지 판단할 수 있다. 만약 드라큘라가 중요한 기능을 삭제하기로 결정하거나, 완전히 프로젝트를 삭제하기로 결정한다면, 늑대인간의 수정되지 않는 사본을 유지한다. 가장 중요하게는 늑대인간이 유용한 수정사항을 생성하거나 공유하기 위해서 드라큘라가 자신의 저장소에 변경사항을 만드는데 늑대인간에게 권한을 줄 필요가 없다.

한 저장소에서 변경사항을 얻어오는 `git pull`와 변경사항을 다른 저장소에 적용하는 `git push`를 사용해서 이와 같은 리뷰와 병합 프로세스는 수작업으로 브랜치에서 수행한다. GitHub는 두 저장소가 변경사항 공유를 제안하는 도구가 있다: 풀 요청(pull request).

드라큘라와 변경사항을 공유하기 위해서, 늑대인간은 풀 요청(pull request)을 생성하고 드라큘라에게 “늑대인간이 변경사항을 저장소와 병합하고자 합니다”라고 통지한다.



풀 요청(pull request)는 병합이 일어나기를 대기하는 병합이다. 드라큘라가 풀 요청을 온라인에서 봤을 때, 늑대인간의 변경사항을 이해하고 논평할 수 있다. 풀 요청이 받아들여지기 앞서서 늑대인간과 드라큘라는 함께 여러차례 토의를 하고, 필요하면 브랜치를 생성한다.

This pull request can be automatically merged.

You can also merge branches on the [command line](#).



만약 친숙하게 들린다면, 이유는 이것이 과학 자체가 동작하는 방식이기 때문이다. 누군가 새로운 방법 혹은 결과를 발표할 때, 다른 과학자들이 즉시 이것 위에 무엇か 구축작업을 시작한다—본질적으로 다른 과학자들이 자신만의 포크를 생성하고 변경사항을 커밋하기 시작한다. 만약 첫번째 과학자가 두번째 작업에 동의한다면, 다음번 논문에 이러한 발견을 통합하는데, 풀 요청을 병합하는 것과 유사하다. 만약 동의하지 않는다면, 두번째 작업에서 구축을 결정할지 혹은 두가지 접근법을 병합을 시도하는 것은 또 다른 과학자에게 달려있다.

코드 리뷰 (Code Review)

소프트웨어 품질을 향상하는 방법은 많이 있다. 가장 효과적인 방법이 코드 리뷰(code review)다. GitHub은 프로젝트 내부에서 혹은 프로젝트 사이에 코드 리뷰를 촉진하기 위해서 풀 요청(pull request)을 사용한다.

풀 요청은 한 브랜치(branch)에서 또 다른 브랜치로 병합을 제안하는 방법이다. 브랜치가 동일한 저장소 혹은 같은 프로젝트의 다른 포크(forks)에 있을 수 있다. 본 학습은 다른 사람들과 효과적으로 일하는 방법에 대한 [Git 브랜치](#) 심화 과정이다.

드라큘라 프로젝트에 기여하고자 로컬 브랜치에 변경사항을 만든 늑대인간의 경우를 다시 살펴보자. GitHub에 변경사항을 만들어 브랜치에 푸쉬한다. 원래 저장소에

변경사항을 푸쉬하거나 만약 원래 저장소에 접근할 수 없다면 포크에 푸쉬한다. 그리고 나면 늑대인간은 웹브라우저의 저장소 페이지를 로드하고, 풀 요청을 할 수 있다. 만약 늑대인간이 몇분 내에 풀 요청을 한다면, 최근에 변경된 브랜치가 풀 요청을 할수 있는 버튼과 함께 저장소 헤더에 눈에 돋보이게 표시된다.

Your recently pushed branches:



만약 늑대인간이 더 이상 눈에 돋보이지 않게 된 브랜치에 풀 요청을 하고자 한다면, 모든 브랜치의 전반적인 상황을 로드하는 저장소 헤더의 링크를 사용할 수 있다.



브랜치 개요에서 각 브랜치별로 버튼이 있어서 풀 요청을 할 수 있다.



드라큘라가 풀 요청을 온라인에서 봤을 때, 늑대인간이 바꿀려고 하는 변경 사항을 볼 수 있다. 드라큘라는 코드 각 라인별로 남기거나, 늑대인간에게 피드백을 주기 위해서 광범위한 풀 요청에 코멘트를 할 수 있다. 늑대인간이 로컬 코드에 변경을 하고 브랜치에 푸쉬할 수도 있다. 풀 요청이 자동으로 변경사항을 반영하여 갱신한다.

풀 요청에 대해서 받는 피드백이 아주 폭넓거나 혹은 최소일 수도 있다. 처음에는 스타일적인 요소에 초점이 맞춰질 수도 있다 – 어디에 줄 바꿈을 넣는지, 변수에 대한 작명 규칙, 혹은 관용법에 맞는 코드 변환. 나중에는 구현을 다룰 수도 있다 - 코드가 함수와 모듈로 어떻게 구조화되는지. 중요하게, 검토자는 작업한 것을 좀더 작게, 논리적인 조각으로 나누어 다시 작성하도록 요청도 한다. 이런 변경 사항이 검토를 쉽게 하고, 프로젝트와 통합하기 쉽고, 버그를 덜 생성하게 한다.

코드를 리뷰하는 것은 더 복잡할 수 있다. 코드 리뷰 작업을 도와주는 도구는 많이 있다. 소프트웨어 린트(lint)는 코드가 스타일을 따르는지 확인하는 도구다. 자동 테스트는 이전 버그가 다시 재현되지 않도록 보장한다. 하지만, 어떠한 도구도 동료에 의한 주의깊은 검토를 받는 것만큼 효과적이지는 않다. 변경사항을 읽어가면서 뜻밖의 원천을 찾아서 세밀히 조사하는 것이 좋은 경험적 방법(heuristics)이다. 리뷰는 초기에 코드를 작성하는 것보다 더 어렵고, 검토자는 한시간 검토 뒤에는 급격하게 효율성이 저하된다. 코드 라인수자는 패치(patch)를 검토하는데 얼마나 오래 시간이 걸리는지에 대한 좋은 근사치가 된다.

대체로 훨씬 더 빠르지만, 프로세스가 과학 출판의 동료 검토와 유사하다. 규모가 큰 공개 소프트웨어 프로젝트에서, 최종적으로 승인되어 병합되기 전에 풀 요청이 수차례 개선되는 것은 매우 흔하다. 이와 같은 작업 방식은 코드의 품질을 유지할 뿐만 아니라, 지식을 전달하는 매우 효과적인 방식도 된다.

늑대인간은 첫번째 변경사항에 대한 검토를 기다리는 동안에, 로컬 저장소에 새로운 브랜치를 생성해서 작업을 계속할 수 있다. 새로운 브랜치에서 새로운 변경사항을 만들고, GitHub에 푸쉬하고, 두번째 풀 요청을 보낸다. 이것이 Git, Mercurial 그리고 다른 최신 버전 제어 시스템이 브랜치를 사용하는 중요한 방식이다. 드라큘라가 늑대인간의 변경사항을 검토하는데 몇일이 소요될 것이다. 검토가 완료될 때까지 정지하기보다, 늑대인간은 다른 브랜치로 전환해서 다른 작업을 하고, 드라큘라의 검토가 최종적으로 도착했을 때, 다시 원래의 작업으로 전환한다. 드라큘라가 변경사항이 마음에 들고, 자신의 프로젝트와 병합하고자 한다면, 버튼을 클릭해서 작업을 완료할 수 있다.

This pull request can be automatically merged.

You can also merge branches on the command line.



변경사항이 특정 브랜치에 승인되고, `master` (혹은 다른 브랜치)에 병합되자마자, 늑대인간은 브랜치를 삭제할 수 있다. 변경사항 자체는 병합되는 곳에 보존된다.

매뉴얼 페이지 (Manual Page)

`man` (manual의 약어) 명령어로 모든 유닉스 시스템 명령어에 대한 도움말을 얻는다. 예를 들어, `cp`에 대한 정보를 찾는 명령어가 다음에 있다.

```
$ man cp
```

화면에 표시되는 출력결과를 “매뉴얼 페이지(man page)”라고 한다.

매뉴얼 페이지는 셸의 기본 파일 뷰어로 표시된다. 대체로 프로그램이 `more`를 호출한다. `more`가 콜론 (':')을 화면에 표시할 때, 다음 페이지를 보려고 스페이스바를, 도움말을 얻는데는 'h'를, 끝내기 위해서는 문자 'q'를 각각 누른다.

사용 지침서보다는 참고자료로 사용되도록 고안되어서, `man` 출력결과는 보통 필요한 모든 것이 갖춰지도록 완전하지만 간결한다. 대부분의 매뉴얼 페이지는 절 (section)으로 나눠진다.

- NAME(이름): 명령어의 이름과 간략한 기술을 포함한다.
- SYNOPSIS(개요): 옵션 및 필수 매개변수를 포함하여 명령어를 어떻게 실행하는지 설명한다. (나중에 구문을 설명한다.)
- DESCRIPTION(기술): 명령어의 모든 옵션에 대한 기술을 포함하고, 개요보다 좀더 자세한 기술. 이번 절은 명령어가 어떻게 동작하는지에 대한 자세한 설명과 예제 사용을 포함할 수도 있다.
- EXAMPLES(예제): 따로 설명이 필요없이 예제가 나온다.
- SEE ALSO(기타 참조): 유용한 다른 명령어나 도움이 되는 다른 원천 정보를 목록으로 출력한다.

다른 절에는 저자(AUTHOR), 버그 보고(REPORTING BUGS), 저작권(COPYRIGHT), 이력(HISTORY), 버그(known BUGS), 호환(COMPATIBILITY) 정보를 포함한다.

개요(Synopsis)를 읽는 방법 우분투 리눅스상에서 cp 명령어에 대한 개요가 다음에 있다.

SYNOPSIS

```
cp [OPTION]... [-T] SOURCE DEST
cp [OPTION]... SOURCE... DIRECTORY
cp [OPTION]... -t DIRECTORY SOURCE...
```

상기 출력결과는 독자에게 명령어를 사용하는 방법이 3가지 있다고 제시한다. 첫 번째 사용례를 살펴보자.

```
cp [OPTION]... [-T] SOURCE DEST
```

[OPTION]이 의미하는 바는 cp 명령은 하나 이상의 옵션 플래그(flags)가 올 수 있다. 꺾쇠괄호 기호로 옵션사항이 되고, 생략(...) 기호로 하나 이상의 옵션 플래그를 사용한다는 의미가 된다. 예를 들어, [-T]가 꺾쇠괄호와 생략부호 뒤에 있다는 사실은 선택 사항이지만, 만약 사용이 된다면 다른 모든 옵션 다음에 와야함을 의미한다.

SOURCE는 원천 파일 혹은 디렉토리를 나타낸다. DEST는 목적 파일 혹은 디렉토리를 나타낸다. 각각의 정확한 의미는 기술(DESCRIPTION) 절 상단에 설명되어 있다.

다른 두가지 사용 예제도 비슷한 방식으로 읽을 수 있다. 주목할 것은 마지막 사용 예제에서 -t 옵션은 필수사항이라는 것이다. (왜냐하면 꺾쇠 팔호가 감싸고 있지 않다.)

기술(DESCRIPTION)항은 명령어와 사용법을 설명하는 문단 몇개로 시작하고 나서 사용가능한 옵션에 대해서 하나씩 기술해 나간다.

The following options are available:

- a Same as -pPR options. Preserves structure and attributes of files but not directory structure.

- f If the destination file cannot be opened, remove it and create a new file, without prompting for confirmation regardless of its permissions. (The -f option overrides any previous -n option.)

The target file is not unlinked before the copy. Thus, any existing access rights will be retained.

... ...

특정 옵션에 대한 도움말 찾기 관심있는 옵션으로 건너뛰고자 한다면, 슬래시 '/'를 사용해서 검색할 수 있다. (`man` 명령어 부분은 아니고, `more`의 기능이다.) 예를 들어, -t 옵션에 대해서 좀더 찾아보고자 한다면, /-t 타이핑하고 리턴키를 누른다. 그 후에 원하는 자세한 정보를 찾을 때까지 'n' 키를 눌러 다음 매칭되는 곳으로 이동한다.

```
-t, --target-directory=DIRECTORY
copy all SOURCE arguments into DIRECTORY
```

이 옵션은 간략한 형식 -t와 긴 형식 --target-directory가 있고 인자도 갖는다. 옵션의 의미는 모든 SOURCE 인자를 DIRECTORY로 복사한다. 그래서, 마지막에 디렉토리를 놓는 것 대신에 목적지를 명시적으로 줄 수 있다.

매뉴얼 페이지의 한계 매뉴얼 페이지는 명령어를 어떻게 실행하는지에 대한 빠른 확인을 하는데 유용하지만, 가독성은 좋지 않은 것으로 유명하다. 만약 매뉴얼 페이지에서 원하는 것을 찾지 못한다면—혹은 검색한 것을 이해하지 못한다면—“유닉스 명령어를 복사”해서 선호하는 검색엔진에 입력해보라: 종종 더 도움이 되는 결과를 준다.

다음이 도움이 될지도... explainshell.com 사이트는 유닉스 명령어를 부분으로 쪼개서 각각이 무엇을 수행하는지 잘 설명한다. 슬프게도, 반대로는 동작하지 않습니다.

권한 (Permissions)

권한(*permissions*)을 사용해서 유닉스는 누가 파일을 읽고, 변경할 수 있고, 실행할 수 있는지 제어한다. 이번 학습 후반부에 윈도우에서 어떻게 권한을 관리하는지도 살펴볼 것이다. 개념은 비슷하지만, 규칙은 다르다.

Nelle과 함께 시작하자. Nelle은 유일한 사용자명(user name), nnemo, 사용자 ID(user ID), 1404을 가지고 있다.

왜 정수형 ID일까? 왜 정수형 ID일까? 다시 한번 대답은 1970년 초 반으로 거슬러 올라간다. alan.turing 같은 문자열은 길이가 변하고, 다른 문자열과 비교하는 것은 많은 명령어를 필요로 한다. 반대로, 정수는 매우 적은 저장공간 (일반적으로 문자 4개)만을 사용하고, 하나의 명령어로 비교를 수행할 수 있다. 연산을 빠르고 간단하게 하기 위해서 프로그래머는 종종 내부적으로 정수를 사용해서 기록을 하고 정수를 표현을 위해서는 사용자가 친근한 텍스트로 변환하는 일종의 조회(lookup) 테이블을 사용한다. 종종 프로그래머는 프로그래머라 사용자

친근한 문자열 부분을 건너뛰고 정수만 사용한다. 마치 연구실에서 작업하는 연구자가 “아나콘다에 대한 알파 반응 실험” 대신에 “28번 실험”이라고 부르는 것과 유사하다.

사용자는 그룹(groups)에 임의의 숫자만큼 속할 수 있다. 각각의 그룹은 중복되지 않는 그룹이름(group name)과 숫자형식 그룹 ID(group ID)를 갖는다. 어느 사용자가 어떤 그룹에 속해있는가하는 목록 정보는 /etc/group 파일에 일반적으로 저장된다. (만약 지금 유닉스 컴퓨터 앞에 있다면, 그 파일을 확인하기 위해서 cat /etc/group 명령어를 실행해보자.)

이제 파일과 디렉토리를 살펴보자. 유닉스 컴퓨터의 모든 파일과 디렉토리는 하나의 소유자와 하나의 그룹에 속한다. 각 파일의 내용물과 함께 운영시스템은 파일을 소유하는 사용자 숫자 ID와 그룹정보를 저장한다.

사용자-그룹 (user-and-group) 모델은 각 파일마다 시스템의 모든 사용자는 다음 세가지 범주 중에 하나에 속하게 된다: 파일의 소유자, 파일 그룹의 일원, 그리고 그 밖의 모든 사람.

이 세가지 범주 각각에 대해서, 컴퓨터는 범주에 있는 사용자가 파일을 읽고, 쓰고, 실행(만약 프로그램이라면 실행)할 수 있는지를 기록한다.

예를 들어, 만약 파일이 다음과 같은 권한집합으로 구성된다면,

user

group

all

read

yes

yes

no

write

yes

no

no

execute

no

no

no

의미하는 바는 다음과 같다.

- 파일 소유자는 파일을 읽고 쓸 수 있지만, 실행할 수는 없다.
- 파일 그룹의 다른 사용자는 읽을 수는 있지만, 변경하거나 실행할 수는 없다.
- 그 밖의 모든 사람은 어떤 것도 할 수 없다.

실제 동작하는 이 모델을 살펴보자. `cd` 명령어로 디렉토리를 `labs`으로 변경하고 `ls -F`를 실행하면, `setup` 이름 끝에 `*`를 놓여진다. `setup`이 실행가능하다는 것을 표현하는 방식이다. 즉, (아마도) 컴퓨터가 실행할 수 있는 무엇이다.

```
$ cd labs  
$ ls -F  
  
safety.txt      setup*      waiver.txt
```

필요하지만 충분하지는 않다. 실행가능하다고 표기가 되었다는 사실이 실질적으로 프로그램의 일종을 담고있다는 의미는 아니다. 다음에 소개되는 명령어를 사용하여 지금 작성하고 있는 HTML 파일을 쉽게 실행가능한 파일로 표시할 수 있다. 사용하고 있는 운영시스템에 따라서 “실행(run)”을 하면 실행 실패(왜냐하면, 컴퓨터가 인식할 수 있는 명령집합을 담고 있지 않다.) 혹은 운영시스템이 자동으로 파일을 (웹브라우저) 같은 응용프로그램으로 열게 한다.

이제 `ls -l` 명령을 실행해보자:

```
$ ls -l
```

```
-rw-rw-r-- 1 vlad bio 1158 2010-07-11 08:22 safety.txt  
-rwxr-xr-x 1 vlad bio 31988 2010-07-23 20:04 setup  
-rw-rw-r-- 1 vlad bio 2312 2010-07-11 08:23 waiver.txt
```

-l 플래그는 ls 명령어가 장문 형식의 리스트 목록을 출력한다. 정보가 많아서 차례로 칼럼을 하나씩 살펴보자.

오른쪽에 파일 이름이 있다. 왼쪽으로 이동하면 그 옆에 가장 최근에 변경된 시간과 날짜 정보가 있다. 백업 시스템과 다른 도구는 이 정보를 다양한 방식으로 이용한다. 하지만, 언제 여러분 (혹은 권한을 가진 다른 사람)이 마지막에 파일을 변경했는지 확인하는데 사용한다.

변경시간 옆에 바이트로 파일 크기와 파일 소유자와 그룹이름이 있다. (이 경우 vlad가 소유자 bio가 그룹이 된다) 지금 두번째 칼럼(각 파일마다 1을 표시)은 건너뛴다. 왜냐하면 가장 관심을 가져야 하는 것이 첫번째 칼럼이기 때문이다. 첫 번째 칼럼은 파일 권한을 보여준다. 즉, 누가 읽고, 쓰고, 실행할 수 있는지 권한을 보여준다.

권한 문자열 중 하나를 좀 더 자세히 살펴보자: -rwxr-xr-x. 첫번째 문자는 무슨 타입(type)인지에 대한 정보를 제공한다. '-'은 정규 파일, 'd'은 디렉토리, 다른 문자는 좀 더 소수의 사람만 이해할 수 있는 것을 의미한다.

다음 세 문자는 파일 소유자가 무슨 권한을 가지고 있는지 알려준다. 여기서 파일 소유자는 파일을 읽고, 쓰고, 실행도 할 수 있다: rwx. 중간의 세쌍둥이는 그룹 권한을 정보를 보여준다. 만약 권한이 없다면, 대쉬로 표현된다. 그래서 r-x은 “읽고, 실행은 하지만, 쓸수는 없다.”는 의미를 가진다. 마지막 세쌍둥이는 파일의 소유자도 파일의 그룹에 있지도 않은 그빠의 누구나 무엇을 할 수 있는지 보여준다. 이 경우, 다시 r-x이여서, 시스템의 그 밖의 모든 사람은 파일의 내용을 볼 수 있고 실행도 할 있지만, 변경할 수는 없다. 권한을 변경하기 위해서, chmod 명령어를 사용한다. (chmod는 “change mode”의 약자) Vlad가 강의하는 과정의 최종 점수에 대한 권한을 보여주는 긴 목록 정보가 다음에 있다.

```
$ ls -l final.grd  
  
-rwxrwxrwx 1 vlad bio 4215 2010-08-29 22:30 final.grd
```

이럴 수가 있나요: 세상의 모든 사람이 읽을 수 있어요—그리고, 더욱 상황이 않좋게는 변경도 할 수 있어요. (거의 확실하게 동작하지 않겠지만, 프로그램으로 성적

파일을 실행할도 수 있어요.) 소유자 권한을 rw-으로 변경하는 명령어는 다음과 같다.

```
$ chmod u=rw final.grd
```

'u'는 사용자(즉, 파일 소유자)의 권한을 변경한다는 신호다. 그리고 rw는 새로운 권한집합이다. ls -l 명령어는 권한 변경이 동작하는 것을 보여준다. 왜냐하면, 소유자 권한이 이제 읽고 쓰는 것으로 설정이 변경되었다.

```
$ ls -l final.grd
```

```
-rw-rwxrwx 1 vlad bio 4215 2010-08-30 08:19 final.grd
```

그룹 권한을 읽을 수만 있도록 변경하기 위해서 chmod을 다시 실행하자.

```
$ chmod g=r final.grd
```

```
$ ls -l final.grd
```

```
-rw-r--rw- 1 vlad bio 4215 2010-08-30 08:19 final.grd
```

그리고 최종적으로 그 밖의 모든 사람(파일의 소유자도 그룹원도 아닌 시스템의 모든 사람)에게는 어떠한 권한도 주지 말자.

```
$ chmod a= final.grd
```

```
$ ls -l final.grd
```

```
-rw-r----- 1 vlad bio 4215 2010-08-30 08:20 final.grd
```

'a'는 그 밖의 모든 사람("all")의 권한을 변경한다는 신호를 준다. "=" 오른쪽에 아무것도 없기 때문에, 그 밖의 모든 사람의 권한은 없다.

또한 권한으로 검색도 할 수 있다. 예를 들어, 사용자가 실행할 수 있는 파일을 찾기 위해서 -type f -perm -u=x을 사용한다.

```
$ find . -type f -perm -u=x
```

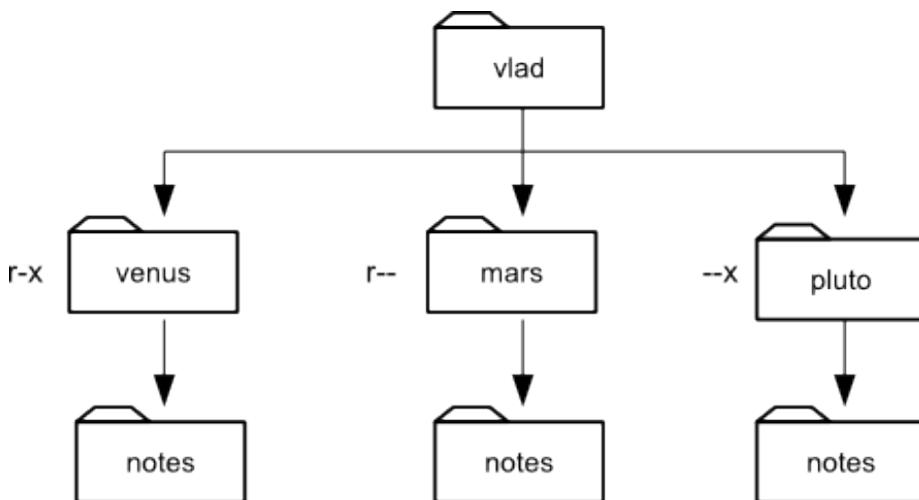
```
./tools/format  
./tools/stats
```

더 진도를 나아가기 전에, `ls -a -l`을 실행해서 평상시에 숨겨져있는 디렉토리 항목을 포함하는 긴 형식 목록 정보를 얻어보자.

```
$ ls -a -l  
  
drwxr-xr-x 1 vlad bio      0 2010-08-14 09:55 .  
drwxr-xr-x 1 vlad bio  8192 2010-08-27 23:11 ..  
-rw-rw-r-- 1 vlad bio  1158 2010-07-11 08:22 safety.txt  
-rwxr-xr-x 1 vlad bio 31988 2010-07-23 20:04 setup  
-rw-rw-r-- 1 vlad bio  2312 2010-07-11 08:23 waiver.txt
```

. 과 .. (현재 디렉토리와 부모 디렉토리)에 대한 권한은 'd'로 시작한다. 하지만, 뒷쪽 권한부분을 살펴보자: 'x'는 “실행(execute)”기능이 활성화 된다는 의미다. 무슨 의미일까? 디렉토리는 프로그램이 아닌데— 어떻게 “실행”할 수 있을까?

사실 'x'는 디렉토리에 대해서 다른 무언가를 의미한다. 디렉토리를 돌아다닐(*traverse*) 수 있는 권리라는 있지만, 내용(콘텐츠)은 볼 수는 없다. 차이가 미묘하다. 그래서 예제를 통해서 살펴보자. Vlad 홈 디렉토리는 venus, mars, pluto라는 3개의 하위 디렉토리를 담고 있다.



디렉토리 각각은 `notes`라는 하위 디렉토리가 있다. 하위의 하위 디렉토리는 다양한 파일을 담고 있다. 만약 `venus` 디렉토리에 대한 사용자 권한이 'r-x'라면, 그러면

만약 사용자가 `ls` 명령어를 사용해서 `venus`와 `venus/notes` 내용(콘텐츠)를 보려고 한다면, 컴퓨터는 양쪽을 모두 사용자가 볼 수 있게 한다. 만약 `mars` 디렉토리의 권한이 'r-'만 있다면, `mars`와 `mars/notes` 디렉토리 내용을 모두 읽을 수 있게 한다. 하지만 `pluto` 디렉토리에 대한 권한이 '-x'만 있으면, `pluto` 디렉토리에 무엇이 있는지 볼 수 없다. `ls pluto` 명령어를 실행하면 내용을 볼 수 있는 권한이 없다고 시스템이 응답한다. 하지만, `pluto/notes` 내부를 보고자 한다면, 시스템이 사용자가 볼 수 있게 해준다. `pluto`를 통과해서 갈 수는 있으나 내부에 무엇이 있는지를 볼 수 없다. 이와 같은 기법이 그 밖의 모든 것을 공개하지 않고, 전체적으로 바깥 세상에 디렉토리의 일부만 보여주는 방법이다.

윈도우(Windows)는 어떨까? 상기 살펴본 것이 유닉스 권한관리의 기본이다. 서두에 언급했듯이, 윈도우에서는 다르게 동작한다. 윈도우에서 권한은 접근 제어 목록(access control lists), ACLs에서 정의된다. ACT은 둘씩 짹지은 목록으로 각각은 “누가(who)”와 “무엇(what)”을 조합한다. 예를 들어, 미이라(Mummy)에게 읽거나 삭제할 수 있는 권한은 주지 않고 파일에 데이터만 추가하는 권한을 줄 수 있고, 프랑켄슈타인에게는 파일이 무엇을 담고 있는지 볼 수 없지만, 파일을 삭제할 수 있는 권한을 줄 수 있다.

유닉스 모델보다 좀더 유연하지만, 작은 시스템에서 이해하고 관리하기는 좀더 복잡하다. (만약 큰 컴퓨터 시스템이 있다면, 어떤것도 관리하거나 이해하기 쉽지 않다.) 최신 유닉스 일부 모델은 ACL 뿐만 아니라 이전 읽기-쓰기-실행 권한 모델도 지원하지만, 거의 사용자가 잘 사용하지는 않는다.

An ACL is a list of pairs, each of which combines a “who” with a “what”. For example, you could give the Mummy permission to append data to a file without giving him permission to read or delete it, and give Frankenstein permission to delete a file without being able to see what it contains.

This is more flexible than the Unix model, but it's also more complex to administer and understand on small systems. (If you have a large computer system, *nothing* is easy to administer or understand.) Some modern variants of Unix support ACLs as well as the older read-write-execute permissions, but hardly anyone uses them.

쉘 변수(Shell Variables)

쉘도 단순한 프로그램이다. 그래서 다른 프로그램처럼 변수도 갖는다. 변수의 역할은 프로그램 실행을 제어한다. 그래서 변수의 값을 변경해서, 쉘과 다른 프로그램의 동작을 변경한다.

`set` 명령어를 실행해서 전형적인 쉘 세션에 있는 변수 몇가지를 살펴보는 것으로 시작하자.

```
$ set

COMPUTERNAME=TURING
HOME=/home/vlad
HOMEDRIVE=C:
HOSTNAME=TURING
HOSTTYPE=i686
NUMBER_OF_PROCESSORS=4
OS=Windows_NT
PATH=/Users/vlad/bin:/usr/local/git/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin
PWD=/home/vlad
UID=1000
USERNAME=vlad
...
```

보게되면, 몇가지가 있다. 사실 상기 보여진 것보다 4배 혹은 5배 더 많다. 컴퓨터에 있는 것을 보기 위해서 `set`를 사용하는 것이 유닉스 조차도 약간 이상해 보인다. 하지만, 만약 어떤 인자도 설정하기 않는다면, 설정할 수 있는 모든 것을 보여준다.

모든 변수는 이름이 있다. 관례로, 항상 존재하는 변수는 대문자 이름이 주어진다. 모든 쉘 변수 같은 UID 같은 숫자처럼 보이는 것 조차도 문자열이다. 필요 시에 문자열을 다른 형으로 변환하는 것은 프로그램에 달려있다. 예를 들어, 만약 컴퓨터가 얼마나 많은 프로세서를 가지고 있는지 알고자 한다면, `NUMBER_OF_PROCESSORS` 변수값을 문자열에서 정수형으로 변환한다.

마찬가지로, `PATH` 같은 변수는 값을 리스트로 저장한다. 이 경우에 관례는 구분자로 콜론(':')을 사용하는 것이다. 만약 프로그램이 경로 정보를 저장한 리스트에서

각 항목 요소를 뽑아내고자 한다면, 변수 문자열 값을 조각내서 쪼개 꺼내는 것은 프로그램이 할 일이다.

경로(PATH) 변수 PATH 변수만 좀 더 자세히 살펴보자. 경로변수 값은 쉘의 검색 경로(search path)를 정의한다. 즉, 특정 디렉토리에 있는 프로그램을 지정하지 않고 탐색할 때, 쉘이 실행 가능한 프로그램 대상으로 찾는 디렉토리 리스트다.

예를 들어, `analyze` 같은 명령어를 탐색할 때, 쉘은 둘 중 `./analyze` 혹은 `/bin/analyze`을 실행할지를 결정해야 한다. 쉘이 사용하는 규칙은 단순하다. PATH 변수에 있는 디렉토리를 순차적으로 쉘이 확인하고, 특정 디렉토리에 요청한 이름을 가진 프로그램을 찾는다. 매칭되는 프로그램을 발견하자마자, 검색을 중지하고 프로그램을 실행한다.

어떻게 동작하는지 보여주기 위해서, 경로(PATH)의 컴포넌트를 한 줄에 하나씩 리스트했다.

```
/Users/vlad/bin  
/usr/local/git/bin  
/usr/bin  
/bin  
/usr/sbin  
/sbin  
/usr/local/bin
```

컴퓨터에 사실 3군데 다른 디렉토리(`/bin/analyze`, `/usr/local/bin/analyze`, `/users/vlad/analyze`)에 `analyze`로 불리는 프로그램이 3개 있다. PATH에 목록으로 올라간 순서로 쉘이 디렉토리를 검색하기 때문에 가장 먼저 `/bin/analyze`을 찾아 실행시킨다. 프로그램의 전체 경로명을 탐색하지 않는다면, `/users/vlad/analyze` 프로그램을 결코 찾지 못한다는 것을 주목하라. 왜냐하면 `/users/vlad` 디렉토리가 PATH 목록에 있지 않기 때문이다.

변수 값 표시하기 변수 HOME 변수 값을 표시하자.

```
$ echo HOME
```

HOME

“HOME”만 출력하는데 원하는 것이 아니다. (설사 실제로 요청한 것이기는 하다.) 대신에 다음과 같이 시도해보자.

```
$ echo $HOME
```

/home/vlad

달러 기호는 변수 이름말고 변수의 값을 원한다고 셸에 알려준다. 달러 기호는 마치 와일드카드처럼 동작한다. 프로그램을 실행하기 전에 요청한 것에 대해서 치환을 헬이 자동으로 한다. 이런 확장기능 덕분에 실제로 실행한 것은 echo /home/vlad이고 정확한 것을 표시한다.

변수 생성하고 변경하기 변수를 생성하는 것은 쉽다. “=”을 사용해서 값을 이름에 할당한다.

```
$ SECRET_IDENTITY=Dracula  
$ echo $SECRET_IDENTITY
```

Dracula

값을 바꾸기 위해서는 단지 새로운 값을 할당한다.

```
$ SECRET_IDENTITY=Camilla  
$ echo $SECRET_IDENTITY
```

Camilla

매번 셸을 실행할 때마다 특정 변수를 자동으로 설정하려고 한다면, 홈 디렉토리에 .bashrc 파일에 명령어를 저장한다. -a 옵션을 특별히 사용하지 않는다면, 앞쪽의

”문자는 ls 명령어가 파일 목록을 출력할 때 이 파일을 제외시킨다. 일반적으로 이러한 것에 대해서는 걱정하고 싶지 않다. 끝쪽의 “rc”는 “run control(실행 제어)”의 약자로 수십년 전에는 매우 중요한 무언가를 의미했지만, 지금은 어떻게 동작하고 왜 사용하는지 이해하지 않고 모든 사람이들이 따라가는 관례가 되었다.

예를 들어, /home/vlad/.bashrc 파일에 다음이 있다.

```
export SECRET_IDENTITY=Dracula
export TEMP_DIR=/tmp
export BACKUP_DIR=$TEMP_DIR/backup
```

상기 3라인은 변수 SECRET_IDENTITY, TEMP_DIR, BACKUP_DIR을 생성하고 내보내기를 해서 쉘이 실행하는 임의의 프로그램도 변수를 볼 수 있다. BACKUP_DIR 정의는 TEMP_DIR 값에 의존하는 것을 주목하세요. 그래서 만약 임시 파일 위치를 변경한다면, 백업이 자동적으로 재지정된다.

alias 명령어를 사용해서 자주 타이핑하는 것에 대한 단축키를 생성하는 것도 흔하다. 예를 들어, 특정 인자 집합으로 /bin/zback을 실행하는데 별칭(alias)으로 backup을 정의할 수도 있다.

```
alias backup=/bin/zback -v --nostir -R 20000 $HOME $BACKUP_DIR
```

별칭(alias)는 많은 타이핑을 줄여주고 그렇게 함으로써 타이핑 실수도 줄여주는 효과가 있다. 자주 사용하는 검색엔진에서 “sample bashrc”를 검색어로 넣고 검색하면 다른 별칭(alias)과 bash 기법에 대한 흥미로운 사실을 많이 알게된다.

원격 작업(Working Remotely)

목표

- SSH가 무엇인지 설명한다.
- SSH 키(SSH Key)가 무엇인지 설명한다.
- SSH 키 페어(SSH Key Pair)를 생성한다.
- SSH 키를 원격 서버에 추가한다.

- SSH 키를 어떻게 사용하는지 배운다.

데스크탑이나 노트북 컴퓨터에 쉘을 사용할 때 무슨일이 생기는 좀더 자세히 살펴보자. 첫번째 단계는 로그인(login)해서 운영체제가 접속한 사용자가 누구이고 무슨 작업을 하도록 허락되었는지를 확인한다. 사용자 이름과 비밀번호를 타이핑해서 수행하게 된다. 즉, 운영체제는 레코드에 저장된 입력한 값을 확인하고, 만약 일치하게 된다면 사용자를 위해서 쉘을 실행한다.

명령어를 타이핑할 때, 타이핑하는 문자를 나타내는 0과 1이 키보드에서 쉘로 전달된다. 쉴은 사용자가 타이핑한 것을 나타내기 위해 문자를 화면에 출력한다. 그리고 나서 만약 타이핑한 것이 명령어면, 쉴은 명령어를 실행하고 출력결과를 (만약 출력할 것이 있다면) 화면에 출력한다.

실험결과 데이터베이스를 관리하는 지하실 서버 같은 다른 컴퓨터에 명령어를 실행하고자 한다면 어떨까? 이를 위해서, 먼저 해당 컴퓨터에 로그인 해야한다. 이를 원격 로그인(remote login)이라고 하고 원격 컴퓨터(remote computer)라고 부른다. 로컬 쉘과 동일하게 사용자를 위해서 원격 컴퓨터에서 명령어를 실행하고 결과를 로컬 컴퓨터 화면에 출력한다.

원격 로그인하기 위해서 사용하는 툴이 시큐어 쉘(secure shell) 즉, SSH다. 특히, `ssh username@computer` 명령어는 SSH를 실행해서 지정한 원격 컴퓨터에 접속한다. 로그인한 후에 원격 컴퓨터의 파일과 디렉토리를 사용하기 위해서 원격 쉘을 이용할 수 있다. `exit`나 컨트롤-D(Control-D)를 타이핑하여 원격 쉘을 끝내고 이전 쉘로 돌아온다.

다음 예제에서, 원격 컴퓨터의 명령어 프롬프트가 \$ 대신에 `moon>`다. 어느 컴퓨터가 무엇을 하는지 좀더 명확히하기 위해서, 원격 컴퓨터에 보내는 명령어와 출력결과를 들여쓰기 한다.

```
$ pwd
```

```
/users/vlad
```

```
$ ssh vlad@moon.euphoric.edu
```

```
Password: *****
```

```
moon> hostname
```

```
moon

moon> pwd

/home/vlad

moon> ls -F

bin/      cheese.txt    dark_side/    rocks.cfg

moon> exit

$ pwd

/users/vlad
```

시큐어 쉘은 원격 쉘(remote shell)을 나타내는 오래된 **rsh** 프로그램과 대조하기 위해서 “시큐어(secure)”라고 부른다. 과거에 모든 사람이 서로를 신뢰하고, 이름만으로 컴퓨터의 모든 칩을 알고 있을 때, 네트워크를 타고 정보를 전송할 때 가장 민감한 정보를 제외하고 어떤 것도 암호화하지 않았다. 하지만, 악의를 가진 사람이 네트워크 트래픽을 볼 수 있고, 사용자이름과 비밀번호를 훔칠 수 있고, 악의를 가지고 이용할 수도 있다는 것을 뜻한다. SSH는 이러한 점을 방지하거나, 최소한 늦추기 위해서 고안되었다. SSH는 다른 컴퓨터 사이에 주고 받는 메시지에 무엇이 있는지 외부인이 볼 수 없도록 정교하고, 수많은 검증을 거친 암호화 프로토콜을 사용한다.

“시큐어 복사(secure copy)”를 나타내는 **scp**로 불리는 동반 프로그램도 **ssh**는 가지고 있다. SSH같은 종류의 연결을 사용하여 원격 컴퓨터에서 혹은 컴퓨터로 파일을 복사할 수 있게 한다. 명령어 이름은 **cp**와 **ssh**를 조합해서 작업을 수행한다. 파일을 복사하기 위해서 원천(소스, source)과 목적 경로를 명기해야한다. 경로명에는 컴퓨터의 이름을 포함할 수도 있다. 만약 컴퓨터 이름을 생략한다면, **scp**는 지금 실행중인 로컬 컴퓨터를 가정한다. 예를 들어, 다음 명령어는 최신 결과를 복사해서 지하실에 있는 백업 서버로 복사하면서 진행사항을 화면에 출력한다.

```
$ scp results.dat vlad@backupserver:backups/results-2011-11-11.dat
Password: *****
```

```
results.dat          100% 9 1.0 MB/s 00:00
```

전체 디렉토리를 복사하는 것도 유사하다. `-r` 옵션을 사용해서 재귀적으로 복사한다는 신호를 전달한다. 예를 들어, 다음 명령어는 백업서버에서 모든 결과를 노트북으로 복사하여 가져온다.

```
$ scp -r vlad@backupserver:backups ./backups
```

```
Password: *****
```

```
results-2011-09-18.dat      100% 7 1.0 MB/s 00:00
results-2011-10-04.dat      100% 9 1.0 MB/s 00:00
results-2011-10-28.dat      100% 8 1.0 MB/s 00:00
results-2011-11-11.dat      100% 9 1.0 MB/s 00:00
```

SSH로 할 수 있는 것이 하나 더 있다. 백업서버에 `backups/results-2011-11-12.dat` 파일을 이미 생성했는지 확인한다고 가정하자. 로그인에서 `ls` 명령어를 타이핑하는 대신에 다음과 같이 할 수 있다.

```
$ ssh vlad@backupserver "ls results"
```

```
Password: *****
```

```
results-2011-09-18.dat  results-2011-10-28.dat
results-2011-10-04.dat  results-2011-11-11.dat
```

SSH가 원격 사용자이름 다음에 인자를 받아서 원격 컴퓨터의 쉘에 전달한다. (단일 인자처럼 보이게 하기 위해서 인용부호 처리를 한다.) 이러한 방식의 인자는 적합한 명령어이기 때문에 원격 쉘이 `ls results`를 실행하고 결과를 다시 사용자의 로컬 쉘의 화면에 출력한다.

SSH 키 (SSH Key)

비밀번호를 반복해서 매번 타이핑하는 것은 성가시다. 특히, 원격에서 실행하려고 하는 명령어가 루프로 반복된다면 더욱 그렇다. 비밀번호를 반복해서 타이핑하는 필요를 제거하기 위해 SSH 키(SSH key)를 생성해서 원격 컴퓨터에 접속하는 것을 항상 신뢰하도록 할 수 있다.

SSH 키는 GitHub 같은 서비스와 공유되는 공개키(public key)와 사용자의 로컬에만 저장되는 개인키(private key)로 짹(pair)으로 되어 있다. 만약 두 키가 일치한다면, 접속이 허락된다.

SSH 키를 지탱하는 암호기술이 공개키로부터 개인키를 역공학(reverse engineering)할 수 없게 보증한다.

SSH 인증을 사용하는 첫 단계는 자신만의 개인키와 공개키 짹을 생성하는 것이다. 로컬 컴퓨터에 SSH 키 짹을 이미 가지고 있을 수도 있다. .ssh 디렉토리로 가서 디렉토리 내부 목록을 출력해서 SSH 키 짹이 존재하는지 확인할 수 있다.

```
$ cd ~/.ssh  
$ ls
```

만약 id_rsa.pub을 본다면, 이미 SSH 키 짹을 가지고 있어서 새로 생성할 필요는 없다.

만약 id_rsa.pub을 볼 수 없다면, 다음 명령어를 사용해서 새로운 SSH 키 짹을 생성할 수 있다. your@email.com 부분을 사용자 여러분의 전자우편 주소로 바꾸도록 확인하세요.

```
$ ssh-keygen -t rsa -C "your@email.com"
```

새로운 키를 어디에 저장할지 물었을 때, 디폴트 장소에 동의하면 “엔터(enter)”를 치세요.

```
Generating public/private rsa key pair.  
Enter file in which to save the key (/Users/username/.ssh/id_rsa):
```

그리고 나서, 선택적인 암호(passphrase)를 입력하도록 요청받을 것이다. SSH 키를 좀 더 안전하게 만드는데 사용될 수 있지만, 만약 여러분이 원하는 것이 매번 비밀 번호를 타이핑하는 것을 피하려고 한다면, “엔터(enter)”를 두 번 쳐서 건너뛸 수 있다.

```
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:
```

SSH 키 생성이 완료되면, 다음 확인 결과를 보게 된다. ~~~ Your identification has been saved in /Users/username/.ssh/id_rsa. Your public key has been saved in /Users/username/.ssh/id_rsa.pub. The key fingerprint is: 01:0f:f4:3b:ca:85:d6:17:a1:7d:f0:68:9d:f0:a2:db your@email.com The key's randomart image is: +-[RSA 2048]---+ | | | | . E + | | . o = . | | . S = o | | o.O . o | | o .+ . | | . o+.. | | .+=o | + -----+ ~~~

상기 임의 예술 이미지가 SSH 키를 일치시키는 대안이지만 여러분은 필요하치 않을 것이다.

아제 생성한 공개키를 접속하려는 서버에 보낼 필요가 있다. cat 명령어로 새로 생성한 공개키의 내용을 화면에 출력하자.

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

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA879BJGY1PTLIuc9/R5MYiN4yc/YiCLcdBpSdzgK9Dt0Bkfe3rSz5cPm
```

출력결과의 내용 콘텐츠를 복사한다. SSH 키를 사용하여 접속하려는 서버에 로그인한다.

```
$ ssh vlad@moon.euphoric.edu  
Password: *****
```

복사한 콘텐트를 ~/.ssh/authorized_keys 끝에 붙여놓는다.

```
moon> nano ~/.ssh/authorized_keys`.
```

콘텐트를 파일에 추가한 후에 원격 컴퓨터에서 로그 아웃하고 다시 로그인한다. 만약 SSH 키 초기 설정이 올바르게 되었다면, 비밀번호를 다시 타이핑할 필요가 없다.

```
moon> exit
```

```
$ ssh vlad@moon.euphoric.edu
```

주요점

- SSH는 사용자명/비밀번호 인증의 안전한 대안이다.
- SSH 키는 공개/개인 키 짹으로 생성된다. 공개 키는 다른 사람, 컴퓨터와 공유될 수 있다. 개인키는 여러분의 로컬 컴퓨터에만 체재한다.

예외 처리 (Exceptions)

가정 설정문(assertions)을 이용하여 코드의 오류를 잡는데 도움이 되지만 파일이 사라지거나 잘못된 파일 형식같은 다른 이유로 뭔가 잘못될 수 있다. 대부분의 최신 프로그래밍 언어는 프로그래머가 예외 처리(exceptions)를 사용해서 만약 모든 것이 정상이면 프로그램이 무엇을 수행하고, 만약 무엇인가 잘못되면 프로그램이 수행하는 것을 구별한다. 예외처리를 하는 것은 양쪽 경우 모두 작성한 코드를 읽기 쉽고, 이해하기 쉽게 한다.

예를 들어, 두개의 별개 파일에서 매개변수(parameter)와 그리드(grid)를 읽어들이고, 만약 파일을 읽는데 둘중에 하나라도 뭔가 잘못되면 오류를 보고하는 코드 일부가 다음에 있다.

```
try:  
    params = read_params(param_file)  
    grid = read_grid(grid_file)  
except:  
    log.error('Failed to read input file(s)')  
    sys.exit(ERROR)
```

`try` 와 `except` 키워드를 사용하여 정상인 경우와 오류 처리 코드를 합쳤다. `if` 와 `else` 와 마찬가지로 함께 동작한다. `try` 다음의 문장은 만약 모든 것이 잘 동작한다면 무엇을 해야하는지를 나타내는 반면에 `except` 다음의 문장은 뭔가 잘못된다면 프로그램이 무엇을 해야하는지 나타낸다.

사실 알지 못하게 앞에서 예외처리를 보았습니다. 왜냐하면, 예외사항이 발생했을 때, 파이썬은 예외사항을 출력하고 프로그램을 정지한다. 예를 들어, 존재하지 않는 파일을 열게 될 때, `IOError`라는 일종의 예외를 발생시킨다. 반면에 존재하지 않는 리스트 항목에 접근할 때는 `IndexError`를 발생시킨다.

```

open('nonexistent-file.txt', 'r')

-----
IOError                                         Traceback (most recent call last)

<ipython-input-13-58cbde3dd63c> in <module>()
----> 1 open('nonexistent-file.txt', 'r')

IOError: [Errno 2] No such file or directory: 'nonexistent-file.txt'

values = [0, 1, 2]
print values[999]

-----
IndexError                                         Traceback (most recent call last)

<ipython-input-14-7fed13afc650> in <module>()
1 values = [0, 1, 2]
----> 2 print values[999]

IndexError: list index out of range

```

만약 단순히 프로그램이 갑자기 중단되는 것 말고 뭔가 다른 것을 원한다면, `try` 와 `except`를 사용하여 직접 오류를 다룰 수 있다.

```

try:
    reader = open('nonexistent-file.txt', 'r')
except IOError:
    print 'Whoops!'

```

Whoops!

파이썬이 상기 코드를 실행할 때, `try` 내부의 문장을 실행한다. 만약 정상적으로 동작하면, `except` 블록 문장을 실행하지 않고 건너 뛴다. 하지만 만약 `try` 블록

내부에 예외사항이 발생하면, 파이썬이 `except`에 명시된 유형과 예외 유형을 비교한다. 만약 둘 사이가 매칭되면, `except` 블록 내부 코드를 실행한다.

존재하지 않는 파일 혹은 읽어들이는데 권한이 없는 프로그램 같은 경우 입출력과 관련된 문제가 발생할 때, 파이썬이 사용하는 예외 유형은 `IOError`이다. `if`문에 많은 문장을 쓸 수 있듯이, `try` 블록에 원하는 만큼 코드를 쓸 수 있다. 또한 몇 가지 다른 유형의 오류도 처리할 수 있다. 예를 들어, 그리드 각지점의 엔트로피를 계산하는 코드가 다음에 있다.

```
try:  
    params = read_params(param_file)  
    grid = read_grid(grid_file)  
    entropy = lee_entropy(params, grid)  
    write_entropy(entropy_file, entropy)  
except IOError:  
    report_error_and_exit('IO error')  
except ArithmeticError:  
    report_error_and_exit('Arithmetic error')
```

`try` 내부 함수 4개를 평소와 마찬가지로 파이썬이 실행한다. 만약 오류가 함수 중에서 발생한다면, 파이썬은 즉시 다음으로 내려가서 `except`에서 해당하는 예외 유형을 검색한다. 만약 예외 유형이 `IOError`이면, 파이썬은 첫번째 오류 처리(error handler)하는 곳으로 간다. 만약 예외 유형이 `ArithmeticError`이면, 대신에 파이썬은 두번째 오류 처리(error handler)하는 곳으로 간다. 파이썬이 일련의 `if/elif/else` 문중에서 하나의 분기만 수행하는 것과 마찬가지로, 여러 예외 유형 중에서 하나만 처리한다.

이와 같이 코드를 배치하여 가독성은 좋게 만들었지만, 무엇인가 중요한 것을 잊어버렸다. `IOError` 분기점에서 출력되는 메시지가 무슨 파일이 문제를 발생시켰는지 알려주지 않는다. 오류에 대한 정보를 저장하기 위해서 파이썬이 생성하는 개체를 잡아서 꼭 붙들 수만 있다면 더욱 잘 할 수 있다.

```
try:  
    params = read_params(param_file)  
    grid = read_grid(grid_file)  
    entropy = lee_entropy(params, grid)
```

```

        write_entropy(entropy_file, entropy)
    except IOError as err:
        report_error_and_exit('Cannot read/write' + err.filename)
    except ArithmeticError as err:
        report_error_and_exit(err.message)

```

`try`에서 뭔가 잘못되면, 파일이 예외 객체를 생성하고, 정보를 채워서 변수 `err`에 할당한다. (변수이름에 관해서 특별한 것은 없다. 원하는 임의의 이름을 사용한다.) 정확하게 무슨 정보가 기록되는지는 어떤 종류의 오류가 발생하느냐에 달려있다. 파일 문서에는 자세하게 각 오류 유형별 특징을 기술하고 있지만, 여기서는 예외 객체만 출력한다. 입출력(I/O) 오류의 경우 물체를 일으킨 파일 이름을 출력한다. 연산 오류의 경우에는 예외 객체에 내장된 메세지를 출력하는데 이것은 어쨌든 파일이 수행했을 것과 동일하다.

예외처리가 어떻게 동작하는지 많은 것을 학습했다. 어떻게 사용할 수 있을까요? 몇몇 프로그래머는 `try` 와 `except`를 사용해서 프로그램에 기본 동작으로 사용한다. 예를 들어, 만약 사용자가 요청한 그리드 파일을 코드가 읽을 수 없다면, 대신에 기본 그리드를 생성한다.

```

try:
    grid = read_grid(grid_file)
except IOError:
    grid = default_grid()

```

다른 프로그래머는 명시적으로 그리드 파일을 테스트하고 제어 흐름에 `if` 와 `else`를 사용한다.

```

if file_exists(grid_file):
    grid = read_grid(grid_file)
else:
    grid = default_grid()

```

거의 취향의 문제이지만, 두번째 스타일을 추천한다. 규칙으로서 예외처리는 예외적인 경우를 다루는데만 사용되어야 한다. 만약 프로그램이 기본 그리드로 어떻게 돌아가는지 알고 있다면, 예상하지 못한 이벤트는 아니다. 설사 동일한 것이라도,

`try`와 `except` 대신에 `if` 와 `else`를 사용하는 것은 코드를 읽는 사람에게 다른 신호를 전달한다.

종종 초보자가 예외처리 스타일에 관한 질문을 한다. 이런 질문을 다루기 전에 상기 예제에 여러분들이 간과한 것이 있다. 예외사항은 실질적으로 장기적 관점으로 처리되어야 한다. 즉시 처리하지 말아야 한다. 코드를 다른 관점으로 살펴보자.

```
try:  
    params = read_params(param_file)  
    grid = read_grid(grid_file)  
    entropy = lee_entropy(params, grid)  
    write_entropy(entropy_file, entropy)  
except IOError as err:  
    report_error_and_exit('Cannot read/write' + err.filename)  
except ArithmeticError as err:  
    report_error_and_exit(err.message)
```

`try` 블록의 4 줄은 모두 함수 호출이다. 4개 함수 모두 예외사항을 잡아서 처리할 수도 있다. 하지만, 만약 내부적으로 처리되지 않은 예외사항이 하나라도 발생한다면, 파이썬은 `except`와 매칭하려고 호출 코드를 검색한다. 만약 거기서 예외처리 코드를 찾지 못한다면, 함수 호출자를 들여다본다. 만약 예외처리기를 찾지 못하고 메인 프로그램으로 다시 돌아온다면, 파이썬 기본 동작은 줄곧 보아왔던 오류 메시지를 출력한다.

이러한 규칙이 [throw low, catch high](#) 규칙의 기원이다. 작성한 프로그램에는 오류가 발생할 수 있는 장소가 많다. 하지만 오류를 눈에 띄게 처리할 수 있는 것은 몇개에 불과하다. 예를 들어, 선형대수 라이브러리는 파이썬 인터프리터에서 직접 호출이 되었는지 혹은 좀더 큰 프로그램의 컴포넌트로 사용되는지 확인할 길이 없다. 후자의 경우 선형대수 라이브러리가 호출하는 프로그램이 명령-라인 인터페이스에서인지 아니면 GUI 인터페이스에서 실행되는지도 알수가 없다. 그래서, 라이브러리는 오류를 그 자체로 다루거나 보고하지 말아야 한다. 왜냐하면 예외사항을 처리할 올바른 방법이 무엇인지 알지 못하기 때문이다. 대신에 예외를 발생(`raise`)시키고, 호출하는 쪽에서 어떻게 처리하는 것이 가장 좋은 것인지 해결하게 한다.

마지막으로, 만약 할 수만 있다면, 예외사항을 발생시킨다. 사실 이렇게 해야 한다.

왜냐하면, 파이썬에서 무언가 잘못되었을 때 신호를 보내는 표준 방법이기 때문이다. 예를 들어, 다음에 그리드를 읽어 들이고, 일관성을 점검하는 함수가 있다.

```
def read_grid(grid_file):
    data = read_raw_data(grid_file)
    if not grid_consistent(data):
        raise Exception('Inconsistent grid: ' + grid_file)
    result = normalize_grid(data)
    return result
```

`raise` 문은 의미있는 오류 메시지로 새로운 예외를 생성한다. `read_grid` 자체는 `try/except` 블록을 포함하고 있지 않기 때문에, 예외사항은 항상 발생하고 함수 밖에서 `read_grid`을 호출자에 의해서 예외를 잡아서 처리해야 한다. 만약 할수만 있다면, 새로운 형식의 예외를 정의할 수 있다. 그리고 그렇게 해야 한다. 그래서 코드의 오류가 다른 사람 코드의 오류와 구별될 수 있다. 하지만, 이렇게 하는 것은 클래스와 객체와 관련되는데 이번 학습 범위 밖이다.

숫자(Numbers)

어떻게 숫자가 저장되는지 살펴보면서 시작해봅시다. 만약 0과 1 두 개의 숫자(digit)만 있다면, 양의 정수를 저장하는 자연스러운 방식은 기수(base)가 2인 이진법을 사용하는 것이다. 그래서 이진수 1001_2 은 십진수로 $(1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 9_{10}$ 이 된다. 마찬가지 방식으로 기호 표기를 위해서 1 비트를 따로 떼어서 음수에 대해서 이 방식을 확장하는 것도 자연스럽니다. 예를 들어, 양수로 0을, 음수로 1을 사용한다면, $+9_{10}$ 은 01001_2 이 될 것이고, -9_{10} 은 11001_{10} 이 될 것이다.

이 표기법에는 두가지 문제가 있다. 첫번째는 이 기법은 0에 대해서 2가지 표현법이 있다. (00000_2 and 10000_2) 반듯이 치명적이지는 않지만, 다음과 같이 코드를 작성할 때, 이 기법이 “자연”스럽다고 주장은 사라져버린다.

```
if (length != +0) and (length != -0)
```

또 다른 문제는 부호 크기 표현(sign and magnitude representation)으로 덧셈과 다른 연산에 필요한 회로는 2의 보수(two's complement)로 불리는 것에 필요한

하드웨어보다도 더 복잡하다. 양의 정수를 미러링하는 대신에 2의 보수는 자동차의 속도계처럼 0 이하로 내려갈 때 다시 이월한다. 만약 숫자마다 4 비트를 사용한다면, 0_{10} 은 0000_2 , -1_{10} 은 1111_2 이 된다. -2_{10} 은 1110_2 , -3_{10} 은 1101_2 , 등등이 되고, 표현할 수 있는 가장 큰 음수는 1000_2 , 즉 -8 이 된다. 그리고 나서 다시 표현이 다시 돌아가서 0111_2 은 7_{10} 이 된다.

이 기법이 직관적이지는 않지만 부호 크기 표현의 “0이 두개 되는” 문제를 해결하고, 하드웨어가 더 싸고 빠르게 할 수 있다. 보너스로, 첫번째 비트를 살펴보는 것만으로 숫자가 양수인지 음수인지 분간할 수 있다. 음수는 1, 양수는 1이 숫자의 첫번째 비트가 된다. 다만 이상한 것은 비대칭성이다. 왜냐하면 0 이 양수로 분류가 되어서 숫자가 -8 에서 7 혹은 -16 에서 15, 등등이 된다. 결과적으로 x 가 유효한 숫자일지라도, $-x$ 는 유효하지 않을 수 있다.

실수(부동 소수점수(floating point numbers)로 불리는데 소수부가 이동할 수 있기 때문이다.)에 대한 좋은 표현방법을 찾는 것은 훨씬 더 어려운 문제다. 문제의 근본적인 원인은 유한 집합의 비트 패턴으로 실수의 무한수를 표현할 수 없기 때문이다. 그리고 정수와는 달리 어떤 값을 표현하더라도 표현할 수 없는 각각 사이에 존재하는 무한수가 있다.

부동 소수점수는 대체로 기호(sign), 절대값(magnitude), 그리고 지수(exponent)로 구성된다. 32비트체계에서 IEEE 754 표준은 1 비트는 기호, 23 비트는 절대값(혹은 소수부/가수), 그리고 8비트는 지수로 정했다. 부동소수점 문제를 시연하기 위해서 심하게 단순한 표현을 사용한다. 소수부분 없는 양수 값만 관심을 가질 것이고, 가수에 3비트, 지수에 2비트만 사용한다.

| Exponent | | | | |
|----------|-----|----|----|----|
| | 00 | 01 | 10 | 11 |
| 000 | 0 | 0 | 0 | 0 |
| 001 | 1 | 2 | 4 | 8 |
| 010 | 2 | 4 | 8 | 16 |
| Mantissa | 011 | 6 | 12 | 24 |
| | 100 | 8 | 16 | 32 |
| | 101 | 10 | 20 | 40 |
| | 110 | 12 | 24 | 48 |
| | 111 | 14 | 28 | 56 |

상기 테이블은 표현할 수 있는 모든 값을 보여준다. 각각은 가수 곱하기 2의 지수2승이다. 예를 들어, 소수점 48은 이진수 111에 이진법 11승으로, 6 곱하기 2의 이진법 3승, 즉 6 곱하기 8이 된다. (IEEE 754 표준같은 실수 부동소수점 표현은 상기 표에 있는 중복이 없지만 여기서 펼치는 주장에는 영향이 없음을 주목하기 바란다.)

주목할 첫번째 것은 저장할 수 없는 많은 값이 있다. 예를 들어 8 그리고 10은 표현할 수 있지만, 9는 표현할 수 없다. 전자 계산기가 1/3같은 소수점 처리에서 갖는 정확하게 동일한 문제다. 소수부에서 0.3333 혹은 0.3334으로 반올림해야 한다.

하지만, 이 기법이 9에 대한 표현법이 없다면, 8+1은 8 혹은 10으로 저장되어야 한다. 이것이 흥미로운 문제를 불러 일으킨다. 만약 8+1이 8이라면, 8+1+1은 무엇이 될까? 왼편에서부터 더한다면, 8+1은 8, 더하기 또 다른 1을 더하면 8이 된다. 하지만, 만약 오른편에서부터 더한다면 1+1은 2가 되고 2+8은 10이 된다. 연산 순서를 바꾸는 것이 옳고 그름의 차이를 만들 수 있다. 무작위성(randomness)이 있으면 안된다. 즉, 특정 연산 순서는 항상 동일한 결과를 만들어 내야 한다. 하지만, 연산 단계가 증가하면서 가장 최선의 순서가 무엇인지 파악하는 것도 어려워진다.

수치해석 분석가는 이런 종류의 문제 해결에 시간을 보낸다. 상기 경우에 만약 더하고자 하는 값을 정렬하고 나서 가장 작은 값부터 차례로 가장 큰 값을 더한다면, 가능한 최선의 정답에 도달할 확률을 높여준다. 역행렬을 구하는 같은 다른 상황에서는 규칙이 훨씬 더 복잡해진다.

균등하지 않은 숫자 라인(number line)에 대한 또 다른 관찰점이 여기에 있다. 표현할 수 있는 값과 값 사이 간격이 균등하지는 않지만, 각 값 사이의 상대적인 간격은 동일한 경우다. 즉, 첫번째 숫자 그룹은 1 씩 떨어져 있고, 구분 간격이 2, 4, 8 그래서 값 사이의 간격 비율은 대략 상수가 된다. 이와 같은 상황이 발생하는 이유는 동일한 고정 집합의 가수를 점점 더 큰 지수로 곱하기 때문이다. 이것이 몇 가지 유용한 정의로 유도한다.

근사할 때 절대 오차(absolute error)는 단순하게 실제 값과 근사 값의 절대 값의 차이다. 반대로, 상대 오차(relative error)는 절대 오차와 근사하려고하는 값의 비율이다. 예를 들어, 만약 근사적으로 1만큼 차이가 있어서 8+1과 56+1이 된다면, 절대 오차는 두 경우 모두 동일하지만, 첫번째 경우는 상대 오류는 $1/9 = 11\%$ 가 되는 반면에 두번째 경우의 상대 오차는 단지 $1/57 = 1.7\%$ 만 된다. 부동 소수점수에 대해서 생각할 때, 절대 오차보다는 상대 오차가 거의 항상 더 유용하다. 결국, 문제의 값이 10억일 때 100만큼 오차가 있다는 것은 거의 의미가 없다.

왜 이것이 중요한지 살펴보기 위해서, 다음의 작은 프로그램을 살펴보자.

```
nines = []
sums = []
current = 0.0
for i in range(1, 10):
    num = 9.0 / (10.0 ** i)
    nines.append(num)
    current += num
    sums.append(current)

for i in range(len(nines)):
    print '%.18f %.18f' % (nines[i], sums[i])
```

루프를 정수 1에서 9까지 반복한다. 이 값을 이용하여 0.9, 0.09, 0.009 등등의 숫자를 생성하고 `vals` 리스트에 담는다. 그리고 나서 이를 숫자의 합을 계산한다. 명확하게 0.9, 0.99, 0.999 등등이 되어야 한다. 하지만, 정말 그럴까?

| | | |
|---|-----------------------|----------------------|
| 1 | 0.900000000000000022 | 0.900000000000000022 |
| 2 | 0.08999999999999997 | 0.9899999999999991 |
| 3 | 0.00899999999999999 | 0.9989999999999999 |
| 4 | 0.0009000000000000000 | 0.999900000000000011 |
| 5 | 0.0000900000000000000 | 0.999990000000000046 |
| 6 | 0.0000090000000000000 | 0.999999000000000082 |
| 7 | 0.0000009000000000000 | 0.999999900000000053 |
| 8 | 0.0000000900000000000 | 0.999999990000000061 |
| 9 | 0.0000000090000000000 | 0.999999999000000028 |

해답이 여기 있다. 첫번째 칼럼은 루프 인덱스, 두번째 칼럼은 0.9, 0.99 등등 값을 계산할 때 실제 얻은 값, 세번째 칼럼은 누적합이다.

주목할 첫번째 것은 누적합에 기여하는 첫번째 값이 이미 약간 오차가 있다. 심지어 가수를 32비트포 표현해도 10진수로 $1/3$ 을 정확하게 표현할 수 있는 이상으로 2 진수로 0.9를 정확하게는 표현할 수 없다. 가수 크기를 두배로 하는 것이 오차를 줄일 수는 있지만 오차를 완전히 제거할 수는 없다.

주목할 두번째 것은 0.0009에 근사가 그 다음의 모든 근사값도 그렇지만, 실질적으로 정확한 것처럼 보인다. 하지만 이것은 잘못 오도될 수 있다. 결국 소수점 아래 18째 자리까지만 출력했다. 누적합의 마지막 숫자 오차에 대해서 증가하는지 감소하는지에 대한 어떤 규칙적인 패턴이 있어 보이지 않는다.

이런 현상이 과학 프로그램을 테스트하는 것을 어렵게 만드는 이유중의 하나다. 만약 함수가 부동 소수점수를 사용한다면, 작성한 함수가 제대로 동작하는지 확인하기 위해서 결과를 무엇과 비교하여야 할까? 만약 `vals`에 있는 처음 몇개의 숫자 합계와 당연히 되어야 하는 이론적 합계와 비교한다면, 설사 리스트를 정확한 값으로 초기화한 후에 정확하게 합을 계산할지라도 정답은 거짓(`False`)이 된다. 이것은 진정으로 어려운 문제이고, 누구도 일반적인 좋은 답을 제시하지 못했다. 문제의 본질은 근사를 사용한다는 것이고 각각의 근사는 그 자체의 장점에서 판단되어야 한다.

하지만, 여러분이 포기하지 말고 할 수 있는 것이 있다. 첫번째 규칙은 계산할 수만 있다면, 해석적인 해답과 컴퓨터로 얻은 값을 비교하라. 예를 들어, 만약 액체 헬륨 방을 행동을 살펴본다면, 무중력 상태에서 정지된 구형 방울에 대한 프로그램 출력결과를 점검하는 데서 시작한다. 이 경우에 정답을 계산할 수 있어야 하고, 만약 작성한 프로그램의 결과와 맞지 않는다면 아마도 다른 상황에도 작동하지 않을 것이다.

두번째 규칙은 좀더 간단한 버전의 코드와 더 훨씬 복잡한 버전의 코드와 비교하는 것이다. 연전달을 계산하는 간단한 알고리즘을 더 복잡하지만 희망적으로 좀더 빠른 알고리즘으로 바꾸려고 한다면, 이전 코드를 버리지 마세요. 대신에 새로운 코드의 정합성을 확인하는 용도로 사용하세요. 그리고, 만약 여러분의 것과 동일한 결과를 계산하는 프로그램을 작성한 누군가와 마주친다면, 데이터를 교환하세요. 모두에게 도움이 된다.

세번째 규칙은 부동 소수점수에 `==` (혹은 `!=`)을 사용하지 마세요. 다른 방식으로 계산된 두 숫자는 아마도 정확하게 같은 비트를 갖지 않을 것이다. 대신에 두 값이 특정 허용 오차 안에 있는지를 점검하고, 만약 허용 오차 안에 있다면, 같다고 처리하세요. 이와 같은 것이 허용 오차를 명시적으로 설정하게 하고, 그 자체로도 유용한다. (실험 결과에 오차 막대를 설정하는 유용한 것과 마찬가지다.)

내가 왜 가르칠까요? (Why I teach)

제 딸 매들린(Madeline)입니다:



몇몇 독립적 연구는 지구상에서 매들린이 가장 귀여운 아이라는 것을 확인했습니다. (+- 5% 오차로, 20번중 19번) 매들리은 저에게 있어서 가장 소중하지만, 매들린이 성인이 될 때쯤이면 전세계는 우리 세대의 근시안적인 결과를 다룰 것입니다. 기후 변화, 자원 부족, 약에 내성을 가진 질병— 목록은 그 외에도 많지만, 저희를 구원할 단 한가지는 좀더 많은 과학과 용기입니다.

이것이 왜 내가 가르치는 이유입니다. 제가 가리치는 이유는 매들린이 살기에 좀 더 적합한 세상을 만드는데 여러분의 도움이 필요하기 때문입니다. 제가 가리치는 이유는 여러분이 더 많은 것을 발견하고 더욱 빨리 발명하는데 여러분이 필요합니다. 그래서 여러분으로부터 물려받을 세상이 지금 저희 세대에서 물려받은 것보다 좀더 나아졌으면 합니다. 과학은 저희 시대의 가장 커다란 모험일뿐만 아니라, 문자 그대로 삶과 죽음의 문제입니다. 여러분에게 가르치는 것이 여러분에게 도움이 되어서 좀더 좋은 것을 하는데 도움이 되었으면 합니다.

시간을 내어주셔서 감사합니다. 항상 아이의 눈으로 세상을 보았으면 합니다.

강사 안내서 (instructors guide)

한 생물종으로서, 두뇌가 어떻게 학습하고, 다양한 교육 방법이 얼마나 효과적이고, 사회적인 필요와 기대가 여러분들이 학습하는 것을 어떻게 바꾸는지에 대해서도

많이 알려져 있다. 하지만, 개인적인 자격으로 대학교에서 강의하는 대부분의 사람들은 이러한 지식이 존재하는지도 모르고, 강의에 접목하지도 않는다. 이것은 마치 단지 의사만 흡연과 암이 연관되었다는 것을 아는 것과 흡사하다.

소프트웨어 카펜트리에서 발견한 근거기반학습(evidence-based learning)에 대한 추천 안내서는 *How Learning Works: Seven Research-Based Principles for Smart Teaching* 다. 이책에서 조언하는 것은 이론, 연구, 경험에 균형을 맞추고 있다. 설사 일부 권장 사항은 진부하지만, 책 자체의 전체적인 설명은 중요하다.

교육과 강사 훈련에 이러한 아이디어를 통합하려고 노력하고 있다. 만약 참여를 원한다면, 연락을 주시기 바랍니다.

일반적인 조언

강의에 관한 일반적인 사항을 적어두는 곳입니다. 소프트웨어 설치 및 설정 문제에 대한 최신정보는 [위키 페이지](#)를 참조바랍니다.

강의 노트

- 예를 들어, 2주에 걸쳐서 오후에 4번 진행되는 2일 이상 진행되는 워크샵을 진행할 때, 이전에 학습한 요약내용을 포함하여 가능하면 연관된 온라인 링크도 함께 학습자에게 워크샵 마지막에 메일을 보내는 것을 권장한다. 다음 워크샵 세션 시작전에 불참한 학습자가 수업을 따라잡게 할 수 있으며 전체 워크샵 맥락에서도 워크샵 수업내용을 제시할 수는 장을 마련해 준다.
- 학습자에게 영어 원문으로 된 현재 강의 자료(버전 5)가 있는 장소 <http://software-carpentry.org/v5/>, 그리고 이전 강의 자료(버전 4)가 있는 장소 <http://software-carpentry.org/v4/> 안내한다. 전자 강의자료는 현재 교육자료로 사용되는 것이고, 후자 강의자료는 동영상, 슬라이드, 그리고 슬라이드 노트를 포함하고 있다. 소프트웨어 카펜트리 [Software Carpentry's FAQ](#) 정 보도 공유한다.
- 교육자료의 원래 출처(예를 들어, 소프트웨어 카펜트리 웹사이트)를 명확히 밝힌다면 교육 자료는 [크리에이티브 커먼즈 - 저작자표시 \(CC-BY\)](#) 라이선스에 따라 자유로이 변경하고 재사용할 수 있다. 하지만 소프트웨어 카펜트리

이름과 로고는 등록상표로 허가없이 사용할 수는 없다. 일반적으로 다음 조건을 만족하는 경우에는 이용허가를 한다. (a) 소프트웨어 카펜트리 핵심 교육 주제를 다룬다. (b) 강사 명단에 최소 1명 이상 소프트웨어 카펜트리 강사가 원칙이지만, 구체적 사항에 관해 협의도 가능하다.

- 워크샵 시작 첫 30-60분은 설치 및 환경설정에 할당하는데 이유는 항상 발생하는 문제이기 때문이다. 사전 워크샵 “업무지원센터(help desk)”를 운영하는 것도 정말 영향을 주지 않습니다. 소프트웨어 설치에 가장 많은 애로를 가질 것 같은 신청가는 아마도 워크샵에 나타나지 않을 것이다. (소프트웨어를 설치하지 않고, 최소 인스톨러도 다운로드하지 않고 나타나는 워크샵 신청자를 되돌려 보내는 공상을 하지만, 실무에서 그렇게 하기는 어렵다.)
- 좋은 소프트웨어 개발 기술이 생산적이고, 재사용가능하고, 재현가능한 연구에 도움이 된다는 것을 강조한다.
- 학습자 질문이나 도움이 필요할 때 빨간색 포스트잇 노트를 노트북이나 모니터에 붙이게 한다. 매번 실습을 시작하기 전에는 포스트잇 노트를 떼어서 내려 놓게 한다. 실습이 완료되면 녹색 포스트잇을 붙이게 하고 도움이 필요하면 빨간색 포스트잇을 붙이게 한다.
- 점심시간과 워크샵 하루의 끝에 학습자에게 한가지 좋은 점(즉, 학습한 것 혹은 즐거웠던 것)을 녹색 포스트잇에 적어서 제출하도록 요청한다. 그리고 나빴던 점(즉, 동작하지 않았던 것, 이해하지 못한 것, 혹은 이미 알고 있는 것)도 빨간색 포스트잇에 적어 제출하도록 요청한다.
- 참가자가 작성한 것을 살펴보는데 몇분정도 걸리지만, 실제 워크샵이 전반적으로 어떻게 돌아가고 있는지 파악하는 빠른 방법이다.
- 워크샵 마지막에는 전체 워크샵을 통해서 한가지 좋은 점과 한가지 나쁜 점을 제시하도록 요청한다. 순서대로 앞으로 나와서 칠판(화이트 보드)에 적는다. 하지만 반복은 허용되지 않습니다. (즉, 항목은 모두 새로운 것이어야 한다.) 대체로 처음 몇개의 부정적인 항목은 무해하다. 즉, 부정적인 항목이 모두 소진된 후에 실질적인 피드백을 받기 시작한다.
- 빨간/녹색 포스트잇 변형으로 이름을 적을 수 있는 태그 라벨과 함께 작은 빨간색/녹색 작은 종이텐트를 만든다. 학습자가 이름 태그 라벨에 자신의 이름을 적고, 강의 속도가 너무 빠르거나 너무 늦으면 빨간색/녹색 종이 텐트를 통해서 피드백을 준다.

- 강의자 일화, 경험, 증거자료를 강의자료로 뒷받침하라. 학습자에게 신뢰를 주고, 학습자로 하여금 자신이 가지고 있는 문제에 어떻게 적용할 수 있는지 이해를 돋는다. 마지막으로 수업이 너무 무미건조하게 진행되는 것을 방지하는 효과도 있다.
- Etherpad나 빔프로젝트 화면 옆에 화이트보드에 지금까지 학습한 명령어 목록을 적어둔다. 학습자(특히 강의 자료를 이미 잘 알고 있어서 자칫 지루해할 수 있는 참가자 대상)에게 Etherpad를 통해서 노트하도록 독려한다. 이렇게 하는 것은 한명의 학습자에 드는 수고를 줄여 주고, 강의자가 전달하는 것을 학습자가 어떻게 생각하고 있는지 볼 수 있는 기회도 마련해 준다. 그리고, 워크샵을 마친 다음에 실제로 강의한 것에 대한 기록도 된다.
- 공동 강사가 강의하지 않을 때, 공동 강사는 Etherpad에 올라온 질문에 답을 하고 강사가 넘어간 핵심사항(핵심사항 뿐 아니라, 강사가 언급하지 않은 명령어나 연관된 요점)을 Etherpad를 통해서 전달한다. 공동 강사가 요점을 중간에 말참견하는 것보다 “강의중인(live)” 강사를 덜 방해하게 되지만, 학습 참여자에게는 두명의 강사로부터 공유 전문지식을 얻게 도와준다.
- 장문의 노트는 강사를 위한 스크립트나 학습자가 자기주도 학습을 위해서 준비된 것이다. 노트를 학습자에게 보여주지 마세요. 대신에 강의를 시작할 때 빈 노트에서 시작하고 수업을 진행하면서 코드를 추가하라. 이렇게 진행하는 방식이 마치 학습자와 앞서나가는 경주하는 것을 막고, 질문에 대한 응답으로 즉흥적으로 수업을 진행하게 돋는다.
- 학습을 완료한 다음에 온라인 학습 자료(워크샵 홈페이지 혹은 <http://software-carpentry.org/v5/>)를 알려줘라. 만약 사전에 학습자료 위치를 알려준다면, 강의하는 동안에 참여자들이 온라인 학습자료를 읽어 수업참여도가 떨어질 수 있다.
- 정말 강사분이 열정적이라면, 강의하고 있는 디렉토리에 있는 SVG 다이어그램을 가까이 배치하여 마크다운 문법)가 사용된다:

```
<<<<< HEAD
lines from local file
=====
lines from remote file
>>>>> 1234567890abcdef1234567890abcdef12345678
```

Mercurial Reference

Set up configuration (only needs to be done once per machine):

On Windows, create a file called %USERPROFILE%\Mercurial.ini (that's spelled \$USERPROFILE/Mercurial.ini if you are in gitbash) containing the following lines:

```
[ui]
username = Your Name <you@some.domain>
editor = your_editor

[extensions]
color =

[color]
mode = win32
```

On OS/X or Linux, create a file called `~/.hgrc` with the contents:

```
[ui]
username = Your Name <you@some.domain>
editor = your_editor

[extensions]
color =
```

Display help text for a Mercurial command:

```
hg help command_name
```

Initialize the current working directory as a repository:

```
hg init
```

Display the status of the repository:

```
hg status
```

Mark specific files to be tracked in the repository:

```
hg add path/to/file_1 path/to/file_2
```

Mark specific directories to be tracked in the repository:

```
hg add path/to/directory_1 path/to/directory_2
```

Mark all untracked files and directories to be tracked in the repository:

```
hg add
```

Stop tracking a specific file:

```
hg forget path/to/file
```

Commit changes in all modified files to the repository's history: (Without `-m` and a message, this command runs a text editor.)

```
hg commit -m "Some message"
```

Commit changes in specific modified files to the repository's history: (Without `-m` and a message, this command runs a text editor.)

```
hg commit path/to/file_1 path/to/file_2 -m "Some message"
```

View the history of the repository:

```
hg log
```

Display differences between the current state of the repository and the last saved state:

```
hg diff
```

Display differences between the current state of a specific file and the last saved state:

```
hg diff path/to/file
```

Display differences between the current state of a specific file and its state in a specific earlier revision:

```
hg diff --rev 27 path/to/file
```

Display changes made to a specific file in a specific earlier revision:

```
hg diff --change 24 path/to/file
```

Erase changes to a specific file since the last commit (modified file is saved with a `.orig` suffix before reverting):

```
hg revert path/to/file
```

Erase changes to a specific file since the last commit without saving `.orig` backup:

```
hg revert --no-backup path/to/file
```

Erase all changes since the last commit:

```
hg revert --all
```

Restore file to its state in a previous revision:

```
hg revert --rev 16 path/to/file
```

To add the URL of a clone of a repository on a remote server, edit (or create) the `.hg/hgrc` in the repository to include:

```
[paths]
default = URL
```

To add the path of a local clone of a repository, edit (or create) the `.hg/hgrc` in the repository to include:

```
[paths]
nickname = path/to/repository
```

Display a repository's clone URLs and/or paths:

```
hg paths
```

Push changes from a local repository to the default clone of the repository:

```
hg push
```

Pull changes from the default clone of the repository:

```
hg pull
```

Pull changes from a local clone of the repository:

```
hg pull path/to/repository
```

Pull changes from a local clone of the repository that you have added a nickname path for:

```
hg pull nickname
```

Update a repository with the changes pulled from a clone of the repository:

```
hg update
```

Merge changes pulled from another clone of the repository using the GUI `kdiff3` to view and resolve conflicts:

```
hg merge --tool=kdiff3
```

Clone a repository from a remote server:

```
hg clone URL
```

Clone a repository from a local path:

```
hg clone path/to/repository
```

Note that when you clone a repository, either by a URL or by a path, the URL/path of the source repository is automatically added to the [paths] section of the new repository's .hg/hgrc file as the default path so that hg push and hg pull just work.

파이썬 참고 정보 (Python Reference)

기본 연산

- `variable = value`을 사용해서 변수에 값을 대입한다.
- `print first, second, third`을 사용해서 값을 화면에 출력한다.
- 파이썬은 1이 아니 0부터 계수한다.
- #으로 주석을 시작한다.
- 블록에 문장은 들여쓴다 (일반적으로 공백 4개).
- `help(thing)`은 도움말을 화면에 출력한다.
- `len(thing)`은 컬렉션(collection) 길이를 산출한다.
- `[value1, value2, value3, ...]`을 통해 리스트를 생성한다.
- `list_name[i]`은 리스트에서 i번째 값을 선택한다.

제어 흐름 (Control Flow)

- `for` 루프를 생성해서 한번에 하나씩 컬렉션 요소를 처리한다:

```
for variable in collection:  
    ...body...
```

- `if`, `elif`, `else`을 사용해서 조건문을 생성한다:

```
if condition_1:
    ...body...
elif condition_2:
    ...body...
else:
    ...body...
```

- `==`을 사용해서 같음(equality)을 테스트한다.
- `X and Y`은 X와 Y가 모두 참(true)일 때만 참이다.
- `X or Y`은 X 혹은 Y가 모두가 참이거나, 둘중 하나가 참이면 참이다.
- `assert condition, message`을 사용해서, 프로그램이 실행될 때 어떤 것이 참인지 점검한다.

함수 (Functions)

- `def name(...params...)`을 통해 새 함수를 정의한다.
- `def name(param=default)`을 통해 매개변수에 기본설정값을 명시한다.
- `name(...values...)`을 사용해서 함수를 호출한다.

라이브러리(Libraries)

- `import libraryname`을 사용해서 프로그램에 라이브러리를 가져온다.
- `sys` 라이브러리는 다음을 포함한다:
 - `sys.argv`: 프로그램이 실행하는데 필요한 명령-라인 인자.
 - `sys.stdin, sys.stdout`: 표준입력과 표준출력.
- `glob.glob(pattern)`은 패턴이 매칭되는 파일 리스트를 반환한다.

배열(rrays)

- import numpy을 통해 넘파이(NumPy) 라이브러리를 적재한다.
- array.shape을 통해 배열 형상을 알 수 있다.
- array[x, y]을 통해 배열에 특정 단일 요소를 선택한다.
- low:high을 통해 슬라이스를 구체화하는데 low에서 high-1까지 요소만 포함된다.
- array.mean(), array.max(), array.min()을 통해 기본 통계량을 계산한다.
- array.mean(axis=0)을 통해 특정 축(axis)에 대한 통계량을 계산한다.

SQL 참고 정보 (SQL Reference)

기본 쿼리 (Basic Queries) 테이블에서 하나 혹은 그 이상 칼럼을 선택한다:

```
SELECT column_name_1, column_name_2 FROM table_name;
```

테이블에서 모든 칼럼을 선택한다:

```
SELECT * FROM table_name;
```

쿼리에 유일한(unique) 결과만 얻는다:

```
SELECT DISTINCT column_name FROM table_name;
```

쿼리에 계산을 수행한다:

```
SELECT column_name_1, ROUND(column_name_2 / 1000.0) FROM table_name;
```

결과를 오름차순으로 정렬한다:

```
SELECT * FROM table_name ORDER BY column_name_1;
```

결과를 오름차순과 내림차순으로 정렬한다:

```
SELECT * FROM table_name ORDER BY column_name_1 ASC, column_name_2 DESC;
```

필터링(Filtering) 조건을 만족하는 데이터만 선택한다:

```
SELECT * FROM table_name WHERE column_name > 0;
```

조건 조합을 만족하는 데이터만 선택한다:

```
SELECT * FROM table_name WHERE (column_name_1 >= 1000) AND (column_name_2 = 'A' OR column_na
```

결측 데이터 (Missing Data) NULL을 사용해서 결측 데이터를 표현한다:

NULL은 0, 거짓(false), 혹은 다른 어떤 특정한 값은 아니다. NULL을 포함하는 연산은 NULL이 된다. 그래서 $1+NULL$, $2>NULL$, $3=NULL$ 은 모두 NULL이 된다.

값이 null인지 테스트한다:

```
SELECT * FROM table_name WHERE column_name IS NULL;
```

값이 null이 아닌지 테스트한다:

```
SELECT * FROM table_name WHERE column_name IS NOT NULL;
```

그룹으로 분류하기(Grouping) 그리고 **총합(Aggregation)** 총합(aggregation) 함수를 사용해서 값을 결합한다:

```
SELECT SUM(column_name_1) FROM table_name;
```

그룹으로 데이터를 결합하고, 그룹에 결합된 값을 계산한다:

```
SELECT column_name_1, SUM(column_name_2), COUNT(*) FROM table_name GROUP BY column_name_1;
```

합병(Joins) 두 테이블에서 데이터를 합병(join)한다:

```
SELECT * FROM table_name_1 JOIN table_name_2 ON table_name_1.column_name = table_name_2.column_name;
```

쿼리 작성하기 (Writing Queries) SQL 명령어는 다음 순서로 결합되어야 한다:

SELECT, FROM, JOIN, ON, WHERE, GROUP BY, ORDER BY.

테이블 생성하기 (Creating Tables) 칼럼 이름과 자료형을 명시해서 테이블을 생성한다. 기본키, 외래키, 제약조건을 포함한다.

```
CREATE TABLE survey(
    taken    INTEGER NOT NULL,
    person   TEXT,
    quant    REAL NOT NULL,
    PRIMARY KEY(taken, quant),
    FOREIGN KEY(person) REFERENCES person(ident)
);
```

프로그래밍 (Programming) 다음 순서로 범용 프로그래밍 언어에서 쿼리를 실행한다:

- 적절한 라이브러리를 적재 (loading the appropriate library)
- 연결을 생성 (creating a connection)
- 커서 생성 (creating a cursor)
- 반복적으로 적용:
 - 쿼리 실행 (execute a query)
 - 일부 혹은 전체 결과 가져오기 (fetch some or all results)
- 커서 폐기 (disposing of the cursor)
- 연결 종료 (closing the connection)

파이썬 예제:

```
import sqlite3
connection = sqlite3.connect("database_name")
cursor = connection.cursor()
cursor.execute("...query...")
for r in cursor.fetchall():
    ...process result r...
cursor.close()
connection.close()
```

프롬프트 인식하는 방법과 끝내는 방법

명령 라인(command line)에서 작업할 때, 프롬프트를 이식하지 못할 수도 있다. 이번 페이지에서 현재 위치를 파악하는 비밀정보(tip)와 만약 원하는 곳에 있지 않을 때 어떻게 탈출할 수 있는지 학습한다.

쉘 (Shell)

- 만약 쉘 프롬프트가 \$이면, 현재 위치는 you are at bash가 된다.

```
[frantere@pupunha shell]$
```

bash에서 탈출하기 위해서는 `exit`를 타이핑하고 ENTER 키를 친다.

- 만약 쉘 프롬프트가 > 이면, 문자열을 지정하는데 쉘 명령 일부로 ' 혹은 "을 타이핑했을 수 있다. 하지만, 문자열을 마무리하기 위해서 상응하는 ' 혹은 "을 타이핑하지 않았다.

```
[frantere@pupunha shell]$ echo 'long line of
>'
```

현재 명령을 중단하려면, CTRL-C를 누른다.

- 만약 쉘 윈도우 하단 왼쪽에 --More--가 보인다면, `more`를 사용해서 파일을 보고있다.

```
0xd/lib/
alsa-lib/
apr-exp
apr-util-1/
aprutil.exp
aspell@
aspell-0.60/
atkmm=1.6/
atkmm-pi2-core/
atkmm-kde.so*
audi/t/
avahi/
awk/
babl-0.1/
binfmt.d/
bitlbee/
bluemar/
bluetooth/
cairo/
cairomm-1.0/
ccache/
chromium/
chromium-browser/
chromium-dev/
cifs-utils/
cloog-isl/
Clucene/
cmake/
color/
colord-plugins/
colord-sensors/
coreutils/
crti.o
crti.o
crtn.o
--More--
```

more에서 나가려면, q를 누른다.

- 만약 쉘 윈도우 하단 유편에 filename, :, (END)가 보인다면, less를 사용해서 파일을 보고 있다.

```
0xd/lib/
alsa-lib/
apr-exp
apr-util-1/
aprutil.exp
aspell@
aspell-0.60/
atkmm=1.6/
atkmm-pi2-core/
atkmm-kde.so*
audi/t/
avahi/
awk/
babl-0.1/
binfmt.d/
bitlbee/
bluemar/
bluetooth/
cairo/
cairomm-1.0/
ccache/
chromium/
chromium-browser/
chromium-dev/
cifs-utils/
cloog-isl/
Clucene/
cmake/
color/
colord-plugins/
colord-sensors/
coreutils/
crti.o
crti.o
crtn.o
:
```

less에서 탈출하려면, q를 누른다.

- 만약 쉘 윈도우 하단 유편에 Manual page가 보인다면, man 페이지를 보고 있다.

```

MAN(1)                                Manual pager utils                               MAN(1)

NAME
    man - an interface to the on-line reference manuals

SYNOPSIS
    man [-C file] [-d] [-D] [-warnings[=warnings]] [-R encoding] [-L locale] [-m system[...]] [-M path] [-S list] [-e extension]
        [-i|-I] [-regex|-wildcard] [-names-only] [-a] [-u] [-no-subpages] [-P pager] [-r prompt] [-7] [-E encoding]
        [-no-hyphenation] [-no-justification] [-p string] [-t] [-Tdevice] [-Hbrowser] [-Xdpi] [-Z] [{section} page ...] ...
    man [-K [-W [-S list]] [-I|-J] [-regex] [section] term ...
    man -f [whatis options] page ...
    man -l [-C file] [-d] [-D] [-warnings[=warnings]] [-R encoding] [-L locale] [-P pager] [-r prompt] [-7] [-E encoding] [-p
        string] [-t] [-Tdevice] [-Hbrowser] [-Xdpi] [-Z] file ...
    man -w [-M [-C file] [-d] [-D] page ...
    man -c [-C file] [-d] [-D] page ...
    man [-?]

DESCRIPTION
    man is the system's manual pager. Each page argument given to man is normally the name of a program, utility or function.
    The manual page associated with each of these arguments is then found and displayed. A section, if provided, will direct man
    to look only in that section of the manual. The default action is to search in all of the available sections following man
    pre-defined order ("1 n 1 8 3 0 2 5 4 9 6 7" by default, unless overridden by the SECTION directive in /etc/man_db.conf),
    and to show only the first page found, even if page exists in several sections.

    The table below shows the section numbers of the manual followed by the types of pages they contain.

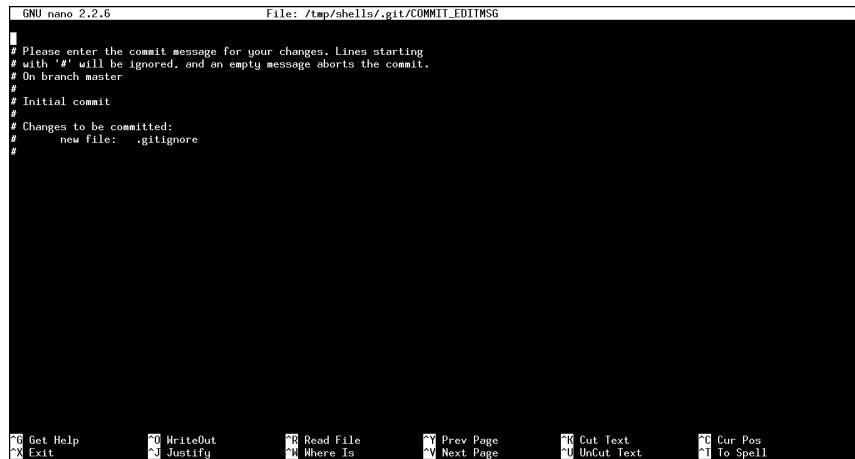
    1 Executable programs or shell commands
    2 Special commands provided by the kernel
    3 Library calls (functions in dynamic libraries)
    4 Special files (usually found in /dev)
    5 File formats and conventions eg /etc/passwd
    6 Games
    7 Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
    8 System administration commands (usually only for root)

```

Manual page man(1) line 1 (press h for help or q to quit)

man에서 탈출하려면, q를 누른다.

- 만약 쉘 원도우 상단에 “GNU nano”가 보인다면, nano 텍스트 편집기에 있다.



nano에서 탈출하려면, CTRL-X를 누른다. 만약 저장하지 못한 변경사항이 있다면, 저장하도록 질문을 받은데 - y를 눌러 저장하거나, n를 눌러 저장하지 않고 끝낸다.

- 쉘 원도우에 각 행 시작지점에 ~가 보인다면, vi 텍스트 편집기에 있다.

```
1 # Please enter the commit message for your changes. Lines starting
2 # with '#' will be ignored, and an empty message aborts the commit.
3 # On branch master
4 #
5 #
6 # Initial commit
7 #
8 # Changes to be committed:
9 #   new file:  .gitignore
10 #
```

vi에서 탈출하려면, 저장하지 않고 나가기는데 :q!을 타이핑한다. 만약 화면에 :q!이 텍스트로 나타난다면, ESC를 누르고 나서, :q!을 타이핑한다.

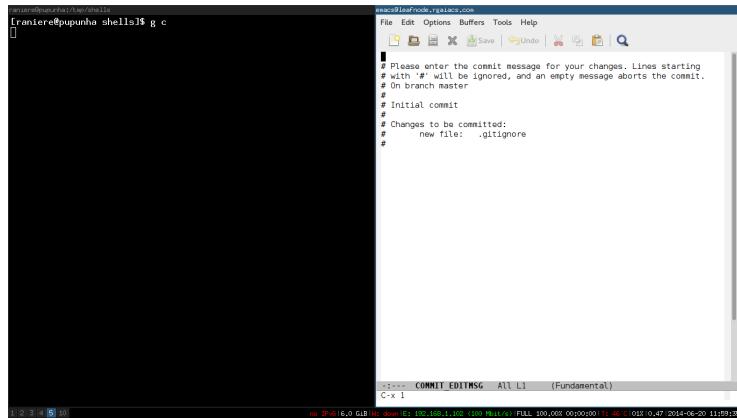
- 만약 셸 윈도우 하단에 (Fundamental) -----이 보인다면, emacs 혹은 xemacs 텍스트 편집기에 있다.

```
1 # Please enter the commit message for your changes. Lines starting
2 # with '#' will be ignored, and an empty message aborts the commit.
3 # On branch master
4 #
5 # Initial commit
6 #
7 # Changes to be committed:
8 #
9 #       new file: .gitignore
10#
```

emacs 혹은 xemacs에서 탈출하려면, CTRL-X CTRL-C을 누른다. 만약 저장되지 않은 변경사항이 있다면, 저장할 것인지 질문을 받게 된다 - 저장하려면 y를 누르거나, n 하고 나서 yes를 타이핑해서 저장하지 않고 끝낸다.

콘솔(console)에서 GUI 호출하기 Emacs에는 그래픽 사용자 인터페이스 모드가 지원된다. 프롬프트에서 호출하면, Emacs를

종료할 때까지 프롬프트가 명령에 응답하지 않는다.



A screenshot of a terminal window titled 'Terminal' on a Mac OS X desktop. The command 'git commit' has been entered, and the terminal is waiting for input. To the right of the terminal is a 'commit message' window from the 'git commit' tool. The window title is 'git commit -m "Initial commit"'. The message area contains the text '# Please enter the commit message for your changes. Lines starting # with '#' will be ignored, and an empty message aborts the commit.' followed by '# On branch master' and '# Changes to be committed:'. A new file named '.gitignore' is listed under the changes. The bottom of the window shows the status bar with 'COMMIT EDITMSG All L1 (Fundamental)' and 'C-x 1'.

파이썬

- 만약 쉘 프롬프트가 >>>이면, python에 있다.



A screenshot of a terminal window titled 'Terminal' on a Mac OS X desktop. The command 'python' has been entered, and the terminal is in the Python interactive shell. The prompt is '>>>'. The window title is 'Python 3.4.1 (default, May 19 2014, 17:23:49) [GCC 4.9.0 20140507 (prerelease)] on linux'. The bottom of the window shows the status bar with 'COMMIT EDITMSG All L1 (Fundamental)' and 'C-x 1'.

python에서 탈출하려면, exit() 혹은 CTRL-D를 타이핑한다.

- 만약 쉘 프롬프트가 ...이면, python 내부 열려있는(unclosed) 환경상태다.

```
[franice@ppunha shell]$ python
Python 3.4.1 (default, May 19 2014, 17:23:49)
[GCC 4.9.0 20140507 (pre-release) on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> def my_function():
...     print(
...     [
```

환경을 중단하려면, CTRL-C를 타이핑한다.

- 만약 셸 프롬프트가 In [123]:○|면, ipython에 있다.

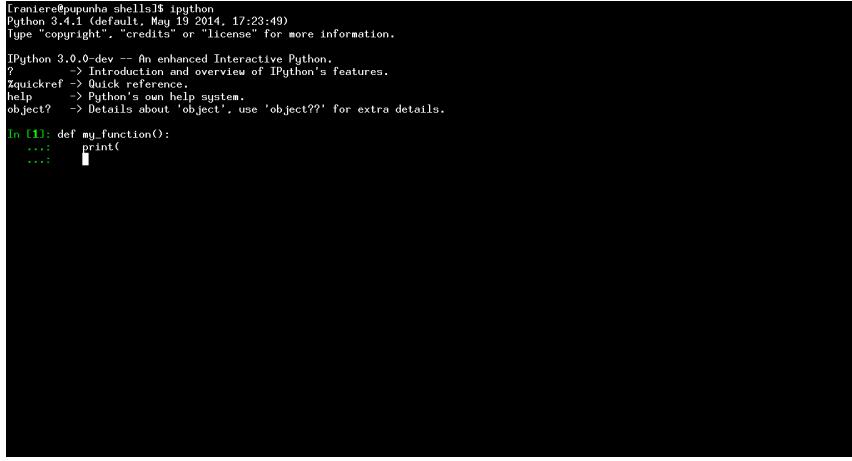
```
[franice@ppunha shell]$ python
Python 3.4.1 (default, May 19 2014, 17:23:49)
[GCC 4.9.0 20140507 (pre-release) on linux
Type "copyright", "credits" or "license" for more information.

IPython 3.0.0-dev -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
quickref -> Quick reference.
help       -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: [
```

ipython에서 탈출하려면, exit(), 혹은 CTRL-D을 누르고 나서 y를 누른다.

- 만약 셸 프롬프트가:○|면, ipython 내부 열려있는(unclosed) 환경에 있다.



```
[Tranier@pupunha shell]$ python
Python 3.4.1 (default, May 19 2014, 17:23:49)
Type "copyright", "credits" or "license" for more information.

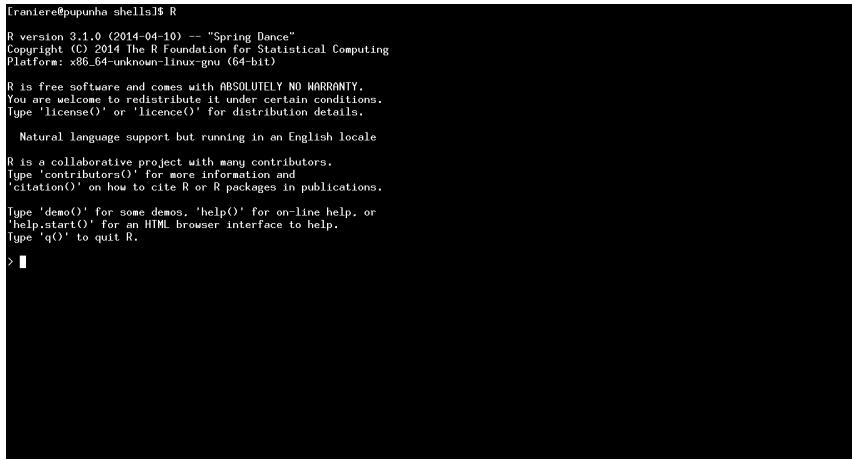
IPython 3.0.0-dev -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
quickref  -> Quick reference.
help       -> Python's own help system.
object?   -> Details about 'object'. Use 'object??' for extra details.

In [1]: def my_function():
...:     print(
...:         "
```

환경을 종단하려면, CTRL-C를 타이핑한다.

R

- 만약 쉘 프롬프트가 >이면, R에 있다.



```
[Tranier@pupunha shell]$ R
R version 3.1.0 (2014-04-10) -- "Spring Dance"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-unknown-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
> 
```

R에서 탈출하려면, q()을 타이핑한다. 만약 작업공간(workspace)을 저장할지 묻는다면, y를 눌러 저장하거나, n를 눌러 저장하지 않는다.

- 만약 쉘 프롬프트가 +라면, R 내부 열려있는(unclosed) 환경에 있다.

```

[franier@pupunha shell]$ R
R version 3.1.0 (2014-04-10) -- "Spring Dance"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-unknown-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> my_function <- function() {
+   [REDACTED]

```

환경을 종단하려면, CTRL-C를 타이핑한다.

R 참고 정보 (R Reference)

기본 연산

- #이 R에서 주석이다.
- x <- 3을 사용해서 값 3을 변수 x에 대입한다.
- R은 1에서 계수하는데, 많은 다른 프로그래밍 언어(예, 파이썬)와 다르다.
- length(thing)는 컬렉션(collection) 길이를 산출한다.
- c(value1, value2, value3)은 벡터를 생성한다.
- container[i]은 container에서 i번째 요소를 선택한다.

현재 환경에 있는 객체 목록을 출력한다. ls()

현재 환경에 있는 객체를 제거한다. rm(x)

현재 환경으로부터 모든 객체를 제거한다. rm(list = ls())

제어흐름 (Control Flow)

- if, elif, else을 사용해서 조건을 생성한다.

```

if(x > 0){
    print("value is positive")
} else if (x < 0){
    print("value is negative")
} else{
    print("value is neither positive nor negative")
}

```

- `for` 루프를 생성해서 한번에 하나씩 컬렉션 요소를 처리한다:

```

for (i in 1:5) {
    print(i)
}

```

상기 루프는 다음을 출력한다:

```

1
2
3
4
5

```

- `==`을 사용해서 같음(equality)을 테스트한다.
 - `3 == 3`은 `TRUE`를 반환한다,
 - `'apple' == 'orange'`은 `FALSE`를 반환한다.
- `X & Y`은 `X`와 `Y`가 모두 참(True)일 때만 참이다.
- `X | Y`은 `X` 혹은 `Y`가 모두가 참이거나, 둘중 하나가 참이면 참(True)이다.

함수(Functions)

- 함수 정의하기:

```

is_positive <- function(integer_value){

```

```

if(interver_value > 0){
    TRUE
} else{
    FALSE
}
}

```

R에서, 함수 마지막 실행 라인이 자동으로 반환된다.

- 함수 인자에 대한 기본값(default value) 명시하기

```

increment_me <- function(value_to_increment, value_to_increment_by = 1){
    value_to_increment + value_to_increment_by
}

```

increment_me(4)은, 5를 반환.

intrement_me(4, 6)은, 10을 반환.

- function_name(function_arguments)을 사용해서 함수를 호출.

– apply 가족(family) 함수:

```

apply()
sapply()
lapply()
mapply()

```

apply(dat, MARGIN = 2, mean)은 dat에 각 칼럼에 대한 평균 (mean)을 반환 한다.

패키지(package)

- install.packages("package-name")을 사용해서 패키지 설치한다.
- update.packages("package-name")을 사용해서 패키지를 갱신한다.
- library("package-name")을 사용해서 패키지를 적재한다.

추천 도서 (Reading)

도서 (Books)

Susan A. Ambrose, Michael W. Bridges, Michele DiPietro, Marsha C. Lovett, and Marie K. Norman

교육에 있어 근거기반실천(evidence-based practices) 최고의 단일 안내서.

Chris Fehily: *SQL: Visual QuickStart Guide* (3rd ed). Peachpit Press, 0321553578, 2002.

95% 실무를 다룰 5% SQL을 기술

Karl Fogel: *Producing Open Source Software: How to Run a Successful Free Software Project.*

공개 소프트웨어 프로젝트가 실제로 어떻게 동작하는지에 대한 안내서.

프로젝트에 커밋(commit) 권한을 획득하는 실용적인 조언, 더 많은 관심을

받게하거나 차이가 너무 커서 해소할 수 없는 경우에 포크(fork)하는 것이

포함됨.

Steve Haddock and Casey Dunn: *Practical Computing for Biologists.* Sinauer, 0878933913, 2010

“다른 90%” 과학 컴퓨팅에 대한 정말 훌륭한 일반 개론서.

Andy Oram and Greg Wilson (eds): *Making Software: What Really Works, and Why We Believe It*

최고의 소프트웨어 공학 연구자들이 각 장을 맡아서 핵심 경험적 결과와

증거를 기술한다. 프로그래머의 생산성에 프로그래밍 언어의 영향에서부터

통계적 기법을 사용해서 소프트웨어 오류를 예측할 수 있는지까지 다양한

주제를 다루고 있다.

Deborah S. Ray and Eric J. Ray: *Unix and Linux: Visual QuickStart Guide.* Peachpit Press, 0321553578

많은 예제를 갖고 있는 유닉스에 대한 친절한 소개서.

논문 (Papers)

Paul F. Dubois: “Maintaining Correctness in Scientific Programs”. *Computing in Science & Engineering*, 1999

종심 방어선을 구축하기 위해서 좋은 프로그래밍 실천사례를 제시한다.

그렇게 함으로써 한 사람이 빼먹은 오류를 다른 사람이 바로잡는다.

Matthew Gentzkow and Jesse M. Shapiro. 2014: “Code and Data for the Social Sciences: A Practical Guide.”

SAS와 엑셀 자료분석에서 재생가능한 방식으로 정리가 잘 된 데이터 위에

유지보수 가능한 스크립트(프로그램) 방식으로 어떻게 옮겨갈 것인지 대한

훌륭한 기술서.

Jo Erskine Hannay, Hans Petter Langtangen, Carolyn MacLeod, Dietmar Pfahl, Janice Singer, and Mark S. Parsons

과학자들이 연구에 컴퓨터를 어떻게 사용하고 얼마의 시간을 사용하는지에

대한 가장 큰 연구 조사.

William Stafford Noble: “A Quick Guide to Organizing Computational Biology Projects”. *PLoS*
과학자가 데이터와 스크립트를 조직화하는 이유와 방법.

Ethan P. White, Elita Baldridge, Zachary T. Brym, Kenneth J. Locey, Daniel J. McGlinn, and
논문 제목이 약속한 것을 정확하게 전달: 다른 과학자가 여러분의 데이터를
사용하기 쉽게 만드는 간단한 실천사례 집합.

Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy,
과학 소프트웨어 개발을 위한 최고 실천사례 집합을 기술하여 과학자가
작성한 소프트웨어 생산성과 신뢰성을 향상하고, 연구와 경험에 든든한
기반을 갖추게 한다.

Greg Wilson: “Software Carpentry: Lessons Learned”. *F1000 Research*, 3(62), 2014, doi:[10.12688/f1000research.62.1000](https://doi.org/10.12688/f1000research.62.1000)
지난 15년에 걸쳐 과학자에게 프로그램을 가르치는 방법에 대해서 경험하고
학습한 것을 기술한다.

용어사전 (Glossary)

절대 오차 (absolute error): 수학적으로 계산한 값과 컴퓨터를 사용하여 유한
근사한 절대값 차이.

절대 경로 (absolute path):

파일 시스템에 특정 장소를 나타내는 경로. 절대 경로는 통상 파일 시스템 루트
디렉토리에 대해 작성되고, (유닉스)에서는 “/”, (마이크로소프트 윈도우)에서는
“\”으로 시작한다. 상대 경로 (relative path) 참조.

접근 제어 목록 (access control list, ACL) : 파일 혹은 디렉토리에 연결되어
누가 무엇을 할 수 있는지를 명시하는 권한 목록.

가색 모델 additive color model): 빨간색, 녹색, 청색 (red, green, and blue)
같은 원색 기여분을 합산하여 색을 표현하는 방법.

집계 함수 (aggregation function): 많은 값을 조합하여 하나 결과값을 생성하는
합계(sum) 혹은 최대값(max) 같은 함수

(라이브러리의) 별명 (alias) : 가져오기(import)할 때 라이브러리 (library)에 별
칭을 부여하는 것.

인자 (argument): 함수나 프로그램이 실행할 때, 함수나 프로그램에 주어진 값.
이 용어는 종종 매개변수 (parameter)와 상호호환되어 (그리고 일관성 없이) 사용
된다.

가정 대입문 (assertion): 프로그램 특정 지점에 참(true)으로 추정되는 표현식. 일반적으로 프로그래머는 오류 확인점검하기 위해서 코드에 가정 대입문을 넣는다. 만약 가정 대입문이 실패하면 (즉, 표현식이 거짓으로 평가되면), 프로그램은 정지하고 오류 메시지를 출력한다. 불변성 (invariant), 사전조건 (precondition), 사후조건(postcondition)을 참조한다.

대입 (assignment): 변수에 값을 연계함으로써, 값을 이름에 할당하는 것.

원자값 (atomic value): 더 작은 조각으로 분해될 수 없는 값. 예를 들어, 숫자 12는 일반적으로 원자(atomic) 값으로 볼 수 있다. (만약 학교 학생에게 덧셈을 가르치지 않는다면, 이 경우에 10과 2로 분해할 수 있다.)

브랜치 (branch): 버전 제어 (version control) 저장소 (repository)에 있는 “평행 우주(parallel universe)”. 일반적으로 프로그래머는 브랜치(branch, 분기)를 사용해서 개발 과정에서 서로에게서 다른 변경사항을 격리한다. 그렇게 함으로써, 한번에 한 문제에 집중할 수 있다. 병합 (merge) 참조.

콜 스택 (call stack): 실행 프로그램 내부에서 활성 함수 호출을 추적하는 자료 구조. 각 호출 변수는 스택 프레임 (stack frame)에 저장된다; 신규 스택 프레임은 각 호출에 대해서 기존 스택 위에 놓여지고, 호출이 완료될 때 폐기된다.

계단식 삭제 (cascading delete): 레코드가 삭제될 때, 데이터베이스에서 그 해당 레코드에 기대고 있는 것을 자동으로 삭제하는 관행. 참조 무결성 (referential integrity) 참조.

대소문자 구별안함 (case insensitive): 대문자와 소문자를 동일한 것처럼 텍스트 문자를 처리. 대소문자 구별 (case sensitive) 참조.

예외처리 (catch an exception): 예외 (exception)가 프로그램 어디선가 발생 (raised)했을 때 처리.

변경 집합 (change set): 하나 혹은 그 이상 파일에 대한 변경 그룹으로 단일 작업으로 버전 제어 (version control) 저장소 (repository)에 커밋 (committed) 된다.

클론 (clone) (a repository): 버전 제어 (version control) 저장소 (repository)에 로컬 사본을 생성. 포크 (fork) 참조.

코드 리뷰 (code review): 소프트웨어 조각 혹은 변경부분에 대한 체계적인 동료 검토. 저장소(repository)에 병합(merged)되기 전에, 동료 검토(peer review)가 통상 풀 요청(pull requests) 단계에서 수행된다.

콤마 구분값 (comma-separated values, CSV): 테이블로 일반적 텍스트를 표현하는데 각 행 값이 콤마로 구분된다.

명령-라인 인터페이스 (command-line interface, CLI) : 일반적으로 REPL로 명령어 타이핑에 기반한 인터페이스. 그래픽 사용자 인터페이스 (graphical user interface) 참조.

주석 (comment): 프로그램에서 사람이 어떻게 진행되고 있는지에 대한 이해를 주려고 작성된 글로, 컴퓨터는 무시한다. 파이썬, R, 유닉스 쉘은 # 문자로 시작해서 라인 끝까지 간다; SQL 주석은 --이고, 다른 언어에는 각자의 관습이 있다.

조건문 (conditional statement): 프로그램 문장으로 테스트가 참 혹은 거짓에 따라 실행될 수도 실행되지 않을 수 있다.

충돌 (conflict): 버전제어시스템 (version control system) 한 사용자가 만든 변경 사항이 다른 사용자가 만든 변경사항과 양립할 수 없는 상황. 충돌을 해소(resolve)하도록 사용자를 돋는 것이 버전 제어의 주요한 작업 중 하나다.

외적 (cross product): 한 집합 모든 요소를 다른 집합 모든 요소와 짹짓는 것.

현재작업디렉토리 (current working directory): 상대 경로 (relative paths)로 계산되는 디렉토리; 동일하게, 이름으로만 파일을 참조하여 찾을 수 있는 장소. 모든 프로세스(process)는 현재작업디렉토리를 갖는다. 통상 현재작업디렉토리는 단축 표기 . (“닷(dot)”으로 발음) 표기된다.

커서 (cursor): 데이터베이스에 작업을 추적하는 포인터.

자료형 (data type): 정수(integer) 혹은 문자열 (character string) 같은 자료 값 유형.

데이터베이스 관리자 (database manager): 관계형 데이터베이스(relational database)를 관리하는 프로그램.

기본 매개변수값 (default parameter value): 명시적으로 어떤 것도 지정되지 않는다면, 매개변수(parameter)에 적용되는 값.

방어적 프로그래밍 (defensive programming): 가능하면 빨리 오류를 잡아내도록 연산작업을 점검하는 프로그램을 작성하는 관행.

구분자 (delimiter): CSV 파일에 칼럼 사이 콤마처럼 각 값을 구분하는데 사용되는 문자(들).

docstring: “문서화 문자열(documentation string)”의 축약형. 파이썬 프로그램에 내장된 텍스트 문서를 지칭한다. 주석과 달리, docstring은 실행 프로그램에 상주하며 인터랙티브 세션에서 조사할 수 있다.

문서화 (documentation): 사람 언어로 텍스트로 작성되어 소프트웨어가 무엇을 수행하며, 어떻게 동작하고, 어떻게 사용하는지 기술한다.

점표기법 (dotted notation): 많은 프로그램 언어에 사용되는 2분 표기법(two-part notation). `thing.component`에서 `component`가 `thing`에 속하는 것을 나타낸다.

빈 문자열 (empty string): 어떤 문자도 포함하지 않아서, 종종 “제로(zero)” 텍스트로 간주되는 문자열.

캡슐화 (encapsulation): 어떤 것의 구현 상세 세부내용을 숨기는 관례. 그렇게 함으로써, 프로그램이 방법(how)보다 무엇(what)에 집중할 수 있다.

예외 (exception): 정상 혹은 예상 프로그램 실행을 파괴하는 이벤트. 대부분 최신 언어는 데이터 일부에서 오동작(또한 예외로 불림)하는 것에 대한 정보를 기록한다. 예외처리 (catch), 예외발생 (raise).

(데이터베이스) 필드(field): 테이블(table)에 각 레코드(record)마다, 특정 자료형을 갖는 데이터 값 집합.

파일 확장자 (filename extension): 마지막 “.” 문자 뒤에 오는 파일이름 부분. 관례로, 이것을 사용해서 파일 형식을 식별한다; `.txt`는 “텍스트 파일”, `.png`는 “휴대용 네트워크 그래픽 파일 (Portable Network Graphics file)” 등등 의미한다. 이러한 관례가 대부분의 운영체제에서 강제되지는 않는다: MP3 소리 파일을 `homepage.html`로 이름을 부여하는 것도 가능하다. 많은 응용프로그램이 파일 확장자를 사용해서 파일 마임 형태(MIME type)을 식별하기 때문에, 파일 확장자를 잘못 명명하게 되면 응용프로그램이 중단되어 실패할 수도 있다.

파일시스템 (filesystem): 파일, 디렉토리, 입출력 장치 (키보드, 모니터 화면) 집합. 파일시스템이 물리적 장치에 여기저기 흩어지거나 많은 파일시스템이 단일 물리 장치에 저장될 수도 있다; 운영체제(operating system)가 접근을 관리한다.

필터 (filter): 자료 흐름을 변환하는 프로그램. 많은 유닉스 명령어-라인 도구는 필터로 작성된다: 표준입력 (standard input)에서 데이터를 읽고, 처리하고, 결과를 표준출력 (standard output)에 쓴다.

플래그 (flag): 옵션을 명시하거나 명령어-라인 프로그램 설정하는 간결한 방법.

관례로, 유닉스 응용프로그램은 -v처럼 대쉬 다음에 문자 하나를 혹은 --verbose처럼 대쉬 2개 다음에 단어를 사용한다. 반면에 도스 응용프로그램은 /V처럼 슬래귀(slash)를 사용한다. 응용프로그램에 따라, 플래그를 -o /tmp/output.txt와 같이 단일 인자 앞에 둘 수 있다.

부동소수점 수 (floating point number): 소수 부분과 지수 부분을 담고 있는 숫자. 정수(integer) 참고.

for 루프 (for loop): 집합, 리스트 혹은 범위에 속한 각 값에 대해 한번씩 실행되는 루프. while 루프 (while loop) 참고.

외래키 (foreign key): 또 다른 테이블에 있는 레코드 (records)를 식별하는데 사용되는 데이터베이스 테이블 (database table)에 있는 하나 혹은 그 이상의 값.

포크 (fork): 서버에 버전제어 (version control) 저장소 (repository)를 복제(clone)하는 것.

함수 몸통 (function body): 함수 내부 실행되는 문장.

함수 호출 (function call): 또 다른 소프트웨어 부분에서 함수를 사용.

함수 합성 (function composition): $f(g(x))$ 처럼, 함수 결과를 즉시 다른 함수에 적용.

그래픽 사용자 인터페이스 (graphical user interface, GUI): 통상 마우스를 사용해서 제어되는 그래픽 사용자 인터페이스. 명령-라인 인터페이스 (command-line interface) 참조.

홈 디렉토리 (home directory): 컴퓨터 시스템 계정과 연관된 기본 디렉토리. 관례로, 사용자 파일 모두가 홈 디렉토리 혹은 하위 디렉토리에 저장된다.

HTTP: 웹페이지와 월드와이드웹 상에 다른 데이터를 공유하는데 사용되는 하이퍼텍스트 전송 프로토콜(Protocol).

불변(immutable): 변경할 수 없는 것. 불변 데이터 값은 생성되면 변경할 수 없다. 변경가능 (mutable) 참고.

가져오기 (import): 라이브러리(library)를 프로그램에 적재.

제자리 연산 (in-place operator): += 같은 연산자로 대입되는 변수가 대입 오른편에 피연산자가 되는 흔한 경우에 제공되는 짧은 표기법. 예를 들어, $x += 3$ 문장은 $x = x + 3$ 과 동일한 의미다.

인덱스 (index): 이미지에 픽셀 하나처럼, 컬렉션(collection)에 단일 값 위치를 지정하는 첨자.

전염성있는 라이센스 (infective license): [GPL](#)같은 라이센스로 자신의 작업물에 저작물을 적용한 사람도 동일한 공유 요건을 두게 만드는 라이센스.

내부 루프 (inner loop): 또다른 루프 내부에 있는 루프. 외부 루프 (outer loop) 참조.

정수 (integer): -12343같은 정수(whole number). 부동소수점 수 (floating-point number) 참조.

불변 (invariant): 일반적으로 가정 대입문(assertion)에 사용되며, 프로그램 실행 동안에 값이 변경되지 않는 표현식. 사전조건 (precondition), 사후조건 (postcondition) 참조.

라이브러리 (library): 관련된 작업 집합을 구현하는 (함수, 클래스, 변수) 코드 조직단위.

루프 몸통 (loop body): for 루프 (for loop) 혹은 while 루프 (while loop) 내부에서 반복되는 문장 혹은 명령어 집합.

루프 변수 (loop variable): 루프 진도를 추적하는 변수.

매크로 (macro): Makefiles에서 변수(variables)가 종종 매크로 불린다. 왜냐하면 읽어들여졌을 때 확장되기 때문이다.

makefile: make 프로그램에 입력 파일. make에 무엇을 수행할지 지시한다.

멤버 (member): 객체(object) 내부에 포함된 변수.

(저장소) 병합(merge) : 저장소 (repository)에 두 변경 집합을 일치시킨다.

메쏘드 (method): 특정 객체(object)에 묶인 함수. 각 객체 메쏘드는 객체가 할 수 있는 것 중 하나 혹은 답할 수 있는 질문 중 하나를 구현한다.

변경가능 (mutable): 변경할 수 있는 것. 변경할 수 있는 데이터 값은 제자리에서 변경될 수 있다. 불변(immutable) 참조.

관념기계 (notional machine): 컴퓨터가 무엇을 할 수 있고, 무엇을 할 것인지에 대해 생각하는데 사용되는 추상 컴퓨터.

직교 (orthogonal): 서로 다른 것과 독립적인 행동과 의미를 갖는 것. 만약 개념 혹은 도구 집합이 직교라면, 어떤 방식으로든지 조합될 수 있다.

외부 루프 (outer loop): 또다른 루프를 포함하는 루프. 내부 루프 (inner loop) 참조.

매개변수 (parameter): 함수 선언에 명명되는 변수로 함수 호출에 전달되는 값을 보관하는데 사용된다. 이 용어는 종종 인자(argument)와 상호호환되어 (그리고 일관성 없이) 사용된다.

부모 디렉토리 (parent directory): 해당 디렉토리를 “담고있는” 디렉토리. 모든 디렉토리는 루트 디렉토리 (root directory)를 제외하고 부모가 있다. 디렉토리 부모는 통상 ... (“닷닷(dot dot)”으로 발음)을 사용해서 표기한다.

가짜 목표 (phony target): Makefile 규칙(rule) 내부 형식으로 생성될 필요가 있는 실제 파일과 상응하지는 않지만, 대신에 의존성(dependencies) 실행을 확실히 하는데 사용되는 플레이스홀더 placeholder 역할을 수행.

파이프 (pipe): 한 프로그램 출력이 다른 프로그램 입력으로 연결. 두개 혹은 그 이상의 프로그램이 이런 방식으로 연결될 때, “파이프라인(pipeline)”으로 불린다.

파이프와 필터 (pipe and filter): 프로그래밍 모형으로 데이터 스트림(streams)을 처리하는 필터(filters)가 끝과 끝을 붙여 연결된다. 파이프와 필터 모형은 유닉스 쉘(shell)에서 폭넓게 사용된다.

사후조건 (postcondition): 함수가 실행을 종료하자마자 함수(혹은 다른 코드 블록)가 보장하는 조건이 참(true)이다. 사후조건은 종종 가정대입문.assertions으로 표현된다.

사전조건 (precondition): 함수 (혹은 다른 코드 블록)가 올바르게 실행되기 위해 참(true)인 조건.

prepared statement: SQL 쿼리 템플릿으로 그곳에 값이 채워질 수 있다.

기본키(primary key): 데이터베이스 테이블(database table)에 있는 하나 혹은 그 이상의 필드(fields)로 그 값이 각 레코드(record)에 유일하다(유니크, unique)고 보장된 것. 즉, 해당 값이 유일하게 항목을 식별한다.

프로세스 (process): 프로그램 실행 인스턴스로 코드, 변수 값, 파일, 네트워크 연결 정보 등을 담고 있다. 프로세스는 운영체제(operating system)가 관리하는 “배우(actor)”다; 통상, 운영체제는 각 프로세스를 한번에 수백분의 몇초 동안 실행해서 운영체제가 여러 프로세스를 동시에 실행하고 있다는 인상을 준다.

프롬프트 (prompt): 컴퓨터가 다음 명령을 대기하고 있다는 것을 보여주기 위해서 REPL에 의한 한 문자 혹은 여러 문자 출력.

프로토콜 (protocol): 한 컴퓨터가 다른 컴퓨터와 어떻게 의사소통하지는 정의하는 규칙 집합. 인터넷 공통 프로토콜에 HTTP와 SSH이 포함된다.

풀요청 (pull request): 버전제어(version control) 저장소(repository)에 생성된 변경 사항으로 병합(merging)을 위해 다른 곳에 제공된다.

쿼리 (query): 값을 읽어 들이지만, 어떤 것도 변경하지 않는 데이터베이스 연산. 쿼리는 SQL로 불리는 특수 목적 언어로 표현된다.

(쉘에서) 인용 (quoting): 쉘이 특수 문자를 해석하지 못하게 다양한 종류의 인용부호를 사용하는 것. 예를 들어, 프로그램에 *.txt 문자열을 전달하기 위해서, (단일 인용부호)로 '*.txt'와 같이 일반적으로 작성하는 것이 필요하다. 그렇게 함으로써 쉘이 와일드카드 문자 *를 확장하지 않는다.

(예외) 발생 (raise): 프로그램에 예외(exception)가 발생했다는 신호를 명시적으로 준다. (예외) 처리(catch) 참조.

읽기-평가-출력 루프 (read-eval-print loop, REPL): 명령-라인 인터페이스 (command-line interface, CLI)로 사용자로부터 명령을 읽고, 실행하고, 결과를 출력하고 다음 명령을 대기한다.

(데이터베이스에서) 레코드 (record): 데이터베이스 테이블 (database table)에서 단일 항목을 구성하는 관련된 값 집합. 통상 행(row)으로 표현됨. 필드(field) 참조.

리다이렉트(redirect): 화면이나 다른 명령에 보내기보다 파일에 명령 출력 결과를 전송함. 혹은 동등하게 파일로부터 명령 입력값을 읽는 것.

참조무결성 (referential integrity): 데이터베이스에 값에 대한 내부 일치성. 만약 한 테이블에 항목이 외래키(foreign key)를 갖고 있지만 상응하는 레코드 (records)가 존재하지 않는다면, 참조무결성이 위반된 것이다.

회귀(regression): 일전에 수정한 버그가 다시 생겨난 것.

정규표현식 (regular expressions, RE): 문자열 집합을 명시하는 패턴. 문자열에서 문자 시퀀스(sequence)를 찾는데 RE가 가장 자주 사용된다.

관계형 데이터베이스 (relational database): 테이블(tables)로 조직화되는 데이터 집합.

상대오차 (relative error): 절대오차 (absolute error) 비율로 근사값으로 실제값을 근사한다.

상대경로(relative path): 현재작업디렉토리 (current working directory)에 대해서 파일 혹은 디렉토리 위치를 명시하는 경로(path). 구분 문자 (유닉스 “/” 혹은 윈도우“\\”)로 시작하지 않는 모든 경로는 상대경로다. 절대경로 (absolute path) 참조.

원격 로그인 (remote login): 네트워크로 컴퓨터에 연결. 즉, 네트워크로 쉘(shell)을 실행. SSH 참조.

원격 저장소 (remote repository): 현재 저장소가 아닌 버전 제어 저장소(repository)로 현재 저장소가 무언가로 연결되거나 미러링(mirroring)된 것.

저장소 (repository): 저장 영역으로 버전제어(version control) 시스템이 누가 언제 무엇을 변경했는지에 대한 수정(revisions)된 과거 파일과 정보를 저장한다.

해소하다 (resolve): 만약 버전제어(version control) 시스템에서 관리되는 파일 한개 혹은 파일 집합에 둘 혹은 그 이상 양립할 수 없는 변경사항이 있다면 충돌(conflicts)을 제거하는 것.

반환문장 (return statement): 함수가 실행을 멈추고, 값을 즉시 호출자(caller)에 반환하는 문장.

수정(revision): 버전제어(version control) 저장소(repository)에 기록된 상태.

RGB(빨간색-녹색-파란색, red-green-blue) : 가색모형(additive model)으로 색을 빨간색, 녹색, 파란색을 조합해서 표현. 각 색깔 값 범위는 일반적으로 0..255 (즉, 1 바이트 정수).

루트 디렉토리 (root directory): 파일시스템(filesystem)에서 최상단 디렉토리. 루트 디렉토리 이름은 유닉스(리눅스 및 맥OS 포함)에서 “/”이고, 마이크로소프트 윈도우에서는 “\\”이 된다.

검색 경로(search path): 프로그램이 실행될 때, 쉘(shell)이 프로그램을 검색하는 디렉토리 목록.

보초값 (sentinel value): 컬렉션(collection)에서 특별한 의미를 갖는 값. 예를 들어, 999가 “연령 미상”을 의미.

(배열) 형상(shape): 배열의 차원으로 벡터로 표현. 예를 들어, 5×3 배열 형상 (array's shape)은 $(5, 3)$ 이다.

쉘(shell): Bash (the Bourne-Again Shell)나 마이크로소프트 윈도우 도스 쉘처럼 명령-라인 인터페이스(command-line interface, CLI)로 사용자가 운영체제(operating system)와 상호작용할 수 있게 한다.

쉘 스크립트(shell script): 재사용을 위해서 파일에 저장된 쉘(shell) 명령어 집합. 쉘 스크립트는 쉘이 실행하는 프로그램이다; “스크립트(script)”라는 이름은 역사적인 연유로 인해서 사용된다.

부호와 크기(sign and magnitude): 숫자를 표현하는 기법(scheme)으로 비트 하나가 부호(양 혹은 음)로 다른 비트들이 숫자의 절대값을 저장한다. 2 보수 (two's complement) 참조.

무소식 고장 (silent failure): 어떠한 경고 메시지도 없이 고장남. 무소식 고장 (silent failure)는 탐지하고 디버깅하기 어렵다.

슬라이스 (slice): 더 커다란 시퀀스의 정규 부분시퀀스(subsequence) 예를 들어, 첫 다섯개 요소 혹은 모든 두번째 요소.

구조적 질의 언어 (SQL, Structured Query Language) : 관계형 데이터베이스 (relational databases)에 연산을 기술하는데 사용되는 특수 목적 언어(special-purpose language)

SQL주입공격 (SQL injection attack): 사용자 입력값에 악성 SQL 문장을 포함한 프로그램으로 공격. 만약 사용자 입력 텍스트가 직접 SQL문장으로 복사되면, 데이터베이스에서 실행될 것이다.

SSH: 컴퓨터 사이에 보안 통신을 위해 사용되는 시큐어 쉘(secure shell)프로토콜 (protocol). 종종 SSH는 컴퓨터 사이 원격 로그인(remote login)으로도 사용된다.

SSH 키 (SSH key): 다른 곳에 컴퓨터나 사용자를 식별하는 디지털 키.

스택프레임 (stack frame): 함수 로컬 변수에 저장공간을 제공하는 자료구조. 함수가 매번 호출될 때, 신규 스택프레임이 생성되고, 콜스택(call stack) 위에 놓여진다. 함수가 반환될 때, 해당 스택프레임은 폐기된다.

표준입력 (standard input, stdin): 프로세스 기본 입력 스트림. 인터랙티브 명령-라인 응용프로그램에서, 표준입력은 통상 키보드에 연결된다; 파이프(pipe)에서는 표준입력은 이전 프로세스 표준출력(stdandard output)으로부터 데이터를 받는다.

표준출력 (standard output, stdout): 프로세스 기본 출력 스트림. 인터랙티브 명령-라인 응용프로그램에서, 표준출력에 전송된 데이터가 화면 스크린에 출력된다; 파이프(pipe)에서는 표준출력이 다음 프로세스에 표준입력(stdandard input)으로 넘겨진다.

스트라이드 (stride): 슬라이스(slice) 연속 요소(element) 사이 오프셋(offset).

문자열(string): “문자 열(character string)”을 줄인 것, 0 혹은 그 이상 문자 시퀀스(sequence).

하위-디렉토리 (sub-directory): 다른 디렉토리 내부에 담겨진 디렉토리.

자동탭완성 (tab completion): 많은 인터랙티브 시스템에서 제공되는 기능으로 텁(Tab)키를 눌러 자동으로 해당 단어 혹은 명령어를 완성한다.

(데이터베이스에서) 테이블(table) : 관계형 데이터베이스(relational database)에 데이터 집합으로 레코드(records) 집합으로 조직화되고, 각 레코드는 동일한 필드(fields)를 갖는다.

시금석 (test oracle): 테스트 결과를 비교할 수 있는 프로그램, 장치, 데이터셋, 혹은 인력.

테스트 주도 개발 (test-driven development, TDD): 테스트할 코드를 작성하기 이전에 단위 테스트를 작성하는 관행.

시간도장(timestamp): 특정 사건(event)이 발생한 시점을 기록.

튜플(tuple): 불변(immutable) 시퀀스(sequence) 값.

2 보수 (two's complement): 자동차 주행 기록처럼 감싸서 숫자를 표현하는 기법(scheme). 그래서, 111...111이 -1을 표현. 부호와 크기 (sign and magnitude) 참조.

사용자 그룹 (user group): 컴퓨터 시스템 상에 사용자 집합.

사용자 그룹 ID(user group ID): 사용자 그룹(user group)을 특정하는 숫자 ID.

사용자 그룹명 (user group name): 사용자 그룹(user group)에 대한 텍스트 명칭.

사용자 ID(user ID): 숫자 ID로 컴퓨터 시스템에 개인 사용자를 특정한다. 사용자명 (user name) 참조.

사용자명(user name): 컴퓨터 시스템에 사용자에 대한 텍스트 명칭. user ID 참조.

변수 (variable): 값 혹은 값 집합과 연결되는 프로그램에 있는 명칭.

버전제어 (version control): 파일 집합에 대한 변경사항을 관리하는 도구. 각 변경집합은 파일에 새로운 수정(revision) 사항을 생성한다; 버전제어시스템은 사용자로 하여금 신뢰성있게 과거 수정사항을 되살리게 하고, 다른 사용자들로 인한 충돌는 변경사항을 관리할 수 있게 돋는다.

while 루프 (while loop): 조건이 참(true)이기만 하면 계속 실행되는 루프. for 루프(for loop) 참조.

와일드카드 (wildcard): 패턴매칭에 사용되는 문자. 유닉스 쉘에서 와일드카드 “*”은 0 혹은 그이상 문자를 매칭한다. 그래서 *.txt은 .txt으로 끝나는 이름을 갖는 모든 파일을 매칭한다.

규칙 (Rules)

- 한주 힘든 일이 종종 한시간 동안 생각할 것을 아낄 수 있게 한다. (A week of hard work can sometimes save you an hour of thought.)
- 일찍 실패하고, 자주 실패하라. (Fail early, fail often.)
- 버그를 가정선언문 혹은 테스트 케이스로 변환하라 (Turn bugs into assertions or tests.)
- 빨간색, 녹색, 리팩터링 (Red, green, refactor.)
- 항상 데이터로부터 초기화하라 (Always initialize from data.)
- 먼저 단순한 것을 테스트하라 (Test the simple things first.)
- 무엇을 동작하기로 했는지 파악하라 (Know what it's supposed to do.)
- 매번 실패하게 만들어라 (Make it fail every time.)
- 빨리 실패하게 만들어라 (Make it fail fast.)
- 사유를 갖고 한번에 하나씩 변경하라 (Change one thing at a time, for a reason.)
- 지금까지 수행한 것을 추적하라 (Keep track of what you've done.)
- 겸손하라 (Be humble.)
- 낮게 던지고 높게 잡는다. (Throw low, catch high.)

라이센스 (Licenses)

강의 교재

모든 소프트웨어 카펜트리 원문과 [xwMOOC](#) 한국어 번역 강의 교재는 크리에이티브 커먼즈 라이센스로 이용할 수 있다.

한국어 이용자는 다음의 권리를 갖습니다:

- **공유**—복제, 배포, 전시, 공연 및 공중송신 (포맷 변경도 포함)
- **변경**—리믹스, 변형, 2차적 저작물의 작성

영리목적으로도 이용이 가능합니다.

아래 조건을 준수하는 한 이 라이선스는 취소되거나 실효되지 않습니다.

다음과 같은 조건을 따라야 합니다:

- **저작권자표시**—적절한 출처“Copyright (c) Software Carpentry, xwMOOC”와, 해당 라이센스 링크를 표시하고, 변경이 있는 경우 공지해야 합니다. 실무적으로 소프트웨어 카펜트리 링크 <http://software-carpentry.org>와 xwMOOC 링크 <http://www.xwmooc.net>를 포함해야 한다. 합리적인 방식으로 이렇게 하면 되지만, 이용 허락권자가 귀하에게 권리를 부여한다거나 귀하의 사용을 허가한다는 내용을 나타내서는 안 됩니다.
- **동일조건변경허락**—이 저작물을 리믹스, 변형하거나 2차적 저작물을 작성하고 그 결과물을 공유할 경우에는 원 저작물과 동일한 조건의 CCL을 적용하여야 합니다.

추가제한금지 이용자는 이 라이선스로 허용된 행위를 제한하는 법적 조건이나 기술적 조치를 부가해서는 안 됩니다.

주의: 퍼블릭 도메인에 해당하는 저작물의 이용 또는 저작재산권의 제한사유에 해당하는 이용의 경우에는 이 라이선스를 따르지 않아도 됩니다. 이 라이선스는 아무런 보증을 하지 않습니다. 또한 저작물의 이용에 필요한 모든 허락이 포함되어 있지 않을 수도 있습니다. 예를 들어 퍼블리시티권, 프라이버시, 인격권 등 다른 권리에 따른 제한이 있을 수 있습니다.

영어 원저작물은 다음과 같은 저작권을 갖습니다:

- to **Share**—to copy, distribute and transmit the work
- to **Remix**—to adapt the work

Under the following conditions:

- **Attribution**—You must attribute the work using “Copyright (c) Software Carpentry” (but not in any way that suggests that we endorse you or your use of the work). Where practical, you must also include a hyperlink to <http://software-carpentry.org>.

With the understanding that:

- **Waiver**—Any of the above conditions can be waived if you get permission from the copyright holder.
- **Other Rights**—In no way are any of the following rights affected by the license:
 - Your fair dealing or fair use rights;
 - The author’s moral rights;
 - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights. *
- **Notice**—For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to <http://creativecommons.org/licenses/by/3.0/>.

For the full legal text of this license, please see <http://creativecommons.org/licenses/by/3.0/legalcode>.

소프트웨어

별도로 명기된 것을 제외하면, 예제 프로그램과 소프트웨어 카펜트리에서 제공된 소프트웨어는 다음 라이센스를 준수한다. OSI-승인 MIT license.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

요약하면

- 이 소프트웨어를 누구라도 무상으로 제한없이 취급해도 좋다. 단, 저작권 표시 및 이 허가 표시를 소프트웨어의 모든 복제물 또는 중요한 부분에 기재해야 한다.
- 저자 또는 저작권자는 소프트웨어에 관해서 아무런 책임을 지지 않는다.

기타 저작물

소프트웨어 카펜트리 행동강령은 웹사이트를 참조했으며, 소프트웨어 카펜트리 소프트웨어 라이센스와 동일한 MIT 라이센스를 이용가능한다.

상표권(Trademark)

“Software Carpentry”와 소프트웨어 카펜트리 로고는 Software Carpentry, Ltd 등록 상표다. “xwMOOC”와 xwMOOC로고는 엑스더블유무크 등록 상표다.

[Email](#) [Twitter](#) [RSS](#) [GitHub](#) [IRC](#) [License](#) [Bug Report](#)