정보교육을 위한 파이썬

정보 탐색

Version 0.0.9-d2

저자: Charles Severance
번역: 이광춘, 한정수
(xwmooc)

출판 이력:

2014년 9월: xwmooc 프로젝트 일환으로 *"정보과학을 위한 파이썬"* 으로 제목 정하고 한국어로 번역 공개

2013년 10월: JSON로 전환, OAuth 사용. 13장, 14장에 주요 개정 신규 시각화 장 추가.

2013년 9월: Amazon CreateSpace 책 출판

2010년 1월 : 미시건 대학 Espresso Book Machine 사용 책 출판

2009년 12월: *Think Python: How to Think Like a Computer Scientist*에서 2장 ~ 10장까지 주요 개정
*Python for Informatics: Exploring Information*을 위해 1장, 11장 ~ 15장 저작

2008년 6월: *Think Python: How to Think Like a Computer Scientist* 제목 바꾸고, 주요 개정.

2007년 8월: *How to Think Like a (Python) Programmer* 제목 바꾸고, 주요 개정.

2002년 4월: *How to Think Like a Computer Scientist* 초판 공개

한국어판 서면

첫 인터넷 웹 브라우저를 만든 마크 앤더슨은 소프트웨어가 세상을 먹고 있다(”Software is eating the world”)는 자극적인 표현으로 2011년 월스트리트 저널에 에세이를 썼고, 카네기멜론 대학의 쟈넷 윙 교수는 이론적 사고(Theoretical Thinking), 실험적 사고(Experimental Thinking))와 더불어 정보적 사고(Computational Thinking)가 현재도 그렇지만 앞으로 인간의 사고를 지배하는 중추적인 역할을 할 것을 주장했다. 이들의 결과는 정보적 사고를 배운 사람과 소프트웨어를 이해하고 활용하는 사람과 그렇지 못한 사람과의 차이는 산업경제의 빈부격차보다 더 큰 디지털 경제의 정보 불평등(Digital Divide)를 야기할 것으로 예측했다.

정부는 ’14년 7월 세계 경제, 사회 환경이 소프트웨어 중심사회로 급격히 변화하고 있으며, 소프트웨어가 혁신과 성장, 가치창출의 중심이 되고, 개인•기업•국가의 경쟁력을 좌우하는 중요한 역할을 하고 있음에도 불구하고, 우리나라는 범정부적, 국민적 관심이 미흡한 상황이라고 진단하고, 미국, 영국, 이스라엘 등 선진국과 마찬가지로, 초•중•고에서 소프트웨어를 필수로 이수할 수 있는 방안을 강구하고 있다.

하지만, 지금까지의 관심은 소프트웨어만 집중되어 왔고, 정보 및 데이터에 대한 부분은 상대적으로 소홀히 다뤄왔다. ”Python for Informatics” 번역을 통해서 컴퓨터 언어를 쉽고 빠르게 그리고 정보 및 데이터에 대한 부분도 효과적으로 학습할 수 있을 것으로 기대한다.

이광춘 한정수 (xwmooc)
http://www.xwmooc.net
경기도 과천
2014년 9월

서면

정보교육을 위한 파이썬: 공개된 책 리믹싱

"출판 혹은 소멸(publish or perish)"를 들어온 학자는 자신만의 신선한 창조로 무에서 만들어내는 것은 무척이나 자연스럽다. 이책은 아무 것도 없는 것에서 시작하는 대신에 Allen B. Downey, Jeff Elkner와 협력자들이 저작한 *Think Python: How to Think Like a Computer Scientist* 책을 "리믹싱(re-mixing)"하는 실험이다.

2009년 12월, 미시건 대학에서 연속해서 5 학기 **SI502 - Networked Programming**을 준비중이었고, 알고리즘과 추상화를 이해하는 대신에 데이터 탐색에 집중하는 파이썬 교과서를 쓸 시점이라고 정했다. SI502 목표는 파이썬을 사용하여 사람들에게 평생 데이터를 다루는 기술을 가르치는 것이다. 대신에, 핵생들 중 누구도 전문적인 컴퓨터 프로그래머를 계획한 사람은 없었다. 대신에 학생들은 도서관원, 관리자, 변호사, 생물학자, 경제학자가 되고자 했는데 자신만의 영역에서 능숙하게 기술을 사용하고자 했다.

수업을 위해서 결코 완벽한 데이터 지향 파이썬 책을 발견할 것 같지 않아서 그런 책을 저작하려고 시작했다. 휴가 기간동안 아무것도 없는 상태에서 새로운 책을 시작하기 3주전 다행스럽게도 교수 회의에서, Atul Prakash 박사가 지난 학기 파이썬 과정을 가르치는데 사용한 *Think Python* 책을 보여주었다. 간결하고 직접적인 설명에 학습하기 쉬운 것에 초점을 맞춘 잘 쓰여진 컴퓨터 과학 교과서였다.

전반적인 책의 구조는 가능한 빠르게 데이터 분석 문제를 다루고, 처음부터 데이터분석에 관한 실전 예제와 연습문제로 바꾸었다.

2-10 장은 *Think Python* 책과 매우 유사하지만 주요 변경사항이 있다. 숫자 중심의 예제와 연습문제는 데이터 지향 연습으로 대체했다. 주제는 순차적으로 제시되어서 점차적으로 정교한 데이터 분석 솔류션을 구축하도록 했다. `try`,

`except` 같은 주제는 앞으로 가져와서 조건문 장의 일부에 제시했다. 함수는 추상화의 첫 수업에 소개되기 보다는 프로그램 복잡성을 다루는데 필요할 때까지 매우 가볍게 다루었다. 거의 모든 사용자 정의 함수는 4장 밖으로 예제 코드와 연습문제를 제거했다. 단어 "재귀(recursion)"[1]는 책의 어디에도 나타나지 않는다.

1장, 11-16장의 모든 콘텐츠는 완전히 새로운 실무 사용과 데이터 분석을 위한 파이썬 간단한 예제에 집중했다. 데이터 분석은 검색과 파싱를 위한 정규 표현식, 사용자 컴퓨터의 작업 자동화, 네트워크 상에서 데이터 가져오기, 데이터로 웹페이지 스크랩핑, 웹서비스 사용하기, XML과 JSON 데이터 파싱, 그리고 SQL(Structured Query Language)을 사용한 데이터베이스 생성 및 사용을 포함한다.

이 모든 변화의 궁극적인 목적은 컴퓨터 과학에서 인포매틱스(informatics)로 전환이고, 설사 전문적인 프로그래머가 되지 않을 지라도 유용한 첫 기술 과목 안으로 의제를 포괄하는 것이다.

이책이 흥미롭고 좀더 탐색하고자 하는 학생은 Allen B. Downey 의 *Think Python* 책을 봐야한다. 두 책간에 많이 겹치는 부분이 있어서, *Think Python*에서 다루는 기술적인 프로그래밍과 알고리즘적 사고에 대한 기술을 빠르게 습득할 것이다. 그리고, 책의 저작 스타일 매우 유사해서, 최소의 노력으로 *Think Python* 을 통해서 빠르게 나아갈 수 있다.

Think Python 저작권자로서, Allen은 GNU 공개 문서 라이센스가 적용된 본인의 책에서 이 책에 적용된 좀더 최근의 크리에이티브 커먼즈 저작자 명시, 동일한 라이선스 적용(CC-BY-SA)변경하도록 허가를 주었다. 공개 문서 라이센스가 GFDL에서 CC-BY-SA(예, 위키피디아) 바뀌는 추세를 따르는 것이다. CC-BY-SA 라이선스를 사용하는 것은 책에 대한 강력한 카피레프트 전통을 유지하면서, 새로운 저자가 재사용해서 자신의 목적에 맞춰 사용하도록 좀더 직접적으로 만드는 것이다. 이 책이 왜 공개 저작물이 미래 교육에 매우 중요하다고 느끼고, 책을 공개 저작권 아래에서 이용가능하게 만든 앞을 내다보는 결정을 내린 Allen B. Downey와 Cambridge University Press에 감사드린다. 저자 노력의 결과에 기뻐하고, 독자는 *모두의* 공동 노력에 즐거워하길 희망한다.

이 책과 관련된 저작권 이슈를 해결하고 처리하는데 인내를 가지고 도움과 안내를 주신 Allen B. Downey와 Lauren Cowles 분께 감사를 표한다.

Charles Severance

[1] 물론 이번 줄은 제외다.

www.dr-chuck.com
Ann Arbor, MI, USA
2013년 9월 9일

Charles Severance는 미시건 대학 정보 학교 부교수다.

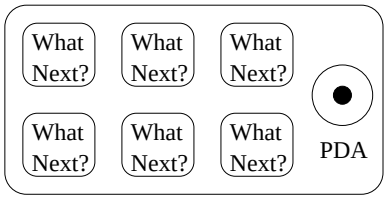
차 례

1 장

왜 프로그래밍을 배워야 하는가?

컴퓨터 프로그램을 만드는 행위(프로그래밍)는 매우 창의적이며 향후 뿌린 것 이상으로 얻을 것이 많다. 프로그램을 만드는 이유는 어려운 자료분석 문제를 해결하려는 것에서부터 다른사람의 문제를 해결해주는데 재미를 느끼는 것까지 다양한 이유가 있다. 이 책을 통해서 *모든 사람*이 어떻게 프로그램을 만드는지를 알고, 프로그램이 어떻게 만드는지를 알게 되면, 새로 습득한 프로그래밍 기술로 하고자 하는 것을 해결할 수 있게 된다.

우리의 일상은 노트북에서부터 스마트폰까지 다양한 종류의 컴퓨터에 둘러싸여 있다. 이러한 컴퓨터를 우리를 위해서 많은 일을 대신해 주는 "개인비서"로 생각한다. 일상생활에서 접하는 컴퓨터 하드웨어는 우리에게 "다음에 무엇을 하면 좋겠습니까?" 라는 질문을 지속적으로 던지게 만들어 졌다.



프로그래머는 운영체제와 하드웨어에 응용 프로그램을 추가했고, 결국 많은 것들을 도와주는 개인 휴대 정보 단말(Persoanl Digital Assistant, PDA)로 진화했다.

사용자 여러분이 컴퓨터에게 "다음 실행해 (do next)"를 컴퓨터가 이해할 수 있는 언어로 지시를 할 수만 있다면, 컴퓨터는 빠르고, 저장소가 커서, 매우 유용하게 사용될 수 있다. 만약 컴퓨터 언어를 알고 있다면, 반복적인 작업을 사람을 대신해서 컴퓨터에 지시할 수 있다. 흥미롭게도, 컴퓨터가 가장 잘 할 수 있는 종

류의 작업들은 종종 사람들이 재미없고, 너무나 지루하다고 생각하는 것이다.

예를 들어, 이번 장의 첫 세 문단을 보고, 가장 많이 나오는 단어를 찾아보고 얼마나 자주 나오는지를 알려주세요. 사람이 몇초내에 단어를 읽고 이해할 수는 있지만, 그 단어가 몇번 나오는지 세는 것은 매우 고생스러운 작업이다. 왜냐하면 사람이 지루하고 반복되는 문제를 해결하는데 적합하지 않기 때문이다. 컴퓨터는 정반대이다. 논문이나 책에서 텍스트를 읽고 이해하는 것은 컴퓨터에게 어렵다. 하지만, 단어를 세고 가장 많이 사용되는 단어를 찾는 것은 컴퓨터에게는 무척이나 쉬운 작업이다.

```
python words.py
Enter file:words.txt
to 16
```

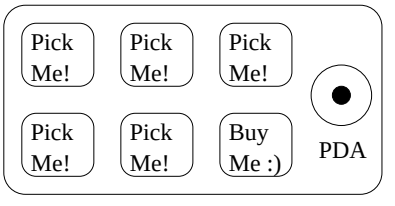
우리의 개인 정보분석 도우미는 이번장의 첫 세 문단에서 단어 "to" 가 가장 많이 사용되었고, 16번 나왔다고 바로 답을 준다.

사람이 잘하지 못하는 점을 컴퓨터가 잘할 수 있다는 사실을 이해하면 왜 "컴퓨터 언어"로 컴퓨터와 대화해야 하는데 능숙해야하는지 알 수 있다. 컴퓨터와 대화할 수 있는 새로운 언어(Python)를 배우게 되면, 지루하고 반복되는 일을 컴퓨터가 처리하고, 사람에게 적합한 일을 하는데 더 많은 시간을 할애할 수 있다. 그래서, 여러분은 직관, 창의성, 창의력을 컴퓨터 파트너와 함께 추진할 수 있다.

제 1 절 창의성과 동기

이책은 직업으로 프로그래밍을 하는 사람을 위해서 저작된 것은 아니지만, 직업적으로 프로그램을 만드는 작업은 개인적으로나 경제적인면에서 꽤 매력적인 일이다. 특히, 유용하며, 심미적이고, 똑똑한 프로그램을 다른 사람이 사용할 수 있도록 만드는 것은 매우 창의적인 활동이다. 다양한 그룹의 프로그래머들이 사용자의 관심과 시선을 차지하기 위해서 경쟁적으로 작성한 다양한 종류의 프로그램이 여러분의 컴퓨터와 개인 휴대 정보 단말기(Personal Digital Assistant, PDA)에 담겨있다. 이렇게 개발된 프로그램은 사용자가 원하는 바를 충족시키고 훌륭한 사용자 경험을 제공하려고 노력한다. 몇몇 상황에서 사용자가 소프트웨어를 골라 구매하게 될 때, 고객의 선택에 대해 프로그래머는 바로 경제적 보상을 받게 된다.

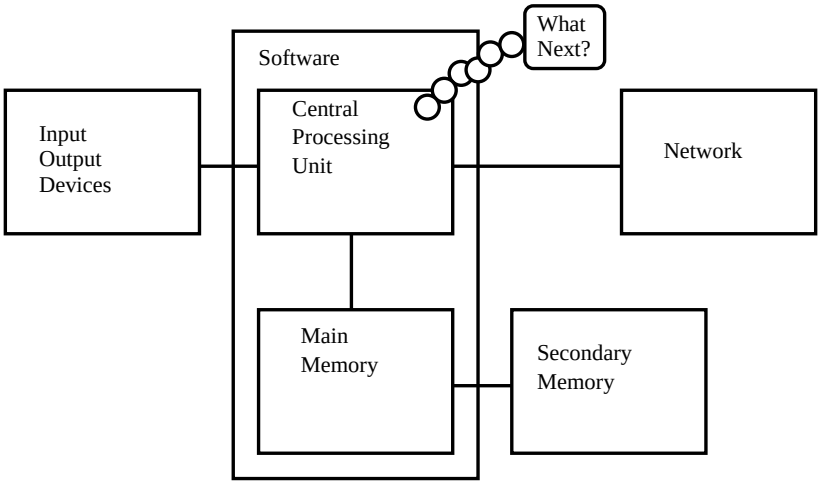
만약 프로그램을 프로그래머 집단의 창의적인 결과물로 생각해본다면, 아마도 다음 그림이 좀더 의미 있는 PDA 컴퓨터로 보일 것이다.



우선은 프로그래을 만드는 주된 동기가 사업을 한다던가 사용자를 기쁘게 한다기보다, 일상생활에서 맞닥뜨리는 자료와 정보를 잘 다뤄 좀더 생산적으로 우리의 삶을 만드는데 초점을 잡아본다. 프로그램을 만들기 시작할 때, 여러분 모두는 프로그래머이면서 동시에 자신이 만든 프로그램의 사용자가 된다. 프로그래머로서 기술을 습득하고 프로그래밍 자체가 좀더 창의적으로 느껴진다면, 여러분은 다른 사람을 위해 프로그램을 개발하게 준비가 된 것이다.

제 2 절 컴퓨터 하드웨어 아키텍처

소프트웨어 개발을 위해 컴퓨터에 지시 명령어를 전달하기 위한 컴퓨터 언어를 학습하기 전에, 컴퓨터가 어떻게 구성되어 있는지 이해할 필요가 있다. 컴퓨터 혹은 핸드폰을 분해해서 안쪽을 살펴보면, 다음과 같은 주요 부품을 확인할 수 있다.



주요 부품의 상위 수준 정의는 다음과 같다.

- **중앙처리장치(Central Processing Unit, CPU):** 다음 무엇을 할까요? (”What is next?”) 명령어를 처리하는 컴퓨터의 주요 부분이다. 만약 컴퓨터 중앙처리장치가 3.0 GHz 라면, 초당 명령어 (다음 무엇을 할까요? What is next?)를 삼백만번 처리할 수 있다는 것이다. CPU 처리속도를 따라서 빠르게 컴퓨터와 어떻게 대화하는지 학습할 것이다.

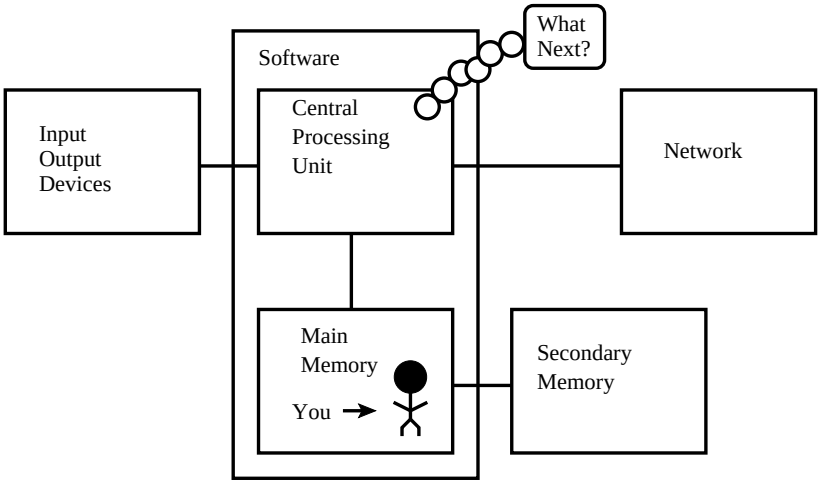
- **주기억장치(Main Memory):** 주기억장치는 중앙처리장치(CPU)가 급하게 명령어를 처리하기 하는데 필요한 정보를 저장하는 용도로 사용된다. 주기억장치는 중앙처리장치만큼이나 빠르다. 그러나 주기억장치에 저장된 정보는 컴퓨터가 꺼지면 자동으로 지워진다.

- **보조 기억장치(Secondary Memory):** 정보를 저장하기 위해 사용되지만, 주기억장치보다 속도는 느리다. 전기가 나갔을 때도 정보를 기억하는 것은 장점이다. 휴대용 USB나 휴대용 MP3 플레이어에 사용되는 USB 플래쉬 메모리나 디스크 드라이브가 여기에 속한다.

- **입출력장치(Input Output Devices):** 간단하게 화면, 키보드, 마우스, 마이크, 스피커, 터치패드가 포함된다. 컴퓨터와 사람이 상호작용하는 모든 방식이 포함된다.
- **네트워크(Network):** 요즘 거의 모든 컴퓨터는 네트워크로 정보를 주고받는 네트워크 연결(Network Connection) 하드웨어가 있다. 네트워크는 항상 "이용가능" 하지 않을지도 모르는 데이터를 저장하고 가져오는 매우 느린 저장소로 볼 수 있다. 그러한 점에서 네트워크는 좀더 느리고, 때때로 신뢰성이 떨어지는 **보조 기억장치(Secondary Memory)**의 한 형태로 볼 수 있다.

주요 부품들이 어떻게 작동하는지에 대한 세세한 사항은 컴퓨터 제조자에게 맡겨져 있지만, 프로그램을 작성할 때 컴퓨터 주요 부품에 대해서 언급되어서, 컴퓨터 전문용어를 습득하고 이해하는 것은 도움이 된다.

프로그래머로서 임무는 자료를 분석하고 문제를 해결하도록, 컴퓨터 자원 각각을 사용하고 조율하는 것이다. 프로그래머로서 대체로 CPU와 "대화"해서 다음 무엇을 실행하라고 지시한다. 때때로 CPU에 주기억장치, 보조기억장치, 네트워크, 혹은 입출력장치도 사용하라고 지시한다.



프로그래머는 컴퓨터의 "다음 무엇을 수행할까요?"에 대한 답을 하는 사람이기도 하다. 하지만, 컴퓨터에 답하기 위해서 5mm 크기로 프로그래머를 컴퓨터에 집어넣고 초당 30억개 명령어로 답을 하게 만드는 것은 매우 불편하다. 그래서, 대신에 미리 컴퓨터에게 수행할 명령문을 작성해야 한다. 이렇게 미리 작성된 명령문 집합을 **프로그램(Program)**이라고 하며, 명령어 집합을 작성하고 명령어 집합이 올바르게 작성될 수 있도록 하는 행위를 **프로그래밍(Programming)**이라고 부른다.

제 3 절 프로그래밍 이해하기

책의 나머지 장을 통해서 책을 읽고 있는 여러분을 프로그래밍 장인으로 인도할 것이다. 종국에는 책을 읽고 있는 여러분 모두 **프로그래머**가 될 것이다. 아마

도 전문적인 프로그래머는 아닐지라도 적어도 자료/정보 분석 문제를 보고 그 문제를 해결할 수 있는 기술을 가지게는 될 것이다.

이런 점에서 프로그래머가 되기 위해서 두 가지 기술이 필요하다.

- 첫째, 파이썬같은 프로그래밍 언어 - 어휘와 문법을 알 필요가 있다. 단어를 새로운 언어에 맞추어 작성할 수 있어야 하며 새로운 언어로 잘 표현된 "문장"으로 어떻게 작성하는지도 알아야 한다.
- 둘째, 스토리(Story)를 말 할 수 있어야 한다. 스토리를 작성할 때, 독자에게 아이디어(idea)를 전달하기 위해서 단어와 문장을 조합합니다. 스토리를 구성할 때 기술적인 면과 예술적인 면이 있는데, 기술적인 면은 쓰기 연습을 반복하고, 피트백을 받아 향상된다. 프로그래밍에서, 우리가 작성하는 프로그램은 "스토리"가 되고, 해결하려고 하는 문제는 "아이디어"에 해당된다.

파이썬과 같은 프로그래밍 언어를 배우게 되면, 자바스크립트나 C++ 같은 두 번째 언어를 배우는 것은 무척이나 쉽다. 새로운 프로그래밍 언어는 매우 다른 어휘와 문법을 갖지만, 문제를 해결하는 기술을 배우면, 다른 모든 프로그래밍 언어를 통해서 동일하게 접근할 수 있습니다.

파이썬 어휘와 문장은 매우 빠르게 학습할 수 있다. 새로운 종류의 문제를 풀기 위해 논리적인 프로그램을 작성하는 것은 더 오래 걸린다. 여러분은 작문을 배우듯이 프로그래밍을 배우게 된다. 프로그래밍을 읽고 설명하는 것으로 시작해서, 간단한 프로그램을 작성하고, 점차적으로 복잡한 프로그램을 작성할 것이다. 어느 순간에 명상에 잠기게 되고, 스스로 패턴이 눈에 들어오게 된다. 그러면, 좀더 자연스럽게 문제를 어떻게 받아들이고, 그 문제를 해결할 수 있는 프로그램을 작성하게 된다. 마지막으로, 그 순간에 도착하게 되면, 프로그래밍은 매우 즐겁고 창의적인 과정이 된다.

파이썬 프로그램의 어휘와 구조로 시작한다. 간단한 예제가 처음으로 언제 프로그램을 읽기 시작했는지를 상기시켜주니 인내심을 가지세요.

제 4 절 단어와 문장

사람 언어와 달리, 파이썬 어휘는 사실 매우 적다. 파이썬 어휘를 예약어(reserved words)로 부른다. 이들 단어는 파이썬에 매우 특별한 의미를 부여한다. 파이썬 프로그램 관점에서 파이썬이 이들 단어를 보게 되면, 파이썬에게는 단 하나의 유일한 의미를 갖는다. 나중에 여러분들이 프로그램을 작성할 때, 자신만의 단어를 작성하는데 이를 **변수(Variable)**라고 한다. 변수 이름을 지을 때 폭넓은 자유를 갖지만, 변수 이름으로 파이썬 예약어를 사용할 수는 없다.

이런 점에서 강아지를 훈련시킬 때 "걸어(walk)", "앉아", "기달려", "가져와" 같은 특별한 어휘를 사용한다. 강아지에게 이와 같은 특별한 예약어를 사용하지 않을 때는, 주인이 특별한 어휘를 사용할 때까지 강아지는 주인을 물끄러미 쳐다보기만 한다. 예를 들어, "더 많은 사람들의 건강을 전반적으로 향상하는

방향으로 동참하여 "걷기(walk)"를 원한다"고 말하면, 강아지가 듣는 것은 "뭐라 뭐라 뭐라 **걷기(walk)** 뭐라"와 같이 들릴 것이다. 왜냐하면 "걸어(walk)"가 강아지 언어에는 예약어[1]이기 때문이다. 이러한 사실이 아마도 개와 고양이사이에는 어떠한 예약어도 존재하지 않는다는 것을 의미할지 모른다.

사람이 파이썬과 대화하는 언어 예약어는 다음과 같다.

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

강아지 사례와 사뭇 다르게 파이썬은 이미 완벽하게 훈련이 되어 있다. 여러분이 "try" 라고 말하면, 매번 "try" 라고 말할 때마다 실패 없이 파이썬은 항시 시도한다.

상기 예약어를 학습하고, 어떻게 잘 사용되는지도 함께 학습할 것이지만, 지금은 파이썬에 말하는 것에 집중할 것이다. 파이썬과 대화하는 것 중 좋은 점은 다음과 같이 인용부호로 감싸 메세지를 던지는 것만으로도 파이썬에 말을 할 수 있다는 것이다.

```
print 'Hello world!'
```

상기 간단한 문장은 파이썬 구문(Syntax)론적으로도 완벽하다. 상기 문장은 예약어 'print'로 시작해서 출력하고자 하는 문자열을 작은 따옴표로 감싸서 올바르게 파이썬에게 전달했다.

제 5 절 파이썬과 대화하기

파이썬으로 우리가 알고 있는 단어를 가지고 간단한 문장을 만들었으니 이제부터는 새로운 언어 기술을 시험하기 위해서 파이썬과 대화를 어떻게 시작하는지 알 필요가 있다.

파이썬과 대화를 시작하기 전에, 파이썬 소프트웨어를 컴퓨터에 설치하고 파이썬을 컴퓨터에서 어떻게 실행하는지를 학습해야 한다. 이번 장에서 다루기에는 너무 구체적이고 자세한 사항이기 때문에 `www.pythonlearn.com`을 참조하는 것을 권한다. 윈도우와 매킨토쉬 시스템 상에서 설치하고 실행하는 방법을 자세한 설치절차와 함께 화면을 캡쳐하여 설명하였다. 설치가 마무리되고 터미널이나 윈도우 명령어 실행창에서 **python**을 타이핑 하게 되면, 파이썬 인터프리터가 인터랙티브 모드로 실행을 시작하고 다음과 같은 것이 화면에 뿌려진다.

```
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

[1] `http://xkcd.com/231/`

파이썬 인터프리터는 >>> 프롬프트를 통해서 여러분에게 요청사항("다음에 파이썬이 무엇을 실행하기를 원합니까?")을 접수받는 방식을 취한다. 파이썬은 여러분과 대화를 나눌 준비가 되었다. 이제 남은 것은 파이썬 언어로 어떻게 말하고 어떻게 파이썬과 대화하는지 아는 것이다.

예를 들어, 여러분이 가장 간단한 파이썬 언어 단어나 문장 조차도 알 수가 없다고 가정하자. 우주 비행사가 저 멀리 떨어진 행성에 착륙해서 행성의 거주민과 대화를 시도할 때 사용하는 간단한 말을 사용해 보자.

```
>>> I come in peace, please take me to your leader
  File "<stdin>", line 1
    I come in peace, please take me to your leader
         ^
SyntaxError: invalid syntax
>>>
```

잘 되는 것 같지 않다. 뭔가 빨리 다른 생각을 내지 않는다면, 행성 거주민은 여러분을 창으로 찌르고, 침으로 바르고, 불위 잘 구워 바베큐로 만들어 저녁으로 먹을 듯 하다.

운 좋게도 기나긴 우주 여행 중 이 책의 복사본을 가지고 와서 다음과 같이 빠르게 타이핑한다고 생각하자.

```
>>> print 'Hello world!'
Hello world!
```

훨씬 좋아보인다. 이제 좀더 커뮤니케이션을 이어갈 수 있을 것으로 보인다.

```
>>> print 'You must be the legendary god that comes from the sky'
You must be the legendary god that comes from the sky
>>> print 'We have been waiting for you for a long time'
We have been waiting for you for a long time
>>> print 'Our legend says you will be very tasty with mustard'
Our legend says you will be very tasty with mustard
>>> print 'We will have a feast tonight unless you say
  File "<stdin>", line 1
    print 'We will have a feast tonight unless you say
                                                      ^
SyntaxError: EOL while scanning string literal
>>>
```

이번 대화는 잠시 동안 잘 진행되다가 여러분이 파이썬 언어로 말하는데 정말 사소한 실수를 저질러 파이썬이 다시 창을 여러분에게 겨눈다.

이 시점에 파이썬은 놀랍도록 복잡하고 강력하며 파이썬과 의사소통을 할 때 사용하는 구문(syntax)은 매우 까다롭다는 것은 알 수 있다. 파이썬은 다른 말로 *안똑똑(Intelligent)*하다. 지금까지 여러분은 자신과 대화를 적절한 구문(syntax)을 사용해서 대화했다.

여러분이 다른 사람이 작성한 프로그램을 사용한다는 것은 파이썬을 사용하는 다른 프로그래머가 파이썬을 중간 매개체로 사용하여 대화한 것으로 볼 수 있다. 프로그램을 만든 저작자가 대화가 어떻게 진행되어져야 하는지를 표현하는

방식이 파이썬이다. 다음 몇 장에 걸쳐서 다른 많은 프로그래머 중의 한명처럼, 파이썬으로 여러분이 작성한 프로그램을 이용하는 사용자와 대화하게 된다.

파이썬 인터프리터와 첫번째 대화를 끝내기 전에, 파이썬 행성의 거주자에게 "안녕히 계세요"를 말하는 적절한 방법도 알아야 한다.

```
>>> good-bye
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'good' is not defined

>>> if you don't mind, I need to leave
  File "<stdin>", line 1
    if you don't mind, I need to leave
             ^
SyntaxError: invalid syntax

>>> quit()
```

상기 처음 두개 시도는 다른 오류 메세지를 출력한다. 두번째 오류는 다른데 이유는 **if**가 예약어이기 때문에 파이썬은 이 예약어를 보고 뭔가 다른 것을 말한다고 생각하지만, 잠시 후 구문이 잘못됐다고 판정하고 오류를 뱉어낸다.

파이썬에 "안녕히 계세요"를 말하는 올바른 방식은 인터렉티브 >>> 프롬프트에서 **quit()**를 입력하는 것이다.

제 6 절 전문용어: 인터프리터와 컴파일러

파이썬은 상대적으로 직접 사람이 읽고 쓸 수도 있고, 컴퓨터도 읽고 처리할 수 있도록 고안된 **하이 레벨(High-level)** 언어이다. 다른 하이 레벨 언어에는 자바, C++, PHP, 루비, 베이직, 펄, 자바스크립트 등 다수가 포함되어 있다. 실제 하드웨어 중앙처리장치(CPU)내에서는 하이레벨 언어를 조금도 이해하지 못한다.

중앙처리장치는 우리가 **기계어(machine-language)**로 부르는 언어만 이해한다. 기계어는 매우 간단하고 솔직히 작성하기에는 매우 귀찮다. 왜냐하면 모두 0과 1로만 표현되기 때문이다.

```
01010001110100100101010000001111
11100110000011101010010101101101
...
```

표면적으로 0과 1로만 되어 있기 때문에 기계어가 간단해 보이지만, 구문은 매우 복잡하고 파이썬보다 훨씬 어렵다. 그래서 매우 소수의 프로그래머만이 기계어로 작성할 수 있다. 대신에, 프로그래머가 파이썬과 자바스크립트 같은 하이 레벨 언어로 작성할 수 있게 다양한 번역기(translator)를 만들었다. 이러한 번역기는 프로그램을 중앙처리장치에 의해서 실제 실행이 가능한 기계어로 변환한다.

기계어는 특정 컴퓨터 하드웨어에 묶여있기 때문에 기계어는 다른 형식의 하드웨어에는 **이식(portable)**되지 않는다. 하이 레벨 언어로 작성된 프로그램은

두 가지 방식으로 이기종의 컴퓨터로 이식이 가능하다. 한 방법은 새로운 하드웨어에 맞게 기계어를 재컴파일(recompile)하는 것이고, 다른 방법은 새로운 하드웨어에 맞는 다른 인터프리터를 이용하는 것이다.

프로그래밍 언어 번역기는 일반적으로 두가지 범주가 있다. (1) 인터프리터 (2) 컴파일러

인터프리터는 프로그래머가 코드를 작성할 때 소스 코드를 읽고, 소스코드를 파싱하고, 즉석에서 명령을 해석한다. 파이썬은 인터프리터다. 따라서, 파이썬을 인터렉트브 모드로 실행할 때, 파이썬 명령문(한 문장)을 작성하면, 파이썬이 즉석에서 처리하고, 사용자가 다른 파이썬 명령어를 입력하도록 준비를 한다.

파이썬 코드의 일부는 나중에 사용될 것이니 파이썬에게 기억하도록 명령한다. 적당한 이름을 골라서 값을 기억시키고, 나중에 그 이름을 호출하여 값을 사용한다. 이러한 목적으로 저장된 값을 참조하는 목적으로 사용되는 표식(label)을 **변수(variable)**라고 한다.

```
>>> x = 6
>>> print x
6
>>> y = x * 7
>>> print y
42
>>>
```

상기 예제에서 파이썬이 값 6 을 기억하고 있다가, 라벨 **x**를 사용하여 나중에 값을 가져오게 만들었다. **print** 예약어를 사용하여 파이썬이 잘 기억하고 있는지를 검증한다. 그리고 **x**를 가져와서 7을 곱하고 새로운 변수 **y**에 값을 집어넣는다. 그리고 **y**에 현재 무슨 값이 저장되었는지 출력하라고 파이썬에게 지시한다.

한줄 한줄 파이썬에 명령어를 입력하고 있지만, 앞쪽 명령문에서 생성된 자료가 뒤쪽 실행 명령문에서 사용될 수 있도록 파이썬은 순차적으로 정렬된 문장으로 처리한다. 방금전 논리적이고 의미있는 순서로 4줄 명령문을 간단하게 한 단락으로 작성했다.

위에서 본 것처럼 파이썬과 인터랙티브하게 대화를 주고받는 것이 **인터프리터**의 본질이다. **컴파일러**가 동작하기 위해서는 먼저 완전한 프로그램을 파일 하나에 담고, 하이 레벨 소스코드를 기계어로 번역하는 과정을 거치고, 마지막으로 나중에 실행되도록 변환된 기계어를 파일에 담는다.

윈도우를 사용한다면, 실행가능한 기계어 프로그램 확장자가 ".exe"(executable), 혹은 ".dll"(dynamically loadable library)임을 확인할 수 있다. 리눅스와 매킨토쉬에는 실행파일을 의미하는 특정 확장자는 없다.

텍스트 편집기에서 실행파일을 열게 되면, 다음과 같이 읽을 수 없는 좀 괴상한 출력결과를 화면상에서 확인한다.

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
```

```
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

기계어를 읽고 쓰는 것은 쉽지 않다. 그래서 C 나 파이썬 같은 하이 레벨 언어로 작성된 프로그램을 기계어로 자동 번역해주는 **인터프리터**와 **컴파일러**가 있다는 것은 멋진 일이다.

컴파일러와, 인터프리터를 논의하는 이 시점에, 파이썬 인터프리터 자체에 대해서 약간 궁금해야 한다. 무슨 언어로 작성되었을까? 컴파일된 언어로 작성되었을까? “python”을 타이핑하게 될 때, 정확하게 무슨 일이 일어나는걸까?

파이썬 인터프리터는 하이 레벨 언어 “C”로 작성되었다. 파이썬 인터프리터 실제 소스 코드를 보려면, `www.python.org` 웹사이트에 가서 여러분의 방식으로 개발할 수 있는 소스코드를 확인할 수 있다. 그래서, 파이썬 그 자체는 프로그램이다. 기계어로 컴파일되어 있어서 파이썬을 여러분의 컴퓨터에 설치(혹은 컴퓨터 제조자가 설치를 대신 해주기도 함)한다는 것은 번역된 파이썬 프로그램 기계어 코드 사본을 여러분 컴퓨터에 복사하는 것에 불과하다. 윈도우 시스템에서 파이썬 실행가능한 기계어 코드는 파일에 다음과 같은 이름을 갖는다.

```
C:\Python27\python.exe
```

지금까지 살펴본 것은 파이썬 프로그래머가 되기 위해서 정말 알 필요가 있는 것 이상이다. 하지만, 때때로 처음에 이런 귀찮은 질문에 바로 답하는 것이 나중에 보상을 한다.

제 7 절 프로그램 작성하기

파이썬 인터프리터에 명령어를 타이핑 하는 것은 파이썬 주요 기능을 알아보는 좋은 방법이지만, 좀더 복잡한 문제를 해결하는데 권하지는 않는다.

프로그램을 작성할 때, 텍스트 편집기를 사용해서 **스크립트(script)**로 불리는 파일에 명령어 집합을 작성한다. 관례로, 파이썬 스크립트 확장자는 `.py`가 된다.

스크립트를 실행하기 위해서, 파이썬 인터프리터에 파일 이름을 넘겨준다. 유니스나 윈도우 명령창에서 `python hello.py`를 입력하게 되면 다음과 같은 결과를 얻는다.

```
csev$ cat hello.py
print 'Hello world!'
csev$ python hello.py
Hello world!
csev$
```

”csev$”은 운영시스템 명령어 프롬프트이고, ”cat hello.py”는 문자열을 출력하는 한줄 파이썬 프로그램을 담고 있는 ”hello.py” 파일을 화면에 출력하라는 명령어입니다.

인터랙트브 모드에서 파이썬 코드 입력하는 방식 대신에 파이썬 인터프리터를 호출해서 ”hello.py” 파일로부터 소스코드를 읽도록 지시합니다.

이 새로운 방식은 파이썬 프로그램을 끝마치기 위해 **quit()**를 사용할 필요가 없다는 점에서 편리합니다. 파일에서 소스코드를 읽을 때, 파일 끝까지 읽게 되면 자동으로 파이썬이 종료됩니다.

제 8 절 프로그램이란 무엇인가?

프로그램(Program)의 가장 본질적인 정의는 특정 작업을 수행할 수 있도록 조작된 일련의 파이썬 문장의 집합이다. 가장 간단한 **hello.py** 스크립트도 프로그램이다. 한줄의 프로그램이 특별히 유익하고 쓸모가 있는 것은 아니지만 엄격한 의미에서 파이썬 프로그램이 맞다.

프로그램을 이해하는 가장 쉬운 방법은 프로그램이 해결하려고 만들어진 문제를 먼저 생각해보고 나서, 그 문제를 풀어가는 프로그램을 살펴보는 것이다.

예를 들어, 페이스북에 게시된 일련의 글에서 가장 자주 사용된 단어에 관심을 가지고 소셜 컴퓨팅 연구를 한다고 생각해 봅시다. 페이스북에 게시된 글들을 쭉 출력해서 가장 흔한 단어를 찾으려고 열심히 들여다 볼 것이지만, 매우 오래 걸리고 실수하기도 쉽다. 하지만 파이썬 프로그램을 작성해서 빨리 정확하게 작업을 마무리한다면 똑똑하게 주말을 재미나게 보낼 수 있다.

예를 들어 자동차(car)와 광대(clown)에 관한 다음 텍스트에서, 가장 많이 나오는 단어가 무엇이며 몇번 나왔는지 세어보세요.

```
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car
```

그리고 나서, 몇 백만줄의 텍스트를 보고서 동일한 일을 한다고 상상해 보자. 솔직히 수작업으로 단어를 세는 것보다 파이썬을 배워 프로그램을 작성하는 것이 훨씬 빠를 것이다.

더 좋은 소식은 이미 텍스트 파일에서 가장 자주 나오는 단어를 찾아내는 간단한 프로그램을 개발했다. 저자가 직접 작성했고, 시험까지 했다. 바로 사용을 할 수 있도록 준비했기 때문에 여러분의 수고도 덜 수 있다.

```
name = raw_input('Enter file:')
handle = open(name, 'r')
text = handle.read()
words = text.split()
counts = dict()

for word in words:
   counts[word] = counts.get(word,0) + 1

bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print bigword, bigcount
```

상기 프로그램을 사용하려고 파이썬을 공부할 필요도 없다. 10장에 걸쳐서 멋진 파이썬 프로그램을 만드는 방법을 배우게 될 것이다. 지금 여러분은 단순 사용자로서 단순히 상기 프로그램을 사용하게 되면, 프로그램의 영리함과 동시에 얼마나 많은 수작업 노력을 줄일 수 있는지 감탄할 것이다. 단순하게 코드를 타이핑해서 **words.py** 파일로 저장하고 실행을 하거나, `http://www.pythonlearn.com/code/`에서 소스 코드를 다운받아 실행하면 된다.

파이썬과 파이썬 언어가 어떻게 여러분(사용자)과 저자(프로그래머)사이에서 중개자 역할을 훌륭히 수행하고 있는지를 보여주는 좋은 사례다. 컴퓨터에 파이썬을 설치한 누구나 사용할 수 있는 공통의 언어로 유용한 명령 순서(즉, 프로그램)를 우리가 주고받을 수 있는 방식이 파이썬이다. 그래서 누구도 *파이썬과* 직접 의사소통하지 않고 *파이썬을 통해서* 서로 의사소통한다.

제 9 절 프로그램 구성요소

다음 몇장에 걸쳐서 파이썬 어휘, 문장구조, 문단구조, 스토리 구조에 대해서 학습할 것이다. 파이썬의 강력한 역량에 대해서 배울 것이고, 유용한 프로그램을 작성하기 위해서 파이썬의 역량을 어떻게 조합할지도 학습할 것이다.

프로그램을 작성하기 위해서 사용하는 개념적인 하위 레벨(low-level) 패턴이 몇 가지 있다. 파이썬 프로그램을 위해서 만들어졌다기 보다는 기계어부터 하이 레벨(high-level) 언어에 이르기까지 모든 언어에도 공통된 사항이기도 하다.

입력: 컴퓨터 바깥 세계에서 데이터를 가져온다. 파일로부터 데이터를 읽을 수도 있고, 마이크나 GPS 같은 센서에서 데이터를 입력받을 수도 있다. 상기 초기 프로그램에서 입력값은 키보드를 사용하여 사용자가 데이터를 입력한 것이다.

출력: 화면에 프로그램 결과값을 출력주거나 파일에 저장한다. 혹은 음악을 연주하거나 텍스트를 읽어 스피커 같은 장치에 데이터를 내보낸다.

순차 실행: 스크립트에 작성된 순서에 맞춰 한줄 한줄 실행된다.

조건 실행: 조건을 확인하고 명령문을 실행하거나 건너뛴다.

반복 실행: 반복적으로 명령문을 실행한다. 대체로 반복 실행시 변화를 수반한다.

재사용: 한벌의 명령문을 작성하여 이름을 부여하고 저장한다. 필요에 따라 프로그램 이름을 불러 몇번이고 재사용한다.

너무나 간단하게 들리지만, 전혀 간단하지는 않다. 단순히 걸음을 "한 다리를 다른 다리 앞에 놓으세요" 라고 말하는 것 같다. 프로그램을 작성하는 "예술"은 기본 요소를 조합하고 엮어 사용자에게 유용한 무언가를 만드는 것이다.

단어를 세는 프로그램은 상기 프로그램의 기본요소를 하나만 빼고 모두 사용하여 작성되었다.

제 10 절 프로그램이 잘못되면?

처음 파이썬과 대화에서 살펴봤듯이, 파이썬 코드를 명확하게 작성해서 의사소통 해야 한다. 작은 차이 혹은 실수는 여러분이 작성한 프로그램을 파이썬이 들여다보다 조기에 포기하게 만든다.

초보 파이썬 프로그래머는 파이썬이 오류에 대해서는 인정사정 보지 않는다고 생각한다. 파이썬이 모든 사람을 좋아하는 것 같지만, 파이썬은 개인적으로만 사람들을 알고, 분노를 간직하고 있다. 이러한 사실로 인해서 파이썬은 여러분이 완벽하게 작성된 프로그램을 받아서 "잘 맞지 않는군요"라고 거절하여 고통을 준다.

```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                       ^
SyntaxError: invalid syntax
>>> primt 'Hello world'
  File "<stdin>", line 1
    primt 'Hello world'
                      ^
SyntaxError: invalid syntax
>>> I hate you Python!
  File "<stdin>", line 1
    I hate you Python!
          ^
SyntaxError: invalid syntax
>>> if you come out of there, I would teach you a lesson
  File "<stdin>", line 1
    if you come out of there, I would teach you a lesson
               ^
SyntaxError: invalid syntax
>>>
```

파이썬과 다퉈봐야 얻을 것은 없어요. 파이썬은 도구고 감정이 없다. 여러분이 필요로 할 때마다 여러분에게 봉사하고 기쁨을 주기 위해서 존재할 뿐이다. 오류 메세지가 심하게 들릴지는 모르지만 단지 파이썬이 도와달라는 요청일 뿐이다. 입력한 것을 쭉 읽어 보고 여러분이 입력한 것을 이해할 수 없다고만 말할 뿐이다.

파이썬은 어떤 면에서 강아지와 닮았다. 맹목적으로 여러분을 사랑하고, 강아지와 마찬가지로 몇몇 단어만 이해하며, 웃는 표정(>>> 명령 프롬프트)으로 여러분이 파이썬이 이해하는 무언가를 말하기만을 기다린다. 파이썬이 "SyntaxError: invalid syntax"을 뱉어낼 때는, 마치 강아지가 꼬리를 흔들면서 "뭔가 말씀하시는 것 같은데요... 주인님 말씀을 이해하지 못하겠어요, 다시 말씀해 주세요(>>>)" 말하는 것과 같다.

여러분이 작성한 프로그램이 점점 유용해지고 복잡해짐에 따라 3가지 유형의 오류와 마주친다.

구문 오류(Syntax Error): 첫번째 마주치는 오류로 고치기 가장 쉽습니다. 구문 오류는 파이썬 문법에 맞지 않는다는 것을 의미한다. 파이썬은 구문오류

가 발생한 줄을 찾아 정확한 위치를 알려준다. 하지만, 파이썬이 제시하는 오류가 그 이전 프로그램 부문에서 발생했을 수도 있기 때문에 파이썬이 제시하는 곳 뿐만 아니라 그 앞쪽도 살펴볼 필요가 있다. 따라서 구문 오류로 파이썬이 지칭하는 행과 문자는 오류를 고치기 위한 시작점으로 의미가 있다.

논리 오류(Logic Error): 논리 오류의 경우 프로그램 구문은 완벽하지만 명령어 실행 순서에 실수가 있거나 혹은 문장이 서로 연관되는 방식에 오류가 있는 것이다. 논리 오류의 예를 들어보자. "물병에서 한모금 마시고, 가방에 넣고, 도서관으로 걸어가서, 물병을 닫는다"

의미론적 오류(Semantic Error): 의미론적 오류는 구문론적으로 완벽하고 올바른 순서로 프로그램의 명령문이 작성되었지만 단순하게 프로그램에 오류가 있다. 프로그램은 완벽하게 작동하지만 여러분이 *의도한 바*를 수행하지는 못한다. 간단한 예로 여러분이 식당으로 가는 방향을 알려주고 있다. " ... 주유소 사거리에 도착했을 때, 왼쪽으로 돌아 1.6km 쭉 가면 왼쪽편에 빨간색 빌딩에 식당이 있습니다." 친구가 매우 늦어 전화로 지금 농장에 있고 헛간으로 걸어가고 있는데 식당을 발견할 수 없다고 전화를 합니다. 그러면 여러분은 "주유소에서 왼쪽으로 혹은 오른쪽으로 돈거야?" 말하면, 그 친구는 "말한대로 완벽하게 따라서 갔고, 말한대로 필기까지 했는데, 왼쪽으로 돌아 1.6km 지점에 주요소가 있다고 했어", 그러면 여러분은 "미안해, 내가 가지고 있는 건 구문론적으로는 완벽한데, 슬프게도 사소하지만 탐지되지 않은 의미론적 오류가 있네!" 라고 말할 것이다.

다시 한번 위 세 종류의 오류에 대해서, 파이썬은 단지 여러분이 요청한 것을 충실히 수행하기 위해서 최선을 다합니다.

제 11 절 학습으로의 여정

책을 읽어 가면서 처음에 개념들이 잘 와 닿지 않는다고 기죽을 필요는 없다. 말하는 것을 배울 때, 처음 몇년 동안 웅얼거리는 것은 문제도 아니다. 간단한 어휘에서 간단한 문장으로 옮겨가데 6개월이 걸리고, 문장에서 문단으로 옮겨가는데 5-6년 이상 걸려도 괜찮다. 흥미로운 완전한 짧은 스토리를 자신의 언어로 작성하는데 몇 년이 걸린다.

파이썬을 빨리 배울 수 있도록 다음 몇장에 걸쳐서 모든 정보를 제공한다. 하지만 새로운 언어를 습득하는 것과 마찬가지로 자연스럽게 느껴지기까지 파이썬을 흡수하고 이해하기까지 시간이 걸린다. 큰 그림(Big Picture)을 이루는 작은 조각들을 정의하는 동안에, 큰 그림을 볼 수 있도록 여러 주제를 방문하고, 또 다시 재방문하면서 혼란이 생길 수도 있다. 이 책은 순차 선형적으로 쓰여져서 본 과정을 선형적으로 배워갈 수도 있지만, 비선형적으로 본 교재를 활용하는 것도 괜찮다. 가볍게 앞쪽과 뒷쪽을 넘나들며 책을 읽을 수도 있다. 구체적이고 세세한 점을 완벽하게 이해하지 않고 고급 과정을 가볍게 읽으면서 프로그래밍의 "왜(Why)"에 대해서 더 잘 이해할 수도 있다. 앞에서 배운 것을 다시 리뷰하고 연습문제를 다시 풀면서 지금 난공불락이라 여겼던 어려운 주제를 통해서 사실 더 많은 것을 학습했다는 것을 깨달을 것이다.

대체적으로 처음 프로그래밍 언어를 배울 때는, 마치 망치로 돌을 내리치고, 끌로 깎아내고 하면서 아름다운 조각품을 만들면서 겪게되는 것과 유사한 몇 번의 ” 유레카, 아 하 ” 순간이 있다.

만약 어떤 것이 특별히 힘들다면, 밤새도록 앉아서 노력하는 것은 별로 의미가 없다. 잠시 쉬고, 낮잠을 자고, 간식을 먹고 다른 사람이나 강아지에게 문제를 설명하고 자문을 구한 후에 깨끗한 정신과 눈으로 돌아와서 다시 시도해보라. 단언컨데 이 책에 있는 프로그래밍 개념을 깨우치게 되면, 돌이켜 생각해보면 프로그래밍은 정말 쉽고 멋지다는 것을 알게 될 것이다. 그래서 단순하게 프로그래밍 언어는 정말 시간을 들여서 배울 가치가 있다.

제 12 절 용어사전

버그(bug): 프로그램 오류

중앙처리장치(central processing unit, CPU): 컴퓨터의 심장, 작성한 프로그램을 실행하는 장치, ”CPU” 혹은 프로세서라고 부른다.

컴파일(compile): 나중에 실행을 위해서 하이레벨 언어로 작성된 프로그램을 로우레벨 언어로 번역한다.

하이레벨 언어(high-level language): 사람이 읽고 쓰기 쉽게 설계된 파이썬과 같은 프로그래밍 언어

인터랙티브 모드(interactive mode): 프롬프트에서 명령어나 표현식을 타이핑함으로써 파이썬 인터프리터를 사용하는 방식

해석한다(interpret): 하이레벨 언어로 작성된 프로그램을 한번에 한줄씩 번역해서 실행한다.

로우레벨 언어(low-level language): 컴퓨터가 실행하기 좋게 설계된 프로그래밍 언어, ”기계어 코드”, ”어셈블리 언어”로 불린다.

기계어 코드(machine code): 중앙처리장치에 의해서 바로 실행될 수 있는 가장 낮은 수준의 언어로 된 소프트웨어

주기억장치(main memory): 프로그램과 데이터를 저장한다. 전기가 나가게 되면 주기억장치에 저장된 정보는 사라진다.

파싱(parse): 프로그램을 검사하고 구문론적 구조를 분석한다.

이식성(portability): 하나 이상의 컴퓨터에서 실행될 수 있는 프로그램의 특성

출력문(print statement): 파이썬 인터프리터가 화면에 값을 출력할 수 있게 만드는 명령문

문제해결(problem solving:) 문제를 만들고, 답을 찾고, 답을 표현하는 과정

프로그램(program:) 컴퓨테이션(Computation)을 명세하는 명령어 집합

프롬프트(prompt): 프로그램이 메세지를 출력하고 사용자가 프로그램에 입력하도록 잠시 멈춘 때.

보조 기억장치(secondary memory): 전기가 나갔을 때도 정보를 기억하고 프로그램을 저장하는 저장소. 일반적으로 주기억장치보다 속도가 느리다. USB의 플래쉬 메모리나 디스크 드라이브가 여기에 속한다.

의미론(semantics): 프로그램의 의미

의미론적 오류(semantic error): 프로그래머가 의도한 것과 다른 행동을 하는 프로그램 오류

소스 코드(source code): 하이레벨 언어로 기술된 프로그램

제 13 절 연습문제

Exercise 1.1 컴퓨터 보조기억장치 기능은 무엇입니까?

a) 프로그램의 모든 연산과 로직을 실행한다.
b) 인터넷을 통해 웹페이지를 불러온다.
c) 파워가 없을 때도 정보를 장시간 저장한다.
d) 사용자로부터 입력정보를 받는다.

Exercise 1.2 프로그램은 무엇입니까?

Exercise 1.3 컴파일러와 인터프리터의 차이점을 설명하세요.

Exercise 1.4 기계어 코드는 다음중 어는 것입니까?

a) 파이썬 인터프리터
b) 키보드
c) 파이썬 소스코드 파일
d) 워드 프로세싱 문서

Exercise 1.5 다음 코드에서 잘못된 점을 설명하세요.

```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                       ^
SyntaxError: invalid syntax
>>>
```

Exercise 1.6 다음 파이썬 프로그램이 실행된 후에, 변수 "X"는 어디에 저장됩니까?

```
x = 123
```

a) 중앙처리장치
b) 주메모리

c) 보조메모리
d) 입력장치
e) 출력장치

Exercise 1.7 다음 프로그램에서 출력되는 것은 무엇입니까?

```
x = 43
x = x + 1
print x
```

a) 43
b) 44
c) x + 1
d) 오류, 왜냐하면 x = x + 1 은 수학적으로 불가능하다.

Exercise 1.8 사람의 어느 능력부위를 예제로 사용하여 다음 각각을 설명하세요. (1) 중앙처리장치, (2) 주메모리, (3) 보조메모리, (4) 입력장치 (5) 출력장치 예를 들어 중앙처리장치에 상응하는 사람의 몸 부위는 어디입니까?

Exercise 1.9 구문오류(”Syntax Error”)는 어떻게 고칩니까?

2 장

변수, 표현식, 스테이트먼트 (Statement)

제 1 절 값(Value)과 형(Type)

값(Value)은 문자와 숫자처럼 프로그램이 다루는 가장 기본이 되는 단위이다. 지금까지 살펴본 값은 `1`, `2` 그리고 `'Hello,World!'` 이다.

상기 값은 다른 **형(Type)**에 속하는데, `2`는 정수, `'Hello,World!'` 는 **문자열(String)**에 속하는데, 문자(Letter)를 일련의 열(sequence)의 형태로 되어 있어서 문자열이라고 부른다. 인용부호에 감싸여 있어서, 여러분과 인터프리터는 문자열을 식별할 수 있다.

`print` 문은 정수에도 사용할 수 있다. `python` 명령어를 실행하여 인터프리터를 구동시키자.

```
python
>>> print 4
4
```

값이 어떤 형인지 확신을 못한다면, 인터프리터가 알려준다.

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

놀랍지도 않게, strings은 `str` 형식이고, 정수는 `int` 형식이다. 다소 명백하지는 않지만, 소숫점을 가진 숫자는 `float` 형식이다. 왜냐하면 이들 숫자가 **부동소수점** 형식으로 표현되기 때문이다.

```
>>> type(3.2)
<type 'float'>
```

`'17'`, `'3.2'` 같은 값은 어떨가? 숫자처럼 보이지만 문자열처럼 인용부호에 감싸여 있다.

```
>>> type('17')
<type 'str'>
>>> type('3.2')
<type 'str'>
```

'17', '3.2' 은 문자열이다.

1,000,000 처럼 아주 큰 정수를 입력할 때, 세자리 숫자마다 콤마(,)를 사용하고 싶을 것이다. 하지만, 파이썬에서 적법하게 정수를 표현하는 것은 아니지만 문법적으로는 적합하다.

```
>>> print 1,000,000
1 0 0
```

하지만, 파이썬 실행 결과는 우리가 기대했던 것이 아니다. 파이썬에서는 1,000,000 을 콤마(’,’)로 구분된 정수로 인식한다. 따라서 사이 사이 공백을 넣어 출력했다.

이 사례가 여러분이 처음 경험하게 되는 의미론적 오류(semantic error)다. 코드가 에러 메세지 없이 실행이되지만, ”올바른(right)” 작동을 하는 것은 아니다.

제 2 절 변수(Variable)

프로그래밍 언어의 가장 강력한 기능 중의 하나는 변수를 다룰 수 있는 능력이다. **변수(Variable)**는 값을 참조하는 이름이다.

할당문(Assignment statement)는 새로운 변수를 생성하고 값을 변수에 할당한다.

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

상기 예제는 세가지 할당사례를 보여준다. 첫 번째 할당 예제는 message 변수에 문자열을 할당한다. 두 번째 예제는 변수 n에 정수 17을 할당한다. 세 번째 예제는 pi변수에 π 근사값을 할당한다.

변수 값을 출력하기 위해서 print문을 사용한다.

```
>>> print n
17
>>> print pi
3.14159265359
```

변수의 형(type)은 변수가 참조하는 값의 형이다.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

제 3 절 변수명(Variable name)과 예약어(keywords)

대체로 프로그래머는 의미있는 변수명을 고른다. 프로그래머는 변수가 사용되는 것에 대해 문서화도 한다.

변수명은 임의로 길 수 있다. 변수명은 문자와 숫자를 포함할 수 있지만, 문자로 변수명을 시작해야 한다. 첫 변수명을 대문자로 사용해도 되지만 소문자로 변수명을 시작하는 것도 좋은 생각이다. (후에 왜 그런지 보게 될 것이다.)

변수명에 밑줄(underscore character, _)이 들어갈 수 있다. 종종 `my_name` 혹은 `airspeed_of_unladen_swallow` 처럼 밑줄은 여러 단어와 함께 사용된다. 변수명을 밑줄로 시작해서 작성할 수 있지만, 다른 사용자가 사용할 라이브러리를 작성하는 경우가 아니라면, 일반적으로 밑줄로 시작하는 변수명은 피한다.

변수명을 적합하게 작성하지 못하다면, 구문 오류가 발생한다.

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` 변수명은 문자로 시작하지 않아서 적합하지 않다. `more@`은 특수문자 (`@`)를 변수명에 포함해서 적합하지 않다. 하지만, `class` 변수명은 뭐가 잘못된 것일까?

구문 오류 이유는 `class`가 파이썬의 예약어 중의 하나라고 밝혀졌다. 인터프리터가 예약어를 사용하여 프로그램 구조를 파악하기 위해서 사용하지만, 변수명으로는 사용할 수 없다.

파이썬에는 31개 키워드[1]가 예약어로 이미 사용중에 있다.

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

상기 예약어 목록을 주머니에 넣고 잘 가지고 다니고 싶을 것이다. 만약 인터프리터가 변수명 중 하나에 대해 불평을 하지만 이유를 모르는 경우, 예약어 목록에 변수명이 있는지 확인해 보세요.

제 4 절 문장(Statement)

문장(statement)은 파이썬 인터프리터가 실행하는 코드 단위다. 지금까지 print, assignment 두 종류의 문장을 살펴봤습니다.

[1] Python 3.0 에서 `exec` 은 더 이상 예약어가 아니지만, `nonlocal` 은 여전히 예약어다.

인터랙트브 모드에서 문장을 입력하면, 인터프리터는 문장을 실행하고, 만약 출력할 것이 있다면 결과를 화면에 출력합니다.

스크립트는 보통 여러줄의 문장으로 구성됩니다. 하나 이상의 문장이 있다면, 스트테이트먼트가 실행되며서 결과가 한번에 하나씩 나타납니다.

예를 들어, 다음의 스크립트를 생각해 봅시다.

```
print 1
x = 2
print x
```

상기 스크립트는 다음 결과를 출력합니다.

```
1
2
```

할당 문장(x=2)은 결과를 출력하지 않습니다.

제 5 절 연산자(Operator)와 피연산자(Operands)

연산자(Operators)는 덧셈, 곱셈 같은 계산(Computation)을 표현하는 특별한 기호입니다. 연산가자 적용되는 값을 **피연산자(operands)**라고 합니다.

다음의 예제에서 보듯이, +, -, *, /, ** 연산자는 덧셈, 뺄셈, 곱셈, 나눗셈, 지수승을 수행합니다.

```
20+32   hour-1   hour*60+minute   minute/60   5**2   (5+9)*(15-7)
```

나눗셈 연산자는 여러분이 기대하는 것을 수행하지 않을 수도 있습니다.

```
>>> minute = 59
>>> minute/60
0
```

minute 값은 59, 보통 59를 60으로 나누면 0 대신에 0.98333 입니다. 이런 차이가 발생하는 이유는 파이썬이 **소수점 이하 버림 나눗셈**[2]을 하기 때문입니다.

두 개의 피연산자가 정수이면, 결과도 정수입니다. 부동 소수점 나눗셈은 소수점 이하를 절사합니다. 그래서 예제에서 소수점 이하 잘라버려 0 이 됩니다.

만약 두 개의 피연산자 중 하나가 부동 소수점 수이면, 파이썬은 부동 소수점 나눗셈을 수행하고 결과는 부동 소수점형 값이 된다.

```
>>> minute/60.0
0.98333333333333328
```

[2]파이썬 3.0 에서 이 나눗셈 값은 소수점입니다. 파이썬 3.0 에 도입된 새로운 연산자//는 정수형 나눗셈을 수행합니다.

제 6 절 표현식(Expression)

표현식 (expression)은 값, 변수, 연산자 조합니다. 값은 자체로 표현식이고, 변수도 동일하다. 따라서 다음 표현식은 모두 적합하다. (변수 x는 사전에 어떤 값이 할당되었다고 가정한다.)

```
17
x
x + 17
```

인터랙티브 모드에서 표현식을 입력하면, 인터프리터는 표현식을 **평가(evaluate)**하고 값을 표시한다.

```
>>> 1 + 1
2
```

하지만, 스크립트에서는 표현식 자체로 어떠한 것도 수행하지는 않는다. 초심자에게 혼란스러운 점이다.

Exercise 2.1 파이썬 인터프리터에 다음 문장을 입력하고 결과를 보세요.

```
5
x = 5
x + 1
```

제 7 절 연산자 적용 우선순위 (Order of Operations)

1개 이상의 연산자가 표현식에 등장할 때, 연산자 평가 순서는 **우선순위 규칙 (rules of precedence)**에 따른다. 수학 연산자에 대해서 파이썬은 수학적 관례를 동일하게 따른다. 영어 두문어 **PEMDAS**는 기억하기 좋은 방식이다.

- **괄호(Parentheses)**는 가장 높은 순위를 가지고 여러분이 원하는 순위에 맞춰 실행할 때 사용한다. 괄호내의 식이 먼저 실행되기 때문에 `2 * (3-1)`은 4가 정답이고, `(1+1)**(5-2)`는 8이다. 괄호를 사용하여 표현식을 좀더 읽기 쉽게 하려고 사용하기도 한다. `(minute * 100) / 60` 는 실행순서가 결과값에 영향을 주지 않지만 가독성이 상대적으로 더 좋다.
- **지수승(Exponentiation)**이 다음으로 높은 우선순위를 가진다. 그래서 `2**1+1`는 4가 아니라 3이고, `3*1**3`는 27이 아니고 3이다.
- **곱셈(Multiplication)과 나눗셈(Division)**은 동일한 우선순위를 가지지만, 덧셈(Addition), 뺄셈(Substraction)보다 높은 우선 순위를 가진다. 덧셈과 뺄샘은 같은 실행 우선순위를 갖는다. `2*3-1`는 4가 아니고 5이고, `6+4/2`는 5가 아니라 8이다.
- 같은 실행 순위를 갖는 연산자는 왼쪽에서부터 오른쪽으로 실행된다. `5-3-1` 표현식은 3이 아니고 1이다. 왜냐하면 `5-3`이 먼저 실행되고 나서 `2`에서 `1`을 빼기 때문이다.

여러분이 의도한 순서대로 연산이 수행될 수 있도록, 좀 의심스러운 경우는 항상 괄호를 사용한다.

제 8 절 나머지 연산자 (Modulus Operator)

나머지 연산자(modulus operator)는 정수에 사용하며, 첫번째 피연산자를 두번째 피연산자가 나눌 때 나머지 값이 생성된다. 파이썬에서 나머지 연산자는 퍼센트 기호(%)다. 구문은 다른 연산자와 동일하다.

```
>>> quotient = 7 / 3
>>> print quotient
2
>>> remainder = 7 % 3
>>> print remainder
1
```

7을 3으로 나누면 몫이 2가 되고 나머지가 1이 된다.

나머지 연산자가 놀랍도록 유용다다. 예를 들어 한 숫자를 다른 숫자로 나눌 수 있는지 없는지를 확인할 수도 있다. `x % y` 값이 0 이라면, `x`를 `y`로 나눌 수 있다.

또한, 숫자에서 가장 오른쪽 숫자를 분리하는데도 사용된다. 예를 들어 `x % 10`은 `x`가 10진수인 경우 가장 오른쪽 숫자를 뽑아낼 수 있고, 동일한 방식으로 `x % 100`은 가장 오른쪽 2개 숫자를 뽑아낼 수도 있다.

제 9 절 문자열 연산자 (String Operator)

`+` 연산자는 문자열에도 동작하지만, 수학에서 말하는 덧셈은 아니다. 대신에 문자열 끝과 끝을 붙이는 **연결(concatenation)** 작업을 수행한다. 예를 들어,

```
>>> first = 10
>>> second = 15
>>> print first+second
25
>>> first = '100'
>>> second = '150'
>>> print first + second
100150
```

상기 프로그램 출력은 `100150` 이다.

제 10 절 사용자에게서 입력값 받기

때때로 키보드를 통해서 사용자로부터 변수에 대한 값을 입력받고 싶을 때가 있다. 키보드[3]로부터 입력값을 받는 `raw_input` 이라는 내장(built-in) 함수를 파이썬에서 제공한다. 입력 함수가 호출되면, 파이썬은 실행을 멈추고 사용자가 무언가 입력하기를 기다린다. 사용자가 Return (리턴) 혹은 Enter (엔터) 키를 누르게 되면 프로그램이 다시 실행되고, `raw_input`은 사용자가 입력한 것을 문자열로 반환한다.

[3]파이썬 3.0에서 입력 함수는 `input`으로 명명되었다.

```
>>> input = raw_input()
Some silly stuff
>>> print input
Some silly stuff
```

사용자로부터 입력 받기 전에 프롬프트에서 사용자가 어떤 값을 입력해야 하는지 정보를 제공하는 것도 좋은 생각이다. 입력을 받기 위해 잠시 멈춰있을 때, 사용자에게 표시되도록 raw_input 함수에 문자열을 전달할 수 있다.

```
>>> name = raw_input('What is your name?\n')
What is your name?
Chuck
>>> print name
Chuck
```

프롬프트의 끝에 \n 은 **개행(newline)**을 의미한다. 개행은 줄을 바꾸게 하는 특수 문자다. 이런 이유 때문에 사용자 입력이 프롬프트 밑에 출력된다.

만약 사용자가 정수를 입력하기를 바란다면, int() 함수를 사용하여 반환되는 값을 정수(int)로 형변환한다.

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

하지만, 사용자가 숫자 문자열이 아닌 다른 것을 입력하게 되면 오류가 발생한다.

```
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int()
```

나중에 이런 종류의 오류를 어떻게 다루는지 배울 것이다.

제 11 절 주석

프로그램이 커지고 복잡해짐에 따라 가독성은 떨어진다. 형식 언어(formal language)는 촘촘하고 코드 일부분도 읽기 어렵고 무슨 역할을 왜 수행하는지 이해하기 어렵다.

이런 이유로 프로그램이 무엇을 하는지를 자연어로 프로그램에 노트를 달아두는 것은 좋은 생각이다. 이런 노트를 **주석(Comments)**이라고 하고 # 기호로 시작한다.

```
# 경과한 시간을 퍼센트로 계산
percentage = (minute * 100) / 60
```

상기 사례의 경우, 주석 자체가 한줄이다. 주석을 프로그램의 맨 뒤에 놓을 수도 있다.

```
percentage = (minute * 100) / 60      # 경과한 시간을 퍼센트로 계산
```

뒤의 모든 것은 무시되기 때문에 프로그램에는 아무런 영향이 없다.

명확하지 않은 코드의 기능을 문서화할 때 주석은 가장 유용하게 된다. 프로그램을 읽는 사람이 코드가 *무엇*을 하는지 이해한다고 가정하는 것은 일리가 있다. *왜* 그런지를 이유를 설명하는 것은 더욱 유용하다.

다음의 주석은 코드와 중복되어 쓸모가 없다.

```
v = 5     # assign 5 to v
```

다음의 주석은 코드에 없는 유용한 정보가 있다.

```
v = 5     # velocity in meters/second.
```

좋은 변수명은 주석을 할 필요를 없게 만들지만, 지나치게 긴 변수명은 읽기 어려운 복잡한 표현식이 될 수 있다. 그래서 상충관계(trade-off)가 존재한다.

제 12 절 연상되는 변수명 만들기

변수를 이름 짓는데 단순한 규칙을 지키고 예약어를 피하기만 하다면, 변수이름을 작명할 수 있는 무척이나 많은 경우의 수가 존재한다. 처음에 이렇게 넓은 선택폭이 오히려 프로그램을 읽는 사람이나 프로그램을 작성하는 사람 모두에게 혼란을 줄 수 있다. 예를 들어, 다음의 3개 프로그램은 각 프로그램이 달성하려 하는 관점에서 동일하지만, 여러분이 읽고 이해하는데는 많은 차이점이 있다.

```
a = 35.0
b = 12.50
c = a * b
print c

hours = 35.0
rate = 12.50
pay = hours * rate
print pay

x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print x1q3p9afd
```

파이썬 인터프리터는 상기 3개 프로그램을 *정확하게 동일하게* 바라보지만, 사람은 이들 프로그램을 매우 다르게 보고 이해한다. 사람은 가장 빨리 두 번째 프로그램의 **의도**를 알아차린다. 왜냐하면 각 변수에 무슨 데이터가 저장될지에 관해서, 프로그래머의 **의도**를 반영하는 변수명을 사용했기 때문이다.

현명하게 선택된 변수명을 *연상기호 변수명("mnemonic variable name")*이라고 한다. 연상되기 좋은 영어 단어 "mnemonic"[4] 은 기억을 돕는다는 뜻이다. 왜 변수를 생성했는지 기억하기 좋게 하기 위해서 연상하기 좋은 변수명을 선택한다.

매우 훌륭하게 들리고, 연상하기 좋은 변수명을 만드는게 좋은 아이디어 같지만, 기억하기 좋은 변수명은 초보 프로그래머가 코드를 파싱(parsing)하고 이해하는데 걸림돌이 되기도 한다. 왜냐하면 31개 밖에 되지 않지만 예약어도 기억하지 못하고, 변수명이 때때로 너무 서술적이라 마치 일반적으로 사용하는 언어처럼 보이고 잘 선택된 변수명처럼 보이지 않기 때문이다.

어떤 데이터를 반복하는 다음 파이썬 코드를 살펴보자. 곧 반복 루프를 살펴보겠지만, 다음 코드가 무엇을 의미하는지 알기 위해서 퍼즐을 풀어보자.

```
for word in words:
    print word
```

무엇이 일어나고 있는 것일까? for, word, in 등등 어느 토큰이 예약어일까? 변수명은 무엇일까? 파이썬은 기본적으로 단어의 개념을 이해할까? 초보 프로그래머는 어느 부분 코드가 이 예제와 동일해야만 *하는지* 그리고, 어느 부분 코드가 프로그래머 선택에 의한 것인지 분간하는데 고생을 한다.

다음의 코드는 위의 코드와 동일하다.

```
for slice in pizza:
    print slice
```

초보 프로그래머가 이 코드를 보고 어떤 부분이 파이썬 예약어이고 어느 부분이 프로그래머가 선택한 변수명인지 알기 쉽다. 파이썬이 피자와 피자조각에 대한 근본적인 이해가 없고, 피자는 하나 혹은 여러 조각으로 구성된다는 근본적인 사실을 알지 못한다는 것은 자명하다.

하지만, 작성한 프로그램이 데이터를 읽고 데이터에 있는 단어를 찾는다면 피자(`pizza`)와 피자조각(`slice`)은 연상하기 좋은 변수명이 아니다. 이것을 변수명으로 선핸하게 되면 프로그램의 의미를 왜곡시킬 수 있다.

좀 시간을 보낸 후에 가장 흔한 예약어에 대해서 알게 될 것이고, 이들 예약어가 어느 순간 여러분에게 눈에 띄게 될 것이다.

for `word` **in** `words`**:**
    print `word`

파이썬에서 정의된 코드 일부분(`for`, `in`, `print`, `:`)은 예약어로 굵게 표시되어 있고, 프로그래머가 생성한 변수명(`word`, `words`)는 굵게 표시되어 있지 않다. 대다수 텍스트 편집기는 파이썬 구문을 인지하고 있어서, 파이썬 예약어와 프로그래머가 작성한 변수를 구분하기 위해서 색깔을 다르게 표시한다. 잠시 후에 여러분은 파이썬을 읽고 변수와 예약어를 빠르게 구분할 수 있을 것이다.

[4] `http://en.wikipedia.org/wiki/Mnemonic`.

제 13 절 디버깅(Debugging)

이 지점에서 여러분이 저지르기 쉬운 구문 오류는 `odd~job`, `US$` 같은 특수문자를 포함해서 잘못된 변수명을 생성하는 것과 `class`, `yield`같은 예약어를 변수명으로 사용하는 것이다.

변수명에 공백을 넣는다면, 파이썬은 연산자 없는 두 개의 피연산자로 생각한다.

```
>>> bad name = 5
SyntaxError: invalid syntax
```

구문 오류에 대해서, 오류 메세지는 그다지 도움이 되지 못한다. 가장 흔한 오류 메세지는 `SyntaxError: invalid syntax`, `SyntaxError: invalid token`인데 둘다 그다지 오류에 대한 많은 정보를 주지는 못한다.

여러분이 많이 범하는 실행 오류는 정의 전에 사용("use before def")하는 것으로 변수에 값을 할당하기 전에 변수를 사용할 경우 발생한다. 여러분이 변수명을 잘못 쓸 때도 발생할 수 있다.

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

변수명은 대소문자를 구분한다. 그래서, `LaTeX`는 `latex`와 같지 않다.

이 지점에서 여러분이 범하기 쉬운 의미론적 오류는 연산자 우선 순위일 것이다. 예를 들어 $\frac{1}{2\pi}$를 계산하기 위해서 다음과 같이 프로그램을 작성하게 되면 ...

```
>>> 1.0 / 2.0 * pi
```

나눗셈이 먼저 일어나서 $\pi/2$이 되는데 의도한 것과 같지 않다. 파이썬으로 하여금 여러분이 작성한 의도를 알게할 수는 없다. 그래서 이런 경우 오류 메세지는 없지만, 여러분은 잘못된 답을 얻게 된다.

제 14 절 용어 설명

할당(assignment): 변수에 값을 할당하는 문장

연결(concatenate): 두 개의 피연산자 끝과 끝을 합치는 것

주석(comment): 다른 프로그래머나 소스코드를 읽는 다른 사람을 위한 프로그램 정보로 프로그램의 실행에는 아무런 영향이 없다.

평가(evaluate): 하나의 값을 만들도록 연산을 실행함으로써 표현식을 간단히 하는 것

표현식(expression): 하나의 결과값을 만드는 변수, 연산자, 값의 조합

부동 소수점(floating-point): 소수점을 가진 숫자를 표현하는 형

버림 나눗셈(floor division) 두 숫자를 나누어 소수점이하 부분을 절사하는 연산자

정수(integer): 완전수를 나타내는 형

예약어(keyword): 컴파일러가 프로그램을 파싱하는데 사용하기 위해서 이미 예약된 단어; if, def, while 같은 예약어를 변수명으로 사용할 수 없다.

연상기호(mnemonic): 기억 보조. 변수에 저장된 것을 기억하기데 도움이 되도록 변수에 연상되는 이름을 부여한다.

나머지 연산자(modulus operator): 퍼센트 기호 (`%`)로 표시되고 정수를 가지고 한 숫자를 다른 숫자로 나누었을 때 나머지를 생성하는 연산자

피연산자(operand): 연산자가 연산을 수행하는 값중의 하나

연산자(operator): 덧셈, 곱셈, 문자열 결합 같은 간단한 연산을 표현하는 특별 기호

우선순위 규칙(rules of precedence): 다수의 연산자와 피연산자를 포함한 표현식이 평가되는 실행 순서를 규정한 규칙 집합

문장(statement): 명령이나 액션을 나타내는 코드 부문. 지금까지 assignment, print 문을 보았다.

문자열(string): 일련의 문자를 나타내는 형식

형(type): 값의 범주. 지금까지 여러분이 살펴본 형은 정수 (type `int`), 부동 소수점수 (type `float`), 문자열 (type `str`) 이다.

값(value): 숫자나 문자 같은 프로그램이 다루는 데이터의 기본 단위중 하나

변수(variable): 값을 참조하는 이름

제 15 절 연습문제

Exercise 2.2 `raw_input`을 사용하여 사용자의 이름을 입력받고 환영하는 프로그램을 작성하세요.

```
Enter your name: Chuck
Hello Chuck
```

Exercise 2.3 급여를 지불하기 위해서 사용자로부터 근로시간과 시간당 임금을 계산하는 프로그램을 작성하세요.

```
Enter Hours: 35
Enter Rate: 2.75
Pay: 96.25
```

지금은 급여가 정확하게 소수점 두자리까지 표현되지 않아도 된다. 만약 원하다면, 파이썬 내장 `round` 함수를 사용하여 소수점 아래 두자리까지 반올림하여 작성할 수 있다.

Exercise 2.4 다음 할당 문장을 실행한다고 가정합시다.

```
width = 17
height = 12.0
```

다음 표현식 각각에 대해서, 표현식의 값(value)과 (표현식 값의) 형(type)을 작성하세요.

1. `width/2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 * 5`

정답을 확인하기 위해서 파이썬 인터프리터를 사용하세요.

Exercise 2.5 사용자로부터 섭씨 온도를 입력받아 화씨온도로 변환하고, 변환된 온도를 출력하는 프로그램을 작성하세요.

3 장

조건부 실행

제 1 절 불 표현식(Boolean expressions)

불 표현식(boolean expression)은 참(True) 혹은 거짓(False)를 지닌 표현식이다. 다음 예제는 == 연산자를 사용하여 두 개 피연산자를 비교하여 값이 동일하면 참(True), 그렇지 않으면 거짓(False)을 산출한다.

```
>>> 5 == 5
True
>>> 5 == 6
False
```

참(True)과 거짓(False)은 불(bool) 형(type)에 속하는 특별한 값이지만, 문자열은 아니다.

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

==연산자는 **비교 연산자(comparison operators)** 중 하나이고, 다른 연산자는 다음과 같다.

```
x != y               # x는 y와 값이 같지 않다.
x > y                # x는 y보다 크다.
x < y                # x는 y보다 작다.
x >= y               # x는 y보다 크거나 같다.
x <= y               # x는 y보다 작거나 같다.
x is y               # x는 y와 같다.
x is not y           # x는 y와 개체가 동일하지 않다.
```

상기 연산자가 친숙할지 모르지만, 파이썬 기호는 수학 기호와 다르다. 일반적인 오류로 비교를 해서 동일하다는 의미로 == 연산자 대신에 =를 사용하는 것이다. = 연산자는 할당 연산자이고, ==연산자는 비교 연산자다. =<, => 같은 비교 연산자는 파이썬에는 없다.

제 2 절 논리 연산자

세개 **논리 연산자(logical operators)**: `and`, `or`, `not` 가 있다. 논리 연산자 의미는 영어 의미와 유사하다. 예를 들어,

```
x > 0 and x < 10
```

`x` 가 0 보다 크다. *그리고(and)*, 10 보다 작으면 참이다.

`n%2 == 0 or n%3 == 0`은 두 조건문 중의 하나만 참이 되면, 즉, 숫자가 2 *혹은 (or)* 3으로 나누어지면 참이다.

마지막으로 `not` 연산자는 불 연산 표현식을 부정한다. `x > y` 가 거짓이면, `not (x > y)`은 참이다. 즉, `x`이 `y` 보다 작거나 같으면 참이다.

엄밀히 말해서, 논리 연산자의 두 피연산자는 모두 불 표현식이지만, 파이썬에서 그다지 엄격하지는 않다. 0 이 아닌 임의의 숫자 모두 "참(true)"으로 해석된다.

```
>>> 17 and True
True
```

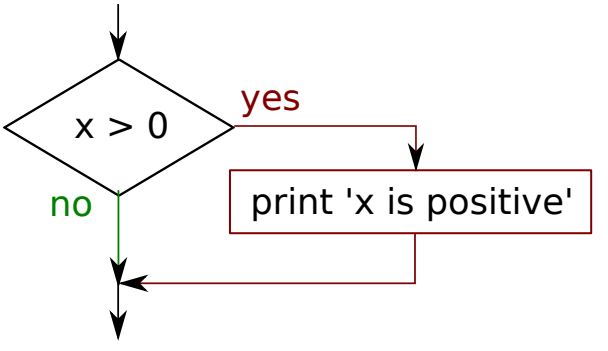
이러한 유연함이 유용할 수 있으나, 혼란을 줄 수도 있으니 유의해서 사용해야 한다. 무슨 일을 하고 있는지 정확하게 알지 못한다면 피하는게 상책이다.

제 3 절 조건문 실행

유용한 프로그램을 작성하기 위해서 거의 항상 조건을 확인하고 조건에 따라 프로그램 실행을 바꿀 수 있어야 한다. **조건문(Conditional statements)**은 그러한 능력을 부여한다. 가장 간단한 형태는 `if` 문이다.

```
if x > 0 :
    print 'x is positive'
```

`if`문 뒤에 불 표현식(boolean expression)을 **조건(condition)**이라고 한다.



만약 조건문이 참이면, 들여쓰기 된 문장이 실행된다. 만약 조건문이 거짓이면, 들여쓰기 된 문장의 실행을 건너뛴다.

if문은 함수 정의, for 반복문과 동일한 구조를 가진다. if문은 콜론(:)으로 끝나는 헤더 머리부문과 들여쓰기된 몸통 블록(block)으로 구성된다. if문처럼 문장이 한 줄 이상에 걸쳐 작성되기 때문에 **복합 문장(compound statements)**이라고 한다.

if문 몸통 부문에 작성되는 실행 문장 숫자에 제한은 없으나 최소한 한 줄은 있어야 한다. 때때로, 몸통 부문에 어떤 문장도 없는 경우가 있다. 아직 코드를 작성하지 않아서 자리만 잡아 놓는 경우로, 아무것도 수행하지 않는 pass문을 사용할 수 있다.

```
if x < 0 :
    pass          # 음수값을 처리 예정!
```

if문을 파이썬 인터프리터에서 타이핑하고 엔터를 치게 되면, 명령 프롬프트가 갈매기 세마리에서 점 세개로 바뀐다. 따라서 다음과 같이 if문 몸통 부분을 작성중에 있다는 것을 나타낸다.

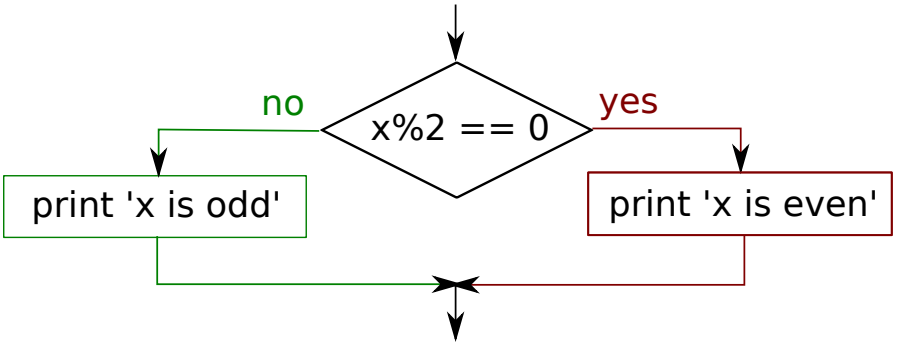
```
>>> x = 3
>>> if x < 10:
...     print 'Small'
...
Small
>>>
```

제 4 절 대안 실행

if문의 두 번째 형태는 **대안 실행(alternative execution)**이다. 대안 실행의 경우 두 가지 경우의 수가 존재하고, 조건이 어느 방향으로 실행할 것인지 결정한다. 구문(Syntax)은 아래와 같다.

```
if x%2 == 0 :
    print 'x is even'
else :
    print 'x is odd'
```

x를 2로 나누었을 때, 0 이되면, x는 짝수이고, 프로그램은 짝수('x is even')라는 결과 메시지를 출력한다. 만약 조건이 거짓이라면, 두 번째 몸통 부문 문장이 실행된다.



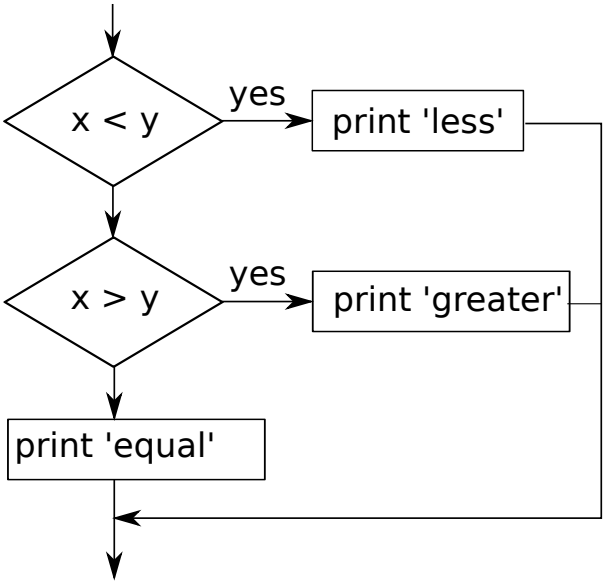
조건은 참 혹은 거짓이어서, 대안 중 하나만 정확하게 실행된다. 대안을 **분기(Branch)**라고도 하는데 이유는 실행 흐름이 분기되기 때문이다.

제 5 절 연쇄 조건문

때때로, 두 가지 이상의 경우의 수가 있으며, 두 가지 이상의 분기가 필요하다. 이와 같은 연산을 표현하는 방식이 **연쇄 조건문(chained conditional)**이다.

```
if x < y:
    print 'x is less than y'
elif x > y:
    print 'x is greater than y'
else:
    print 'x and y are equal'
```

elif는 "else if"의 축약어이다. 이번에도 단 한번의 분기만 실행된다.



elif 문의 갯수에 제한은 없다. else 절이 있다면, 거기서 끝마쳐야 하지만, 연쇄 조건문에 필히 있어야 하는 것은 아니다.

```
if choice == 'a':
    print 'Bad guess'
elif choice == 'b':
    print 'Good guess'
elif choice == 'c':
    print 'Close, but not correct'
```

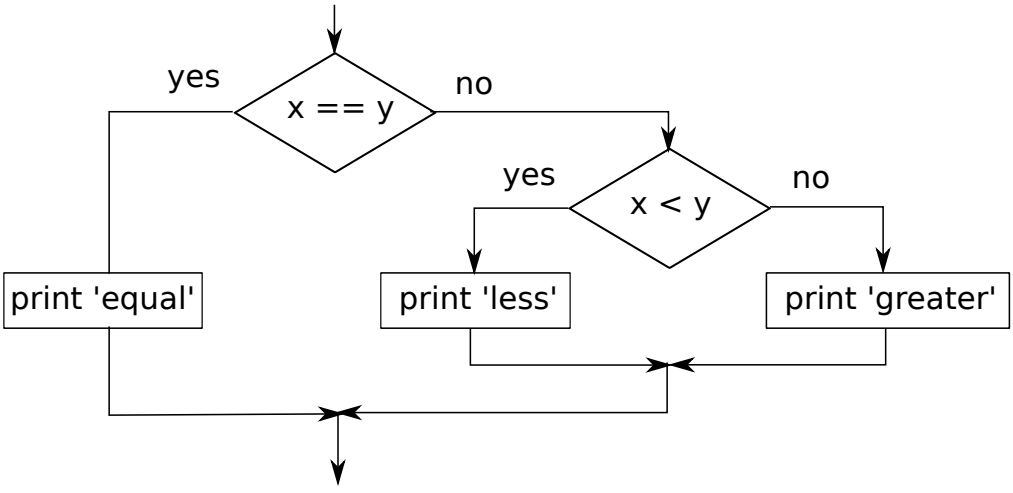
각 조건은 순서대로 점검한다. 만약 첫 번째가 거짓이면, 다음을 점검하고 계속 점검해 나간다. 순서대로 진행 중에 하나의 조건이 참이면, 해당 분기가 수행되고, if문 전체는 종료된다. 설사 하나 이상의 조건이 참이라고 하더라도, 첫 번째 참 분기만 수행된다.

제 6 절 중첩 조건문

하나의 조건문이 조건문 내부에 중첩될 수 있다. 다음과 같이 삼분 예제를 작성할 수 있다.

```
if x == y:
    print 'x and y are equal'
else:
    if x < y:
        print 'x is less than y'
    else:
        print 'x is greater than y'
```

바깥 조건문에는 두 개의 분기가 있다. 첫 분기는 간단한 문장을 담고 있다. 두 번째 분기는 자체가 두 개의 분기를 가지고 있는 또 다른 `if`문을 담고 있다. 자체로 둘다 조건문이지만, 두 분기 모두 간단한 문장이다.



문장을 들여쓰는 것이 구조를 명확히 하지만, **중첩 조건문의 경우 가독성이 급격히 저하된다. 일반적으로, 가능하면 중첩 조건문을 피하는 것을 권장한다.**

논리 연산자를 사용하여 중첩 조건문을 간략히 할 수 있다. 예를 들어, 단일 조건문으로 가지고 앞의 코드를 다음과 같이 재작성할 수 있다.

```
if 0 < x:
    if x < 10:
        print 'x is a positive single-digit number.'
```

`print`**문은 두 개 조건문을 통과될 때만 실행돼서,** `and` **연산자와 동일한 효과를 거둘 수 있다.**

```
if 0 < x and x < 10:
    print 'x is a positive single-digit number.'
```

제 7 절 try와 catch를 활용한 예외 처리

함수 raw_input와 int을 사용하여 앞에서 사용자가 타이핑한 숫자를 읽어 정수로 파싱하는 프로그램 코드를 살펴보았다. 또한 이렇게 코딩하는 것이 얼마나 위험한 것인지도 살펴보았다.

```
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int()
>>>
```

파이썬 인터프리터에서 상기 문장을 실행하면, 인터프리터에서 새로운 프롬프트로 되고, "이런(oops)" 잠시 후에, 다음 문장 실행으로 넘어간다.

하지만, 만약 코드가 파이썬 스크립트로 실행이 되어 오류가 발생하면, 역추적해서 그 지점에서 즉시 멈추게 된다. 다음에 오는 문장은 실행하지 않는다.

화씨 온도를 섭씨 온도로 변환하는 간단한 프로그램이 있다.

```
inp = raw_input('Enter Fahrenheit Temperature:')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print cel
```

이 코드를 실행해서 적절하지 않은 입력값을 넣게 되면, 다소 불친절한 오류 메시지와 함께 간단히 작동을 멈춘다.

```
python fahren.py
Enter Fahrenheit Temperature:72
22.2222222222

python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: invalid literal for float(): fred
```

이런 종류의 예측하거나, 예측하지 못한 오류를 다루기 위해서 파이썬에는 "try / except"로 불리는 조건 실행 구조가 내장되어 있다. try와 except의 기본적인 생각은 일부 명령문에 문제가 있다는 것을 사전에 알고 있고, 만약 그 때문에 오류가 발생하게 된다면 대신 프로그램에 추가해서 명령문을 실행한다는 것이다. except 블록의 문장은 오류가 없다면 실행되지 않는다.

문장 실행에 대해서 파이썬 try, except 기능을 보험으로 생각할 수도 있다.

온도 변환기 프로그램을 다음과 같이 재작성한다.

```
inp = raw_input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
```

```
    print cel
except:
    print 'Please enter a number'
```

파이썬은 try 블록 문장을 우선 실행한다. 만약 모든 것이 순조롭다면, except 블록을 건너뛰고, 다음 코드를 실행한다. 만약 try 블록에서 except이 발생하면, 파이썬은 try 블록에서 빠져 나와 except블록 문장을 수행한다.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.2222222222

python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

try문으로 예외사항을 다루는 것을 예외 처리한다(catching an exception)고 부른다. 예제에서 except 절에서는 단순히 오류 메시지를 출력만 한다. 대체로, 예외 처리를 통해서 오류를 고치거나, 재시작하거나, 최소한 프로그램이 정상적으로 종료될 수 있게 한다.

제 8 절 논리 연산식의 단락(Short circuit) 평가

x >= 2 and (x/y) > 2와 같은 논리 표현식을 파이썬에서 처리할 때, 왼쪽에서부터 오른쪽으로 표현식을 평가한다. and 정의 때문에 x 가 2보다 작다면, x >= 2은 거짓(False)으로, 전체적으로 (x/y) > 2 이 참(True) 혹은 거짓(False) 이냐에 관계없이 거짓(False)이 된다.

나머지 논리 표현식을 평가해도 나아지는 것이 없다고 파이썬이 자동으로 탐지할 때, 평가를 멈추고 나머지 논리 표현식에 대한 연산도 중지한다. 최종값이 이미 결정되었기 때문에 더 이상의 논리 표현식의 평가가 멈출 때, 이를 단락(Short-circuiting) 평가라고 한다.

좋은 점처럼 보일 수 있지만, 단락 행동은 가디언 패턴(guardian pattern)으로 불리는 좀 더 똑똑한 기술로 연계된다. 파이썬 인터프리터의 다음 코드를 살펴보자.

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: integer division or modulo by zero
>>>
```

세번째 연산은 실패하는데 이유는 (x/y) 연산을 평가할 때 y 가 0 이어서 실행 오류 발생한다. 하지만, 두 번째 예제의 경우 실패하지 않는데 이유는 x >= 2 이 거짓(False) 으로, 전체가 거짓(False) 이 되어 단락(Short-circuiting) 평가 규칙에 의해 (x/y) 평가는 실행되지 않아 오류도 발생하지 않는다.

평가 오류를 발생하기 전에 가디언(gardian) 평가식을 전략적으로 배치해서 논리 표현식을 다음과 같이 구성한다.

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

첫 번째 논리 표현식은 x >= 2 이 거짓(False) 이라 and에서 멈춘다. 두 번째 논리 표현식은 x >= 2 이 참(True), y != 0은 거짓(False) 이라 (x/y)까지 갈 필요가 없다. 세 번째 논리 표현식은 (x/y) 연산이 끝난 후에 y != 0 이 수행되어서 오류가 발생한다.

두 번째 표현식에서 y 가 0 이 아닐 때만, (x/y)을 실행하도록 y != 0 이 가디언(gardian) 역할을 수행한다고 말할 수 있다.

제 9 절 디버깅(Debugging)

오류가 발생했을 때, 파이썬 화면에 출력되는 역추적(traceback)에는 상당한 정보가 담겨있다. 하지만,특히 스택에 많은 프레임이 있는 경우 엄청나게 보여 엄두가 나지 않을 수도 있다. 대체로 가장 유용한 정보는 다음과 같은 것이 있다.

- 어떤 종류의 오류인가.
- 어디서 발생했는가.

구문 오류는 대체로 발견하기 쉽지만, 몇 가지는 애매하다. 공백(space)과 탭(tab)의 차이가 눈에 보이지 않아 통상 무시하고 넘어가기 쉽기 때문에 공백 오류를 잡아내기가 까다롭다.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
```

```
    y = 6
    ^
SyntaxError: invalid syntax
```

상기 예제 문제는 두 번째 줄에 한 칸 공백이 들여써서 발생하는 것이다. 하지만, y에 오류 메시지가 있는데 프로그래머를 잘못된 곳으로 인도한다. 대체로 오류 메시지는 문제가 어디에서 발견되었는지를 지칭하지만, 실제 오류는 코드 앞에 종종 선행하는 줄에 있을 수 있다.

동일한 문제가 실행 오류에도 있다. 데시벨(decibels)로 신호 대비 잡음비를 계산한다고 가정하자. 공식은 $SNR_{db} = 10\log_{10}(P_{signal}/P_{noise})$ 이다. 파이썬에서 아래와 같이 작성할 수 있다.

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power / noise_power
decibels = 10 * math.log10(ratio)
print decibels
```

하지만, 실행하게 되면, 다음과 같은 오류 메시지[1]가 발생한다.

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
OverflowError: math range error
```

오류 메지지가 5번째 줄에 있다고 지칭하지만, 잘못된 것은 없다. 실제 오류를 발견하기 위해서, 출력값이 0 인 ratio 값을 print문을 사용해서 출력하는 것이 도움이 된다. 문제는 4번째 줄에 있는데, 왜냐하면 두 정수를 나눌 때 내림 나눗셈을 했기 때문입니다. signal_power 와 noise_power 를 부동 소수점값으로 표현하는게 해결책이다.

대체로, 오류 메시지는 문제가 어디에서 발견되었는지를 알려주지만, 종종 문제의 원인이 어디에서 발생했는지는 알려주지 않는다.

제 10 절 용어 정의

몸통 부문(body): 복합 문장 내부에 일련의 문장문

불 표현식(boolean expression): 참(True) 혹은 거짓(False)의 값을 가지는 표현식

분기(branch): 조건문에서 대안 문장의 한 흐름

연쇄 조건문(chained conditional): 일련의 대안 분기가 있는 조건문

비교 연산자(comparison operator): 피연산자를 ==, !=, >, <, >=, <=로 비교하는 연산자

[1] 파이썬 3.0에서는 오류 메시지가 발생하지 않는다. 정수 피연산자인 경우에도 나눗셈 연산자가 부동 소수점 나눗셈을 수행한다.

조건문(conditional statement): 조건에 따라 명령의 흐름을 제어하는 명령문

조건(condition): 조건문에서 어느 분기를 실행할지 결정하는 불 표현식

복합문(compound statement): 머리부문(head)과 몸통부문(body)으로 구성된 문장. 머리부문은 콜론(:)으로 끝나며, 몸통부문은 머리부문을 기준으로 들여쓰기로 구별된다.

가디언 패턴(guardian pattern): 단락(short circuit) 행동을 잘 이용하도록 논리 표현식을 구성하는 것

논리 연산자(logical operator): 불 표현식을 결합하는 연산자 중의 하나 (`and`, `or`, `not`)

중첩 조건문(nested conditional): 하나의 조건문이 다른 조건문 분기에 나타나는 조건문.

역추적(traceback): 예외 사항이 발생했을 때 실행되고, 출력되는 함수 리스트

단락(short circuit): 나머지 표현식 평가를 할 필요없이 최종 결과를 알기 때문에, 파이썬이 논리 표현식 평가를 진행하는 중간에 평가를 멈출 때.

제 11 절 연습문제

Exercise 3.1 40시간 이상 일할 경우 시급을 1.5배 더 종업원에게 지급하는 봉급계산 프로그램을 다시 작성하세요.

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

Exercise 3.2 `try`, `except`를 사용하여 봉급계산 프로그램을 다시 작성하세요. 숫자가 아닌 입력값을 잘 처리해서 숫자 아닌 입력값이 들어왔을 때 메시지를 출력하고 정상적으로 프로그램을 종료하도록 합니다. 다음이 프로그램 출력 결과를 보여줍니다.

```
Enter Hours: 20
Enter Rate: nine
Error, please enter numeric input

Enter Hours: forty
Error, please enter numeric input
```

Exercise 3.3 0.0과 1.0 사이의 점수를 출력하는 프로그램을 작성하세요. 만약 점수가 범위 밖이면 오류를 출력합니다. 만약 점수가 0.0과 1.0 사이라면, 다음의 테이블에 따라 등급을 출력합니다.

```
Score   Grade
>= 0.9     A
```

```
>= 0.8     B
>= 0.7     C
>= 0.6     D
< 0.6    F

Enter score: 0.95
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

상기 보이는 것처럼 반복적으로 프로그램을 실행해서 다양한 다른 입력값을 테스트해 보세요.

4 장

함수

제 1 절 함수 호출

프로그래밍 문맥에서, 함수(function)는 연산을 수행하는 명명된 일련의 문장이다. 함수를 정의할 때, 이름과 일련의 문장을 명기한다. 나중에, 함수를 이름으로 "호출(call)"한다. 이미 함수 호출(function call)의 예제를 살펴봤다.

```
>>> type(32)
<type 'int'>
```

함수명은 `type`이다. 괄호안의 표현식을 함수의 인자(argument)라고 한다. 인자는 함수 입력으로 함수 내부로 전달되는 값이나 변수이다. 앞선 `type` 함수에 대한 결과는 인자의 형(type)이다.

통상 함수가 인자를 "받아" 결과를 "반환"한다. 결과를 결과값(return value)이라 부른다.

제 2 절 내장(Built-in) 함수

함수를 정의할 필요없이 사용할 수 있는 내장함수가 파이썬에는 많다. 공통의 문제를 해결할 수 있는 함수를 파이썬을 창시자(Guido van Rossum)가 작성해서 누구나 사용할 수 있도록 파이썬에 포함했다.

`max`와 `min` **함수는 리스트 최소값과 최대값을 각각 계산해서 출력한다.**

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

`max` **함수는 문자열의 "가장 큰 문자", 상기 예제에서 "w",** `min` **함수는 최소 문자를, 상기 예제에서는 공백, 출력한다.**

매우 자주 사용되는 또 다른 내장 함수는 얼마나 많은 항목이 있는지 출력하는 len 함수가 있다. 만약 len 함수의 인수가 문자열이면 문자열에 있는 문자 갯수를 반환한다.

```
>>> len('Hello world')
11
>>>
```

이들 함수는 문자열에만 국한된 것이 아니라, 뒷장에서 보듯이 다양한 자료형에 사용된다.

내장함수 이름은 사전에 점유된 예약어로 취급해야 한다. 예를 들어 "max"를 변수명으로 사용하지 말아야 한다.

제 3 절 형(type) 변환 함수

이런 형(type)에서 저런 형(type)으로 값을 변환하는 내장 함수가 파이썬에는 있다. int함수는 임의의 값을 입력받아 변환이 가능하면 정수형으로 변환하고, 그렇지 않으면 오류가 발생한다.

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

int는 부동 소수점 값을 정수로 변환할 수 있지만 소수점 이하를 절사한다.

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

float는 정수와 문자열을 부동 소수점으로 변환한다.

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

마지막으로, str은 인자를 문자열로 변환한다.

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

제 4 절 난수(Random numbers)

동일한 입력을 받을 때, 대부분의 컴퓨터는 매번 동일한 출력값을 생성하기 때문에 결정적(deterministic)이라고 한다. 결정론이 대체로 좋은 것이다. 왜냐하면,

동일한 결과를 얻는데 동일한 계산을 기대하기 때문입니다. 하지만, 어떤 응용 프로그램에 대해서 컴퓨터가 예측불가능하길 바란다. 게임이 좋은 예가 되고, 더 많은 예는 얼마든지 많다.

진실되게 프로그램을 비결정론적으로 만드는 것이 쉽지 않은 것으로 밝혀졌지만, 적어도 비결정론적인 것처럼 보이게 하는 방법은 있다. 의사 난수(pseudo-random numbers)를 생성하는 알고리즘을 사용하는 것이 방법 중의 하나다. 의사 난수는 이미 결정된 연산에 의해서 생성된다는 점에서 진정한 의미의 난수는 아니지만, 이렇게 생성된 숫자만 봐서는 진정한 난수와 구별하는 것은 불가능에 가깝다.

`random` **모듈안에 의사 난수를 생성하는 함수가 있다. (이하 의사 난수 대신 "랜덤(random)"으로 간략히 부르기로 한다.)**

`random` **함수는 0.0 과 1.0 사이 부동 소수점 난수를 반환한다.** `random` **함수는 0.0은 포함하지만 1.0은 포함하지 않는다. 매번** `random` **함수를 호출할 때 마다, 이미 생성된 아주 긴 난수열에서 하나씩 하나씩 뽑아 쓰다. 사례로 다음 반복문을 실행하자.**

```
import random

for i in range(10):
    x = random.random()
    print x
```

상기 프로그램은 0.0 에서 1.0 구간(단, 1.0은 포함하지 않음)에서 10개 난수 리스트를 생성한다.

```
0.301927091705
0.513787075867
0.319470430881
0.285145917252
0.839069045123
0.322027080731
0.550722110248
0.366591677812
0.396981483964
0.838116437404
```

Exercise 4.1 여러분의 컴퓨터에 프로그램을 실행해서, 어떤 난수가 생성되는지 살펴보세요. 한번 이상 프로그램을 실행하여 보고, 어떤 난수가 생성되는지 다시 살펴보세요.

`random` **함수는 난수를 다루는 많은 함수 중의 하나다.** `randint` **함수는 최저** `(low)` **와 최고** `(high)` **매개 변수를 입력받아 최저값** `(low)` **과 최고값** `(high)` **사이 (최저값과, 최저값 모두 포함) 정수를 반환한다.**

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

무작위로 특정 열에서 하나의 요소를 뽑아내기 위해, choice를 사용한다.

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

또한 random 모듈은 정규분포, 지수분포, 감마분포 및 기타 연속형 분포에서 난수를 생성하는 함수도 제공한다.

제 5 절 수학 함수

파이썬은 가장 친숙한 수학 함수를 제공하는 수학 모듈이 있다. 수학 모듈을 사용하기 전에, 수학 모듈 가져오기를 실행한다.

```
>>> import math
```

상기 명령문은 **math** 라는 모듈 객체를 생성한다. 모듈 객체를 출력하면, 모듈 객체에 대한 정보를 얻을 수 있다.

```
>>> print math
<module 'math' from '/usr/lib/python2.5/lib-dynload/math.so'>
```

모듈 객체는 모듈에 정의된 함수와 변수를 담고 있다. 함수 중에서 한 함수에 접근하기 위해서는, 모듈 이름과 함수 이름을 명시해야 하는데, 둘은 점(구두점)으로 구분된다. 이런 형식을 점 표기법(**dot notation**)이라고 부른다.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

첫 예제는 신호-대-소음비의 로그 밑이 **10** 을 계산한다. 수학 모듈이 자연 로그를 log 함수를 호출해서 사용할 수 있도록 제공한다.

두 번째 예제는 라디안의 사인값을 찾는 것이다. 변수의 이름이 힌트를 주는데, sin과 다른 삼각함수(cos, tan 등)는 라디안을 인자로 받는다. 도(**degree**)에서 라디안(**radian**)으로 변환하기 위해서 **360**으로 나누고 2π를 곱한다.

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

math.pi 표현식은 수학 모듈에서 pi 변수를 얻는데, π 값과 비교하여 **15** 자리 수까지 정확하고 근사적으로 수렴한다.

삼각함수를 배웠다면, 앞선 연산 결과를 **2**에 루트를 씌우고 **2**로 나누어 비교 검증한다.

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

제 6 절 신규 함수 추가

지금까지 파이썬 설치 시 함께 있는 함수만 사용했지만 새로운 함수를 추가하는 것도 가능하다. 함수 정의(function definition)는 신규 함수명과 함수가 호출될 때 실행될 일련의 문장을 명세한다. 신규로 함수를 정의하면, 프로그램 실행 중에 반복해서 함수를 재사용할 수 있다.

다음에 예제가 있다.

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'
```

def는 "이것이 함수 정의다"를 표시하는 예약어다. 함수 이름은 print_lyrics 이다. 함수 이름을 명명 규칙은 변수명과 동일하다. 문자, 숫자, 그리고 몇몇 문장 부호는 사용할 수 있지만, 첫 문자가 숫자는 될 수 없다. 함수 이름으로 예약어를 사용할 수 없고, 함수 이름과 동일한 변수명은 피해야 한다.

함수명 뒤 빈 괄호는 이 함수가 어떠한 인자도 갖지 않는다는 것을 나타낸다. 나중에, 입력값으로 인자를 가지는 함수를 작성해 볼 것이다.

함수 정의 첫번째 줄을 머리 부문(헤더, header), 나머지 부문을 몸통 부문(바디, body)라고 부른다. 머리 부문은 콜론(:)으로 끝나고, 몸통 부문은 들여쓰기해야 한다. 파이썬 관례로 들여쓰기는 항상 4칸 공백이다. 몸통 부문에는 제약 없이 문장을 작성할 수 있다.

print문의 문자열은 이중 인용부호로 감싼다. 단일 인용부호나, 이중 인용부호나 차이는 없다. 대부분의 경우 단일 인용부호를 사용하고, 단일 인용부호가 문자열에 나타나는 경우, 이중 인용부호를 사용하여 단일 인용부호가 출력되게 감싼다.

만약 함수 정의를 인터렉티브 모드에서 타이핑을 하면, 함수 정의가 끝나지 않았다는 것을 의미로 생략부호(...)가 출력된다.

```
>>> def print_lyrics():
...     print "I'm a lumberjack, and I'm okay."
...     print 'I sleep all night and I work all day.'
...
```

함수 정의를 끝내기 위해서 빈 줄을 입력한다. (스크립트에서는 반듯이 필요한 것은 아니다.)

함수를 정의하게 되면 동일한 이름의 변수도 생성된다.

```
>>> print print_lyrics
<function print_lyrics at 0xb7e99e9c>
>>> print type(print_lyrics)
<type 'function'>
```

print_lyrics 값은 'function' 형을 가지는 함수 객체(function object)다.

신규 함수를 호출하는 구문은 내장 함수의 경우와 동일하다.

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

함수를 정의하면, 또 다른 함수 내부에서 사용이 가능하다. 예를 들어, 이전 후렴구를 반복하기 위해 `repeat_lyrics` **함수를 작성할 수 있다.**

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

그리고 나서, `repeat_lyrics` **함수를 호출한다.**

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

하지만, 이것이 실제 노래가 불려지는 것은 아니다.

제 7 절 함수 정의와 사용법

앞 절의 코드 조각을 모아서 작성한 전체 프로그램은 다음과 같다.

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

상기 프로그램에는 두개의 함수(`print_lyrics`**,** `repeat_lyrics`**)가 있다. 함수 정의는 다른 문장처럼 수행되지만, 함수 객체를 생성한다는 점에서 차이가 있다. 함수 내부 문장은 함수가 호출되기 전까지 수행되지 않고, 함수 정의는 출력값도 생성하지 않는다.**

예상하듯이, 함수를 실행하기 전에 함수를 생성해야 한다. 다시 말해서, 처음으로 호출되기 전에 함수 정의가 실행되어야 한다.

Exercise 4.2 상기 프로그램의 마지막 줄을 최상단으로 옮겨서 함수 정의 전에 호출되도록 프로그램을 고쳐보세요. 프로그램을 실행서 오류 메시지를 확인하세요.

Exercise 4.3 함수 호출을 맨 마지막으로 옮기고, `repeat_lyrics` 함수 정의 뒤에 `print_lyrics` 함수를 옮기세요. 프로그램을 실행하게 되면 무슨 일이 발생하나요?

제 8 절 실행 흐름

처음으로 함수가 사용되기 전에 정의되었는지를 확인하기 위해서, 명령문 실행 순서를 파악해야 하는데 이를 실행 흐름(flow of execution)이라고 한다.

프로그램 실행은 항상 프로그램 첫 문장부터 시작한다. 명령문은 한번에 하나씩 위에서 아래로 실행된다.

함수 *정의(definitions)*가 프로그램 실행 순서를 바꾸지는 않는다. 하지만, 함수 내부의 문장은 함수가 호출될 때까지 실행이 되지 않는다는 것을 기억하자.

함수 호출은 프로그램 실행 흐름을 우회하는 것과 같다. 다음 문장으로 가기 전에, 실행 흐름은 함수 몸통 부문을 실행하고는 건너 뛰기를 시작한 지점으로 다시 돌아온다.

함수가 또 다른 함수를 호출한다는 것을 기억할 때까지는 매우 간단하게 들린다. 함수 중간에서 프로그램이 또 다른 함수의 문장을 수행할지도 모른다. 하지만, 새로운 함수를 실행하는 중간에 프로그램이 또 다른 함수를 실행할지도 모른다!

다행스럽게도, 파이썬은 프로그램 실행 위치를 정확히 추적한다. 그래서, 함수가 실행을 완료할 때마다, 프로그램을 함수를 호출해서 떠난 지점으로 정확히 되돌려 놓는다. 프로그램이 마지막에 도달했을 때, 프로그램은 종료한다.

이렇게 복잡한 이야기의 교훈은 무엇일까요? 프로그램을 읽을 때, 위에서부터 아래로 읽을 필요는 없다. 때때로, 실행 흐름을 따르는 것이 좀더 이치에 맞는다.

제 9 절 매개 변수(parameter)와 인수(argument)

지금까지 살펴본 몇몇 내장 함수는 인자를 요구한다. 예를 들어, `math.sin` 함수를 호출할 때, 숫자를 인자로 넘겨야 한다. 어떤 함수는 2개 이상의 인수를 받는다. `math.pow` 는 밑과 지수 2개의 인자가 필요하다.

인자는 함수 내부에서 매개 변수(parameters)로 불리는 변수로 할당된다. 하나의 인자를 받는 사용자 정의 함수(user-defined function)가 예제로 있다.

```
def print_twice(bruce):
    print bruce
    print bruce
```

사용자 정의 함수는 인자를 받아 매개변수 `bruce`에 할당한다. 함수가 호출될 때, 매개변수의 값(무엇이든 관계 없이)을 두번 출력합니다.

사용자 정의 함수는 출력 가능한 임의의 값에 작동한다.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
```

```
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

내장함수에 적용되는 동일한 구성 규칙이 사용자 정의 함수에도 적용되어서, print_twice 함수 인자로 표현식 어떤 종류도 가능하다.

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

함수가 호출되기 전에 인자에 대한 평가는 완료되어, 예제에서 'Spam '*4과 math.cos(math.pi)은 단지 1회만 평가된다.

변수도 인자로 사용이 가능하다.

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

인수자 넘기는 변수명(michael)은 매개 변수명(bruce)과 아무런 연관이 없다. 무슨 값이 호출된든지 호출하는 측과 상관이 없다. 여기 print_twice 함수에서는 누구나 bruce라고 부르면 된다.

제 10 절 결과있는 함수(fruitful function)와 빈 함수(void function)

수학 함수와 같은 몇몇 함수는 결과를 만들어 낸다. 좀더 좋은 이름이 없어서, 결과를 만들어 내는 함수를 결과있는 함수(fruitful functions)라고 명명한다. print_twice와 같이 액션을 수행하지만, 결과를 만들어 내지 않는 함수를 빈 함수(void functions)라고 부른다.

결과있는 함수를 호출할 때는 결과값을 가지고 뭔가를 하려고 한다. 예를 들어, 결과값을 변수에 할당하거나, 표현식의 일부로 사용할 수 있다.

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

인터랙티브 모드에서 함수를 호출할 때, 파이썬은 결과를 화면에 출력한다.

```
>>> math.sqrt(5)
2.2360679774997898
```

하지만, 스크립트에서 결과있는 함수를 호출하고 변수에 결과값을 저장하지 않으면 반환되는 결과값은 안개속에 사라져간다!

```
math.sqrt(5)
```

이 스크립트는 5의 제곱근을 계산하지만, 변수에 결과값을 저장하거나, 화면에 출력하지 않아서 그다지 유용하지는 않다.

빈 함수(Void functions)는 화면에 출력하거나 무엇인가 다른 효과를 가지지만, 반환값이 없다. 빈 함수를 사용하여 결과에 변수를 할당하면, None으로 불리는 특별한 값을 얻게 된다.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print result
None
```

None 값은 자신만의 특별한 값을 가지며, 문자열 'None' 과는 같지 않다.

```
>>> print type(None)
<type 'NoneType'>
```

함수에서 결과를 반환하기 위해서, 함수 내부에 return문을 사용한다. 예를 들어, 두 숫자를 더해서 결과를 반환하는 addtwo라는 간단한 함수를 작성할 수 있다.

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print x
```

상기 스크립트가 실행될 때 print 문은 "8"을 출력한다. 왜냐하면, 3과 5를 인수로 받는 addtwo 함수가 호출되기 때문이다. 함수 내부에 매개 변수 a, b는 각각 3, 5이다. addtwo 함수는 두 숫자 덧셈을 수행하고 added라는 로컬 변수에 저장하고, return문을 사용해서 덧셈 결과를 반환하고, x 라는 변수에 할당해서 출력한다.

제 11 절 왜 함수를 사용하는가?

프로그램을 함수로 나누는 고생을 할 가치가 있는지 명확하지 않을 수 있다. 다음에 여기 몇 가지 이유가 있다.

- 문장을 그룹으로 만들어 새로운 함수로 명명하는 것이 프로그램을 읽고, 이해하고, 디버그하기 좋게한다.
- 함수는 반복 코드를 제거해서 프로그램을 작고 콤팩트하게 만든다. 나중에 프로그램에 수정사항이 생기면, 단지 한 곳에서만 수정을 하면 된다.
- 긴 프로그램을 함수로 나누어 작성하는 것은 작은 부분에서 버그를 수정할 수 있게 하고, 이를 조합해서 전체적으로 동작하는 프로그램을 만들 수 있다.

- 잘 설계된 함수는 종종 많은 프로그램에서 유용하게 사용된다. 잘 설계된 프로그램을 작성하고 디버그를 해서 오류가 없이 만들게 되면, 나중에 재사용도 용이하다.

책의 나머지 부분에서 이 개념을 설명하는 함수 정의를 종종 사용한다. "리스트에서 가장 작은 값을 찾아내는 것"과 같이 아이디어를 적절하게 추상화하여 함수를 작성하는 것이 함수를 만들고 사용하는 기술의 일부가 된다. 나중에, 리스트에서 가장 작은 값을 찾아내는 코드를 보여 줄 것입니다. 리스트를 인수로 받아 가장 작은 값을 반환하는 `min` 함수를 작성해서 여러분에게 보여드릴 것이다.

제 12 절 디버깅

텍스트 편집기로 스크립트를 작성한다면 공백과 탭으로 몇번씩 문제에 봉착했을 것입니다. 이런 문제를 피하는 가장 최선의 방식은 절대 탭을 사용하지 말고 공백(스페이스)를 사용하는 것이다. 파이썬을 인식하는 대부분의 텍스트 편집기는 디폴트로 이런 기능을 지원하지만, 몇몇 텍스트 편집기는 이런 기능을 지원하지 않아 탭과 공백 문제를 야기한다.

탭과 공백은 통상 눈에 보이지 않기 때문에 디버그를 어렵게 한다. 자동으로 들여쓰기를 해주는 편집기를 프로그램 작성 시 사용한다.

프로그램을 실행하기 전에 저장하는 것을 잊지 마세요. 몇몇 개발 환경은 자동 저장 기능을 지원하지만 그렇지 않는 것도 있다. 이런 이유 때문에 텍스트 편집기에서 작성한 개발 프로그램과 실행운영하고 있는 프로그램이 같지 않을 수도 있다.

동일하고 잘못된 프로그램을 반복적으로 실행한다면, 디버깅은 오래 걸릴 수 있다.

작성하고 있는 코드와 실행하는 코드가 일치하는지 필히 확인하자. 확신을 하지 못한다면, 프로그램의 첫줄에 `print 'hello'` 을 넣어서 실행해 보자. `hello` 를 보지 못한다면, 작성하고 있는 프로그램과 실행하고 있는 프로그램은 다른 것이다.

제 13 절 용어정의

알고리즘(algorithm): 특정 범주의 문제를 해결하는 일반적인 프로세스

인자(argument): 함수가 호출될 때 함수에 제공되는 값. 이 값은 함수 내부에 상응하는 매개 변수에 할당된다.

몸통 부문(body): 함수 정의 내부에 일련의 문장

구성(composition): 좀더 큰 표현식의 일부분으로 표현식을 사용하거나, 좀더 큰 문장의 일부로서의 문장

결정론적(deterministic): 동일한 입력값이 주어지고 실행될 때마다 동일한 행동을 하는 프로그램에 관련된 것.

점 표기법(dot notation): 점과 함수명으로 모듈명을 명세함으로써 다른 모듈의 함수를 호출하는 구문.

실행 흐름(flow of execution): 프로그램 실행 동안 명령문이 실행되는 순서.

결과있는 함수(fruitful function): 반환값을 가지는 함수.

함수(function): 유용한 연산을 수행하는 이름을 가진 일련의 명령문. 함수는 인수를 가질 수도 갖지 않을 수도 있고, 결과값을 생성할 수도 생성하지 않을 수도 있다.

함수 호출(function call): 함수를 실행하는 명령문. 함수 이름과 인자 리스트로 구성된다.

함수 정의(function definition): 신규 함수를 정의하는 명령문으로 이름, 매개변수, 실행 명령문을 명세한다.

함수 객체(function object): 함수 정의로 생성되는 값. 함수명은 함수 객체를 참조하는 변수다.

머리 부문(header): 함수 정의의 첫번째 줄

가져오기 문(import statement): 모듈 파일을 읽어 모듈 개체를 생성하는 명령문

모듈 개체(module object): `import`문에 의해서 생성된 모듈에 정의된 코드와 데이터에 접근할 수 있는 값

매개 변수(parameter): 인자로 전달된 값을 참조하기 위해 함수 내부에 사용되는 이름

의사 난수(pseudorandom): 난수처럼 보이는 일련의 숫자와 관련되어 있지만, 결정론적 프로그램에 의해 생성된다.

반환 값(return value): 함수의 결과. 함수 호출이 표현식으로 사용된다면, 반환값은 표현식의 값이 된다.

빈 함수(void function): 반환값을 갖지 않는 함수

제 14 절 연습문제

Exercise 4.4 파이썬 "def" 키워드의 목적은 무엇입니까?

a) "다음의 코드는 정말 좋다"라는 의미를 가진 속어
b) 함수의 시작을 표시한다.
c) 다음의 들여쓰기 코드 부문은 나중을 위해 저장되야 된다는 것을 표시한다.
d) b와 c 모두 사실
e) 위 모두 거짓

Exercise 4.5 다음 파이썬 프로그램은 무엇을 출력할까요?

```
def fred():
   print "Zap"

def jane():
   print "ABC"

jane()
fred()
jane()
```

a) Zap ABC jane fred jane
b) Zap ABC Zap
c) ABC Zap jane
d) ABC Zap ABC
e) Zap Zap Zap

Exercise 4.6 프로그램 작성 시 (hours과 rate)을 매개 변수로 갖는 함수 computepay을 생성하여, 초과근무에 대해서는 50% 초과 근무수당을 지급하는 봉급 계산 프로그램을 다시 작성하세요.

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

Exercise 4.7 매개 변수로 점수를 받아 문자열로 등급을 반환하는 computegrade 함수를 사용하여 앞장의 등급 프로그램을 다시 작성하세요.

```
Score   Grade
> 0.9     A
> 0.8     B
> 0.7     C
> 0.6     D
<= 0.6    F

Program Execution:

Enter score: 0.95
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C
```

```
Enter score: 0.5
F
```

반복적으로 프로그램을 실행해서 다양한 다른 입력값을 테스트해 보세요.

5 장

반복(Iteration)

제 1 절 변수 갱신

대입문의 흔한 패턴은 변수를 갱신하는 대입문이다. 변수의 새로운 값은 이전 값에 의존한다.

```
x = x+1
```

상기 예제는 "현재 값 `x`에 1을 더해서 `x`를 새로운 값으로 갱신한다."

만약 존재하지 않는 변수를 갱신하면, 오류가 발생한다. 왜냐하면 `x`에 값을 대입하기 전에 파이썬이 오른쪽을 먼저 평가하기 때문이다.

```
>>> x = x+1
NameError: name 'x' is not defined
```

변수를 갱신하기 전에 간단한 변수 대입으로 통상 먼저 초기화(initialize)한다.

```
>>> x = 0
>>> x = x+1
```

1을 더해서 변수를 갱신하는 것을 증가(increment)라고 하고, 1을 빼서 변수를 갱신하는 것을 감소(decrement)라고 한다.

제 2 절 `while`문

종종 반복적인 작업을 자동화하기 위해서 컴퓨터를 사용한다. 오류 없이 동일하거나 비슷한 작업을 반복하는 일은 컴퓨터가 사람보다 잘한다. 반복이 매우 흔한 일이어서, 파이썬에서 반복 작업을 쉽게 하도록 몇가지 언어적 기능을 제공한다.

파이썬에서 반복의 한 형태가 `while`문이다. 다음은 5 에서부터 거꾸로 세어서 마지막에 "Blastoff(발사)!"를 출력하는 간단한 프로그램이다.

```
n = 5
while n > 0:
    print n
    n = n-1
print 'Blastoff(발사)!'
```

마치 영어를 읽듯이 while을 읽어 내려갈 수 있다. n이 0 보다 큰 동안에 n의 값을 출력하고 n 값에서 1만큼 뺀다. 0 에 도달했을 때, while문을 빠져나가 Blastoff(발사)!"를 화면에 출력한다.

좀더 형식을 갖춰 정리하면, 다음이 while문에 대한 실행 흐름에 대한 정리다.

1. 조건을 평가해서 참(True) 혹은 거짓(False)를 산출한다.
2. 만약 조건이 거짓이면, while문을 빠져나가 다음 문장을 계속 실행한다.
3. 만약 조건이 참이면, 몸통 부문의 문장을 실행하고 다시 처음 1번 단계로 돌아간다.

3번째 단계에서 처음으로 다시 돌아가는 반복을 하기 때문에 이런 종류의 흐름을 루프(loop)이라고 부른다. 매번 루프 몸통 부문을 실행할 때마다, 이것을 반복(iteration)이라고 한다. 상기 루프에 대해서 "5번 반복했다"고 말한다. 즉, 루프 몸통 부문이 5번 수행되었다는 의미가 된다.

루프 몸통 부문은 필히 하나 혹은 그 이상의 변수값을 바꾸어서 종국에는 조건식이 거짓(false)이 되어 루프가 종료되게 만들어야 한다. 매번 루프가 실행될 때마다 상태를 변경하고 언제 루프가 끝날지 제어하는 변수를 반복 변수(iteration variable)라고 한다. 만약 반복 변수가 없다면, 루프는 영원히 반복될 것이고, 결국 무한 루프(infinite loop)에 빠질 것이다.

제 3 절 무한 루프

프로그래머에게 무한한 즐거움의 원천은 아마도 "거품내고, 헹구고, 반복" 이렇게 적혀있는 샴프 사용법 문구가 무한루프라는 것을 알아차릴 때일 것이다. 왜냐하면, 얼마나 많이 루프를 실행해야 하는지 말해주는 반복 변수(iteration variable)가 없어서 무한 반복하기 때문입니다.

숫자를 꺼꾸로 세는 (countdown) 예제는 루프가 끝나는 것을 증명할 수 있다. 왜냐하면 n값이 유한하고, n이 매번 루프를 돌 때마다 작아져서 결국 0 에 도달할 것이기 때문이다. 다른 경우 반복 변수가 전혀 없어서 루프가 명백하게 무한 반복한다.

제 4 절 무한 반복과 break

가끔 몸통 부문을 절반 진행할 때까지 루프를 종료해야하는 시점인지 확신하지 못한다. 이런 경우 의도적으로 무한 루프를 작성하고 break 문을 사용하여 루프를 빠져 나온다.

다음 루프는 명백하게 무한 루프(infinite loop)가 되는데 이유는 while문 논리 표현식이 단순히 논리 상수 참(True)으로 되어 있기 때문이다.

```
n = 10
while True:
    print n,
    n = n - 1
print 'Done!'
```

실수하여 상기 프로그램을 실행한다면, 폭주하는 파이썬 프로세스를 어떻게 멈추는지 빨리 배우거나, 컴퓨터의 전원 버튼이 어디에 있는지 찾아야 할 것이다. 표현식 상수 값이 참(True)이라는 사실로 루프 상단 논리 연산식이 항상 참 값이여서 프로그램이 영원히 혹은 배터리가 모두 소진될 때까지 실행된다.

이것이 역기능 무한 루프라는 것은 사실이지만, 유용한 루프를 작성하기 위해는 이 패턴을 여전히 이용할 것이다. 이를 위해서 루프 몸통 부문에 break문을 사용하여 루프를 빠져나가는 조건에 도달했을 때, 루프를 명시적으로 빠져나갈 수 있도록 주의깊게 코드를 추가해야 한다.

예를 들어, 사용자가 done을 입력하기 전까지 사용자로부터 입력값을 받는다고 가정해서 프로그램 코드를 다음과 같이 작성한다.

```
while True:
    line = raw_input('> ')
    if line == 'done':
        break
    print line
print 'Done!'
```

루프 조건이 항상 참(True)이여서 break문이 호출될 때까지 루프는 반복적으로 실행된다.

매번 프로그램이 꺾쇠 괄호로 사용자에게 명령문을 받을 준비를 한다. 사용자가 done을 타이핑하면, break문이 실행되어 루프를 빠져나온다. 그렇지 않은 경우 프로그램은 사용자가 무엇을 입력하든 메아리처럼 입력한 것을 그대로 출력하고 다시 루프 처음으로 되돌아 간다. 다음 예제로 실행한 결과가 있다.

```
> hello there
hello there
> finished
finished
> done
Done!
```

while 루프를 이와 같은 방식으로 작성하는 것이 흔한데 프로그램 상단에서 뿐만 아니라 루프 어디에서나 조건을 확인할 수 있고 피동적으로 "이벤트가 발생할 때까지 계속 실행" 대신에, 적극적으로 "이벤트가 생겼을 때 중지"로 멈춤 조건을 표현할 수 있다.

제 5 절 continue로 반복 종료

때때로 루프를 반복하는 중간에서 현재 반복을 끝내고, 다음 반복으로 즉시 점프하여 이동하고 싶을 때가 있다. 현재 반복 루프 몸통 부분 전체를 끝내지 않고 다음 반복으로 건너뛰기 위해서 continue문을 사용한다.

사용자가 "done"을 입력할 때까지 입력값을 그대로 복사하여 출력하는 루프 예제가 있다. 하지만 파이썬 주석문처럼 해쉬(#)로 시작하는 줄은 출력하지 않는다.

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print line
print 'Done!'
```

continue문이 추가된 새로운 프로그램을 샘플로 실행했다.

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

해쉬 기호(#)로 시작하는 줄을 제외하고 모든 줄을 출력한다. 왜냐하면, continue문이 실행될 때,현재 반복을 종료하고 while문 처음으로 돌아가서 다음 반복을 실행하게 되어서 print문을 건너뛴다.

제 6 절 for문을 사용한 명확한 루프

때때로, 단어 리스트나, 파일의 줄, 숫자 리스트 같은 사물의 집합에 대해 루프를 반복할 때가 있다. 루프를 반복할 사물 리스트가 있을 때, for문을 사용해서 *확정 루프(definite loop)*를 구성한다.

while문을 *불확정 루프(indefinite loop)*라고 하는데, 왜냐하면 어떤 조건이 거짓(False)가 될 때까지 루프가 단순히 계혹해서 돌기 때문이다. 하지만, for루프는 확정된 항목의 집합에 대해서 루프가 돌게 되어서 집합에 있는 항목만큼만 실행이 된다.

for문이 있고, 루프 몸통 부문으로 구성된다는 점에서 for루프 구문은 while 루프 구문과 비슷하다.

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print 'Happy New Year:', friend
print 'Done!'
```

파이썬 용어로, 변수 friends는 3개의 문자열을 가지는 리스트고, for 루프는 리스트를 하나씩 하나씩 찾아서 리스트에 있는 3개 문자열 각각에 대해 출력을 실행하여 다음 결과를 얻게 된다.

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

for 루프를 영어로 번역하는 것이 while문을 번역하는 것과 같이 직접적이지는 않다. 하지만, 만약 friends를 집합(set)으로 생각한다면 다음과 같다. friends라고 명명된 집합에서 friend 각각에 대해서 한번씩 for 루프 몸통 부문에 있는 문장을 실행하라.

for 루프를 살펴보면, for와 in은 파이썬 예약어이고 friend와 friends는 변수이다.

```
for friend in friends:
    print 'Happy New Year', friend
```

특히, friend는 for 루프의 반복 변수(iteration variable)다. 변수 friend는 루프가 매번 반복할 때마다 변하고, 언제 for 루프가 완료되는지 제어한다. 반복 변수는 friend 변수에 저장된 3개 문자열을 순차적으로 훑고 간다.

제 7 절 루프 패턴

종종 for문과 while문을 사용하여, 리스트 항목, 파일 콘텐츠를 훑어 자료에 있는 가장 큰 값이나 작은 값 같은 것을 찾는다.

for나 while 루프는 일반적으로 다음과 같이 구축된다.

- 루프가 시작하기 전에 하나 혹은 그 이상의 변수를 초기화
- 루프 몸통부분에 각 항목에 대해 연산을 수행하고, 루프 몸통 부분의 변수 상태를 변경
- 루프가 완료되면 결과 변수의 상태 확인

루프 패턴의 개념과 작성을 시연하기 위해서 숫자 리스트를 사용한다.

7.1 계수(counting)와 합산 루프

예를 들어, 리스트의 항목을 세기 위해서 다음과 같이 for 루프를 작성한다.

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print 'Count: ', count
```

루프가 시작하기 전에 변수 count를 0 으로 설정하고, 숫자 목록을 훑어 갈 for 루프를 작성한다. 반복(iteration) 변수는 itervar라고 하고, 루프에서 itervar 을 사용되지 않지만, itervar는 루프를 제어하고 루프 몸통 부문 리스트의 각 값에 대해서 한번만 실행되게 한다.

루프 몸통 부문에 리스트의 각 값에 대해서 변수 count 값에 1을 더한다. 루프가 실행될 때, count 값은 "지금까지" 살펴본 값의 횟수가 된다.

루프가 종료되면, count 값은 총 항목 숫자가 된다. 총 숫자는 루프 맨마지막에 얻어진다. 루프를 구성해서, 루프가 끝났을 때 기대했던 바를 얻었다.

숫자 집합의 갯수를 세는 또 다른 비슷한 루프는 다음과 같다.

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print 'Total: ', total
```

상기 루프에서, 반복 변수(iteration variable)가 사용되었다. 앞선 루프에서처럼 변수 count에 1을 단순히 더하는 대신에, 각 루프가 반복을 수행하는 동안 실제 숫자 (3, 41, 12, 등)를 작업중인 합계에 덧셈을 했다. 변수 total을 생각해보면, total은 "지금까지 값의 총계다." 루프가 시작하기 전에 total은 어떤 값도 살펴본 적이 없어서 0 이다. 루프가 도는 중에는 total은 작업중인 총계가 된다. 루프의 마지막 단계에서 total은 리스트에 있는 모든 값의 총계가 된다.

루프가 실행됨에 따라, total은 각 요소의 합계로 누적된다. 이 방식으로 사용되는 변수를 누산기(accumulator)라고 한다.

계수(counting) 루프나 합산 루프는 특히 실무에서 유용하지는 않다. 왜냐하면 리스트에서 항목의 개수와 총계를 계산하는 len()과 sum()가 각각 내장 함수로 있기 때문이다.

7.2 최대값과 최소값 루프

리스트나 열(sequence)에서 가장 큰 값을 찾기 위해서, 다음과 같이 루프를 작성한다.

```
largest = None
print 'Before:', largest
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print 'Loop:', itervar, largest
print 'Largest:', largest
```

프로그램을 실행하면, 출력은 다음과 같다.

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
```

```
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

변수 largest는 "지금까지 본 가장 큰 수"로 생각할 수 있다. 루프 시작 전에 largest 값은 상수 None이다. None은 "빈(empty)" 변수를 표기하기 위해서 변수에 저장하는 특별한 상수 값이다.

루프 시작 전에 지금까지 본 가장 큰 수는 None이다. 왜냐하면 아직 어떤 값도 보지 않았기 때문이다. 루프가 실행되는 동안에, largest 값이 None이면, 첫 번째 본 값이 지금까지 본 가장 큰 값이 된다. 첫번째 반복에서 itervar는 3 이 되는데 largest 값이 None이여서 즉시, largest값을 3 으로 갱신한다.

첫번째 반복 후에 largest는 더 이상 None이 아니다. itervar > largest인지를 확인하는 복합 논리 표현식의 두 번째 부분은 "지금까지 본" 값 보다 더 큰 값을 찾게 될 때 자동으로 동작한다. "심지어 더 큰" 값을 찾게 되면 변수 largest에 새로운 값으로 대체한다. largest가 3에서 41, 41에서 74로 변경되어 출력되어 나가는 것을 확인할 수 있다.

루프의 끝에서 모든 값을 훑어서 변수 largest는 리스트의 가장 큰 값을 담고 있다.

최소값을 계산하기 위해서는 코드가 매우 유사하지만 작은 변화가 있다.

```
smallest = None
print 'Before:', smallest
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print 'Loop:', itervar, smallest
print 'Smallest:', smallest
```

변수 smallest는 루프 실행 전에, 중에, 완료 후에 "지금까지 본 가장 작은" 값이 된다. 루프 실행이 완료되면, smallest는 리스트의 최소 값을 담게 된다.

계수(counting)과 합산에서와 마찬가지로 파이썬 내장함수 max()와 min()은 이런 루프문 작성을 불필요하게 만든다.

다음은 파이썬 내장 min() 함수의 간략 버전이다.

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

가장 적은 코드로 작성한 함수 버전은 파이썬에 이미 내장된 min 함수와 동등하게 만들기 위해서 모든 print문을 삭제했다.

제 8 절 디버깅

좀더 큰 프로그램을 작성할 때, 좀더 많은 시간을 디버깅에 보내는 자신을 발견할 것이다. 좀더 많은 코드는 버그가 숨을 수 있는 좀더 많은 장소와 오류가 발생할 기회가 있다는 것을 의미한다.

디버깅 시간을 줄이는 한 방법은 " 이분법에 따라 디버깅(debugging by bisection)" 하는 것이다. 예를 들어, 프로그램에 100 줄이 있고 한번에 하나씩 확인한다면, 100 번 단계가 필요하다.

대신에 문제를 반으로 나눈다. 프로그램 정확히 중간이나, 중간부분에서 점검한다. `print`문이나, 검증 효과를 갖는 상응하는 대용물을 넣고 프로그램을 실행한다.

중간지점 점검 결과 잘못 되었다면 문제는 양분한 프로그램 앞부분에 틀림없이 있다. 만약 정확하다면, 문제는 프로그램 뒷부분에 있다.

이와 같은 방식으로 점검하게 되면, 검토 해야하는 코드의 줄수를 절반으로 계속 줄일 수 있다. 단계가 100 번 걸리는 것에 비해 6번 단계 후에 이론적으로 1 혹은 2 줄로 문제 코드의 범위를 좁힐 수 있다.

실무에서, "프로그램의 중간"이 무엇인지는 명확하지 않고, 확인하는 것도 가능하지 않다. 프로그램 코드 라인을 세서 정확히 가운데를 찾는 것은 의미가 없다. 대신에 프로그램 오류가 생길 수 있는 곳과 오류를 확인하기 쉬운 장소를 생각하세요. 점검 지점 앞뒤로 버그가 있을 곳과 동일하게 생각하는 곳을 중간 지점으로 고르세요.

제 9 절 용어정의

누산기(accumulator): 더하거나 결과를 누적하기 위해 루프에서 사용되는 변수

계수(counter): 루프에서 어떤 것이 일어나는 횟수를 기록하는데 사용되는 변수. 카운터를 0 으로 초기화하고, 어떤 것의 "횟수"를 셀 때 카운터를 증가시킨다.

감소(decrement): 변수 값을 감소하여 갱신

초기화(initialize): 갱신될 변수의 값을 초기 값으로 할당

증가(increment): 변수 값을 증가시켜 갱신 (통상 1씩)

무한 루프(infinite loop): 종료 조건이 결코 만족되지 않거나 종료 조건이 없는 루프

반복(iteration): 재귀함수 호출이나 루프를 사용하여 명령문을 반복 실행

제 10 절 연습문제

Exercise 5.1 사용자가 “done”을 입력할 때까지 반복적으로 숫자를 읽는 프로그램을 작성하세요. “done”이 입력되면, 총계, 갯수, 평균을 출력하세요. 만약 숫자가 아닌 다른 것을 입력하게되면, `try`와 `except`를 사용하여 사용자 실수를 탐지해서 오류 메시지를 출력하고 다음 숫자로 건너 뛰게 하세요.

```
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
Enter a number: 7
Enter a number: done
16 3 5.33333333333
```

Exercise 5.2 위에서처럼 숫자 목록을 사용자로부터 입력받는 프로그램을 작성하세요. 평균값 대신에 숫자 목록 최대값과 최소값을 출력하세요.

6 장

문자열

제 1 절 문자열은 순서(sequence)다.

문자열은 여러 문자들의 순서다. 꺾쇠 연산자로 한번에 하나씩 문자에 접근한다.

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

두 번째 문장은 변수 `fruit`에서 1번 위치 문자를 추출하여 변수 `letter`에 대입한다. 꺾쇠 표현식을 인덱스(index)라고 부른다. 인덱스는 순서(sequence)에서 사용자가 어떤 문자를 원하는지 표시한다.

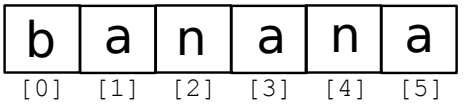
하지만, 여러분이 기대한 것은 아니다.

```
>>> print letter
a
```

대부분의 사람에게 `'banana'`의 첫 분자는 `a`가 아니라 `b`다. 하지만, 파이썬 인텍스는 문자열 처음부터 오프셋(offset)[1] 이다. 첫 글자 오프셋은 0 이다.

```
>>> letter = fruit[0]
>>> print letter
b
```

그래서, `b`가 `'banana'`의 0 번째 문자가 되고 `a`가 첫번째, `n`이 두번째 문자가 된다.



인덱스로 문자와 연산자를 포함하는 어떤 표현식도 사용가능지만, 인덱스 값은 정수여야만 한다. 정수가 아닌 경우 다음과 같은 결과를 얻게 된다.

[1] 컴퓨터에서 어떤 주소로부터 간격을 두고 떨어진 주소와의 거리. 기억 장치가 페이지 혹은 세그먼트 단위로 나누어져 있을 때 하나의 시작 주소로부터 오프셋만큼 떨어진 위치를 나타내는 것이다. [네이버 지식백과] 오프셋 [offset] (IT용어사전, 한국정보통신기술협회)

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

제 2 절 len함수 사용 문자열 길이 구하기

len 함수는 문자열의 문자 갯수를 반환하는 내장함수다.

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

문자열의 가장 마지막 문자를 얻기 위해서, 아래와 같이 시도하려 싶을 것이다.

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

인덱스 오류 (IndexError) **이유는 'banana' 에 6번 인텍스 문자가 없기 때문이다. 0 에서부터 시작했기 때문에 6개 문자는 0 에서부터 5 까지 번호가 매겨졌다. 마지막 문자를 얻기 위해서 length에서 1을 빼야 한다.**

```
>>> last = fruit[length-1]
>>> print last
a
```

대안으로 음의 인텍스를 사용해서 문자열 끝에서 역으로 수를 셀 수 있다. 표현식 fruit[-1]은 마지막 문자를 fruit[-2]는 끝에서 두 번째 등등 활용할 수 있다.

제 3 절 루프를 사용한 문자열 운행법

연산의 많은 경우에 문자열을 한번에 한 문자씩 처리한다. 종종 처음에서 시작해서, 차례로 각 문자를 선택하고, 선택된 문자에 임의 연산을 수행하고, 끝까지 계속한다. 이런 처리 패턴을 운행법(traversal)라고 한다. 운행법을 작성하는 한 방법이 while 루프다.

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print letter
    index = index + 1
```

while 루프가 문자열을 운행하여 문자열을 한줄에 한 글자씩 화면에 출력한다. 루프 조건이 index < len(fruit)이여서, index가 문자열 길이와 같을 때, 조건은 거짓이 되고, 루프의 몸통 부문은 실행이 되지 않는다. 파이썬이 접근한 마지막 len(fruit)-1 인텍스 문자로, 문자열의 마지막 문자다.

Exercise 6.1 문자열의 마지막 문자에서 시작해서, 문자열 처음으로 역진행하면서 한줄에 한자씩 화면에 출력하는 while 루프를 작성하세요.

운행법을 작성하는 또 다른 방법은 for 루프다.

```
for char in fruit:
    print char
```

루프를 매번 반복할 때, 문자열 다음 문자가 변수 char에 대입된다. 루프는 더 이상 남겨진 문자가 없을 때까지 계속 실행된다.

제 4 절 문자열 슬라이스(slice)

문자열의 일부분을 슬라이스(slice)라고 한다. 문자열 슬라이스를 선택하는 것은 문자를 선택하는 것과 유사하다.

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print s[6:13]
Python
```

[n:m] 연산자는 n번째 문자부터 m번째 문자까지의 문자열 - 첫 번째는 포함하지만 마지막은 제외 - 부분을 반환한다.

콜론 앞 첫 인텍스를 생략하면, 문자열 슬라이스는 문자열 처음부터 시작한다. 두 번째 인텍스를 생략하면, 문자열 슬라이스는 문자열 끝까지 간다.

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

만약 첫번째 인텍스가 두번째보다 크거나 같은 경우 결과는 인용부호로 표현되는 빈 문자열(empty string)이 된다.

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

빈 문자열은 어떤 문자도 포함하지 않아서 길이가 0 이 되지만, 이것을 제외하고 다른 문자열과 동일하다.

Exercise 6.2 fruit이 문자열로 주어졌을 때, fruit[:]의 의미는 무엇인가요?

제 5 절 문자열은 불변이다.

문자열 내부에 있는 문자를 변경하려고 대입문 왼쪽편에 [] 연산자를 사용하고 싶은 유혹이 있을 것이다. 예를 들어 다음과 같다.

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

이 경우 "객체(object)"는 문자열이고, 대입하고자 하는 문자는 "항목(item)"이다. 지금으로서 객체는 값과 동일하지만, 나중에 객체 정의를 좀더 상세화할 것이다. 항목은 순서 값 중의 하나다.

오류 이유는 문자열은 불변(immutable)이기 때문이다. 따라서 기존 문자열을 변경할 수 없다는 의미다. 최선의 방법은 원래 문자열을 변형한 새로운 문자열을 생성하는 것이다.

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
Jello, world!
```

새로운 첫 문자에 `greeting` 문자열 슬라이스를 연결한다. 원래 문자열에는 어떤 영향도 주지 않는 새로운 문자열을 생성되었다.

제 6 절 루프 돌기(looping) 계수(counting)

다음 프로그램은 문자열에 문자 `a`가 나타나는 횟수를 계수한다.

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print count
```

상기 프로그램은 계수기(counter)라고 부르는 또다른 연산 패턴을 보여준다. 변수 `count`는 0 으로 초기화 되고, 매번 `a`를 찾을 때마다 증가한다. 루프를 빠져나갔을 때, `count`는 결과 값 즉, `a`가 나타난 총 횟수를 담고 있다.

Exercise 6.3 문자열과 문자를 인자(argument)로 받도록 상기 코드를 `count`라는 함수로 캡슐화(encapsulation)하고 일반화하세요.

제 7 절 `in` 연산자

연산자 `in` 은 불 연산자로 두 개의 문자열을 받아, 첫 번째 문자열이 두 번째 문자열의 일부이면 참(`True`)을 반환한다.

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

제 8 절 문자열 비교

비교 연산자도 문자열에서 동작한다. 두 문자열이 같은지를 살펴보다.

```
if word == 'banana':
    print  'All right, bananas.'
```

다른 비교 연산자는 단어를 알파벳 순서로 정렬하는데 유용하다.

```
if word < 'banana':
    print 'Your word,' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word,' + word + ', comes after banana.'
else:
    print 'All right, bananas.'
```

파이썬은 사람과 동일하는 방식으로 대문자와 소문자를 다루지 않는다. 모든 대문자는 소문자 앞에 온다.

```
Your word, Pineapple, comes before banana.
```

이러한 문제를 다루는 일반적인 방식은 비교 연산을 수행하기 전에 문자열을 표준 포맷으로 예를 들어 모두 소문자, 변환하는 것입니다. 경우에 따라서 "Pineapple"로 무장한 사람들로부터 여러분을 보호해야하는 것을 명심하세요.

제 9 절 string 메쏘드

문자열은 파이썬 객체(objects)의 한 예다. 객체는 데이터(실제 문자열 자체)와 메쏘드(methods)를 담고 있다. 메쏘드는 객체에 내장되고 어떤 객체의 인스턴스(instance)에도 사용되는 사실상 함수다.

객체에 대해 이용가능한 메쏘드를 보여주는 dir 함수가 파이썬에 있다. type 함수는 객체의 형(type)을 보여 주고, dir은 객체에 사용될 수 있는 메쏘드를 보여준다.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<type 'str'>
>>> dir(stuff)
['capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> string

    Return a copy of the string S with only its first character
    capitalized.
>>>
```

dir 함수가 메쏘드 목록을 보여주고, 메쏘드에 대한 간단한 문서 정보는 help를 사용할 수 있지만, 문자열 메쏘드에 대한 좀더 좋은 문서 정보는 **docs.python.org/library/string.html**에서 찾을 수 있다.

인자를 받고 값을 반환한다는 점에서 메쏘드(method)를 호출하는 것은 함수를 호출하는 것과 유사하지만, 구문은 다르다. 구분자로 점을 사용해서 변수명에 메쏘드명을 붙여 메쏘드를 호출한다.

예를 들어, upper 메쏘드는 문자열을 받아 모두 대문자로 변환된 새로운 문자열을 반환한다.

함수 구문 upper(word) 대신에, word.upper() 메쏘드 구문을 사용한다.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

이런 형태의 점 표기법은 메쏘드 이름(upper)과 메쏘드가 적용되는 문자열 이름(word)을 명세한다. 빈 괄호는 메쏘드가 인자가 없다는 것을 나타낸다.

메쏘드를 부르는 것을 호출(invocation)이라고 부른다. 상기의 경우, word에 upper 메쏘드를 호출한다고 말한다.

예를 들어, 문자열안에 문자열의 위치를 찾는 find라는 문자열 메쏘드가 있다.

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
```

상기 예제에서, word 문자열의 find 메쏘드를 호출하여 매개 변수로 찾고자 하는 문자를 넘긴다.

find 메쏘드는 문자뿐만 아니라 부속 문자열(substring)도 찾을 수 있다.

```
>>> word.find('na')
2
```

두 번째 인자로 어디서 검색을 시작할지 인텍스를 넣을 수 있다.

```
>>> word.find('na', 3)
4
```

한 가지 자주 있는 작업은 strip 메쏘드를 사용해서 문자열 시작과 끝의 공백(공백 여러개, 탭, 새줄)을 제거하는 것이다.

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

startswith 메쏘드는 참, 거짓 같은 불 값(boolean value)을 반환한다.

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

startswith가 대소문자를 구별하는 것을 요구하기 때문에 `lower` **메쏘드를 사용해서 검증을 수행하기 전에, 한 줄을 입력받아 모두 소문자로 변환하는 것이 필요하다.**

```
>>> line = 'Please have a nice day'
>>> line.startswith('p')
False
>>> line.lower()
'please have a nice day'
>>> line.lower().startswith('p')
True
```

마지막 예제에서 문자열이 문자 "p"로 시작하는지를 검증하기 위해서, `lower` **메쏘드를 호출하고 나서 바로** `startswith` **메쏘드를 사용한다. 주의깊게 순서만 다룬다면, 한 줄에 다수 메쏘드를 호출할 수 있다.**

Exercise 6.4 앞선 예제와 유사한 함수인 `count`로 불리는 문자열 메쏘드가 있다. `docs.python.org/library/string.html`에서 `count` 메쏘드에 대한 문서를 읽고, 문자열 `'banana'`의 문자가 몇 개인지 계수하는 메쏘드 호출 프로그램을 작성하세요.

제 10 절 문자열 파싱(Parsing)

종종, 문자열을 들여다 보고 특정 부속 문자열(substring)을 찾고 싶다. 예를 들어, 아래와 같은 형식으로 작성된 일련의 라인이 주어졌다고 가정하면,

```
From stephen.marquard@ uct.ac.za Sat Jan  5 09:14:16 2008
```

각 라인마다 뒤쪽 전자우편 주소(즉, `uct.ac.za`**)만 뽑아내고 싶을 것이다.** `find` **메쏘드와 문자열 슬라이싱(string sliceing)을 사용해서 작업을 수행할 수 있다.**

우선, 문자열에서 골뱅이(@, at-sign) 기호의 위치를 찾는다. 그리고, 골뱅이 기호 *뒤* 첫 공백 위치를 찾는다. 그리고 나서, 찾고자 하는 부속 문자열을 뽑아내기 위해서 문자열 슬라이싱을 사용한다.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print atpos
21
>>> sppos = data.find(' ',atpos)
>>> print sppos
31
>>> host = data[atpos+1:sppos]
>>> print host
uct.ac.za
>>>
```

find 메쏘드를 사용해서 찾고자 하는 문자열의 시작 위치를 명세한다. 문자열 슬라이싱(slicing)할 때, 골뱅기 기호 뒤부터 빈 공백을 *포함하지 않는* 위치까지 문자열을 뽑아낸다.

find 메쏘드에 대한 문서는 **docs.python.org/library/string.html** 에서 참조 가능하다.

제 11 절 서식 연산자

서식 연산자(format operator), %는 문자열 일부를 변수에 저장된 값으로 바꿔 문자열을 구성한다. 정수에 서식 연산자가 적용될 때, %는 나머지 연산자가 된다. 하지만 첫 피연산자가 문자열이면, %은 서식 연산자가 된다.

첫 피연산자는 서식 문자열 format string로 두번째 피연산자가 어떤 형식으로 표현되는지를 명세하는 하나 혹은 그 이상의 서식 순서 format sequence를 담고 있다. 결과값은 문자열이다.

예를 들어, 형식 순서 '%d'의 의미는 두번째 피연산자가 정수 형식으로 표현됨을 뜻한다. (d는 "decimal"을 나타낸다.)

```
>>> camels = 42
>>> '%d' % camels
'42'
```

결과는 문자열 '42'로 정수 42와 혼동하면 안 된다.

서식 순서는 문자열 어디에도 나타날 수 있어서 문장 중간에 값을 임베드(embed)할 수 있다.

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

만약 문자열 서식 순서가 하나 이상이라면, 두번째 인자는 튜플(tuple)이 된다. 서식 순서 각각은 순서대로 튜플 요소와 매칭된다.

다음 예제는 정수 형식을 표현하기 위해서 '%d', 부동 소수점 형식을 표현하기 위해서 '%g', 문자열 형식을 표현하기 위해서 '%s'을 사용한 사례다. 여기서 왜 부동 소수점 형식이 '%f'대신에 '%g'인지는 질문하지 말아주세요.

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

튜플 요소 숫자는 문자열 서식 순서의 숫자와 일치해야 하고, 요소의 자료형(type)도 서식 순서와 일치해야 한다.

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

상기 첫 예제는 충분한 요소 개수가 되지 않고, 두 번째 예제는 자료형이 맞지 않는다.

서식 연산자는 강력하지만, 사용하기가 어렵다. 더 많은 정보는 `docs.python.org/lib/typesseq-strings.html`에서 찾을 수 있다.

제 12 절 디버깅

프로그램을 작성하면서 배양해야 하는 기술은 항상 자신에게 질문을 하는 것이다. "여기서 무엇이 잘못 될 수 있을까?" 혹은 "내가 작성한 완벽한 프로그램을 망가뜨리기 위해 사용자는 무슨 엄청난 일을 할 것인가?"

예를 들어 앞장의 반복 `while` 루프를 시연하기 위해 사용한 프로그램을 살펴봅시다.

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print line

print 'Done!'
```

사용자가 입력값으로 빈 공백 줄을 입력하게 될 때 무엇이 발생하는지 살펴봅시다.

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#' :
```

빈 공백줄이 입력될 때까지 코드는 잘 작동합니다. 그리고 나서, 0 번째 문자가 없어서 트레이스백(traceback)이 발생한다. 입력줄이 비어있을 때, 코드 3번째 줄을 "안전"하게 만드는 두 가지 방법이 있다.

하나는 빈 문자열이면 거짓(`False`)을 반환하도록 `startswith` 메쏘드를 사용하는 것이다.

```
    if line.startswith('#') :
```

가디언 패턴(guardian pattern)을 사용한 `if`문으로 문자열에 적어도 하나의 문자가 있는 경우만 두번째 논리 표현식이 평가되도록 코드를 작성한다.

```
    if len(line) > 0 and line[0] == '#' :
```

제 13 절 용어정의

계수기(counter): 무언가를 계수하기 위해서 사용되는 변수로 일반적으로 0 으로 초기화하고 나서 증가한다.

빈 문자열(empty string): 두 인용부호로 표현되고, 어떤 문자도 없고 길이가 0 인 문자열.

서식 연산자(format operator): 서식 문자열과 튜플을 받아, 서식 문자열에 지정된 서식으로 튜플 요소를 포함하는 문자열을 생성하는 연산자, `%`.

서식 순서(format sequence): `%d`처럼 어떤 값의 서식으로 표현되어야 하는지를 명세하는 "서식 문자열" 문자 순서.

서식 문자열(format string): 서식 순서를 포함하는 서식 연산자와 함께 사용되는 문자열.

플래그(flag): 조건이 참인지를 표기하기 위해 사용하는 불 변수(boolean variable)

호출(invocation): 메쏘드를 호출하는 명령문.

불변(immutable): 순서의 항목에 대입할 수 없는 특성.

인덱스(index): 문자열의 문자처럼 순서(sequence)에 항목을 선택하기 위해 사용되는 정수 값.

항목(item): 순서에 있는 값의 하나.

메쏘드(method): 객체와 연관되어 점 표기법을 사용하여 호출되는 함수.

객체(object): 변수가 참조하는 무엇. 지금은 "객체"와 "값"을 구별없이 사용한다.

검색(search): 찾고자 하는 것을 찾았을 때 멈추는 운행법 패턴.

순서(sequence): 정돈된 집합. 즉, 정수 인덱스로 각각의 값이 확인되는 값의 집합.

슬라이스(slice): 인덱스 범위로 지정되는 문자열 부분.

운행법(traverse): 순서(sequence)의 항목을 반복적으로 훑기, 각각에 대해서는 동일한 연산을 수행.

제 14 절 연습문제

Exercise 6.5 다음 문자열을 파이썬 코드를 작성하세요.

```
str = 'X-DSPAM-Confidence:  0.8475'
```

find 메쏘드와 문자열 슬라이싱을 사용하여 콜론(:) 문자 뒤 문자열을 뽑아내고 float 함수를 사용하여 뽑아낸 문자열을 부동 소수점 숫자로 변환하세요.

Exercise 6.6 https://docs.python.org/2.7/library/stdtypes.html#string-methods에서 문자열 메쏘드 문서를 읽어보세요. 어떻게 동작하는가를 이해도를 확인하기 위해서 몇개를 골라 실험을 해보세요. strip과 replace가 특히 유용합니다.

문서는 좀 혼동스러울 수 있는 구문을 사용합니다. 예를 들어, find(sub[, start[, end]])의 꺾쇠기호는 선택(옵션) 인수를 나타냅니다. 그래서, sub은 필수지만, start은 선택 사항이고, 만약 start가 인자로 포함된다면, end는 선택이 된다.

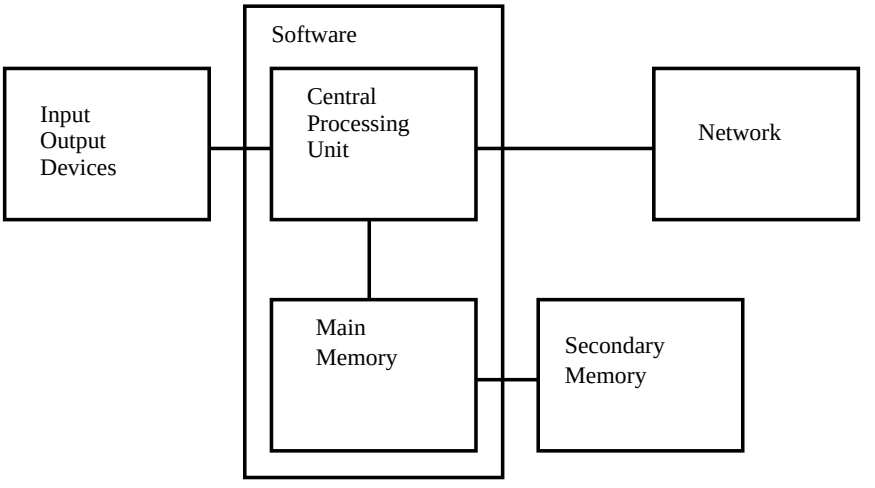
7 장

파일

제 1 절 영속성(Persistence)

지금까지, 프로그램을 어떻게 작성하고 조건문, 함수, 반복을 사용하여 중앙처리장치(CPU, Central Processing Unit)에 프로그래머의 의도를 커뮤니케이션하는지 학습했다. 주기억장치(Main Memory)에 어떻게 자료구조를 생성하고 사용하는지도 배웠다. CPU와 주기억장치는 소프트웨어가 동작하고 실행되는 곳이고, 모든 "생각(thinking)"이 발생하는 장소다.

하지만, 앞서 하드웨어 아키텍처를 논의했던 기억을 되살린다면, 전원이 꺼지게 되면, CPU와 주기억장치에 저장된 모든 것이 지워진다. 지금까지 작성한 프로그램은 파이썬을 배우기 위한 일시적으로 재미로 연습한 것에 불과하다.



이번 장에서 보조 기억장치(Secondary Memory) 혹은 파일을 가지고 작업을 시작한다. 보조 기억장치는 전원이 꺼져도 지워지지 않는다. 혹은, USB 플래쉬 드라이브를 사용한 경우에는 프로그램으로부터 작성한 데이터는 시스템에서 제거되어 다른 시스템으로 전송될 수 있다.

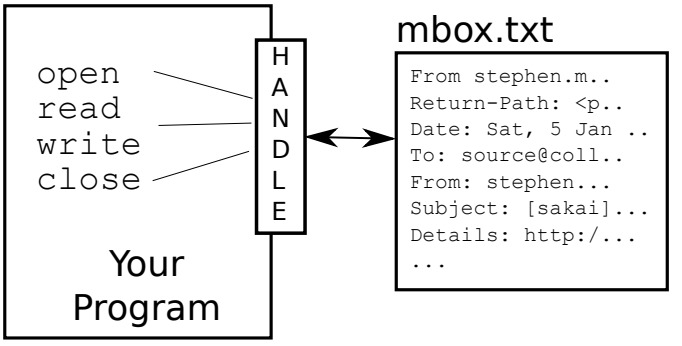
우선 텍스트 편집기로 작성한 텍스트 파일을 읽고 쓰는 것에 초점을 맞출 것이다. 나중에 데이터베이스 소프트웨어를 통해서 읽고 쓰도록 설계된 바이너리 파일 데이터베이스를 가지고 어떻게 작업하는지를 살펴볼 것이다.

제 2 절 파일 열기

하드 디스크 파일을 읽거나 쓸려고 할 때, 파일을 열여야(open) 한다. 파일을 열 때 각 파일 데이터가 어디에 저장되었는지를 알고 있는 운영체제와 커뮤니케이션 한다. 파일을 열 때, 운영체제에 파일이 존재하는지 확인하고 이름으로 파일을 찾도록 요청한다. 이번 예제에서, 파이썬을 시작한 동일한 폴더에 저장된 `mbox.txt` 파일을 연다. **`www.py4inf.com/code/mbox.txt`** 에서 파일을 다운로드할 수 있다.

```
>>> fhand = open('mbox.txt')
>>> print fhand
<open file 'mbox.txt', mode 'r' at 0x1005088b0>
```

`open`이 성공하면, 운영체제는 파일 핸들(file handle)을 반환한다. 파일 핸들(file handle)은 파일에 담겨진 실제 데이터는 아니고, 대신에 데이터를 읽을 수 있도록 사용할 수 있는 "핸들(handle)"이다. 요청한 파일이 존재하고, 파일을 읽을 수 있는 적절한 권한이 있다면 이제 핸들이 여러분에게 주어졌다.



파일이 존재하지 않는다면, `open`은 역추적(traceback) 파일 열기 오류로 실패하고, 파일 콘텐츠에 접근할 핸들도 얻지 못한다.

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'stuff.txt'
```

나중에 `try`와 `except`를 가지고, 존재하지 않는 파일을 열려고 하는 상황을 좀더 우아하게 처리할 것이다.

제 3 절 텍스트 파일과 라인

파이썬 문자열이 문자 순서(sequence)로 간주 되듯이 마찬가지로 텍스트 파일은 라인 순서(sequence)로 생각될 수 있다. 예를 들어, 다음은 오픈 소스 프로젝

트 개발 팀에서 다양한 참여자들의 전자우편 활동을 기록한 텍스트 파일 샘플이다.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

상호 의사소통한 전자우편 전체 파일은 **www.py4inf.com/code/mbox.txt** 에서 접근 가능하고, 간략한 버젼 파일은 **www.py4inf.com/code/mbox-short.txt**에서 얻을 수 있다. 이들 파일은 다수 전자우편 메시지를 담고 있는 파일로 표준 포맷으로 되어 있다. "From "으로 시작하는 라인은 메시지 본문과 구별되고, "From: "으로 시작하는 라인은 본문 메시지의 일부다. 더 자세한 정보는 **en.wikipedia.org/wiki/Mbox**에서 찾을 수 있다.

파일을 라인으로 쪼개기 위해서, 새줄(newline) 문자로 불리는 "줄의 끝(end of the line)"을 표시하는 특수 문자가 있다.

파이썬에서, 문자열 상수 역슬래쉬-n(`\n`)으로 새줄(newline) 문자를 표현한다. 두 문자처럼 보이지만, 사실은 단일 문자다. 인터프리터에 "stuff"에 입력한 후 변수를 살펴보면, 문자열에 `\n`가 있다. 하지만, `print`문을 사용하여 문자열을 출력하면, 문자열이 새줄 문자에 의해서 두 줄로 쪼개지는 것을 볼 수 있다.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print stuff
Hello
World!
>>> stuff = 'X\nY'
>>> print stuff
X
Y
>>> len(stuff)
3
```

문자열 `'X\nY'`의 길이는 *3* 이다. 왜냐하면 새줄(newline) 문자도 한 문자이기 때문이다.

그래서, 파일 라인을 볼 때, 라인 끝을 표시하는 새줄(newline)로 불리는 눈에 보이지 않는 특수 문자가 각 줄의 끝에 있다고 *상상*할 필요가 있다.

그래서, 새줄(newline) 문자는 파일에 있는 문자를 라인으로 분리한다.

제 4 절 파일 읽어오기

파일 핸들(file handle)이 파일 자료를 담고 있지 않지만, `for` 루프를 사용하여 파일 각 라인을 읽고 라인수를 세는 것을 쉽게 구축할 수 있다.

```
fhand = open('mbox.txt')
count = 0
for line in fhand:
    count = count + 1
print 'Line Count:', count

python open.py
Line Count: 132045
```

파일 핸들을 for 루프 순서(sequence)로 사용할 수 있다. for 루프는 단순히 파일 라인 수를 세고 전체 라인수를 출력한다. for 루프를 대략 일반어로 풀어 말하면, "파일 핸들로 표현되는 파일 각 라인마다, count 변수에 1 씩 더한다"

open 함수가 전체 파일을 바로 읽지 못하는 이유는 파일이 수 기가 바이트 파일 크기를 가질 수도 있기 때문이다. open 문장은 파일 크기에 관계없이 파일을 여는데 시간이 동일하게 걸린다. 실질적으로 for 루프가 파일로부터 자료를 읽어오는 역할을 한다.

for 루프를 사용해서 이 같은 방식으로 파일을 읽어올 때, 새줄(newline) 문자를 사용해서 파일 자료를 라인 단위로 쪼갠다. 파이썬에서 새줄(newline) 문자까지 각 라인 단위로 읽고, for 루프가 매번 반복할 때마다 line 변수에 새줄(newline)을 마지막 문자로 포함한다.

for 루프가 데이터를 한번에 한줄씩 읽어오기 때문에, 데이터를 저장할 주기억장치 저장공간을 소진하지 않고, 매우 큰 파일을 효과적으로 읽어서 라인을 셀 수 있다. 각 라인별로 읽고, 세고, 그리고 나서 폐기되기 때문에, 매우 적은 저장공간을 사용해서 어떤 크기의 파일도 상기 프로그램을 사용하여 라인을 셀 수 있다.

만약 주기억장치 크기에 비해서 상대적으로 작은 크기의 파일이라는 것을 안다면, 전체 파일을 파일 핸들로 read 메쏘드를 사용해서 하나의 문자열로 읽어올 수 있다.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print len(inp)
94626
>>> print inp[:20]
From stephen.marquar
```

상기 예제에서, mbox-short.txt 전체 파일 콘텐츠(94,626 문자)를 변수 inp로 바로 읽었다. 문자열 슬라이싱을 사용해서 inp에 저장된 문자열 자료 첫 20 문자를 출력한다.

파일이 이런 방식으로 읽혀질 때, 모든 라인과 새줄(newline)문자를 포함한 모든 문자는 변수 inp에 할당된 매우 큰 문자열이다. 파일 데이터가 컴퓨터 주기억장치가 안정적으로 감당해 낼 수 있을때만, 이런 형식의 open 함수가 사용될 수 있다는 것을 기억하라.

만약 주기억장치가 감당해 낼 수 없는 매우 파일 크기가 크다면, for나 while 루프를 사용해서 파일을 쪼개서 읽는 프로그램을 작성해야 한다.

제 5 절 파일 검색

파일 데이터를 검색할 때, 흔한 패턴은 파일을 읽고, 대부분 라인은 건너뛰고, 특정 기준을 만족하는 라인만 처리하는 것이다. 간단한 검색 메카니즘을 구현하기 위해서 파일을 읽는 패턴과 문자열 메쏘드를 조합한다.

예를 들어, 파일을 읽고, "From:"으로 시작하는 라인만 출력하고자 한다면, startswith 문자열 메쏘드를 사용해서 원하는 접두사로 시작하는 라인만을 선택한다.

```
fhand = open('mbox-short.txt')
for line in fhand:
    if line.startswith('From:') :
        print line
```

이 프로그램이 실행하면 다음 출력값을 얻는다.

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

"From:"으로만 시작하는 라인만 출력하기 때문에 출력값은 훌륭해 보인다. 하지만, 왜 여분으로 빈 라인이 보이는 걸까? 원인은 눈에 보이지 않는 새줄(newline) 문자 때문이다. 각 라인이 새줄(newline)로 끝나서 변수 line에 새줄(newline)이 포함되고 `print`문이 추가로 새줄(newline)을 추가해서 결국 우리가 보기에는 두 줄 효과가 나타난다.

마지막 문자를 제외하고 모든 것을 출력하기 위해서 라인 슬라이싱(slicing)을 할수 있지만, 좀더 간단한 접근법은 다음과 같이 문자열 오른쪽 끝에서부터 공백을 벗겨내는 rstrip 메쏘드를 사용하는 것인다.

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:') :
        print line
```

프로그램을 실행하면, 다음 출력값을 얻는다.

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

파일 처리 프로그램이 점점 더 복잡해짐에 따라 continue를 사용해서 검색 루프(search loop)를 구조화할 필요가 있다. 검색 루프의 기본 아이디어는 "흥미로운" 라인을 집중적으로 찾고, "흥미롭지 않은" 라인은 효과적으로 건너뛰는 것이다. 그리고 나서 흥미로운 라인을 찾게되면, 그 라인에서 특정 연산을 수행하는 것이다.

다음과 같이 루프를 구성해서 흥미롭지 않은 라인은 건떠뛰는 패턴을 따르게 한다.

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:') :
        continue
    # Process our 'interesting' line
    print line
```

프로그램의 출력값은 동일하다. 흥미롭지 않는 라인은 "From:"으로 시작하지 않는 라인이라 continue문을 사용해서 건너뛴다. "흥미로운" 라인 (즉, "From:"으로 시작하는 라인)에 대해서는 연산처리를 수행한다.

find 문자열 메쏘드를 사용해서 검색 문자열이 라인 어디에 있는지를 찾아주는 텍스트 편집기 검색기능을 모사(simulation)할 수 있다. find 메쏘드는 다른 문자열 내부에 검색하는 문자열이 있는지 찾고, 문자열 위치를 반환하거나, 만약 문자열이 없다면 -1을 반환하기 때문에, "@uct.ac.za"(남아프리카 케이프 타운 대학으로부터 왔다) 문자열을 포함하는 라인을 검색하기 위해 다음과 같이 루프를 작성한다.

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1 :
        continue
    print line
```

출력결과는 다음과 같다.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

제 6 절 사용자가 파일명을 선택하게 만들기

매번 다른 파일을 처리할 때마다 파이썬 코드를 편집하고 싶지는 않다. 매번 프로그램이 실행될 때마다, 파일명을 사용자가 입력하도록 만드는 것이 좀더 유

용할 것이다. 그래서 파이썬 코드를 바꾸지 않고, 다른 파일에 대해서도 동일한 프로그램을 사용하도록 만들자.

다음과 같이 raw_input을 사용해서 사용자로부터 파일명을 읽어 프로그램을 실행하는 것이 단순하다.

```
fname = raw_input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
print 'There were', count, 'subject lines in', fname
```

사용자로부터 파일명을 읽고 변수 fname에 저장하고, 그 파일을 연다. 이제 다른 파일에 대해서도 반복적으로 프로그램을 실행할 수 있다.

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

다음 절을 엿보기 전에, 상기 프로그램을 살펴보고 자신에게 다음을 질문해 보자. "여기서 어디가 잘못될 수 있는가?" 혹은 "이 작고 멋진 프로그램에 트레이스백(traceback)을 남기고 바로 끝나게 하여, 결국 사용자 눈에는 좋지 않은 프로그램이라는 인상을 남길 수 있도록 우리의 친절한 사용자는 무엇을 할 수 있을까?

제 7 절 try, except, open 사용하기

제가 여러분에게 엿보지 말라고 말씀드렸습니다. 이번이 마지막 기회입니다. 사용자가 파일명이 아닌 뭔가 다른 것을 입력하면 어떻게 될까요?

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
IOError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
IOError: [Errno 2] No such file or directory: 'na na boo boo'
```

웃지마시구요, 사용자는 결국 여러분이 작성한 프로그램을 망가뜨리기 위해 고의든 악의를 가지든 가능한 모든 수단을 강구할 것입니다. 사실, 소프트웨어 개

발팀의 중요한 부분은 품질 보증(Quality Assurance, QA)이라는 조직이다. 품질보증 조직은 프로그래머가 만든 소프트웨어를 망가뜨리기 위해 가능한 말도 안 되는 것을 합니다.

사용자가 소프트웨어를 제품으로 구매하거나, 주문형으로 개발하는 프로그램에 대해 월급을 지급하던지 관계없이 품질보증 조직은 프로그램이 사용자에게 전달되기 전까지 프로그램 오류를 발견할 책임이 있다. 그래서 품질보증 조직은 프로그래머의 최고의 친구다.

프로그램 오류를 찾았기 때문에, `try`/`except` 구조를 사용해서 오류를 우아하게 고쳐봅시다. `open` 호출이 잘못될 수 있다고 가정하고, `open` 호출이 실패할 때를 대비해서 다음과 같이 복구 코드를 추가한다.

```
fname = raw_input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print 'File cannot be opened:', fname
    exit()

count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
print 'There were', count, 'subject lines in', fname
```

`exit`함수가 프로그램을 끝낸다. 결코 돌아오지 않는 함수를 호출한 것이다. 이제 사용자 혹은 품질 보증 조직에서 올바르지 않거나 어처구니 없는 파일명을 입력했을 때, "catch"로 잡아서 우아하게 복구한다.

```
python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

파이썬 프로그램을 작성할 때 `open` 호출을 보호하는 것은 `try`, `except`의 적절한 사용 예제가 된다. "파이썬 방식(Python way)"으로 무언가를 작성할 때, "파이썬스러운(Pythonic)"이라는 용어를 사용한다. 상기 파일을 여는 예제는 파이썬스러운 방식의 좋은 예가 된다고 말한다.

파이썬에 좀더 자신감이 생기게 되면, 다른 파이썬 프로그래머와 동일한 문제에 대해 두 가지 동치하는 해답을 가지고 어떤 접근법이 좀더 "파이썬스러운지"에 대한 현답을 찾는데도 관여하게 된다.

"좀더 파이썬스럽게" 되는 이유는 프로그래밍이 엔지니어링적인 면과 예술적인 면을 동시에 가지고 있기 때문이다. 항상 무언가를 단지 작동하는 것에만 관심이 있지 않고, 프로그램으로 작성한 해결책이 좀더 우아하고, 다른 동료에 의해서 우아한 것으로 인정되기를 또한 원합니다.

제 8 절 파일에 쓰기

파일에 쓰기 위해서는 두 번째 매개 변수로 'w' 모드로 파일을 열어야 한다.

```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

파일이 이미 존재하는데 쓰기 모드에서 파일을 여는 것은 이전 데이터를 모두 지워버리고, 깨끗한 파일 상태에서 다시 시작되니 주의가 필요하다. 만약 파일이 존재하지 않는다면, 새로운 파일이 생성된다.

파일 핸들 객체의 write 메쏘드는 데이터를 파일에 저장한다.

```
>>> line1 = 'This here's the wattle,\n'
>>> fout.write(line1)
```

다시 한번, 파일 객체는 마지막 포인터가 어디에 있는지 위치를 추적해서, 만약 write 메쏘드를 다시 호출하게 되면, 새로운 데이터를 파일 끝에 추가한다.

라인을 끝내고 싶을 때, 명시적으로 새줄(newline) 문자를 삽입해서 파일에 쓰도록 라인 끝을 필히 관리해야 한다.

print문은 자동적으로 새줄(newline)을 추가하지만, write 메쏘드는 자동적으로 새줄(newline)을 추가하지는 않는다.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
```

파일 쓰기가 끝났을 때, 파일을 필히 닫아야 한다. 파일을 닫는 것은 데이터 마지막 비트까지 디스크에 물리적으로 쓰여져서, 전원이 나가더라도 자료가 유실되지 않는 역할을 한다.

```
>>> fout.close()
```

파일 읽기로 연 파일을 닫을 수 있지만, 몇개 파일을 열어 놓았다면 약간 단정치 못하게 끝날 수 있습니다. 왜냐하면 프로그램이 종료될 때 열린 모든 파일이 닫혀졌는지 파이썬이 확인하기 때문이다. 파일에 쓰기를 할 때는, 파일을 명시적으로 닫아서 예기치 못한 일이 발생할 여지를 없애야 한다.

제 9 절 디버깅

파일을 읽고 쓸 때, 공백 때문에 종종 문제에 봉착한다. 이런 종류의 오류는 공백, 탭, 새줄(newline)이 눈에 보이지 않기 때문에 디버그하기도 쉽지 않다.

```
>>> s = '1 2\t 3\n 4'
>>> print s
1 2  3
 4
```

내장함수 repr이 도움이 될 수 있다. 인자로 임의 객체를 잡아 객체 문자열 표현으로 반환한다. 문자열 공백문자는 역슬래쉬 순서(sequence)로 나타냅니다.

```
>>> print repr(s)
'1 2\t 3\n 4'
```

디버깅에 도움이 될 수 있다.

여러분이 봉착하는 또 다른 문제는 다른 시스템에서는 라인 끝을 표기하기 위해서 다른 문자를 사용한다는 점이다. 어떤 시스템은 `\n` 으로 새줄(newline)을 표기하고, 다른 시스템은 `\r`으로 반환 문자(return character)를 사용한다. 둘 다 모두 사용하는 시스템도 있다. 파일을 다른 시스템으로 이식한다면, 이러한 불일치가 문제를 야기한다.

대부분의 시스템에는 A 포맷에서 B 포멧으로 변환하는 응용프로그램이 있다. `wikipedia.org/wiki/Newline` 에서 응용프로그램을 찾을 수 있고, 좀더 많은 것을 읽을 수 있다. 물론, 여러분이 직접 프로그램을 작성할 수도 있다.

제 10 절 용어정의

잡기(catch): `try`와 `except` **문을 사용해서 프로그램이 끝나는 예외 상황을 방지하는 것.**

새줄(newline): 라인의 끝을 표기 위한 파일이나 문자열에 사용되는 특수 문자.

파이썬스러운(Pythonic): 파이썬에서 우아하게 작동하는 기술. "try와 catch를 사용하는 것은 파일이 없는 경우를 복구하는 파이썬스러운 방식이다."

품질 보증(Quality Assurance, QA): 소프트웨어 제품의 전반적인 품질을 보증하는데 집중하는 사람이나 조직. 품질 보증은 소프트웨어 제품을 시험하고, 제품이 시장에 출시되기 전에 문제를 확인하는데 관여한다.

텍스트 파일(text file): 하드디스크 같은 영구 저장소에 저장된 일련의 문자 집합.

제 11 절 연습문제

Exercise 7.1 파일을 읽고 한줄씩 파일의 내용을 모두 대문자로 출력하는 프로그램을 작성하세요. 프로그램을 싱행하면 다음과 같이 보일 것입니다.

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN  5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
 BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
 SAT, 05 JAN 2008 09:14:16 -0500
```

`www.py4inf.com/code/mbox-short.txt`에서 파일을 다운로드 받으세요.

Exercise 7.2 파일명을 입력받아, 파일을 읽고, 다음 형식의 라인을 찾는 프로그램을 작성하세요.

```
X-DSPAM-Confidence: 0.8475
```

"X-DSPAM-Confidence:"로 시작하는 라인을 만나게 되면, 부동 소수점 숫자를 뽑아내기 위해 해당 라인을 별도로 보관하세요. 라인 수를 세고, 라인으로부터 스팸 신뢰값의 총계를 계산하세요. 파일의 끝에 도달할 했을 때, 평균 스팸 신뢰도를 출력하세요.

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745

Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

`mbox.txt`와 `mbox-short.txt` 파일에 작성한 프로그램을 시험하세요.

Exercise 7.3 때때로, 프로그래머가 지루해지거나, 약간 재미를 목적으로, 프로그램에 무해한 **부활절 달걀**(Easter Egg, `en.wikipedia.org/wiki/Easter_egg_(media)`)을 넣습니다. 사용자가 파일명을 입력하는 프로그램을 변형시켜, 'na na boo boo'로 파일명을 정확하게 입력했을 때, 재미있는 메시지를 출력하는 프로그램을 작성하세요. 파일이 존재하거나, 존재하지 않는 다른 모든 파일에 대해서도 정상적으로 작동해야 합니다. 여기 프로그램을 실행한 견본이 있습니다.

```
python egg.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python egg.py
Enter the file name: missing.tyxt
File cannot be opened: missing.tyxt

python egg.py
Enter the file name: na na boo boo
NA NA BOO BOO TO YOU - You have been punk'd!
```

프로그램에 부활절 달걀을 넣도록 격려하지는 않습니다. 단지 연습입니다.

8 장

리스트 (List)

제 1 절 리스트는 순서(sequence)다.

문자열처럼, 리스트(list)는 값의 순서(sequence)다. 문자열에서, 값은 문자지만, 리스트에서는 임의 자료형(type)도 될 수 있다. 리스트 값은 요소(elements)나 때때로 항목(items)으로 불린다.

신규 리스틀 생성하는 방법은 여러 가지가 있다. 가장 간단한 방법은 꺾쇠 괄호 ([와])로 요소를 감싸는 것이다.

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

첫번째 예제는 4개 정수 리스트다. 두번째 예제는 3개 문자열 리스트다. 문자열 요소가 동일한 자료형(type)일 필요는 없다. 다음 리스트는 문자열, 부동 소수점 숫자, 정수, (아!) 또 다른 리스트를 담고 있다.

```
['spam', 2.0, 5, [10, 20]]
```

또 다른 리스트 내부에 리스트가 중첩(nested)되어 있다.

어떤 요소도 담고 있지 않는 리스트를 빈 리스트(empty list)라고 부르고, 빈 꺾쇠 괄호("[]")로 생성한다.

예상했듯이, 리스트 값을 변수에 대입할 수 있다.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

제 2 절 리스트는 변경가능하다.

리스트 요소에 접근하는 구문은 문자열 문자에 접근하는 것과 동일한 꺾쇠 괄호 연산자다. 꺽쇠 괄호 내부 표현식은 인덱스를 명세한다. 기억할 것은 인덱스는

0 에서부터 시작한다는 것이다.

```
>>> print cheeses[0]
Cheddar
```

문자열과 달리, 리스트 항목 순서를 바꾸거나, 리스트에 새로운 항목을 다시 대입할 수 있기 때문에 리스트는 변경가능하다. 꺾쇠 괄호 연산자가 대입문 왼쪽 편에 나타날 때, 새로 대입될 리스트 요소를 나타낸다.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

리스트 numbers 첫번째 요소는 123 값을 가지고 있었으나, 이제 5 값을 가진다.

리스트를 인덱스와 요소의 관계로 생각할 수 있다. 이 관계를 매핑(mapping)이라고 부른다. 각각의 인덱스는 요소 중 하나에 대응("maps to")된다.

리스트 인덱스는 문자열 인덱스와 동일한 방식으로 동작한다.

- 어떠한 정수 표현식도 인덱스로 사용할 수 있다.
- 존재하지 않는 요소를 읽거나 쓰려고 하면, 인덱스 오류 (IndexError)가 발생한다.
- 인덱스가 음의 값이면, 리스트 끝에서부터 역으로 센다.

in 연산자도 또한 리스트에서 동작하니 사용할 수 있다.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

제 3 절 리스트 운행법

리스트 요소를 운행하는 가장 흔한 방법은 for문을 사용하는 것이다. 문자열에서 사용한 것과 구문은 동일하다.

```
for cheese in cheeses:
    print cheese
```

리스트 요소를 읽기만 한다면 이것만으로도 잘 동작한다. 하지만, 리스트 요소를 쓰거나, 갱신하는 경우, 인덱스가 필요하다. 리스트 요소를 쓰거나 갱신하는 일반적인 방법은 range와 len 함수를 조합하는 것이다.

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

상기 루프는 리스트를 운행하고 각 요소를 갱신한다. `len` 함수는 리스트 요소 갯수를 반환한다. `range` 함수는 0 에서 $n-1$ 까지 리스트 인덱스를 반환한다. 여기서, n은 리스트 길이다. 매번 루프가 반복될 때마다, `i`는 다음 요소 인덱스를 얻는다. 몸통 부문 대입문은 `i`를 사용해서 요소의 이전 값을 읽고 새 값을 대입한다.

빈 리스트에 대해서 `for`문은 결코 몸통 부문을 실행하지 않는다.

```
for x in empty:
    print 'This never happens.'
```

리스트가 또 다른 리스트를 담을 수 있지만, 중첩된 리스트는 여전히 요소 하나로 간주된다. 다음 리스트 길이는 4 이다.

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

제 4 절 리스트 연산자

+ 연산자는 리스트를 결합한다.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

유사하게 * 연산자는 주어진 횟수만큼 리스트를 반복한다.

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

첫 예제는 `[0]`을 4회 반복한다. 두 번째 예제는 `[1, 2, 3]` 리스트를 3회 반복한다.

제 5 절 리스트 슬라이스(List slices)

슬라이스 연산자는 리스트에도 또한 동작한다.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

첫 번째 인덱스를 생략하면, 슬라이스는 처음부터 시작한다. 두 번째 인덱스를 생략하면, 슬라이스는 끝까지 간다. 그래서 양쪽의 인덱스를 생략하면, 슬라이스 결과는 전체 리스트를 복사한 것이 된다.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

리스트는 변경이 가능하기 때문에 리스트를 접고, 돌리고, 훼손하는 연산을 수행하기 전에 복사본을 만들어 두는 것이 유용하다.

대입문 왼편의 슬라이스 연산자로 복수의 요소를 갱신할 수 있다.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print t
['a', 'x', 'y', 'd', 'e', 'f']
```

제 6 절 리스트 메쏘드

파이썬은 리스트에 연산하는 메쏘드를 제공한다. 예를 들어, 덧붙이기 (append) **메쏘드는 리스트 끝에 신규 요소를 추가한다.**

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print t
['a', 'b', 'c', 'd']
```

확장 (extend) **메쏘드는 인자로 리스트를 받아 모든 요소를 리스트에 덧붙인다.**

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print t1
['a', 'b', 'c', 'd', 'e']
```

상기 예제는 t2 **리스트를 변경없이 그냥 둔다.**

정렬 (sort) **메쏘드는 낮음에서 높음으로 리스트 요소를 정렬한다.**

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print t
['a', 'b', 'c', 'd', 'e']
```

대부분의 리스트 메쏘드는 보이드(void)여서, 리스트를 변경하고 None**을 반환한다. 우연히** t = t.sort() **이렇게 작성한다면, 결과에 실망할 것이다.**

제 7 절 요소 삭제

리스트 요소를 삭제하는 방법이 몇 가지 있다. 리스트 요소 인덱스를 알고 있다면, 팝 (pop) **메쏘드를 사용한다.**

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print t
['a', 'c']
>>> print x
b
```

팝(pop) 메쏘드는 리스트를 변경하여 제거된 요소를 반환한다. 인덱스를 주지 않으면, 마지막 요소를 지우고 반환한다.

요소에서 제거된 값이 필요없다면, del 연산자를 사용한다.

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
['a', 'c']
```

(인덱스가 아닌) 제거할 요소값을 알고 있다면, 제거 (remove) **메쏘드를 사용한다.**

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print t
['a', 'c']
```

제거 (remove) **메쏘드의 반환값은** None**이다.**

하나 이상의 요소를 제거하기 위해서, 슬라이스 인덱스(slice index)와 del**을 사용한다.**

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print t
['a', 'f']
```

마찬가지로, 슬라이스는 두 번째 인덱스를 포함하지 않는 두 번째 인덱스까지 모든 요소를 선택한다.

제 8 절 리스트와 함수

루프를 작성하지 않고도 리스트를 빠르게 살펴볼 수 있는 리스트에 적용할 수 있는 내장함수가 많이 있다.

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print len(nums)
6
>>> print max(nums)
74
>>> print min(nums)
3
>>> print sum(nums)
154
>>> print sum(nums)/len(nums)
25
```

리스트 요소가 숫자일 때만, `sum()` 함수는 동작한다. `max()`, `len()`, 등등 다른 함수는 문자열 리스트나, 비교 가능한 다른 자료형(type) 리스트에 사용될 수 있다.

리스트를 사용해서, 앞서 작성한 프로그램을 다시 작성해서 사용자가 입력한 숫자 목록 평균을 계산한다.

우선 리스트 없이 평균을 계산하는 프로그램:

```
total = 0
count = 0
while ( True ) :
    inp = raw_input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print 'Average:', average
```

상기 프로그램에서, `count` 와 `sum` 변수를 사용해서 반복적으로 사용자가 숫자를 입력하면 값을 저장하고, 지금까지 사용자가 입력한 누적 합계를 계산한다.

단순하게, 사용자가 입력한 각 숫자를 기억하고 내장함수를 사용해서 프로그램 마지막에 합계와 갯수를 계산한다.

```
numlist = list()
while ( True ) :
    inp = raw_input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print 'Average:', average
```

루프가 시작되기 전 빈 리스트를 생성하고, 매번 숫자를 입력할 때마다 숫자를 리스트에 추가한다. 프로그램 마지막에 간단하게 리스트 총합을 계산하고, 평균을 산출하기 위해서 입력한 숫자 개수로 나눈다.

제 9 절 리스트와 문자열

문자열은 문자 순서(sequence)이고, 리스트는 값 순서(sequence)이다. 하지만 리스트 문자는 문자열과 같지는 않다. 문자열에서 리스트 문자로 변환하기 위해서, `list`를 사용한다.

```
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

list는 내장함수 이름이기 때문에, 변수명으로 사용하는 것을 피해야 한다. l을 사용하면 1 처럼 보이기 때문에 피한다. 그래서, t를 사용했다.

list 함수는 문자열을 각각의 문자로 쪼갠다. 문자열 단어로 쪼개려면, 분할 (split) 메쏘드를 사용할 수 있다.

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
>>> print t[2]
the
```

분할 (split) 메쏘드를 사용해서 문자열을 리스트 토큰으로 쪼개면, 인덱스 연산자('[]')를 사용하여 리스트의 특정 단어를 볼 수 있다.

옵션 인자로 단어 경계로 어떤 문자를 사용할 것인지 지정하는데 사용되는 구분자 (delimiter)를 활용하여 분할 (split) 메쏘드를 호출한다. 다음 예제는 구분자로 하이픈('-')을 사용한 사례다.

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

합병 (join) 메쏘드는 분할 (split) 메쏘드의 역이다. 문자열 리스트를 받아 리스트 요소를 연결한다. 합병 (join)은 문자열 메쏘드여서, 구분자를 호출하여 매개 변수로 넘길 수 있다.

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

상기의 경우, 구분자가 공백 문자여서 결합 (join) 메쏘드가 단어 사이에 공백을 넣는다. 공백없이 문자열을 결합하기 위해서, 구분자로 빈 문자열 ''을 사용한다.

제 10 절 라인 파싱하기(Parsing)

파일을 읽을 때 통상, 단지 전체 라인을 출력하는 것 말고 뭔가 다른 것을 하고자 한다. 종종 "흥미로운 라인을" 찾아서 라인을 파싱(parse)하여 흥미로운 *부분*을 찾고자 한다. "From "으로 시작하는 라인에서 요일을 찾고자 하면 어떨까?

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

이런 종류의 문제에 직면했을 때, 분할 (split) 메쏘드가 매우 효과적이다. 작은 프로그램을 작성하여 "From "으로 시작하는 라인을 찾고 분할 (split) 메쏘드로 파싱하고 라인의 흥미로운 부분을 출력한다.

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From ') : continue
    words = line.split()
    print words[2]
```

if 문의 축약 형태를 사용하여 continue 문을 if문과 동일한 라인에 놓았다. if 문 축약 형태는 continue 문을 들여쓰기를 다음 라인에 한 것과 동일하다.

프로그램은 다음을 출력한다.

```
Sat
Fri
Fri
Fri
    ...
```

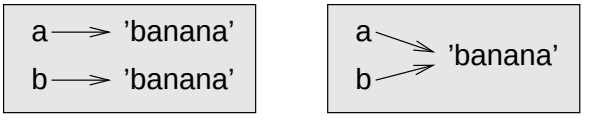
나중에, 매우 정교한 기술에 대해서 학습해서 정확하게 검색하는 비트(bit) 수준 정보를 찾아 내기 위해서 작업할 라인을 선택하고, 어떻게 해당 라인을 뽑아낼 것이다.

제 11 절 객체와 값(value)

다음 대입문을 실행하면,

```
a = 'banana'
b = 'banana'
```

a 와 b 모두 문자열을 참조하지만, 두 변수가 *동일한* 문자열을 참조하는지 알 수 없다. 두 가지 가능한 상태가 있다.



한 가지 경우는 a 와 b가 같은 값을 가지는 다른 두 객체를 참조하는 것이다. 두 번째 경우는 같은 객체를 참조하는 것이다.

두 변수가 동일한 객체를 참조하는지를 확인하기 위해서, is 연산자가 사용된다.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

이 경우, 파이썬은 하나의 문자열 객체를 생성하고 a 와 b 모두 동일한 객체를 참조한다.

하지만, 리스트 두 개를 생성할 때, 객체가 두 개다.

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

상기의 경우, 두 개의 리스트는 동등하다고 말할 수 있다. 왜냐하면 동일한 요소를 가지고 있기 때문이다. 하지만, 같은 객체는 아니기 때문에 동일하지는 않다. 두 개의 객체가 동일하다면, 두 객체는 또한 등등하다. 하지만, 동등하다고 해서 반듯이 동일하지는 않다.

지금까지 "객체(object)"와 "값(value)"을 구분 없이 사용했지만, 객체가 값을 가진다라고 말하는 것이 좀더 정확하다. `a = [1,2,3]` 을 실행하면, `a` 는 특별한 순서 요소값을 갖는 리스트 객체로 참조한다. 만약 또 다른 리스트가 동일한 요소를 가진다면, 그 리스트는 같은 값을 가진다고 말한다.

제 12 절 에일리어싱(Aliasing)

`a`가 객체를 참조하고, `b = a` 대입하다면, 두 변수는 동일한 객체를 참조한다.

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

객체와 변수의 연관짖는 것을 참조(reference)라고 한다. 상기의 경우 동일한 객체에 두 개의 참조가 있다.

하나 이상의 참조를 가진 객체는 한개 이상의 이름을 갖게 되어서, 객체가 에일리어스(aliased) 되었다고 한다.

만약 에일리어스된 객체가 변경 가능하면, 변화의 여파는 다른 객체에도 파급된다.

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

이와 같은 행동이 유용하기도 하지만, 오류를 발생시키기도 쉽다. 일반적으로, 변경가능한 객체(mutable object)로 작업할 때 에일리어싱을 피하는 것이 안전하다.

문자열 같이 변경 불가능한 객체에 에일리어싱은 그렇게 문제가 되지 않는다.

```
a = 'banana'
b = 'banana'
```

상기 예제에서, `a` 와 `b`가 동일한 문자열을 참조하든 참조하지 않든 거의 차이가 없다.

제 13 절 리스트 인수

리스트를 함수에 인자로 전달할 때, 함수는 리스트에 참조를 얻는다. 만약 함수가 리스트 매개 변수를 변경한다면, 호출자는 변경된 것을 보게된다. 예를 들어, `delete_head`**는 리스트에서 첫 요소를 제거한다.**

```
def delete_head(t):
    del t[0]
```

다음에 `delete_head` **함수가 사용된 예제가 있다.**

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```

매개 변수 `t`**와 변수** `letters`**는 동일한 객체에 대한 에일리어스(aliases)다.**

리스트를 변경하는 연산자와 신규 리스트를 생성하는 연산자를 구별하는 것이 중요하다. 예를 들어, 덧붙이기 `(append)` **메쏘드는 리스트를 변경하지만,** `+` **연산자는 신규 리스트를 생성한다.**

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1
[1, 2, 3]
>>> print t2
None

>>> t3 = t1 + [3]
>>> print t3
[1, 2, 3]
>>> t2 is t3
False
```

리스트를 변경하는 함수를 작성할 때, 이 차이는 매우 중요하다. 예를 들어, 다음 함수는 리스트의 머리 부문(head)을 삭제하지 않는다.

```
def bad_delete_head(t):
    t = t[1:]               # 틀림(WRONG)!
```

슬라이스 연산자는 새로운 리스트를 생성하고 대입문을 통해서 `t`**가 참조하게 하지만, 어떤 것도 인자로 전달된 리스트에는 영향도 주지 못한다.**

대안은 신규 리스트를 생성하고 반환하는 함수를 작성하는 것이다. 예를 들어, `tail`**은 리스트의 첫 요소를 제외하고 모든 요소를 반환한다.**

```
def tail(t):
    return t[1:]
```

상기 함수는 원 리시트를 변경하지는 않는다. 다음에 사용 예시가 있다.

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print rest
['b', 'c']
```

Exercise 8.1 리스트를 인자로 받아 리스트를 변경하여, 첫 번째 요소와 마지막 요소를 제거하고 None을 반환하는 chop 함수를 작성하게요.

그리고 나서, 리스트를 인자로 받아 처음과 마지막 요소를 제외한 나머지 요소를 새로운 리스트로 반환하는 middle 함수를 작성하세요.

제 14 절 디버깅

부주의한 리스트 사용이나 변경가능한 객체를 사용하는 경우 디버깅을 오래 할 수 있다. 다음에 일반적인 함정 유형과 회피하는 방법을 소개한다.

1. **대부분의 리스트 메쏘드는 인자를 변경하고, None을 반환한다. 이는 새로운 문자열을 반환하고 원 문자열은 그대로 두는 문자열의 경우와 정반대다.**

   다음과 같이 문자열 코드를 쓰는데 익숙해져 있다면,

   ```
   word = word.strip()
   ```

   다음과 같이 리스트 코드를 작성하고 싶은 유혹이 있을 것이다.

   ```
   t = t.sort()            # 틀림(WRONG)!
   ```

   정렬 (sort) **메쏘드는** None**을 반환하기 때문에, 리스트** t**로 수행한 다음 연산은 실패한다.**

   리스트 메쏘드와 연산자를 사용하기 전에, 문서를 주의깊게 읽고, 인터랙티브 모드에서 시험하는 것을 권한다. 리스트가 문자열과 같은 다른 순서(sequence)와 공유하는 메쏘드와 연산자는 `docs.python.org/lib/typesseq.html` **에 문서화되어 있다. 변경가능한 순서(sequence)에만 적용되는 메쏘드와 연산자는** `docs.python.org/lib/typesseq-mutable.html`**에 문서화되어 있다.**

2. **관용구를 선택하고 고수하라.**

   리스트와 관련된 문제 일부는 리스트를 가지고 할 수 있는 것이 너무 많다는 것이다. 예를 들어, 리스트에서 요소를 제거하기 위해서, pop**,** remove**,** del**, 혹은 슬라이스 대입(slice assignment)도 사용할 수 있다. 요소를 추가하기 위해서** 덧붙이기 (append) **메쏘드나** + **연산자를 사용할 수 있다. 하지만 다음이 맞다는 것을 잊지 마세요.**

   ```
   t.append(x)
   t = t + [x]
   ```

   하지만, 다음은 잘못됐다.

   ```
   t.append([x])           # 틀림(WRONG)!
   t = t.append(x)         # 틀림(WRONG)!
   t + [x]                 # 틀림(WRONG)!
   t = t + x               # 틀림(WRONG)!
   ```

인터랙티브 모드에서 각각을 연습해 보고 제대로 이해하고 있는지 확인해 보세요. 마지막 한개만 실행 오류를 하고, 다른 세가지는 모두 작동하지만, 잘못된 것을 수행함을 주목하세요.

3. 에일리어싱을 회피하기 위해서 사본 만들기.

   인자를 변경하는 정렬 (sort) 같은 메쏘드를 사용하지만, 원 리스트도 보관되길 원한다면, 사본을 만든다.

```
orig = t[:]
t.sort()
```

   상기 예제에서 원 리스트는 그대로 둔 상태로 새로 정렬된 리스트를 반환하는 내장함수 sorted를 사용할 수 있다. 하지만 이 경우에는, 변수명으로 sorted를 사용하는 것을 피해야 한다!

4. 리스트, 분할 (split), 파일

   파일을 읽고 파싱할 때, 프로그램이 중단될 수 있는 입력값을 마주할 수많은 기회가 있다. 그래서 파일을 훑어 "건초더미에서 바늘"을 찾는 프로그램을 작성할 때 사용한 가디언 패턴(guardian pattern)을 다시 살펴보는 것은 좋은 생각이다.

   파일 라인에서 요일을 찾는 프로그램을 다시 살펴보자.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

   각 라인을 단어로 나누었기 때문에, startswith를 사용하지 않고, 라인에 관심있는 단어가 있는지 살펴보기 위해서 단순하게 각 라인의 첫 단어를 살펴본다. 다음과 같이 continue 문을 사용해서 "From"이 없는 라인을 건너 뛴다.

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    if words[0] != 'From' : continue
    print words[2]
```

   프로그램이 훨씬 간단하고, 파일 끝에 있는 새줄(newline)을 제거하기 위해 rstrip을 사용할 필요도 없다. 하지만, 더 좋아졌는가?

```
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

   작동하는 것 같지만, 첫줄에 Sat 를 출력하고 나서 역추적 오류(traceback error)로 프로그램이 정상 동작에 실패한다. 무엇이 잘못되었을까? 어딘가 엉망이 된 데이터가 있어 우아하고, 총명하며, 매우 파이썬스러운 프로그램을 망가뜨린건가?

오랜 동안 프로그램을 응시하고 머리를 짜내거나, 다른 사람에게 도움을 요청할 수 있지만, 빠르고 현명한 접근법은 print문을 추가하는 것이다. print문을 넣는 가장 좋은 장소는 프로그램이 동작하지 않는 라인 앞이 적절하고, 프로그램 실패를 야기할 것 같은 데이터를 출력한다.

이 접근법이 많은 라인을 출력하지만, 즉석에서 문제에 대해서 손에 잡히는 단서는 최소한 준다. 그래서 words를 출력하는 출력문을 5번째 라인 앞에 추가한다. "Debug:"를 접두어로 라인에 추가하여, 정상적인 출력과 디버그 출력을 구분한다.

```
for line in fhand:
    words = line.split()
    print 'Debug:', words
    if words[0] != 'From' : continue
    print words[2]
```

프로그램을 실행할 때, 많은 출력결과가 스크롤되어 화면 위로 지나간다. 마지막에 디버그 결과물과 역추적(traceback)을 보고 역추적(traceback) 바로 앞에서 무슨 일이 생겼는지 알 수 있다.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

각 디버그 라인은 리스트 단어를 출력하는데, 라인을 분할 (split)해서 단어로 만들 때 얻어진다. 프로그램이 실패할 때 리스트 단어는 비었다 '[]'. 텍스트 편집기로 파일을 열어 살펴보면 그 지점은 다음과 같다.

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

프로그램이 빈 라인을 만났을 때, 오류가 발생한다. 물론, 빈 라인은 '0' 단어 ("zero words")다. 프로그램을 작성할 때, 왜 이것을 생각하지 못했을까? 첫 단어(word[0])가 "From"과 일치하는지 코드가 점검할 때, "인덱스 범위 오류(index out of range)"가 발생한다.

물론, 첫 단어가 없다면 첫 단어 점검을 회피하는 가디언 코드(guardian code)를 삽입하기 최적 장소이기는 하다. 코드를 보호하는 방법은 많다. 첫 단어를 살펴보기 전에 단어의 갯수를 확인하는 방법을 택한다.

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
```

```
# print 'Debug:', words
if len(words) == 0 : continue
if words[0] != 'From' : continue
print words[2]
```

변경한 코드가 실패해서 다시 디버그할 경우를 대비해서, `print`문을 제거하는 대신에 `print`문을 주석 처리한다. 그리고 나서, 단어가 '0' 인지를 살펴보고 만약 '0' 이면, 파일 다음 라인으로 건너뛰도록 `continue`문을 사용하는 가디언 문장(guardian statement)을 추가한다.

두 개 `continue`문이 "흥미롭고" 좀더 처리가 필요한 라인 집합을 정제하도록 돕는 것으로 생각할 수 있다. 단어가 없는 라인은 "흥미 없어서" 다음 라인으로 건너뛴다. 첫 단어에 "From"이 없는 라인도 "흥미 없어서" 건너뛴다.

변경된 프로그램이 성공적으로 실행되어서, 아마도 올바르게 작성된 것으로 보인다. 가디언 문장(guardian statement)이 `words[0]`가 정상작동할 것이라는 것을 확인해 주지만, 충분하지 않을 수도 있다. 프로그램을 작성할 때, "무엇이 잘못 될 수 있을까?"를 항상 생각해야만 한다.

Exercise 8.2 상기 프로그램의 어느 라인이 여전히 적절하게 보호되지 않은지를 생각해 보세요. 텍스트 파일을 구성해서 프로그램이 실패하도록 만들 수 있는지 살펴보세요. 그리고 나서, 프로그램을 변경해서 라인이 적절하게 보호되게 하고, 새로운 텍스트 파일을 잘 다룰 수 있도록 시험하세요.

Exercise 8.3 두 `if` 문 없이, 상기 예제의 가디언 코드(`guardian code`)를 다시 작성하세요. 대신에 단일 `if`문과 `and` 논리 연산자를 사용하는 복합 논리 표현식을 사용하세요.

제 15 절 용어정의

에일리어싱(aliasing): 하나 혹은 그 이상의 변수가 동일한 객체를 참조하는 상황.

구분자(delimiter): 문자열이 어디서 분할되어져야 할지를 표기하기 위해서 사용되는 문자나 문자열.

요소(element): 리스트 혹은 다른 순서(sequence) 값의 하나로 항목(item)이라고도 한다.

동등한(equivalent): 같은 값을 가짐.

인덱스(index): 리스트의 요소를 지칭하는 정수 값.

동일한(identical): 동등을 함축하는 같은 객체임.

리스트(list): 순서(sequence) 값.

리스트 운행법(list traversal): 리스트의 각 요소를 순차적으로 접근함.

중첩 리스트(nested list): 또 다른 리스트의 요소인 리스트.

객체(object): 변수가 참조할 수 있는 무엇. 객체는 자료형(type)과 값(value)을 가진다.

참조(reference): 변수와 값의 연관.

제 16 절 연습문제

Exercise 8.4 `www.py4inf.com/code/romeo.txt`에서 파일 사본을 다운로드 받으세요.

`romeo.txt` 파일을 열어, 한 줄씩 읽어들이는 프로그램을 작성하세요. 각 라인마다 분할 `(split)` 함수를 사용하여 라인을 단어 리스트로 쪼개세요.

각 단어마다, 단어가 이미 리스트에 존재하는지를 확인하세요. 만약 단어가 리스트에 없다면, 리스트에 새 단어로 추가하세요.

프로그램이 완료되면, 알파벳 순으로 결과 단어를 정렬하고 출력하세요.

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

Exercise 8.5 전자우편 데이터를 읽어 들이는 프로그램을 작성하세요. "From"으로 시작하는 라인을 발견했을 때, 분할 `(split)` 함수를 사용하여 라인을 단어로 쪼개세요. "From" 라인의 두번째 단어, 누가 메시지를 보냈는지에 관심이 있다.

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

"From" 라인을 파싱하여 각 "From"라인의 두번째 단어를 출력한다. 그리고 나서, "From:"이 아닌 "From"라인 갯수를 세고, 끝에 갯수를 출력한다.

여기 몇 줄을 삭제한 출력 예시가 있다.

```
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

Exercise 8.6 사용자가 숫자 리스트를 입력하고, 입력한 숫자 중에 최대값과 최소값을 출력하고 사용자가 ”done”을 입력할 때 종료하는 프로그램을 다시 작성하세요. 사용자가 입력한 숫자를 리스트에 저장하고, `max()` 과 `min()` 함수를 사용하여 루프가 끝나면, 최대값과 최소값을 출력하는 프로그램을 작성하세요.

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```

9 장

딕셔너리(Dictionaries)

딕셔너리(dictionary)는 리스트 같지만 좀더 일반적이다. 리스트에서 위치(인덱스)는 정수이지만, 딕셔너리에서는 인덱스는 임의 자료형(type)이 될 수 있다.

딕셔너리를 인덱스 집합(키(keys)라고 부름)에서 값(value) 집합으로 사상(mapping)하는 것으로 생각할 수 있다. 각각의 키는 값에 대응한다. 키와 값을 연관시키는 것을 키-값 페어(key-value pair)라고 부르고, 종종 항목(item)으로도 부른다.

한 예제로, 영어 단어에서 스페인 단어에 매핑되는 사전을 만들 것이다. 키와 값은 모두 문자열이다.

`dict` **함수는 항목이 전혀 없는 사전을 신규로 생성한다.** `dict`**는 내장함수명이어서, 변수명으로 사용하는 것을 피해야 한다.**

```
>>> eng2sp = dict()
>>> print eng2sp
{}
```

구불구불한 괄호 `{}`**는 빈 딕셔너리를 나타낸다. 딕셔너리에 항목을 추가하기 위해서 꺾쇠 괄호를 사용한다.**

```
>>> eng2sp['one'] = 'uno'
```

상기 라인은 키 `'one'`**에서 값** `'uno'`**를 매핑하는 항목을 생성한다. 딕셔너리를 다시 출력하면, 키와 값 사이에 콜론(:)을 가진 키-값 페어(key-value pair)를 볼 수 있다.**

```
>>> print eng2sp
{'one': 'uno'}
```

출력 형식이 또한 입력 형식이다. 예를 들어, 세개 항목을 가진 신규 딕셔너리를 생성할 수 있다.

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

`eng2sp`**을 출력하면, 놀랄 것이다.**

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

키-값 페어(key-value pair) 순서가 같지 않다. 사실 동일한 사례를 여러분의 컴퓨터에서 입력하면, 다른 결과를 얻게 된다. 일반적으로, 딕셔너리 항목 순서는 예측 가능하지 않다.

딕셔너리 요소가 정수 인덱스로 색인되지 않아서 문제되지는 않는다. 대신에, 키를 사용해서 상응하는 값을 찾을 수 있다.

```
>>> print eng2sp['two']
'dos'
```

'two' 키는 항상 값 'dos'에 상응되어서 딕셔너리 항목 순서는 문제가 되지 않는다.

만약 키가 딕셔너리에 존재하지 않으면, 예외 오류가 발생한다.

```
>>> print eng2sp['four']
KeyError: 'four'
```

len 함수를 딕셔너리에 사용해서, 키-값 페어(key-value pair) 항목 개수를 반환한다.

```
>>> len(eng2sp)
3
```

in 연산자도 딕셔너리에 작동되는데, 어떤 것이 딕셔너리 *키(key)*에 있는지 알려준다. (값(value)으로 나타내는 것은 충분히 좋지는 않다.)

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

딕셔너리에 무엇이 값으로 있는지 확인하기 위해서, values 메쏘드를 해서 리스트로 값을 반환받고 나서 in 연산자를 사용하여 확인한다.

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

in 연산자는 리스트와 딕셔너리에 각기 다른 알고리즘을 사용한다. 리스트에 대해서 선형 검색 알고리즘을 사용한다. 리스트가 길어짐에 따라 검색 시간은 리스트 길이에 비례하여 길어진다. 딕셔너리에 대해서 파이썬은 해쉬 테이블(hash table)로 불리는 놀라운 특성을 가진 알고리즘을 사용한다. 얼마나 많은 항목이 딕셔너리에 있는지에 관계없이 in 연산자는 대략 동일한 시간이 소요된다. 왜 해쉬 함수가 마술 같은지에 대해서는 설명하지 않지만, wikipedia.org/wiki/Hash_table 에서 좀더 많은 정보를 얻을 수 있다.

Exercise 9.1 words.txt 단어를 읽어서 딕셔너리에 키로 저장하는 프로그램을 작성하세요. 값이 무엇이든지 상관없습니다. 딕셔너리에 문자열을 확인하는 가장 빠른 방법으로 in 연산자를 사용할 수 있습니다.

제 1 절 계수기(counter) 집합으로서 딕셔너리

문자열이 주어진 상태에서, 각 문자가 얼마나 나타나는지를 센다고 가정합시다. 몇 가지 방법이 아래에 있습니다.

1. 26개 변수를 알파벳 문자 각각에 대해 생성한다. 그리고 나서 아마도 연쇄 조건문을 사용하여 문자열을 훑고 해당 계수기(counter)를 하나씩 증가한다.
2. 26개 요소를 가진 리스트를 생성한다. 내장함수 `ord`를 사용해서 각 문자를 숫자로 변환한다. 리스트 안에 인덱스로 숫자를 사용해서 적당한 계수기(counter)를 증가한다.
3. 키(key)로 문자, 계수기(counter)로 해당 값(value)을 갖는 딕셔너리를 생성한다. 처음 문자를 만나면, 딕셔너리에 항목으로 추가한다. 추가한 후에는 기존 항목 값을 증가한다.

상기 3개 옵션은 동일한 연산을 수행하지만, 각각은 다른 방식으로 연산을 구현한다.

구현(implementation)은 연산(computation)을 수행하는 방법이다. 어떤 구현 방법이 다른 것 보다 낫다. 예를 들어, 딕셔너리 구현의 장점은 사전에 문자열에서 어떤 문자가 나타날지 몰라도 된다. 다만 나타날 문자에 대한 공간만 준비하면 된다는 것이다.

다음에 딕셔너리로 구현한 코드가 있다.

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print d
```

계수기(counter) 혹은 빈도에 대한 통계 용어인 히스토그램(histogram)을 효과적으로 계산해보자.

`for` 루프는 문자열을 훑는다. 매번 루프를 반복할 때마다 딕셔너리에 문자 `c`가 없다면, 키 `c`와 초기값 1을 가진 새로운 항목을 생성한다. 문자 `c`가 이미 딕셔너리에 존재한다면, `d[c]`을 증가한다.

다음 프로그램 실행 결과가 있다.

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

히스토그램은 문자 `'a'`, `'b'`는 1회, `'o'`는 2회 등등 나타남을 보여준다.

딕셔너리에는 키와 디폴트(default) 값을 갖는 `get` 메쏘드가 있다. 딕셔너리에 키가 나타나면, `get` 메쏘드는 해당 값을 반환하고, 해당 값이 없으면 디폴트 값을 반환한다. 예를 들어,

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print counts.get('jan', 0)
100
>>> print counts.get('tim', 0)
0
```

get 메쏘드를 사용해서 상기 히스토그램 루프를 좀더 간결하게 작성할 수 있다. get 메쏘드는 딕셔너리에 키가 존재하지 않는 경우를 자동적으로 다루기 때문에, if문을 없애 4줄을 1줄로 줄일 수 있다.

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c,0) + 1
print d
```

계수기(counter) 루프를 단순화하려고 get메쏘드를 사용하는 것은 파이썬에서 흔히 사용되는 "숙어(idiom)"가 되고, 책 후반부까지 많이 사용할 것이다. 시간을 가지고서 잠시 if 문과 in 연산자를 사용한 루프와 get메쏘드를 사용한 루프를 비교해 보세요. 동일한 연산을 수행하지만, 하나가 더 간결한다.

제 2 절 딕셔너리와 파일

딕셔너리의 흔한 사용법 중의 하나는 텍스트로 작성된 파일에 단어 빈도를 세는 것이다. **http://shakespeare.mit.edu/Tragedy/romeoandjuliet/romeo_juliet.2.2.html** 사이트 덕분에 *로미오와 쥴리엣(Romeo and Juliet)* 텍스트 파일에서 시작합시다.

처음 연습으로 구두점이 없는 짧고 간략한 텍스트 버젼을 사용한다. 나중에 구두점이 포함된 전체 텍스트로 작업할 것이다.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

파일 라인을 읽고, 각 라인을 단어 리스트로 쪼개고, 루프를 돌려 사전을 이용하여 각 단어의 빈도수를 세는 파이썬 프로그램을 작성한다.

두 개의 for 루프를 사용한다. 외곽 루프는 파일 라인을 읽고, 내부 루프는 특정 라인의 단어 각각에 대해 반복한다. 하나의 루프는 *외곽* 루프가 되고, 또 다른 루프는 *내부* 루프가 되어서 중첩 루프(nested loops)라고 불리는 패턴 사례다.

외곽 루프가 한번 반복을 할 때마다 내부 루프는 모든 반복을 수행하기 때문에 내부 루프는 "좀더 빨리" 반복을 수행하고 외곽 루프는 좀더 천천히 반복을 수행하는 것으로 생각할 수 있다.

두 중첩 루프의 조합이 입력 파일의 모든 라인에 있는 모든 단어의 빈도를 계수(count)하도록 보증합니다.

```
fname = raw_input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print 'File cannot be opened:', fname
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print counts
```

프로그램을 실행하면, 정렬되지 않은 해쉬 순서로 모든 단어의 빈도수를 출력합니다. `romeo.txt` **파일은** **`www.py4inf.com/code/romeo.txt`**에서 다운로드 가능하다.

```
python count1.py
Enter the file name: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

가장 높은 빈도 단어와 빈도수를 찾기 위해서 딕셔너리를 훑는 것이 불편하다. 좀더 도움이 되는 출력결과를 만들려고 파이썬 코드를 추가하자.

제 3 절 반복과 딕셔너리

`for`**문에 순서(sequence)로서 딕셔너리를 사용한다면, 딕셔너리 키를 훑는다. 루프는 각 키와 해당 값을 출력한다.**

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    print key, counts[key]
```

출력은 다음과 같다.

```
jan 100
chuck 1
annie 42
```

다시 한번, 키는 특별한 순서가 없다.

이 패턴을 사용해서 앞서 기술한 다양한 루프 숙어를 구현한다. 예를 들어, 딕셔너리에서 10 보다 큰 값을 가진 항목을 모두 찾고자 한다면, 다음과 같이 코드를 작성한다.

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    if counts[key] > 10 :
        print key, counts[key]
```

for 루프는 딕셔너리 *키(keys)*를 반복한다. 그래서, 인덱스 연산자를 사용해서 각 키에 상응하는 *값(value)*을 가져와야 한다. 여기 출력값이 있다.

```
jan 100
annie 42
```

10 이상 값만 가진 항목만 볼 수 있다.

알파벳 순으로 키를 출력하고자 한다면, 딕셔너리 객체의 keys 메쏘드를 사용해서 딕셔너리 키 리스트를 생성한다. 그리고 나서 리스트를 정렬하고, 정렬된 리스트를 루프 돌리고, 아래와 같이 정렬된 순서로 키/값 페어(key/value pair)를 출력한다.

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = counts.keys()
print lst
lst.sort()
for key in lst:
    print key, counts[key]
```

다음에 출력결과가 있다.

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

처음에 keys 메쏘드로부터 얻은 정렬되지 않은 키 리스트가 있고, 그리고 나서 for 루프로 정렬된 키/값 페어(key/value pair)가 있다.

제 4 절 고급 텍스트 파싱

romeo.txt 파일을 사용한 상기 예제에서, 수작업으로 모든 구두점을 제거해서 가능한 단순하게 만들었다. 실제 텍스트는 아래 보여지는 것처럼 많은 구두점이 있다.

```
But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,
```

파이썬 split 함수는 공백을 찾고 공백으로 구분되는 토큰으로 단어를 처리하기 때문에, "soft!" 와 "soft"는 다른 단어로 처리되고 각 단어에 대해서 별도 딕셔너리 항목을 생성한다.

파일에 대문자가 있어서, "who"와 "Who"를 다른 단어, 다른 빈도수를 가진 것으로 처리한다.

lower, punctuation, translate 문자열 메쏘드를 사용해서 상기 문제를 해결할 수 있다. translate 메쏘드가 가장 적합하다. translate 메쏘드에 대한 문서가 다음에 있다.

```
string.translate(s, table[, deletechars])
```

(만약 존재한다면) deletechars 에 있는 모든 문자를 삭제한다. 그리고 나서 순서수(ordinal)로 색인된 각 문자를 번역하는 256-문자열 테이블(table)을 사용해서 문자를 번역한다. 만약 테이블이 None 이면, 문자 삭제 단계만 수행된다.

table을 명세하지는 않을 것이고, deletechars 매개변수를 사용해서 모든 구두점을 삭제할 것이다. 파이썬이 "구두점"으로 간주하는 문자 리스트를 출력하게 할 것이다.

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

프로그램을 다음과 같은 수정을 한다.

```
import string                                           # 신규 코드

fname = raw_input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print 'File cannot be opened:', fname
    exit()

counts = dict()
for line in fhand:
    line = line.translate(None, string.punctuation)     # 신규 코드
    line = line.lower()                                 # 신규 코드
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print counts
```

translate 메쏘드를 사용해서 모든 구두점을 제거했고, lower 메쏘드를 사용해서 라인을 소문자로 수정했다. 나머지 프로그램은 변경된게 없다. 파이썬 2.5 이전 버젼에는 translate 메쏘드가 첫 매개변수로 None을 받지 않아서 translate 메쏘드를 호출하기 위해서 다음 코드를 사용하세요.

```
print a.translate(string.maketrans(' ',' '), string.punctuation
```

"파이썬 예술(Art of Python)" 혹은 "파이썬스럽게 생각하기(Thinking Pythonically)"를 배우는 일부분은 파이썬은 흔한 자료 분석 문제에 대해서 내장 기능을 가지고 있는 것을 깨닫는 것이다. 시간이 지남에 따라, 충분한 예제 코드를 보고 충분한 문서를 읽는다. 작업을 편하게 할 수 있게 이미 다른 사람이

작성한 코드가 존재하는지를 살펴보기 위해서 어디를 찾아봐야 하는지도 알게 될 것이다.

다음은 출력결과의 축약 버젼이다.

```
Enter the file name: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

출력결과는 여전히 다루기 힘들어 보입니다. 파이썬을 사용해서 정확히 찾고자 는 하는 것을 찾았으나 파이썬 튜플(tuples)에 대해서 학습할 필요성을 느껴진다. 튜플을 학습할 때, 다시 이 예제를 살펴볼 것이다.

제 5 절 디버깅

점점 더 큰 데이터로 작업함에 따라, 수작업으로 데이터를 확인하거나 출력을 통해서 디버깅을 하는 것이 어려울 수 있다. 큰 데이터를 디버깅하는 몇가지 방법이 있다.

입력값을 줄여라(Scale down the input):] 가능하면, 데이터 크기를 줄여라. 예를 들어, 프로그램이 텍스트 파일을 읽는다면, 첫 10줄로 시작하거나, 찾을 수 있는 작은 예제로 시작하라. 데이터 파일을 편집하거나, 프로그램을 수정해서 첫 `n` 라인만 읽도록 프로그램을 변경하라.

오류가 있다면, `n`을 줄여서 오류를 재현하는 가장 작은 값으로 만들어라. 그리고 나서, 오류를 찾고 수정해 나감에 따라 점진적으로 늘려나가라.

요약값과 자료형을 확인하라(Check summaries and types): 전체 데이터를 출력하고 검증하는 대신에 데이터를 요약하여 출력하는 것을 생각하라: 예를 들어, 딕셔너리 항목의 숫자 혹은 리스트 숫자의 총계

실행 오류(runtime errors)의 일반적인 원인은 올바른 자료형(right type)이 아니기 때문이다. 이런 종류의 오류를 디버깅하기 위해서, 종종 값의 자료형을 출력하는 것으로 충분하다.

자가 진단 작성(Write self-checks): 종종 오류를 자동적으로 검출하는 코드를 작성한다. 예를 들어, 리스트 숫자의 평균을 계산한다면, 결과값은 리스트의 가장 큰 값보다 클 수 없고, 가장 작은 값보다 작을 수 없다는 것을 확인할 수 있다. "완전히 비상식적인" 결과를 탐지하기 때문에 "건전성 검사(sanity check)"라고 부른다.

또 다른 검사법은 두가지 다른 연산의 결과를 비교해서 일치하는지 살펴보는 것이다. "일치성 검사(consistency check)"라고 부른다.

고급 출력(Pretty print the output): 디버깅 출력결과를 서식화하는 것은 오류 발견을 용이하게 한다.

다시 한번, 발판(scaffolding)을 만드는데 들인 시간이 디버깅에 소비되는 시간을 줄일 수 있다.

제 6 절 용어정의

딕셔너리(dictionary): 키(key)에서 해당 값으로 매핑(mapping)

해쉬테이블(hashtable): 파이썬 딕셔너리를 구현하기 위해 사용된 알고리즘

해쉬 함수(hash function): 키에 대한 위치를 계산하기 위해서 해쉬테이블에서 사용되는 함수

히스토그램(histogram): 계수기(counter) 집합.

구현(implementation): 연산(computation)을 수행하는 방법

항목(item): 키-값 페어(key-value pair)에 대한 또 다른 이름.

키(key): 키-값 페어(key-value pair)의 첫번째 부분으로 딕셔너리에 나타나는 객체.

키-값 페어(key-value pair): 키에서 값으로 매핑 표현.

룩업(lookup): 키를 가지고 해당 값을 찾는 딕셔너리 연산.

중첩 루프(nested loops): 루프 "내부"에 하나 혹은 그 이상의 루프가 있음. 외곽 루프가 1회 실행될 때, 내부 루프는 전체 반복을 완료함.

값(value): 키-값 페어(key-value pair)의 두번째 부분으로 딕셔너리에 나타나는 객체. 앞에서 사용한 단어 "값(value)" 보다 더 구체적이다.

제 7 절 연습문제

Exercise 9.2 커밋(commit)이 무슨 요일에 수행되었는지에 따라 전자우편 메세지를 구분하는 프로그램을 작성하세요. "From"으로 시작하는 라인을 찾고, 3번째 단어를 찾아서 요일별 횟수를 계수(count)하여 저장하세요. 프로그램 끝에 딕셔너리 내용을 출력하세요. (순서는 문제가 되지 않습니다.)

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

Sample Execution:
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

Exercise 9.3 전자우편 로그(log)를 읽고, 히스토그램을 생성하는 프로그램을 작성하세요. 딕셔너리를 사용해서 전자우편 주소별로 얼마나 많은 전자우편이 왔는지를 계수(count)하고 딕셔너리를 출력합니다.

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
```

```
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

Exercise 9.4 상기 프로그램에 누가 가장 많은 전자우편 메시지를 가졌는지 알아내는 코드를 추가하세요.

결국, 모든 데이터를 읽고, 딕셔너리를 생성한다. 최대 루프(장 7.2 참조)를 사용해서 딕셔너리를 훑어서 누가 가장 많은 전자우편 메시지를 갖는지, 그리고 그 사람이 얼마나 많은 메시지를 갖는지 출력한다.

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

Exercise 9.5 다음 프로그램은 주소 대신에 도메인 이름을 기록한다. 누가 메일을 보냈는지 대신(즉, 전체 전자우편 주소) 메시지가 어디에서부터 왔는지 출처를 기록한다. 프로그램 마지막에 딕셔너리 내용을 출력한다.

```
python schoolcount.py
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

10 장

튜플(Tuples)

제 1 절 튜플은 불변이다.

튜플(tuple)[1]은 리스트와 마찬가지로 순서(sequence) 값이다. 튜플에 저장된 값은 임의 자료형(type)이 될 수 있고, 정수로 색인 된다. 중요한 차이점은 튜플은 불변(immutable)하다는 것이다. 튜플은 또한 비교 가능(comparable)하고 해쉬형(hashable)이다. 따라서, 리스트 값을 정렬할 수 있고, 파이썬 딕셔너리 키 값으로 튜플을 사용할 수 있다.

구문론적으로, 튜플은 콤마로 구분되는 리스트 값이다.

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

꼭 필요하지는 않지만, 파이썬 코드를 봤을 때, 가독성을 높여 튜플을 빠르게 알아볼 수 있도록 괄호로 튜플을 감싸는 것이 일반적이다.

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

단일 요소를 가진 튜플을 생성하기 위해서 마지막 콤마를 포함해야 한다.

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

콤마가 없는 경우 파이썬에서는 `('a')`을 괄호를 가진 문자열 표현으로 간주하여 문자열로 평가한다.

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

튜플을 구축하는 다른 방법은 내장함수 `tuple`을 사용하는 것이다. 인자가 없는 경우, 빈 튜플을 생성한다.

[1] 재미난 사실: 단어 "튜플(tuple)"은 가변 길이 (한배, 두배, 세배, 네배, 다섯배, 여섯배, 일곱배 등) 숫자열에 붙여진 이름에서 유래한다.

```
>>> t = tuple()
>>> print t
()
```

만약 인자가 문자열, 리스트 혹은 튜플 같은 순서(sequence)인 경우, `tuple`에 호출한 결과는 요소 순서(sequence)를 가진 튜플이 된다.

```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

튜플(`tuple`)이 생성자 이름이기 때문에 변수명으로 튜플 사용을 피해야 한다.

대부분의 리스트 연산자는 튜플에서도 사용 가능하다. 꺾쇠 연산자가 요소를 색인한다.

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

그리고, 슬라이스 연산자(slice operator)는 요소 범위를 선택한다.

```
>>> print t[1:3]
('b', 'c')
```

하지만, 튜플 요소 중 하나를 변경하고 하면, 오류가 발생한다.

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

튜플 요소를 변경할 수는 없지만, 튜플을 다른 튜플로 교체는 할 수 있다.

```
>>> t = ('A',) + t[1:]
>>> print t
('A', 'b', 'c', 'd', 'e')
```

제 2 절 튜플 비교하기

비교 연산자는 튜플과 다른 순서(sequence)와 함께 쓸 수 있다. 파이썬은 각 순서(sequence)에서 비교를 첫 요소부터 시작한다. 만약 두 요소가 같다면, 다음 요소 비교를 진행하며 서로 다른 요소를 찾을 때까지 계속한다. 후속 요소가 아무리 큰 값이라고 하더라도 비교 고려대상에서 제외된다.

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

정렬 (`sort`) **함수도 동일한 방식으로 동작한다. 첫 요소를 먼저 정렬하지만, 동일한 경우 두 번째 요소를 정렬하고, 그 후속 요소를 동일한 방식으로 정렬한다.**

이 기능이 다음 DSU라고 불리는 패턴이 된다.

데코레이트 (Decorate) : 순서(sequence)에서 요소 앞에 하나 혹은 그 이상의 키를 가진 튜플 리스트를 구축해서 순서(sequence)를 장식한다.

정렬 (Sort) : 파이썬 내장 함수 `sort`를 사용한 튜플 리스트를 정렬한다.

언데코레이트 (Undecorate) : 순서(sequence)의 정렬된 요소만 추출하여 장식을 지웁니다.

예를 들어, 단어 리스트가 있고 가장 긴 단어부터 가장 짧은 단어 순으로 정렬한다고 가정하자.

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
   t.append((len(word), word))

t.sort(reverse=True)

res = list()
for length, word in t:
    res.append(word)

print res
```

첫번째 루프는 튜플 리스트를 생성하고, 각 튜플은 단어 앞에 길이 정보를 가진다.

정렬(`sort`)함수는 첫번째 요소, 길이를 우선 비교하고, 동률일 경우에만 두 번째 요소를 고려한다. 정렬(`sort`) 함수의 인자 `reverse=True`는 내림차순으로 정렬한다는 의미다.

두 번째 루프는 튜플 리스트를 운행하여 훑고, 길이에 따라 내림차순으로 단어 리스트를 생성한다. 그래서, 5 문자 단어는 역 알파벳 순으로 정렬되어 있다. 다음 리스트에서 "what"이 "soft" 보다 앞에 나타난다.

프로그램의 출력은 다음과 같다.

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

물론, 파이썬 리스트로 변환하여 내림차순으로 정렬된 문장은 시적인 의미를 많이 잃어버렸다.

제 3 절 튜플 대입(Tuple Assignment)

파이썬 언어의 독특한 구문론적인 기능중의 하나는 대입문의 왼편에 튜플을 놓을 수 있다는 것이다. 왼편이 순서(sequence)인 경우 한번에 하나 이상의 변수에 대입할 수 있다.

다음 예제에서, 순서(sequence)인 두개 요소를 갖는 리스트가 있다. 하나의 문장으로 순서(sequence)의 첫번째와 두번째 요소를 변수 `x`와 `y`에 대입한다.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

마술이 아니다. 파이썬은 *대략* 튜플 대입 구문을 다음과 같이 해석한다.[2]

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

스타일적으로 대입문 왼편에 튜플을 사용할 때, 괄호를 생략한다. 하지만 다음은 동일하게 적합한 구문이다.

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

튜플 대입문을 사용하는 특히 똑똑한 응용사례는 단일 문장으로 두 변수 값을 교환(swap)하는 것이다.

```
>>> a, b = b, a
```

양쪽 문장이 모두 튜플이지만, 왼편은 튜플 변수이고 오른편은 튜플 표현식이다. 오른편 각각의 값이 왼편 해당 변수에 대입된다. 대입이 이루어지기 전에 오른편의 모든 표현식이 평가된다.

왼편의 변수 갯수와 오른편의 값의 갯수는 동일해야 한다.

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

좀더 일반적으로, 오른편은 임의 순서(문자열, 리스트 혹은 튜플)가 될 수 있다. 예를 들어, 전자우편 주소를 사용자 이름과 도메인으로 분할하기 위해서 다음과 같이 프로그램을 작성할 수 있다.

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

[2] 파이썬은 구문을 문자 그대로 해석하지는 않는다. 예를 들어, 동일한 것을 딕셔너리로 작성한다면, 기대한 것처럼 동작하지는 않는다.

분할 (split) 함수로부터 반환되는 값은 두개 요소를 가진 리스트다. 첫번째 요소는 uname에 두번째 요소는 domain에 대입된다.

```
>>> print uname
monty
>>> print domain
python.org
```

제 4 절 딕셔너리와 튜플

딕셔너리에는 튜플 리스트를 반환하는 items 메쏘드가 있다. 각 튜플은 키-값 페어(key-value pair)다.[3]

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = d.items()
>>> print t
[('a', 10), ('c', 22), ('b', 1)]
```

딕셔너리로부터 기대했듯이, 항목은 특별한 순서가 없다.

하지만 튜플 리스트는 리스트여서 비교가 가능하기 때문에, 튜플 리스트를 정렬할 수 있다. 딕셔너리를 튜플 리스트로 변환하는 것이 키로 정렬된 딕셔너리 내용을 출력할 수 있게 한다.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = d.items()
>>> t
[('a', 10), ('c', 22), ('b', 1)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

새로운 리스트는 키 값으로 오름차순 알파벳 순으로 정렬된다.

제 5 절 딕셔너리로 다중 대입

items 함수, 튜플 대입, for문을 조합해서, 단일 루프로 딕셔너리의 키와 값을 운행하여 훑는 멋진 코드 패턴을 만들 수 있다.

```
for key, val in d.items():
    print val, key
```

상기 루프에는 두개의 반복 변수(iteration variables)가 있다. items 함수가 튜플 리스트를 반환하고, key, val는 튜플 대입하여 딕셔너리에 있는 각각의 키-값 페어(key-value pair)를 성공적으로 반복한다.

매번 루프를 반복할 때마다, key와 value는 딕셔너리(여전히 해쉬 순으로 되어 있음)의 다음 키-값 페어(key-value pair)로 진행한다.

[3] 파이썬 3.0으로 가면서 살짝 달라졌다.

루프의 출력결과는 다음과 같다.

```
10 a
22 c
1 b
```

다시 한번 해쉬 키 순서다. (즉, 특별한 순서가 없다.)

두 기술을 조합하면, 딕셔너리 내용을 키-값 페어(key-value pair)에 저장된 *값*의 순서로 정렬하여 출력할 수 있다.

이것을 수행하기 위해서, 각 튜플이 `(value, key)` **형태인 튜플 리스트를 작성한다.** `items` **메쏘드를 사용하여 리스트** `(key, value)` **튜플을 만든다. 하지만 이번에는 키가 아닌 값으로 정렬한다. 키-값(key-value) 튜플 리스트를 생성하면, 역순으로 리스트를 정렬하고 새로운 정렬 리스트를 출력하는 것은 쉽다.**

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
...     l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

조심스럽게 각 튜플 첫번째 요소로 값(value)을 갖는 튜플 리스트를 생성했다. 튜플 리스트를 정렬하여 값으로 정렬된 딕셔너리가 생성되었다.

제 6 절 가장 빈도수가 높은 단어

로미오와 쥴리엣 2장 2막 **텍스트 파일로 다시 돌아와서, 텍스트에서 가장 빈도수가 높은 단어를 10개를 출력하기 위해서 상기 학습한 기법을 사용하여 프로그램을 보강해보자.**

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
    line = line.translate(None, string.punctuation)
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

# Sort the dictionary by value
lst = list()
```

```
for key, val in counts.items():
    lst.append( (val, key) )

lst.sort(reverse=True)

for key, val in lst[:10] :
    print key, val
```

파일을 읽고 각 단어를 문서의 단어 빈도수에 매핑(사상)하여 딕셔너리를 계산하는 프로그램 첫 부분은 바뀌지 않는다. 하지만, `counts` 를 단순히 출력하는 대신에 `(val, key)` 튜플 리스트를 생성하고 역순으로 리스트를 정렬한다.

값이 첫 위치에 있기 때문에, 비교 목적으로 값을 사용한다. 만약 동일한 값을 가진 튜플이 하나이상 존재한다면, 두번째 요소(키, key)를 살펴본다. 그래서 값이 동일한 경우 키의 알파벳 순으로 추가 정렬된다.

마지막에 다중 대입 반복을 수행하는 멋진 `for` 루프를 작성한다. 그리고, 리스트 슬라이스(`lst[:10]`)를 통해 가장 빈도수가 높은 상위 10개 단어를 출력한다.

이제 단어 빈도 분석을 위해서 작성한 프로그램의 마지막 출력결과는 원하는 바를 완수한 것처럼 보인다.

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

복잡한 데이터 파싱과 분석 작업이 이해하기 쉬운 19줄 파이썬 프로그램으로 수행된 사실이 왜 파이썬이 정보 탐색 언어로서 좋은 선택인지 보여준다.

제 7 절 딕셔너리 키로 튜플 사용하기

튜플은 해쉬형(hashable)이고, 리스트는 그렇지 못하기 때문에, 만약 딕셔너리에 사용할 복합(composite)키를 생성하려면, 키로 튜플을 사용해야 한다.

만약 성(last-name)과 이름(first-name)을 가지고 전화번호에 사상(매핑, mapping)하는 전화번호부를 생성하려고 하면, 복합키와 마주친다. 변수 `last`, `first`, `number`을 정의했다고 가정하면, 다음과 같이 딕셔너리 대입문을 작성할 수 있다.

```
directory[last,first] = number
```

꺾쇠 괄호 표현은 튜플이다. 딕셔너리를 훑기 위해서 `for` 루프에 튜플 대입을 사용한다.

```
for last, first in directory:
    print first, last, directory[last,first]
```

상기 루프가 튜플인 directory에 키를 훑는다. 각 튜플 요소를 last, first에 대입하고 나서, 이름과 해당 전화번호를 출력한다.

제 8 절 순서(sequence) : 문자열, 리스트, 튜플

여기서 리스트 튜플에 초점을 맞추었지만, 이장의 거의 모든 예제가 또한 리스트의 리스트, 튜플의 튜플, 리스트 튜플에도 동작한다. 가능한 모든 조합을 열거하는 것을 피하기 위해서, 순서의 순서(sequences of sequences)에 대해서 논의하는 것이 때로는 쉽다.

대부분의 문맥에서 다른 종류의 순서(문자열, 리스트, 튜플)는 상호 호환해서 사용될 수 있다. 그런데 왜 그리고 어떻게 다른 것보다 이것을 선택해야 될까?

명확한 것부터 시작하자. 문자열은 요소가 문자여야 하기 때문에 다른 순서(sequence)보다 제약이 따른다. 또한 문자열은 불변(immutable)이다. 새로운 문자열을 생성하는 것과 반대로, 문자열에 있는 문자를 변경하고자 한다면, 대신에 문자 리스트를 사용하는 것을 생각할 수 있다.

리스트는 튜플보다 좀더 일반적으로 사용된다. 이유는 대체로 변경가능(mutable)하기 때문이다. 하지만, 다음 몇가지 경우에 튜플이 좀더 선호된다.

1. return 문처럼 어떤 맥락에서, 리스트보다 튜플을 생성하는 것이 구문론적으로 더 간략하다. 다른 맥락에서는 리스트가 더 선호될 수 있다.
2. 딕셔너리 키로서 순서(sequence)를 사용하려면, 튜플이나 문자열같은 불변 자료형(immutable type)을 사용해야 한다.
3. 함수에 인자로 순서(sequence)를 전달하려면, 튜플을 사용하는 것이 에일리어싱(aliasing)으로 생기는 예기치 못한 행동에 대한 가능성을 줄여준다.

튜플은 불변(immutable)이어서, 기존 리스트를 변경하는 sort, reverse 같은 메쏘드를 제공하지는 않는다. 하지만, 파이썬이 제공하는 내장함수 sorted, reversed 를 통해서, 매개 변수로 임의 순서(sequence)를 전달 받아서, 같은 요소를 다른 순서로 정렬된 새로운 리스트를 반환한다.

제 9 절 디버깅

리스트, 딕셔너리, 튜플은 자료 구조(data structures)로 일반적으로 알려져 있다. 이번장에서 리스트 튜플, 키로 튜플, 값으로 리스트를 담고 있는 딕셔너리 같은 복합 자료 구조를 보기 시작했다. 복합 자료 구조는 유용하지만, 저자가 작명한 모양 오류(shape errors)라고 불리는 오류에 노출되어 있다. 즉, 자료

구조가 잘못된 자료형(type), 크기, 구성일 경우 오류가 발생한다. 혹은 코드를 작성하고, 자료의 모양이 생각나지 않는 경우도 오류의 원인이 된다.

예를 들어, 정수 하나를 가진 리스트를 기대하고, (리스트가 아닌) 일반 정수를 넘긴다면, 작동하지 않는다.

프로그램을 디버깅할 때, 정말 어려운 버그를 잡으려고 작업을 한다면, 다음 네 가지를 시도할 수 있다.

코드 읽기(reading): 코드를 면밀히 조사하고, 스스로에게 다시 읽어 주고, 코드가 자신이 작성한 의도를 담고 있는지 점검하라.

실행(running): 변경해서 다른 버젼을 실행해서 실험하라. 종종, 프로그램이 적절한 곳에 적절한 것을 보여준다면, 문제가 명확하다. 발판(scaffolding)을 만들기 위해서 때때로 시간을 들일 필요도 있다.

반추(ruminating): 생각의 시간을 갖자. 어떤 종류의 오류인가: 구문, 실행, 의미론(semantic). 오류 메시지로부터 혹은 프로그램 출력결과로부터 무슨 정보를 얻을 수 있는가? 어떤 종류 오류가 지금 보고 있는 문제를 만들었을까? 문제가 나타나기 전에, 마지막으로 변경한 것은 무엇인가?

퇴각(retreating): 어느 시점에선가, 최선은 물러서서, 최근의 변경을 다시 원복하는 것이다. 잘 동작하고 이해하는 프로그램으로 다시 돌아가서, 다시 프로그램을 작성한다.

초보 프로그래머는 종종 이들 활동 중 하나에 사로잡혀 다른 것을 잊곤 한다. 활동 각각은 고유한 실패 방식과 함께 온다.

예를 들어, 프로그램을 정독하는 것은 문제가 인쇄상의 오류에 있다면 도움이 되지만, 문제가 개념상 오해에 뿌리를 두고 있다면 그다지 도움이 되지 못한다. 만약 작성한 프로그램을 이해하지 못한다면, 100번 읽을 수는 있지만, 오류를 발견할 수는 없다. 왜냐하면, 오류는 여러분 머리에 있기 때문입니다.

만약 작고 간단한 테스트를 진행한다면, 실험을 수행하는 것이 도움이 될 수 있다. 하지만, 코드를 읽지 않거나, 생각없이 실험을 수행한다면, 프로그램이 작동될 때까지 무작위 변경하여 개발하는 "랜덤 워크 프로그램(random walk programming)" 패턴에 빠질 수 있다. 말할 필요없이 랜덤 워크 프로그래밍은 시간이 오래 걸린다.

생각할 시간을 가져야 한다. 디버깅은 실험 과학 같은 것이다. 문제가 무엇인지에 대한 최소한 한 가지 가설을 가져야 한다. 만약 두개 혹은 그 이상의 가능성이 있다면, 이러한 가능성 중에서 하나라도 줄일 수 있는 테스트를 생각해야 한다.

휴식 시간을 가지는 것은 생각하는데 도움이 된다. 대화를 하는 것도 도움이 된다. 문제를 다른 사람 혹은 자신에게도 설명할 수 있다면, 질문을 마치기도 전에 답을 종종 발견할 수 있다.

하지만, 오류가 너무 많고 수정하려는 코드가 매우 크고, 복잡하다면 최고의 디버깅 기술도 무용지물이다. 가끔, 최선의 선택은 퇴각하는 것이다. 작동하고 이해하는 곳까지 후퇴해서 프로그램을 간략화하라.

초보 프로그래머는 종종 퇴각하기를 꺼려한다. 왜냐하면, 설사 잘못되었지만, 한줄 코드를 지울 수 없기 때문이다. 삭제하지 않는 것이 기분을 좋게 한다면, 다시 작성하기 전에 프로그램을 다른 파일에 복사하라. 그리고 나서, 한번에 조금씩 붙여넣어라.

정말 어려운 버그(hard bug)를 발견하고 고치는 것은 코드 읽기, 실행, 반추, 때때로 퇴각을 요구한다. 만약 이들 활동 중 하나도 먹히지 않는다면, 다른 것들을 시도해 보세요.

제 10 절 용어정의

비교가능한(comparable): 동일한 자료형의 다른 값과 비교하여 큰지, 작은지, 혹은 같은지를 확인하기 위해서 확인할 수 있는 자료형(type). 비교가능한(comparable) 자료형은 리스트에 넣어서 정렬할 수 있다.

자료 구조(data structure): 연관된 값의 집합, 종종 리스트, 딕셔너리, 튜플 등으로 조직화된다.

DSU: "decorate-sort-undecorate,"의 약어로 리스트 튜플을 생성, 정렬, 결과 일부 추출을 포함하는 패턴.

모음(gather): 가변-길이 인자 튜플을 조합하는 연산.

해쉬형(hashable): 해쉬 함수를 가진 자료형(type). 정수, 소수점, 문자열 같은 불변형은 해쉬형이다. 리스트나 딕셔너리 처럼 변경가능한 형은 해쉬형이 아니다.

스캐터(scatter): 순서(sequence)를 리스트 인자로 다루는 연산.

(자료 구조의) 모양 (shape (of a data structure)): 자료 구조의 자료형(type), 크기, 구성을 요약.

싱글톤(singleton): 단일 요소를 가진 리스트 (혹은 다른 순서(sequence)).

튜플(tuple): 불변 요소들의 순서 (sequence).

튜플 대입(tuple assignment): 오른편 순서(sequence)와 왼편 튜플 변수를 대입. 오른편이 평가되고나서 각 요소들은 왼편의 변수에 대입된다.

제 11 절 연습문제

Exercise 10.1 앞서 작성한 프로그램을 다음과 같이 수정하세요. "From"라인을 읽고 파싱하여 라인에서 주소를 뽑아내세요. 딕셔너리를 사용하여 각 사람으로부터 메시지 숫자를 계수(count)한다.

모든 데이터를 읽은 후에 가장 많은 커밋(commit)을 한 사람을 출력하세요. 딕셔너리로부터 리스트 (count, email) 튜플을 생성하고 역순으로 리스트를 정렬한 후에 가장 많은 커밋을 한 사람을 출력하세요.

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

Exercise 10.2 이번 프로그램은 각 메시지에 대한 하루 중 시간의 분포를 계수(count)한다. "From" 라인으로부터 시간 문자열을 찾고 콜론(:) 문자를 사용하여 문자열을 쪼개서 시간을 추출합니다. 각 시간별로 계수(count)를 누적하고 아래에 보여지듯이 시간 단위로 정렬하여 한 라인에 한시간씩 계수(count)를 출력합니다.

```
Sample Execution:
python timeofday.py
Enter a file name: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

Exercise 10.3 파일을 읽고, 빈도(frequencey)에 따라 내림차순으로 *문자(letters)*를 출력하는 프로그램을 작성하세요. 작성한 프로그램은 모든 입력을 소문자로 변환하고 a-z 문자만 계수(count)한다. 공백, 숫자, 문장기호 a-z를 제외한 다른 어떤 것도 계수하지 않습니다. 다른 언어로 구성된 텍스트 샘플을 구해서 언어마다 문자 빈도가 어떻게 변하는지 살펴보세요. 결과를 `wikipedia.org/wiki/Letter_frequencies` 표와 비교하세요.

11 장

정규 표현식

지금까지 파일을 훑어서 패턴을 찾고, 관심있는 라인에서 다양한 비트(bits)를 뽑아냈다. `split`, `find` 같은 문자열 메쏘드를 사용하였고, 라인에서 일정 부분을 뽑아내기 위해서 리스트와 문자열 슬라이싱(slicing)을 사용했다.

검색하고 추출하는 작업은 너무 자주 있는 일이어서 파이썬은 상기와 같은 작업을 매우 우아하게 처리하는 정규 표현식(regular expressions)으로 불리는 매우 강력한 라이브러리를 제공한다. 정규 표현식을 책의 앞부분에 소개하지 않은 이유는 정규 표현식 라이브러리가 매우 강력하지만, 약간 복잡하고, 구문에 익숙해지는데 시간이 필요하기 때문이다.

정규표현식은 문자열을 검색하고 파싱하는데 그 자체가 작은 프로그래밍 언어다. 사실, 책 전체가 정규 표현식을 주제로 쓰여진 책이 몇권 있다. 이번 장에서는 정규 표현식의 기초만을 다룰 것이다. 정규 표현식의 좀더 자세한 사항은 다음을 참조하라.

http://en.wikipedia.org/wiki/Regular_expression

http://docs.python.org/library/re.html

정규 표현식 라이브러리는 사용하기 전에 프로그램을 가져오기(import)해야 한다. 정규 표현식 라이브러리의 가장 간단한 쓰임은 `search()` 검색 함수다. 다음 프로그램은 검색 함수의 사소한 사용례를 보여준다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line) :
        print line
```

파일을 열고, 각 라인을 루프로 반복해서 정규 표현식 `search()` 메쏘드를 호출하여 문자열 "From"이 포함된 라인만 출력한다. 상기 프로그램은 진정으로 강력한 정규 표현식 기능을 사용하지 않았다. 왜냐하면, `line.find()` 메쏘드를 가지고 동일한 결과를 쉽게 구현할 수 있기 때문이다.

정규 표현식의 강력한 기능은 문자열에 해당하는 라인을 좀더 정확하게 제어하기 위해서 검색 문자열에 특수문자를 추가할 때 확인될 수 있다. 매우 적은 코드를 작성할지라도, 정규 표현식에 특수 문자를 추가하는 것만으로도 정교한 일치(matching)와 추출이 가능하게 한다.

예를 들어, 탈자 기호(caret)는 라인의 "시작"과 일치하는 정규 표현식에 사용된다. 다음과 같이 "From:"으로 시작하는 라인만 일치하도록 응용프로그램을 변경할 수 있다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line) :
        print line
```

"From:" 문자열로 *시작*하는 라인만 일치할 수 있다. 여전히 매우 간단한 프로그램으로 문자열 라이브러리에서 `startswith()` 메쏘드로 동일하게 수행할 수 있다. 하지만, 무엇을 정규 표현식과 매칭하는가에 대해 특수 액션 문자를 담아 좀더 많은 제어를 할수 있게 하는 정규 표현식 개념을 소개하기에는 충분하다.

제 1 절 정규 표현식의 문자 매칭

좀더 강력한 정규 표현식을 작성할 수 있는 다른 특수문자는 많이 있다. 가장 자주 사용되는 특수 문자는 임의 문자를 매칭하는 마침표다.

다음 예제에서 정규 표현식 "F..m:"은 "From:", "Fxxm:", "F12m:", "F!@m:' 같은 임의 문자열을 매칭한다. 왜냐하면 정규 표현식 마침표 문자가 임의의 문자와 매칭되기 때문이다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line) :
        print line
```

정규 표현식에 "*", "+' ' 문자를 사용하여 문자가 원하는만큼 반복을 나타내는 기능과 결합되었을 때는 더욱 강력해진다. "*", "+' ' 특수 문자가 검색 문자열에 문자 하나만을 매칭하는 대신에 별표 기호인 경우 0 혹은 그 이상의 매칭, 더하기 기호인 경우 1 혹은 그 이상의 문자의 매칭을 의미한다.

다음 예제에서 반복 와일드 카드(wild card) 문자를 사용하여 매칭하는 라인을 좀더 좁힐 수 있다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:.+@', line) :
        print line
```

검색 문자열 "^From:.+@" 은 "From:" 으로 시작하고, ".+" 하나 혹은 그 이상의 문자들, 그리고 @ 기호와 매칭되는 라인을 성공적으로 찾아낸다. 그래서 다음 라인은 매칭이 될 것이다.

From: stephen.marquard @uct.ac.za

콜론(:)과 @ 기호 사이의 모든 문자들을 매칭하도록 확장하는 것으로 ".+" 와이드 카드를 간주할 수 있다.

From: .+ @

더하기와 별표 기호를 "밀어내기(pushy)" 문자로 생각하는 것이 좋다. 예를 들어, 다음 문자열은 ".+" 특수문자가 다음에 보여주듯이 밖으로 밀어내는 것처럼 문자열 마지막 @ 기호를 매칭한다.

From: stephen.marquard@uct.ac.za, csev@umich.edu, and cwen @iupui.edu

다른 특수문자를 추가함으로서 별표나 더하기 기호가 너무 "탐욕(greedy)"스럽지 않게 만들 수 있다. 와일드 카드 특수문자의 탐욕스러운 기능을 끄는 것에 대해서는 자세한 정보를 참조바란다.

제 2 절 정규 표현식 사용 데이터 추출

파이썬으로 문자열에서 데이터를 추출하려면, findall() 메쏘드를 사용해서 정규 표현식과 매칭되는 모든 부속 문자열을 추출할 수 있다. 형식에 관계없이 임의 라인에서 전자우편 주소 같은 문자열을 추출하는 예제를 사용하자. 예를 들어, 다음 각 라인에서 전자우편 주소를 뽑아내고자 한다.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

각각의 라인에 대해서 다르게 쪼개고, 슬라이싱하면서 라인 각각의 형식에 맞추어 코드를 작성하고는 싶지는 않다. 다음 프로그램은 findall() 메쏘드를 사용하여 전자우편 주소가 있는 라인을 찾아내고 하나 혹은 그 이상의 주소를 뽑아낸다.

```
import re
s = 'Hello from csev@umich.edu to cwen@iupui.edu about the meeting @2PM'
lst = re.findall('\S+@\S+', s)
print lst
```

findall() 메쏘드는 두번째 인자 문자열을 찾아서 전자우편 주소처럼 보이는 모든 문자열을 리스트로 반환한다. 공백이 아닌 문자 (nS)와 매칭되는 두 문자 순서(sequence)를 사용한다.

프로그램의 출력은 다음과 같다.

```
['csev@umich.edu', 'cwen@iupui.edu']
```

정규 표현식을 해석하면, 적어도 하나의 공백이 아닌 문자, @과 적어도 하나 이상의 공백이 아닌 문자를 가진 부속 문자열을 찾는다. 또한, "nS+" 특수 문자는 가능한 많이 공백이 아닌 문자를 매칭한다. (정규 표현식에서 "탐욕(greedy)" 매칭이라고 부른다.)

정규 표현식은 두번 매칭(csev@umich.edu, cwen@iupui.edu)하지만, 문자열 "@2PM"은 매칭을 하지 않는다. 왜냐하면, @ 기호 앞에 공백이 아닌 문자가 하나도 없기 때문이다. 프로그램의 정규 표현식을 사용해서 파일에 모든 라인을 읽고 다음과 같이 전자우편 주소처럼 보이는 모든 문자열을 출력한다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line)
    if len(x) > 0 :
        print x
```

각 라인을 읽어 들이고, 정규 표현식과 매칭되는 모든 부속 문자열을 추출한다. findall() 메쏘드는 리스트를 반환하기 때문에, 전자우편 처럼 보이는 부속 문자열을 적어도 하나 찾아서 출력하기 위해서 반환 리스트 요소 숫자가 0 보다 큰지 여부를 간단히 확인한다.

mbox.txt 파일에 프로그램을 실행하면, 다음 출력을 얻는다.

```
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['<postmaster@collab.sakaiproject.org>']
['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
```

전자우편 주소 몇몇은 "<", ";" 같은 잘못된 문자가 앞과 뒤에 붙어있다. 문자나 숫자로 시작하고 끝나는 문자열 부분만 관심있다고 하자.

그러기 위해서, 정규 표현식의 또 다른 기능을 사용한다. 매칭하려는 다중 허용 문자 집합을 표기하기 위해서 꺾쇠 괄호를 사용한다. 그런 의미에서 "nS"은 공백이 아닌 문자 집합을 매칭하게 한다. 이제 매칭하려는 문자에 관해서 좀더 명확해졌다.

여기 새로운 정규 표현식이 있다.

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

약간 복잡해졌다. 왜 정규 표현식이 자신만의 언어인가에 대해서 이해할 수 있다. 이 정규 표현식을 해석하면, 0 혹은 그 이상의 공백이 아닌 문자("nS*")로 하나의 소문자, 대문자 혹은 숫자("[a-zA-Z0-9]")를 가지며, @ 다음에 0 혹은 그 이상의 공백이 아닌 문자("nS*")로 하나의 소문자, 대문자 혹은 숫자("[a-zA-Z0-9]")로 된 부속 문자열을 찾는다. 0 혹은 그 이상의 공백이 아닌 문자를 나타내기 위해서 "+"에서 "*"으로 바꿨다. 왜냐하면 "[a-zA-Z0-9]" 자체가 이미

하나의 공백이 아닌 문자이기 때문이다. "*", "+"는 단일 문자에 별표, 더하기 기호 왼편에 즉시 적용됨을 기억하세요.

프로그램에 정규 표현식을 사용하면, 데이터가 훨씬 깔끔해진다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('[a-zA-Z0-9]\S*@\S*[a-zA-Z]', line)
    if len(x) > 0 :
        print x

...
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

"source@collab.sakaiproject.org" 라인에서 문자열 끝에 ">;" 문자를 정규 표현식으로 제거한 것을 주목하세요. 정규 표현식 끝에 "[a-zA-Z]"을 추가하여서 정규 표현식 파서가 찾는 임의 문자열은 문자로만 끝나야 되기 때문이다. 그래서, "sakaiproject.org>;"에서 ">"을 봤을 때, "g"가 마지막 맞는 매칭이 되고, 거기서 마지막 매칭을 마치고 중단한다.

프로그램의 출력은 리스트의 단일 요소를 가진 문자열로 파이썬 리스트이다.

제 3 절 검색과 추출 조합하기

다음과 같은 "X-" 문자열로 시작하는 라인의 숫자를 찾고자 한다면,

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

임의의 라인에서 임의 부동 소수점 숫자가 아니라 상기 구문을 가진 라인에서만 숫자를 추출하고자 한다.

라인을 선택하기 위해서 다음과 같이 정규 표현식을 구성한다.

```
^X-.*: [0-9.]+
```

정규 표현식을 해석하면, '^'에서 "X-"으로 시작하고, ".*"에서 0 혹은 그이상의 문자를 가지며, 콜론(":")이 나오고 나서 공백을 만족하는 라인을 찾는다. 공백 뒤에 "[0-9.]+"에서 숫자 (0-9) 혹은 점을 가진 하나 혹은 그 이상의 문자가 있여야 한다. 꺽쇠 기호 사이에 마침표는 실제 마침표만 매칭함을 주목하기 바란다. (즉, 꺾쇠 기호 사이는 와일드 카드 문자가 아니다.)

관심을 가지고 있는 특정한 라인과 매우 정확하게 매칭이되는 매우 빠듯한 정규 표현식으로 다음과 같다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line) :
        print line
```

프로그램을 실행하면, 잘 걸러져서 찾고자 하는 라인만 볼 수 있다.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
```

하지만, 이제 `rsplit` **사용해서 숫자를 뽑아내는 문제를 해결해야 한다.** `rsplit` **을 사용하는 것이 간단해 보이지만, 동시에 라인을 검색하고 파싱하기 위해서 정규 표현식의 또 다른 기능을 사용할 수 있다.**

괄호는 정규 표현식의 또 다른 특수 문자다. 정규 표현식에 괄호를 추가한다면, 문자열이 매칭될 때, 무시된다. 하지만, `findall()` **을 사용할 때, 매칭할 전체 정규 표현식을 원할지라도, 정규 표현식을 매칭하는 부속 문자열의 부분만을 뽑아낸다는 것을 괄호가 표시한다.**

그래서, 프로그램을 다음과 같이 수정한다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X\S*: ([0-9.]+)', line)
    if len(x) > 0 :
        print x
```

`search()` **을 호출하는 대신에, 매칭 문자열의 부동 소수점 숫자만 뽑아내는** `findall()` **에 원하는 부동 소수점 숫자를 표현하는 정규 표현식 부분에 괄호를 추가한다.**

프로그램의 출력은 다음과 같다.

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
..
```

숫자가 여전히 리스트에 있어서 문자열에서 부동 소수점으로 변환할 필요가 있지만, 흥미로운 정보를 찾아 뽑아내기 위해서 정규 표현식의 강력한 힘을 사용했다.

이 기술을 활용한 또 다른 예제로, 파일을 살펴보면, 폼(form)을 가진 라인이 많다.

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

상기 언급한 동일한 기법을 사용하여 모든 변경 번호(라인의 끝에 정수 숫자)를 추출하고자 한다만, 다음과 같이 프로그램을 작성할 수 있다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9.]+)', line)
    if len(x) > 0:
        print x
```

작성한 정규 표현식을 해석하면, "Details:"로 시작하는 ".*"에 임의의 문자들로, "rev="을 포함하고 나서, 하나 혹은 그 이상의 숫자를 가진 라인을 찾는다. 전체 정규 표현식을 만족하는 라인을 찾고자 하지만, 라인 끝에 정수만을 추출하기 위해서 "[0-9]+"을 괄호로 감쌌다.

프로그램을 실행하면, 다음 출력을 얻는다.

```
['39772']
['39771']
['39770']
['39769']
...
```

"[0-9]+"은 "탐욕(greedy)"스러워서, 숫자를 추출하기 전에 가능한 큰 문자열 숫자를 만들려고 한다는 것을 기억하라. 이런 "탐욕(greedy)"스러운 행동으로 인해서 왜 각 숫자로 모두 5자리 숫자를 얻은 이유가 된다. 정규 표현식 라이브러리는 양방향으로 파일 처음이나 끝에 숫자가 아닌 것을 마주칠 때까지 뻗어나간다.

이제 정규 표현식을 사용해서 각 전자우편 메시지의 요일에 관심이 있었던 책 앞의 연습 프로그램을 다시 작성한다. 다음 형식의 라인을 찾는다.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

그리고 나서, 각 라인의 요일의 시간을 추출하고자 한다. 앞에서 `split`를 두번 호출하여 작업을 수행했다. 첫번째는 라인을 단어로 쪼개고, 다섯번째 단어를 뽑아내서, 관심있는 두 문자를 뽑아내기 위해서 콜론 문자에서 다시 쪼갰다.

작동을 할지 모르지만, 실질적으로 정말 부서지기 쉬운 코드로 라인이 잘 짜여져 있다고 가정하에 가능하다. 잘못된 형식의 라인이 나타날 때도 결코 망가지지 않는 프로그램을 담보하기 위해서 충분한 오류 검사기능을 추가하거나 커다란 **try/except** 블록을 넣으면, 참 읽기 힘든 **10-15** 라인 코드로 커질 것이다.

다음 정규 표현식으로 훨씬 간결하게 작성할 수 있다.

```
^From .* [0-9][0-9]:
```

상기 정규 표현식을 해석하면, 공백을 포함한 "From "으로 시작해서, ".*"에 임의의 갯수의 문자, 그리고 공백, 두 개의 숫자 "[0-9][0-9]" 뒤에 콜론(:) 문자를 가진 라인을 찾는다. 일종의 찾고 있는 라인에 대한 정의다.

findall()을 사용해서 단지 시간만 뽑아내기 위해서, 두 숫자에 괄호를 다음과 같이 추가한다.

```
^From .* ([0-9][0-9]):
```

작업 결과는 다음과 같이 프로그램에 반영한다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0 : print x
```

프로그램을 실행하면, 다음 출력 결과가 나온다.

```
['09']
['18']
['16']
['15']
...
```

제 4 절 이스케이프(Escape) 문자

라인의 처음과 끝을 매칭하거나, 와일드 카드를 명세하기 위해서 정규 표현식의 특수 문자를 사용했기 때문에, 정규 표현식에 사용된 문자가 "정상(normal)"적인 문자임을 표기할 방법이 필요하고 달러 기호와 탈자 기호(^) 같은 실제 문자를 매칭하고자 한다.

역슬래쉬('\')를 가진 문자를 앞에 덧붙여서 문자를 단순히 매칭하고자 한다고 나타낼 수 있다. 예를 들어, 다음 정규표현식으로 금액을 찾을 수 있다.

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

역슬래쉬 달러 기호를 앞에 덧붙여서, 실제로 "라인 끝(end of line)" 매칭 대신에 입력 문자열의 달러 기호와 매칭한다. 정규 표현식 나머지 부분은 하나 혹은 그 이상의 숫자 혹은 소수점 문자를 매칭한다. *주목*: 꺾쇠 괄호 내부에 문자는 "특수 문자"가 아니다. 그래서 "[0-9.]"은 실제 숫자 혹은 점을 의미한다. 꺾쇠 괄호 외부에 점은 "와일드 카드(wild-card)" 문자이고 임의의 문자와 매칭한다. 꺾쇠 괄호 내부에서 점은 점일 뿐이다.

제 5 절 요약

지금까지 정규 표현식의 표면을 긁은 정도지만, 정규 표현식 언어에 대해서 조금 학습했다. 정규 표현식은 특수 문자로 구성된 검색 문자열로 "매칭(matching)"

정의하고 매칭된 문자열로부터 추출된 결과물을 정규 표현식 시스템과 프로그래머가 의도한 바를 의사소통하는 것이다. 다음에 특수 문자 및 문자 시퀀스의 일부가 있다.

^
라인의 처음을 매칭.

$
라인의 끝을 매칭.

.
임의의 문자를 매칭(와일드 카드)

n**s**
공백 문자를 매칭.

n**S**
공백이 아닌 문자를 매칭.(n**s** 의 반대).

*
바로 앞선 문자에 적용되고 0 혹은 그 이상의 앞선 문자와 매칭을 표기함.

*?
바로 앞선 문자에 적용되고 0 혹은 그 이상의 앞선 문자와 매칭을 "탐욕적이지 않은(non-greedy) 방식"으로 표기함.

+
바로 앞선 문자에 적용되고 1 혹은 그 이상의 앞선 문자와 매칭을 표기함.

+?
바로 앞선 문자에 적용되고 1 혹은 그 이상의 앞선 문자와 매칭을 "탐욕적이지 않은(non-greedy) 방식"으로 표기함.

[aeiou]
명세된 집합 문자에 존재하는 단일 문자와 매칭. 다른 문자는 안되고, "a", "e", "i", "o", "u" 문자만 매칭되는 예제.

[a-z0-9]
음수 기호로 문자 범위를 명세할 수 있다. 소문자이거나 숫자인 단일 문자만 매칭되는 예제.

[^A-Za-z]
집합 표기의 첫문자가 ^인 경우, 로직을 거꾸로 적용한다. 대문자나 혹은 소문자가 아닌 임의 단일 문자만 매칭하는 예제.

( )
괄호가 정규표현식에 추가될 때, 매칭을 무시한다. 하지만 `findall()`을 사용할 때 전체 문자열보다 매칭된 문자열의 상세한 부속 문자열을 추출할 수 있게 한다.

n**b**
빈 문자열을 매칭하지만, 단어의 시작과 끝에만 사용된다.

n**B**
빈 문자열을 매칭하지만, 단어의 시작과 끝이 아닌 곳에 사용된다.

n**d**
임의 숫자와 매칭하여 [0-9] 집합에 상응함.

n**D**
임의 숫자가 아닌 문자와 매칭하여 [^0-9] 집합에 상응함.

제 6 절 유닉스 사용자를 위한 보너스

정규 표현식을 사용하여 파일을 검색 기능은 1960년대 이래로 유닉스 운영 시스템에 내장되어 여러가지 형태로 거의 모든 프로그래밍 언어에서 이용가능하다.

사실, `search()` **예제에서와 거의 동일한 기능을 하는 grep (Generalized Regular Expression Parser)으로 불리는 유닉스 내장 명령어 프로그램이 있다. 그래서, 맥킨토시나 리눅스 운영 시스템을 가지고 있다면, 명령어 창에서 다음 명령어를 시도할 수 있다.**

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

`grep`**을 사용하여,** `mbox-short.txt` **파일 내부에 "From:" 문자열로 시작하는 라인을 보여준다.** `grep` **명령어를 가지고 약간 실험을 하고** `grep`**에 대한 문서를 읽는다면, 파이썬에서 지원하는 정규 표현식과** `grep`**에서 지원되는 정규 표현식과 차이를 발견할 것이다. 예를 들어,** `grep` **공백이 아닌 문자 "n**S**"을 지원하지 않는다. 그래서 약간 더 복잡한 집합 표기 "[^]"을 사용해야 한다. "[^]"은 간단히 정리하면, 공복을 제외한 임의의 문자와 매칭한다.**

제 7 절 디버깅

만약 특정 메쏘드의 정확한 이름을 기억해 내기 위해서 빠르게 생각나게 하는 것이 필요하다면 도움이 많이 될 수 있는 간단하고 초보적인 내장 문서가 파이썬에 포함되어 있다. 내장 문서 도움말은 인터랙티브 모드의 파이썬 인터프리터에서 볼 수 있다.

`help()`**를 사용하여 인터랙티브 도움을 받을 수 있다.**

```
>>> help()

Welcome to Python 2.6!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.
```

```
Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help> modules
```

특정한 모듈을 사용하고자 한다면, `dir()` 명령어를 사용하여 다음과 같이 모듈의 메쏘드를 찾을 수 있다.

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

또한, `dir()` 명령어를 사용하여 특정 메쏘드에 대한 짧은 문서 도움말을 얻을 수 있다.

```
>>> help (re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
>>>
```

내장 문서는 광범위하지 않아서, 급하거나, 웹 브라우저나 검색엔진에 접근할 수 없을 때 도움이 될 수 있다.

제 8 절 용어정의

부서지기 쉬운 코드(brittle code): 입력 데이터가 특정한 형식일 경우에만 작동하는 코드. 하지만 올바른 형식에서 약간이도 벗아나게 되면 깨지기 쉽니다. 쉽게 부서지기 때문에 "부서지기 쉬운 코드(brittle code)"라고 부른다.

욕심쟁이 매칭(greedy matching): 정규 표현식의 "+", "*" 문자는 가능한 큰 문자열을 매칭하기 위해서 밖으로 확장하는 개념.

grep: 정규 표현식에 매칭되는 파일을 탐색하여 라인을 찾는데 대부분의 유닉스 시스템에서 사용가능한 명령어. "Generalized Regular Expression Parser"의 약자.

정규 표현식(regular expression): 좀더 복잡한 검색 문자열을 표현하는 언어. 정규 표현식은 특수 문자를 포함해서 검색 라인의 처음 혹은 끝만 매칭하거나 많은 비슷한 것을 매칭한다.

와일드 카드(wild card): 임의 문자를 매칭하는 특수 문자. 정규 표현식에서 와이드 카드 문자는 마침표 문자다.

제 9 절 연습 문제

Exercise 11.1 유닉스의 `grep` 명령어를 모사하는 간단한 프로그램을 작성하세요. 사용자가 정규 표현식을 입력하고 정규 표현식에 매칭되는 라인수를 셈하는 프로그램입니다.

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author

$ python grep.py
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-

$ python grep.py
Enter a regular expression: java$
mbox.txt had 4218 lines that matched java$
```

Exercise 11.2 다음 형식의 라인만을 찾는 프로그램을 작성하세요.

```
New Revision: 39772
```

그리고, 정규 표현식과 `findall()` 메쏘드를 사용하여 각 라인으로부터 숫자를 추출하세요. 숫자들의 평균을 구하고 출력하세요.

```
Enter file:mbox.txt
38549.7949721

Enter file:mbox-short.txt
39756.9259259
```

12 장

네트워크 프로그램

지금까지 책의 많은 예제는 파일을 읽고 파일의 정보를 찾는데 집중했지만, 다양한 많은 정보의 원천이 인터넷에 있다.

이번 장에서는 웹브라우져로 가장하고 HTTP 프로토콜(HyperText Transport Protocol,HTTP)을 사용하여 웹페이지를 검색할 것이다. 웹페이지 데이터를 읽고 파싱할 것이다.

제 1 절 하이퍼 텍스트 전송 프로토콜(HyperText Transport Protocol - HTTP)

웹에 동력을 공급하는 네트워크 프로토콜은 실제로 매우 단순하다. 파이썬에는 소켓 (sockets)이라고 불리는 내장 지원 모듈이 있다. 파이썬 프로그램에서 소켓 모듈을 통해서 네트워크 연결을 하고, 데이터 검색을 매우 용이하게 한다.

소켓(socket)은 단일 소켓으로 두 프로그램 사이에 양방향 연결을 제공한다는 점을 제외하고 파일과 매우 유사하다. 동일한 소켓에 읽거나 쓸 수 있다. 소켓에 무언가를 쓰게 되면, 소켓의 다른 끝에 있는 응용프로그램에 전송된다. 소켓으로부터 읽게 되면, 다른 응용 프로그램이 전송한 데이터를 받게 된다.

하지만, 소켓의 다른쪽 끝에 프로그램이 어떠한 데이터도 전송하지 않았는데 소켓을 읽으려고 하면, 단지 앉아서 기다리기만 한다. 만약 어떠한 것도 보내지 않고 양쪽 소켓 끝의 프로그램 모두 기다리기만 한다면, 모두 매우 오랜 시간동안 기다리게 될 것이다.

인터넷으로 통신하는 프로그램의 중요한 부분은 특정 종류의 프로토콜을 공유하는 것이다. 프로토콜(protocol)은 정교한 규칙의 집합으로 누가 메시지를 먼저 보내고, 메세지로 무엇을 하며, 메시지에 대한 응답은 무엇이고, 다음에 누가 메세지를 보내고 등등을 포함한다. 이런 관점에서 소켓 끝의 두 응용프로그램이 함께 춤을 추고 있으니, 다른 사람 발을 밟지 않도록 확인해야 한다.

네트워크 프로토콜을 기술하는 문서가 많이 있다. 하이퍼텍스트 전송 프로토콜(HyperText Transport Protocol)은 다음 문서에 기술되어 있다.

http://www.w3.org/Protocols/rfc2616/rfc2616.txt

매우 상세한 176 페이지나 되는 장문의 복잡한 문서다. 흥미롭다면 시간을 가지고 읽어보기 바란다. RFC2616에 36 페이지를 읽어보면, GET 요청(request)에 대한 구문을 발견하게 된다. 꼼꼼히 읽게 되면, 웹서버에 문서를 요청하기 하기 위해서, 80 포트로 `www.py4inf.com` 서버에 연결을 하고 나서 다음 양식 한 라인을 전송한다.

```
GET http://www.py4inf.com/code/romeo.txt HTTP/1.0
```

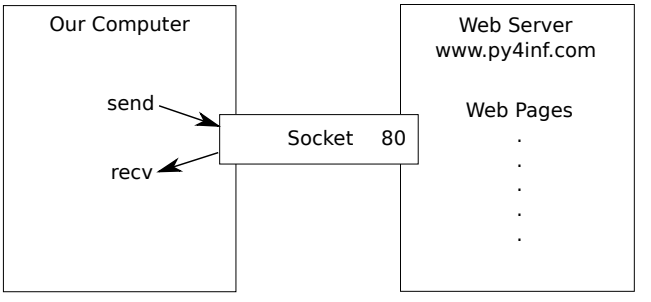
두번째 매개변수는 요청하는 웹페이지가 된다. 그리고 또한 빈 라인도 전송한다. 웹서버는 문서에 대한 헤더 정보와 빈 라인 그리고 문서 본문으로 응답한다.

제 2 절 세상에서 가장 간단한 웹 브라우져(Web Browser)

아마도 HTTP 프로토콜이 어떻게 작동하는지 알아보는 가장 간단한 방법은 매우 간단한 파이썬 프로그램을 작성하는 것이다. 웹서버에 접속하고 HTTP 프로토콜 규칙에 따라 문서를 요청하고 서버가 다시 보내주는 결과를 보여주는 것이다.

```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('www.py4inf.com', 80))
mysock.send('GET http://www.py4inf.com/code/romeo.txt HTTP/1.0\n\n')

while True:
    data = mysock.recv(512)
    if ( len(data) < 1 ) :
        break
    print data

mysock.close()
```

처음에 프로그램은 `www.py4inf.com` 서버에 80 포트로 연결한다."웹 브라우져" 역할로 작성된 프로그램이 하기 때문에 HTTP 프로토콜은 GET 명령어를 공백 라인과 함께 보낸다.



공백 라인을 보내자 마자, 512 문자 덩어리의 데이터를 소켓에서 받아 더 이상 읽을 데이터가 없을 때까지(즉, recv()이 빈 문자열을 반환한다.) 데이터를 출력하는 루프를 작성한다.

프로그램 실행결과 다음을 얻을 수 있다.

```
HTTP/1.1 200 OK
Date: Sun, 14 Mar 2010 23:52:41 GMT
Server: Apache
Last-Modified: Tue, 29 Dec 2009 01:31:22 GMT
ETag: "143c1b33-a7-4b395bea"
Accept-Ranges: bytes
Content-Length: 167
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

출력결과는 웹서버가 문서를 기술하기 위해서 보내는 헤더(header)로 시작한다. 예를 들어, `Content-Type` 헤더는 문서가 일반 텍스트 문서(`text/plain`)임을 표기한다.

서버가 헤더를 보낸 후에, 빈 라인을 추가해서 헤더 끝임을 표기하고 나서 실제 파일`romeo.txt`을 보낸다.

이 예제를 통해서 소켓을 통해서 저수준(low-level) 네트워크 연결을 어떻게 하는지 확인할 수 있다. 소켓을 사용해서 웹서버, 메일 서버 혹은 다른 종류의 서버와 통신할 수 있다. 필요한 것은 프로토콜을 기술하는 문서를 찾고 프로토콜에 따라 데이터를 주고 받는 코드를 작성하는 것이다.

하지만, 가장 흔히 사용하는 프로토콜은 HTTP (즉, 웹) 프로토콜이기 때문에, 파이썬에는 HTTP 프로토컬을 지원하기 위해 특별히 설계된 라이브러리가 있다. 이것을 통해서 웹상에서 데이터나 문서를 검색을 쉽게 할 수 있다.

제 3 절 HTTP를 통해서 이미지 가져오기

상기 예제에서는 파일에 새줄(newline)이 있는 일반 텍스트 파일을 가져왔다. 그리고 나서, 프로그램을 실행해서 데이터를 단순히 화면에 복사했다. HTTP를 사용하여 이미지를 가져오도록 비슷하게 프로그램을 작성할 수 있다. 프로그램 실행 시에 화면에 데이터를 복사하는 대신에, 데이터를 문자열로 누적하고, 다음과 같이 헤더를 잘라내고 나서 파일에 이미지 데이터를 저장한다.

```
import socket
import time

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('www.py4inf.com', 80))
mysock.send('GET http://www.py4inf.com/cover.jpg HTTP/1.0\n\n')


count = 0
picture = "";
```

```
while True:
    data = mysock.recv(5120)
    if ( len(data) < 1 ) : break
    # time.sleep(0.25)
    count = count + len(data)
    print len(data),count
    picture = picture + data

mysock.close()

# Look for the end of the header (2 CRLF)
pos = picture.find("\r\n\r\n");
print 'Header length',pos
print picture[:pos]

# Skip past the header and save the picture data
picture = picture[pos+4:]
fhand = open("stuff.jpg","wb")
fhand.write(picture);
fhand.close()
```

프로그램을 실행하면, 다음과 같은 출력을 생성한다.

```
$ python urljpeg.py
2920 2920
1460 4380
1460 5840
1460 7300
...
1460 62780
1460 64240
2920 67160
1460 68620
1681 70301
Header length 240
HTTP/1.1 200 OK
Date: Sat, 02 Nov 2013 02:15:07 GMT
Server: Apache
Last-Modified: Sat, 02 Nov 2013 02:01:26 GMT
ETag: "19c141-111a9-4ea280f8354b8"
Accept-Ranges: bytes
Content-Length: 70057
Connection: close
Content-Type: image/jpeg
```

상기 url에 대해서, `Content-Type` **헤더가 문서 본문이 이미지(**`image/jpeg`**)를 나타내는 것을 볼 수 있다. 프로그램이 완료되면, 이미지 뷰어로** `stuff.jpg` **파일을 열어서 이미지 데이터를 볼 수 있다.**

프로그램을 실행하면, `recv()` **메쏘드를 호출할 때 마다 5120 문자는 전달받지 못하는 것을 볼 수 있다.** `recv()` **호출하는 순간마다 웹서버에서 네트워크로 전송되는 가능한 많은 문자를 받을 뿐이다. 매번 5120 문자까지 요청하지만, 1460 혹은 2920 문자만 전송받는다.**

결과값은 네트워크 속도에 따라 달라질 수 있다. recv() 메쏘드 마지막 호출에는 스트림 마지막인 1681 바이트만 받았고, recv() 다음 호출에는 0 길이 문자열을 전송받아서, 서버가 소켓 마지막에 close() 메쏘드를 호출하고 더이상의 데이터가 없다는 신호를 준다.

주석 처리한 time.sleep()을 풀어줌으로써 recv() 연속 호출을 늦출 수 있다. 이런 방식으로 매번 호출 후에 0.25초 기다리게 한다. 그래서, 사용자가 recv() 메쏘드를 호출하기 전에 서버가 먼저 도착할 수 있어서 더 많은 데이터를 보낼 수가 있다. 정지 시간을 넣어서 프로그램을 다시 실행하면 다음과 같다.

```
$ python urljpeg.py
1460 1460
5120 6580
5120 11700
...
5120 62900
5120 68020
2281 70301
Header length 240
HTTP/1.1 200 OK
Date: Sat, 02 Nov 2013 02:22:04 GMT
Server: Apache
Last-Modified: Sat, 02 Nov 2013 02:01:26 GMT
ETag: "19c141-111a9-4ea280f8354b8"
Accept-Ranges: bytes
Content-Length: 70057
Connection: close
Content-Type: image/jpeg
```

recv() 메쏘드 호출의 처음과 마지막을 제외하고, 매번 새로운 데이터를 요청할 때마다 이제 5120 문자가 전송된다.

서버 send() 요청과 응용프로그램 recv() 요청 사이에 버퍼가 있다. 프로그램에 지연을 넣어 실행하게 될 때, 어느 지점엔가 서버가 소켓 버퍼를 채우고 응용프로그램이 버퍼를 비울 때까지 잠시 멈춰야 된다. 송신하는 응용프로그램 혹은 수신하는 응용프로그램을 멈추게 하는 행위를 "흐름 제어(flow control)"이라고 한다.

제 4 절 urllib 사용하여 웹페이지 가져오기

수작업으로 소켓 라이브러리를 사용하여 HTTP로 데이터를 주고 받을 수 있지만, urllib 라이브러리를 사용하여 파이썬에서 동일한 작업을 수행하는 좀더 간편한 방식이 있다.

urllib을 사용하여 파일처럼 웹페이지를 다룰 수가 있다. 단순하게 어느 웹페이지를 가져올 것인지만 지정하면 urllib 라이브러리가 모든 HTTP 프로토콜과 헤더관련 사항을 처리해 준다.

웹에서 romeo.txt 파일을 읽도록 urllib를 사용하여 작성한 상응 코드는 다음과 같다.

```
import urllib

fhand = urllib.urlopen('http://www.py4inf.com/code/romeo.txt')
for line in fhand:
   print line.strip()
```

urllib.urlopen을 사용하여 웹페이지를 열게 되면, 파일처럼 다룰 수 있고 for 루프를 사용하여 데이터를 읽을 수 있다.

프로그램을 실행하면, 파일 내용 출력결과만을 볼 수 있다. 헤더정보는 여전히 전송되었지만, urllib 코드가 헤더를 받아 내부적으로 처리하고, 사용자에게는 단지 데이터만 반환한다.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

예제로, romeo.txt 데이터를 가져와서 파일의 각 단어 빈도를 계산하는 프로그램을 다음과 같이 작성할 수 있다.

```
import urllib

counts = dict()
fhand = urllib.urlopen('http://www.py4inf.com/code/romeo.txt')
for line in fhand:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word,0) + 1
print counts
```

다시 한번, 웹페이지를 열게 되면, 로컬 파일처럼 웹페이지를 읽을 수 있다.

제 5 절 HTML 파싱과 웹 스크래핑

파이썬 urllib 활용하는 일반적인 사례는 웹 스크래핑(scraping)이다. 웹 스크래핑은 웹브라우저를 가장한 프로그램을 작성하는 것이다. 웹페이지를 가져와서, 패턴을 찾아 페이지 내부의 데이터를 꼼꼼히 조사한다. 예로, 구글같은 검색엔진은 웹 페이지의 소스를 조사해서 다른 페이지로 가는 링크를 추출하고, 그 해당 페이지를 가져와서 링크 추출하는 작업을 반복한다. 이러한 기법으로 구글은 웹상의 거의 모든 페이지를 거미(spiders)줄처럼 연결한다.

구글은 또한 발견한 웹페이지에서 특정 페이지로 연결되는 링크 빈도를 사용하여 얼마나 중요한 페이지인지를 측정하고 검색결과에 페이지가 얼마나 높은 순위로 노출되어야 하는지 평가한다.

제 6 절 정규 표현식 사용 HTML 파싱하기

HTML을 파싱하는 간단한 방식은 정규 표현식을 사용하여 특정한 패턴과 매칭되는 부속 문자열을 반복적으로 찾아 추출하는 것이다.

여기 간단한 웹페이지가 있다.

```
<h1>The First Page</h1>
<p>
If you like, you can switch to the
<a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>.
</p>
```

모양 좋은 정규표현식을 구성해서 다음과 같이 상기 웹페이지에서 링크를 매칭하고 추출할 수 있다.

```
href="http://.+?"
```

작성된 정규 표현식은 "href="http://"로 시작하고, 하나 이상의 문자를 ".+?" 가지고 큰 따옴표를 가진 문자열을 찾는다. ".+?"에 물음표가 갖는 의미는 매칭이 "욕심쟁이(greedy)" 방식보다 "비욕심쟁이(non-greedy)" 방식으로 수행됨을 나타낸다. 비욕심쟁이(non-greedy) 매칭방식은 가능한 *가장 적게* 매칭되는 문자열을 찾는 방식이고, 욕심 방식은 가능한 *가장 크게* 매칭되는 문자열을 찾는 방식이다.

추출하고자 하는 문자열이 매칭된 문자열의 어느 부분인지를 표기하기 위해서 정규 표현식에 괄호를 추가하여 다음과 같이 프로그램을 작성한다.

```
import urllib
import re

url = raw_input('Enter - ')
html = urllib.urlopen(url).read()
links = re.findall('href="(http://.*?)"', html)
for link in links:
    print link
```

`findall` 정규 표현식 메쏘드는 정규 표현식과 매칭되는 모든 문자열 리스트를 추출하여 큰 따옴표 사이에 링크 텍스트만을 반환한다.

프로그램을 실행하면, 다음 출력을 얻게된다.

```
python urlregex.py
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm

python urlregex.py
Enter - http://www.py4inf.com/book.htm
http://www.greenteapress.com/thinkpython/thinkpython.html
http://allendowney.com/
http://www.py4inf.com/code
http://www.lib.umich.edu/espresso-book-machine
http://www.py4inf.com/py4inf-slides.zip
```

정규 표현식은 HTML이 예측가능하고 잘 구성된 경우에 멋지게 작동한다. 하지만, "망가진" HTML 페이지가 많아서, 정규 표현식만을 사용하는 솔류션은 유효한 링크를 놓치거나 잘못된 데이터만 찾고 끝날 수 있다.

이 문제는 강건한 HTML 파싱 라이브러리를 사용해서 해결될 수 있다.

제 7 절 BeautifulSoup 사용한 HTML 파싱

HTML을 파싱하여 페이지에서 데이터를 추출할 수 있는 파이썬 라이브러리는 많이 있다. 라이브러리 각각은 강점과 약점이 있어서 사용자 필요에 따라 취사 선택한다.

예로, 간단하게 HTML 입력을 파싱하여 BeautifulSoup 라이브러리를 사용하여 링크를 추출할 것이다. 다음 웹사이트에서 BeautifulSoup 코드를 다운로드 받아 설치할 수 있다.

www.crummy.com

BeautifulSoup 라이브러리를 다운로드 받아 "설치"하거나 `BeautifulSoup.py` **파일을 응용프로그램과 동일한 폴더에 저장한다.**

HTML이 XML 처럼 보이고 몇몇 페이지는 XML로 되도록 꼼꼼하게 구축되었지만, 일반적으로 대부분의 HTML이 깨져서 XML 파서가 HTML 전체 페이지를 잘못 구성된 것으로 간주하고 받아들이지 않는다. BeautifulSoup 라이브러리는 결점 많은 HTML 페이지에 내성이 있어서 사용자가 필요로하는 데이터를 쉽게 추출할 수 있게 한다.

`urllib`**를 사용하여 페이지를 읽어들이고,** `BeautifulSoup`**를 사용해서 앵커 태그 (**`a`**)로부터** `href` **속성을 추출한다.**

```
import urllib
from BeautifulSoup import *

url = raw_input('Enter - ')
html = urllib.urlopen(url).read()
soup = BeautifulSoup(html)

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
   print tag.get('href', None)
```

프로그램이 웹 주소를 입력받고, 웹페이지를 열고, 데이터를 읽어서 BeautifulSoup 파서에 전달하고, 그리고 나서 모든 앵커 태그를 불러와서 각 태그별로 `href` **속성을 출력한다.**

프로그램을 실행하면, 아래와 같다.

```
python urllinks.py
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm

python urllinks.py
Enter - http://www.py4inf.com/book.htm
http://www.greenteapress.com/thinkpython/thinkpython.html
http://allendowney.com/
http://www.si502.com/
http://www.lib.umich.edu/espresso-book-machine
http://www.py4inf.com/code
http://www.pythonlearn.com/
```

BeautifulSoup을 사용하여 다음과 같이 각 태그별로 다양한 부분을 뽑아낼 수 있다.

```
import urllib
from BeautifulSoup import *

url = raw_input('Enter - ')
html = urllib.urlopen(url).read()
soup = BeautifulSoup(html)

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
   # Look at the parts of a tag
   print 'TAG:',tag
   print 'URL:',tag.get('href', None)
   print 'Content:',tag.contents[0]
   print 'Attrs:',tag.attrs
```

상기 프로그램은 다음을 출력합니다.

```
python urllink2.py
Enter - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Content: [u'\nSecond Page']
Attrs: [(u'href', u'http://www.dr-chuck.com/page2.htm')]
```

HTML을 파싱하는데 BeautifulSoup이 가진 강력한 기능을 예제로 보여줬다. 좀더 자세한 사항은 `www.crummy.com` 에서 문서와 예제를 살펴보세요.

제 8 절 urllib을 사용하여 바이너리 파일 읽기

이미지나 비디오 같은 텍스트가 아닌 (혹은 바이너리) 파일을 가져올 때가 종종 있다. 일반적으로 이런 파일 데이터를 출력하는 것은 유용하지 않다. 하지만, `urllib`을 사용하여, 하드 디스크 로컬 파일에 URL 사본을 쉽게 만들 수 있다.

이 패턴은 URL을 열고, `read`를 사용해서 문서 전체 내용을 다운로드하여 문자열 변수(`img`)에 다운로드하고, 그리고 나서 다음과 같이 정보를 로컬 파일에 쓴다.

```
img = urllib.urlopen('http://www.py4inf.com/cover.jpg').read()
fhand = open('cover.jpg', 'w')
fhand.write(img)
fhand.close()
```

작성된 프로그램은 네트워크로 모든 데이터를 한번에 읽어서 컴퓨터 주기억장치 `img` 변수에 저장하고, `cover.jpg` 파일을 열어 디스크에 데이터를 쓴다. 이 방식은 파일 크기가 사용자 컴퓨터의 메모리 크기보다 작다면 정상적으로 작동한다.

하지만, 오디오 혹은 비디오 파일 대용량이면, 상기 프로그램은 멈추거나 사용자 컴퓨터 메모리가 부족할 때 극단적으로 느려질 수 있다. 메모리 부족을 회피하기 위해서, 데이터를 블럭 혹은 버퍼로 가져와서, 다음 블럭을 가져오기 전에 디스크에 각각의 블럭을 쓴다. 이런 방식으로 사용자가 가진 모든 메모리를 사용하지 않고 어떠한 크기 파일도 읽어올 수 있다.

```
import urllib

img = urllib.urlopen('http://www.py4inf.com/cover.jpg')
fhand = open('cover.jpg', 'w')
size = 0
while True:
    info = img.read(100000)
    if len(info) < 1 : break
    size = size + len(info)
    fhand.write(info)

print size,'characters copied.'
fhand.close()
```

상기 예제에서, 한번에 100,000 문자만 읽어 오고, 웹에서 다음 100,000 문자를 가져오기 전에 `cover.jpg` 파일에 읽어온 문자를 쓴다.

프로그램 실행 결과는 다음과 같다.

```
python curl2.py
568248 characters copied.
```

UNIX 혹은 매킨토시 컴퓨터를 가지고 있다면, 다음과 같이 상기 동작을 수행하는 명령어가 운영체제 자체에 내장되어 있다.

```
curl -O http://www.py4inf.com/cover.jpg
```

`curl`은 "copy URL"의 단축 명령어로 두 예제는 `curl` 명령어와 비슷한 기능을 구현해서, **`www.py4inf.com/code`** 사이트에 `curl1.py`, `curl2.py` 이름으로 올라가 있다. 동일한 작업을 좀더 효과적으로 수행하는 `curl3.py` 샘플 프로그램도 있어서 실무적으로 작성하는 프로그램에 이런한 패턴을 이용하여 구현할 수 잇다.

제 9 절 용어정의

BeautifulSoup: 파이썬 라이브러리로 HTML 문서를 파싱하고 브라우저가 일반적으로 생략하는 HTML의 불완전한 부분을 보정하여 HTML 문서에서 데이터를 추출한다. **`www.crummy.com`** 사이트에서 BeautifulSoup 코드를 다운로드 받을 수 있다.

포트(port): 서버에 소켓 연결을 만들 때, 사용자가 무슨 응용프로그램을 연결하는지 나타내는 숫자. 예로, 웹 트래픽은 통상 80 포트, 전자우편은 25 포트를 사용한다.

스크래핑(scraping): 프로그램이 웹브라우저를 가장하여 웹페이지를 가져와서 웹 페이지의 내용을 검색한다. 종종 프로그램이 한 페이지의 링크를 따라 다른 페이지를 찾게 된다. 그래서, 웹페이지 네트워크 혹은 소셜 네트워크 전체를 훑을 수 있다.

소켓(socket): 두 응용프로그램 사이 네트워크 연결. 두 응용프로그램은 양 방향으로 데이터를 주고 받는다.

스파이더(spider): 검색 색인을 구축하기 위해서 한 웹페이지를 검색하고, 그 웹페이지에 링크된 모든 페이지 검색을 반복하여 인터넷에 있는 거의 모든 웹페이지를 가져오기 위해서 사용되는 검색엔진 행동.

제 10 절 연습문제

Exercise 12.1 소켓 프로그램 `socket1.py`을 변경하여 임의 웹페이지를 읽을 수 있도록 URL을 사용자가 입력하도록 바꾸세요. `split('/')`을 사용하여 URL을 컴포턴트로 쪼개서 소켓 `connect` 호출에 대해 호스트 명을 추출할 수 있다. 사용자가 적절하지 못한 형식 혹은 존재하지 않는 URL을 입력하는 경우를 처리할 있도록 `try`, `except`를 사용하여 오류 검사기능을 추가하세요.

Exercise 12.2 소켓 프로그램을 변경하여 전송받은 문자를 계수(count)하고 3000 문자를 출력한 후에 그이상 텍스트 출력을 멈추게 하세요. 프로그램은 전체 문서를 가져와야 하고, 전체 문자를 계수(count)하고, 문서 마지막에 문자 계수(count)결과를 출력해야 합니다.

Exercise 12.3 `urllib`을 사용하여 이전 예제를 반복하세요. (1) 사용자가 입력한 URL에서 문서 가져오기 (2) 3000 문자까지 화면에 보여주기 (3) 문서의 전체 문자 계수(count)하기. 이 연습문제에서 헤더에 대해서는 걱정하지 말고, 단지 문서 본문에서 첫 3000 문자만 화면에 출력하세요.

Exercise 12.4 `urllinks.py` 프로그램을 변경하여 가져온 HTML 문서에서 문단 (p) 태그를 추출하고 프로그램의 출력물로 문단을 계수(count)하고 화면에 출력하세요. 문단 텍스트를 화면에 출력하지 말고 단지 숫자만 셉니다. 작성한 프로그램을 작은 웹페이지 뿐만 아니라 조금 큰 웹 페이지에도 테스트해 보세요.

Exercise 12.5 (고급) 소켓 프로그램을 변경하여 헤더와 빈 라인 다음에 데이터만 보여지게 하세요. `recv`는 라인이 아니라 문자(새줄(newline)과 모든 문자)를 전송받는다는 것을 기억하세요.

13 장

웹서비스 사용하기

프로그램을 사용하여 HTTP상에서 문서를 가져와서 파싱하는 것이 익숙해지면, 다른 프로그램(즉, 브라우져에서 HTML로 보여지지 않는 것)에서 활용되도록 특별히 설계된 문서를 생성하는 것은 그다지 오래 걸리지 않는다.

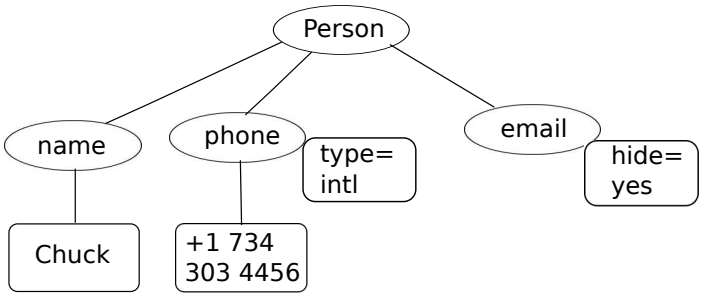
웹상에서 데이터를 교환할 때 두 가지 형식이 많이 사용된다. XML("eXtensible Markup Language")은 오랜 기간 사용되어져 왔고 문서-형식(document-style) 데이터를 교환하는데 가장 적합하다. 딕셔너리, 리스트 혹은 다른 내부 정보를 프로그램으로 서로 교환할 때, JSON(JavaScript Object Notation, `www.json.org`)을 사용한다. 두 가지 형식에 대해 모두 살펴볼 것이다.

제 1 절 XML(eXtensible Markup Language)

XML은 HTML과 매우 유사하지만, XML이 좀더 HTML보다 구조화되었다. 여기 XML 문서 샘플이 있다.

```
<person>
  <name>Chuck</name>
  <phone type="intl">
     +1 734 303 4456
   </phone>
   <email hide="yes"/>
</person>
```

종종 XML문서를 나무 구조(tree structure)로 생각하는 것이 도움이 된다. 최상단 `person` 태그가 있고, `phone` 같은 다른 태그는 부모 노드의 *자식(children)* 노드로 표현된다.



제 2 절 XML 파싱

다음은 XML을 파싱하고 XML에서 데이터 요소를 추출하는 간단한 응용프로그램이다.

```
import xml.etree.ElementTree as ET

data = '''
<person>
  <name>Chuck</name>
  <phone type="intl">
     +1 734 303 4456
   </phone>
   <email hide="yes"/>
</person>'''

tree = ET.fromstring(data)
print 'Name:',tree.find('name').text
print 'Attr:',tree.find('email').get('hide')
```

fromstring을 호출하여 XML 문자열 표현을 XML 노드 '나무(tree)'로 변환한다. XML이 나무구조로 되었을 때, XML에서 데이터 일부분을 추출하기 위해서 호출하는 메쏘드가 연달아 있다.

find 함수는 XML 나무를 훑어서 특정한 태그와 매칭되는 노드(node)를 검색한다. 각 노드는 텍스트, 속성(즉, hide 같은), 그리고 "자식(child)" 노드로 구성된다. 각 노드는 노드 나무의 최상단이 될 수 있다.

```
Name: Chuck
Attr: yes
```

ElementTree같은 XML 파서를 사용하는 것은 장점이 있다. 상기 예제의 XML은 매우 간단하지만, 적합한 XML에 관해서 규칙이 많이 있고, XML 구문 규칙에 얽매이지 않고 ElementTree를 사용해서 XML에서 데이터를 추출할 수 있다.

제 3 절 노드 반복하기

종종 XML이 다중 노드를 가지고 있어서 모든 노드를 처리하는 루프를 작성할 필요가 있다. 다음 프로그램에서 모든 user 노드를 루프로 반복한다.

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
    <users>
        <user x="2">
            <id>001</id>
            <name>Chuck</name>
        </user>
        <user x="7">
            <id>009</id>
            <name>Brent</name>
        </user>
    </users>
</stuff>'''

stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print 'User count:', len(lst)

for item in lst:
    print 'Name', item.find('name').text
    print 'Id', item.find('id').text
    print 'Attribute', item.get('x')
```

findall 메쏘드는 파이썬 리스트의 하위 나무를 가져온다. 리스트는 XML 나무에서 user 구조를 표현한다. 그리고 나서, for 루프를 작성해서 각 user 노드 값을 확인하고 name, id 텍스트 요소와 user 노드에서 x 속성도 출력한다.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

제 4 절 JSON(JavaScript Object Notation)

JSON 형식은 자바스크립트 언어에서 사용되는 객체와 배열 형식에서 영감을 얻었다. 하지만 파이썬이 자바스크립트 이전에 개발되어서 딕셔너리와 리스트의 파이썬 구문이 JSON 구문에 영향을 주었다. 그래서 JSON 포맷이 거의 파이썬 리스트와 딕셔너리의 조합과 일치한다.

상기 간단한 XML에 대략 상응하는 JSON으로 작성한 것이 다음에 있다.

```
{
  "name" : "Chuck",
  "phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456"
   },
```

```
   "email" : {
     "hide" : "yes"
   }
}
```

몇가지 차이점에 주목하세요. 첫째로 XML에서는 "phone" 태그에 "intl"같은 속성을 추가할 수 있다. JSON 에서는 단지 키-값 페어(key-value pair)다. 또한 XML "person" 태그는 사라지고 외부 중괄호 세트로 대체되었다.

일반적으로 JSON 구조가 XML 보다 간단하다. 왜냐하면, JSON 이 XML보다 적은 역량을 보유하기 때문이다. 하지만 JSON 이 딕셔너리와 리스트의 조합에 *직접* 매핑된다는 장점이 있다. 그리고, 거의 모든 프로그래밍 언어가 파이썬 딕셔너리와 리스트에 상응하는 것을 갖고 있어서, JSON 이 협업하는 두 프로그램 사이에서 데이터를 교환하는 매우 자연스러운 형식이 된다.

XML 에 비해서 상대적으로 단순하기 때문에, JSON 이 응용프로그램 간 거의 모든 데이터를 교환하는데 있어 빠르게 선택되고 있다.

제 5 절 JSON 파싱하기

딕셔너리(객체)와 리스트를 중첩함으로써 JSON을 생성한다. 이번 예제에서, user 리스트를 표현하는데, 각 user가 키-값 페어(key-value pair, 즉, 딕셔너리)다. 그래서 리스트 딕셔너리가 있다.

다음 프로그램에서 내장된 json 라이브러리를 사용하여 JSON을 파싱하여 데이터를 읽어온다. 이것을 상응하는 XML 데이터, 코드와 비교해 보세요. JSON은 조금 덜 정교해서 사전에 미리 리스트를 가져오고, 리스트가 사용자이고, 각 사용자가 키-값 페어 집합임을 알고 있어야 한다. JSON은 좀더 간략(장점)하고 하지만 좀더 덜 서술적(단점)이다.

```
import json

input = '''
[
  { "id" : "001",
    "x" : "2",
    "name" : "Chuck"
  } ,
  { "id" : "009",
    "x" : "7",
    "name" : "Chuck"
  }
]'''

info = json.loads(input)
print 'User count:', len(info)

for item in info:
    print 'Name', item['name']
    print 'Id', item['id']
    print 'Attribute', item['x']
```

JSON과 XML에서 데이터를 추출하는 코드를 비교하면, json.loads()을 통해서 파이썬 리스트를 얻는다. for 루프로 파이썬 리스트를 훑고, 리스트 내부의 각 항목은 파이썬 딕셔너리로 각 사용자별 다양한 정보를 추출하기 위해서 파이썬 인덱스 연산자를 사용한다. JSON을 파싱하면, 네이티브 파이썬 객체와 구조가 생성된다. 반환된 데이터가 단순히 네이티브 파이썬 구조체이기 때문에, 파싱된 JSON을 활용하는데 JSON 라이브러리를 사용할 필요는 없다.

프로그램 출력은 정확하게 상기 XML 버젼과 동일한다.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

일반적으로 웹서비스에 대해서 XML에서 JSON으로 옮겨가는 산업 경향이 뚜렷하다. JSON이 프로그래밍 언어에서 이미 갖고 있는 네이티브 자료 구조와 좀더 직접적이며 간단히 매핑되기 때문에, JSON을 사용할 때 파싱하고 데이터 추출하는 코드가 더욱 간단하고 직접적이다. 하지만 XML 이 JSON 보다 좀더 자기 서술적이고 XML 이 강점을 가지는 몇몇 응용프로그램 분야가 있다. 예를 들어, 대부분의 워드 프로세서는 JSON보다는 XML을 사용하여 내부적으로 문서를 저장한다.

제 6 절 API(Application Program Interfaces, 응용 프로그램 인터페이스)

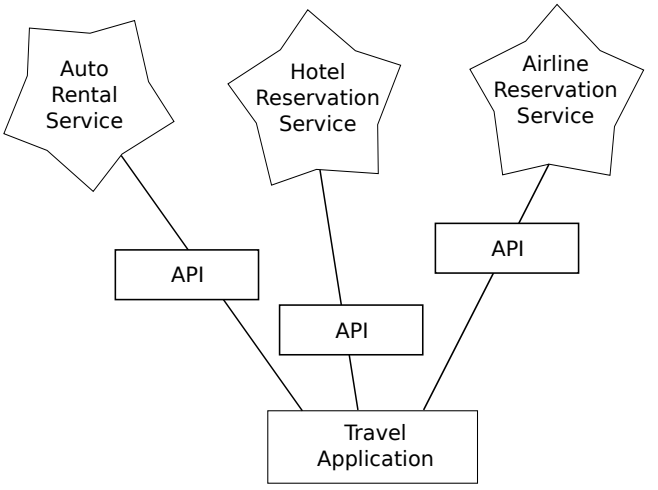
이제 HTTP를 사용하여 응용프로그램간에 데이터를 교환할 수 있게 되었다. 또한, XML 혹은 JSON을 사용하여 응용프로그램간에도 복잡한 데이터를 주고받을 수 있는 방법을 습득했다.

다음 단계는 상기 학습한 기법을 사용하여 응용프로그램 간에 "계약(contract)"을 정의하고 문서화한다. 응용프로그램-대-응용프로그램 계약에 대한 일반적 명칭은 API 응용 프로그램 인터페이스(Application Program Interface) 다. API를 사용할 때, 일반적으로 하나의 프로그램이 다른 응용 프로그램에서 사용할 수 있는 가능한 서비스 집합을 생성한다. 또한, 다른 프로그램이 서비스에 접근하여 사용할 때 지켜야하는 API (즉, "규칙")도 게시한다.

다른 프로그램에서 제공되는 서비스에 접근을 포함하여 프로그램 기능을 개발할 때, 이러한 개발법을 SOA, Service-Oriented Architecture(서비스 지향 아키텍처)라고 부른다. SOA 개발 방식은 전반적인 응용 프로그램이 다른 응용 프로그램 서비스를 사용하는 것이다. 반대로, SOA가 아닌 개발방식은 응용 프로그램이 하나의 독립된 응용 프로그램으로 구현에 필요한 모든 코드를 담고 있다.

웹을 사용할 때 SOA 사례를 많이 찾아 볼 수 있다. 웹사이트 하나를 방문해서 비행기표, 호텔, 자동차를 단일 사이트에서 예약완료한다. 호텔관련 데이터는

물론 항공사 컴퓨터에 저장되어 있지 않다. 대신에 항공사 컴퓨터는 호텔 컴퓨터와 계약을 맺어 호텔 데이터를 가져와서 사용자에게 보여준다. 항공사 사이트를 통해서 사용자가 호텔 예약을 동의할 경우, 항공사 사이트에서 호텔 시스템의 또다른 웹서비스를 통해서 실제 예약을 한다. 전체 거래(transaction)를 완료하고 카드 결재를 진행할 때, 다른 컴퓨터가 프로세스에 관여하여 처리한다.



서비스 지향 아키텍쳐는 많은 장점이 있다. (1) 항상 단 하나의 데이터만 유지관리한다. 이중으로 중복 예약을 원치 않는 호텔 같은 경우에 매우 중요하다. (2) 데이터 소유자가 데이터 사용에 대한 규칙을 정한다. 이러한 장점으로, SOA 시스템은 좋은 성능과 사용자 요구를 모두 만족하기 위해서 신중하게 설계되어야 한다.

응응프로그램이 웹상에 이용가능한 API로 서비스 집합을 만들 때, 웹서비스(web services)라고 부른다.

제 7 절 구글 지오코딩 웹서비스(Google Geocoding Web Service)

구글이 자체적으로 구축한 대용량 지리 정보 데이터베이스를 누구나 이용할 수 있게 하는 훌륭한 웹서비스가 있다. “Ann Arbor, MI” 같은 지리 검색 문자열을 지오코딩 API 에 넣으면, 검색 문자열이 의미하는 지도상에 위치와 근처 주요 지형지물 정보를 나름 최선을 다해서 예측 제공한다.

지오코딩 서비스는 무료지만 사용량이 제한되어 있어서, 상업적 응용프로그램에 API를 무제한 사용할 수는 없다. 하지만, 최종 사용자가 자유형식 입력 박스에 위치정보를 입력하는 설문 데이터가 있다면, 구글 API를 사용하여 데이터를 깔끔하게 정리하는데는 유용하다.

구글 지오코딩 API 같은 무료 API를 사용할 때, 자원 사용에 대한 지침을 준수해야 한다. 너무나 많은 사람이 서비스를 남용하게 되면, 구글은 무료 서비스를 중단하거나, 상당부분 줄일 수 있다. (When you are using a free API like Google’s

geocoding API, you need to be respectful in your use of these resources. If too many people abuse the service, Google might drop or significantly curtail its free service.

서비스에 대해서 자세한 사항을 온라인 문서를 정독할 수 있지만, 무척 간단해서 브라우저에 다음 URL을 입력해서 테스트까지 할 수 있다.

`http://maps.googleapis.com/maps/api/geocode/json?sensor=false&address=Ann+Arbor%2C+MI`

웹프라우저에 붙여넣기 전에, URL만 뽑아냈고 URL에서 모든 공백을 제거했는지 확인하세요.

다음은 간단한 응용 프로그램이다. 사용자가 검색 문자열을 입력하고 구글 지오코딩 API를 호출하여 반환된 JSON에서 정보를 추출한다.

```
import urllib
import json

serviceurl = 'http://maps.googleapis.com/maps/api/geocode/json?'

while True:
    address = raw_input('Enter location: ')
    if len(address) < 1 : break

    url = serviceurl + urllib.urlencode({'sensor':'false',
          'address': address})
    print 'Retrieving', url
    uh = urllib.urlopen(url)
    data = uh.read()
    print 'Retrieved',len(data),'characters'

    try: js = json.loads(str(data))
    except: js = None
    if 'status' not in js or js['status'] != 'OK':
        print '==== Failure To Retrieve ===='
        print data
        continue

    print json.dumps(js, indent=4)

    lat = js["results"][0]["geometry"]["location"]["lat"]
    lng = js["results"][0]["geometry"]["location"]["lng"]
    print 'lat',lat,'lng',lng
    location = js['results'][0]['formatted_address']
    print location
```

프로그램이 사용자로부터 검색 문자열을 받는다. 적절히 인코딩된 매개 변수로 검색문자열을 변환하여 URL을 만든다. 그리고 나서 urllib을 사용하여 구글 지오코딩 API에서 텍스트를 가져온다. 고정된 웹페이지와 달리, 반환되는 데이터는 전송한 매개변수와 구글 서버에 저장된 지리정보 데이터에 따라 달라진다.

JSON 데이터를 가져오면, json 라이브러리로 파싱하고 전송받은 데이터가 올바른지 확인하는 몇가지 절차를 거친 후에 찾고자 하는 정보를 추출한다.

프로그램 출력결과는 다음과 같다. (몇몇 JSON 출력은 의도적으로 삭제했다.)

```
$ python geojson.py
Enter location: Ann Arbor, MI
Retrieving http://maps.googleapis.com/maps/api/
  geocode/json?sensor=false&address=Ann+Arbor%2C+MI
Retrieved 1669 characters
{
    "status": "OK",
    "results": [
        {
            "geometry": {
                "location_type": "APPROXIMATE",
                "location": {
                    "lat": 42.2808256,
                    "lng": -83.7430378
                }
            },
            "address_components": [
                {
                    "long_name": "Ann Arbor",
                    "types": [
                        "locality",
                        "political"
                    ],
                    "short_name": "Ann Arbor"
                }
            ],
            "formatted_address": "Ann Arbor, MI, USA",
            "types": [
                "locality",
                "political"
            ]
        }
    ]
}
lat 42.2808256 lng -83.7430378
Ann Arbor, MI, USA
Enter location:
```

다양한 구글 지오코딩 API의 XML과 JSON을 좀더 살펴보기 위해서 `www.py4inf.com/code/geojson.py`, `www.py4inf.com/code/geoxml.py`을 다운로드 받아보기 바란다.

제 8 절 보안과 API 사용

상용업체 API를 사용하기 위해서는 일종의 "API키(API key)"가 일반적으로 필요하다. 서비스 제공자 입장에서 누가 서비스를 사용하고 있으며 각 사용자가 얼마나 사용하고 있는를 알고자 한다. 상용 API 제공업체는 서비스에 대한 무료 사용자와 유료 사용자에 대한 구분을 두고 있다. 특정 기간 동안 한 개인 사용자가 사용할 수 있는 요청수에 대해 제한을 두는 정책을 두고 있다.

때때로 API키를 얻게 되면, API를 호출할 때 POST 데이터의 일부로 포함하거나 URL의 매개변수로 키를 포함시킨다.

또 다른 경우에는 업체가 서비스 요청에 대한 보증을 강화해서 공유키와 비밀번호를 암호화된 메시지 형식으로 보내도록 요구한다. 인터넷을 통해서 서비스 요청을 암호화하는 일반적인 기술을 OAuth라고 한다. `http://www.oauth.net` 사이트에서 OAuth 프로토콜에 대해 더 많은 정보를 만날 수 있다.

트위터 API가 점차적으로 가치있게 됨에 따라 트위터가 공개된 API에서 API를 매번 호출할 때마다 OAuth 인증을 거치도록 API를 바꿔었다. 다행스럽게도 편리한 OAuth 라이브러리가 많이 있다.

그래서 명세서를 읽고 아무것도 없는 상태에서 OAuth 구현하는 것을 필할 수 있게 되었다. 이용 가능한 라이브러리는 복잡성도 다양한만틈 기능적으로도 다양하다. OAuth 웹사이트에서 다양한 OAuth 라이브러리 정보를 확인할 수 있다.

다음 샘플 프로그램으로 `www.py4inf.com/code` 사이트에서 twurl.py, hidden.py, oauth.py, twitter1.py 파일을 다운로드 받아서 컴퓨터 한 폴더에 저장한다.

프로그램을 사용하기 위해서 트위터 계정이 필요하고, 파이썬 코드를 응용프로그램으로 인증하고, 키, 암호, 토큰, 토큰 암호를 설정해야 한다. hidden.py 파일을 편집하여 4개 문자열을 파일에 적절한 변수에 저장한다.

```
def auth() :
    return { "consumer_key" : "h7L...GNg",
        "consumer_secret" : "dNK...7Q",
        "token_key" : "101...GI",
        "token_secret" : "H0yM...Bo" }
```

트위터 웹서비스는 다음 URL을 사용하여 접근한다.

`https://api.twitter.com/1.1/statuses/user_timeline.json`

하지만, 모든 비밀 정보가 추가되면, URL은 다음과 같이 보인다.

```
https://api.twitter.com/1.1/statuses/user_timeline.json?count=2
&oauth_version=1.0&oauth_token=101...SGI&screen_name=drchuck
&oauth_nonce=09239679&oauth_timestamp=1380395644
&oauth_signature=rLK...BoD&oauth_consumer_key=h7Lu...GNg
&oauth_signature_method=HMAC-SHA1
```

OAuth 보안 요구사항을 충족하기 위해 추가된 다양한 매개 변수 의미를 좀더 자세히 알고자 한다면, OAuth 명세서를 읽어보기 바란다.

트위터로 실행한 프로그램에서 oauth.py, twurl.py 두개 파일에 모든 복잡함을 감추었다. hidden.py에 암호를 설정해서 URL을 twurl.augment() 함수에 전송했고, 라이브러리 코드는 URL에 필요한 매개 변수를 추가했다.

twitter1.py 프로그램은 특정 트위터 사용자 타임라인을 가져와서 JSON 형식 문자열로 반환한다. 단순하게 문자열의 첫 250 문자만 출력한다.

```
import urllib
import twurl

TWITTER_URL='https://api.twitter.com/1.1/statuses/user_timeline.json'

while True:
    print ''
    acct = raw_input('Enter Twitter Account:')
    if ( len(acct) < 1 ) : break
    url = twurl.augment(TWITTER_URL,
        {'screen_name': acct, 'count': '2'} )
    print 'Retrieving', url
    connection = urllib.urlopen(url)
    data = connection.read()
    print data[:250]
    headers = connection.info().dict
    # print headers
    print 'Remaining', headers['x-rate-limit-remaining']
```

프로그램을 실행하면 다음 결과물을 출력한다.

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 17:30:25 +0000 2013","
id":384007200990982144,"id_str":"384007200990982144",
"text":"RT @fixpert: See how the Dutch handle traffic
intersections: http:\/\/t.co\/tIiVWtEhj4\n#brilliant",
"source":"web","truncated":false,"in_rep
Remaining 178

Enter Twitter Account:fixpert
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 18:03:56 +0000 2013",
"id":384015634108919808,"id_str":"384015634108919808",
"text":"3 months after my freak bocce ball accident,
my wedding ring fits again! :)\n\nhttps:\/\/t.co\/2XmHPx7kgX",
"source":"web","truncated":false,
Remaining 177

Enter Twitter Account:
```

반환된 타임라인 데이터와 함께 트위터는 또한 HTTP 응답 헤더에 요청사항에 대한 메타 데이터도 반환한다. 특히 헤더에 있는 x-rate-limit-remaining정보는 한동안 서비스를 이용 못하게 되기 전까지 얼마나 많은 요청을 할 수 있는가하는 정보를 담고 있다. API 요청을 매번 할 때마다 남은 숫자가 줄어드는 것을 확인할 수 있다.

다음 예제에서, 사용자의 트위터 친구 정보를 가져와서 JSON 파싱을 하고 친구에 대한 정보를 추출한다. 파싱 후에 JSON을 가져 와서, 좀더 많은 필드를 추출할 때 데이터를 자세히 살펴보는데 도움이 되도록 문자 4개로 들여쓰기한 "보기좋은 출력(pretty-print)"을 한다.

```
import urllib
import twurl
```

```
import json

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

while True:
    print ''
    acct = raw_input('Enter Twitter Account:')
    if ( len(acct) < 1 ) : break
    url = twurl.augment(TWITTER_URL,
        {'screen_name': acct, 'count': '5'} )
    print 'Retrieving', url
    connection = urllib.urlopen(url)
    data = connection.read()
    headers = connection.info().dict
    print 'Remaining', headers['x-rate-limit-remaining']
    js = json.loads(data)
    print json.dumps(js, indent=4)

    for u in js['users'] :
        print u['screen_name']
        s = u['status']['text']
        print '  ',s[:50]
```

JSON은 중첩된 파이썬 리스트와 딕셔너리 집합이기 때문에, 인덱스 연산과 `for` 루프를 조합해서 매우 적은 양의 코드로 반환된 데이터 구조를 훑어볼 수 있다.

프로그램 결과는 다음과 같다. (페이지에 맞도록 몇몇 데이터 항목을 줄였다.)

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/friends ...
Remaining 14
{
    "next_cursor": 1444171224491980205,
    "users": [
        {
            "id": 662433,
            "followers_count": 28725,
            "status": {
                "text": "@jazzychad I just bought one .__.",
                "created_at": "Fri Sep 20 08:36:34 +0000 2013",
                "retweeted": false,
            },
            "location": "San Francisco, California",
            "screen_name": "leahculver",
            "name": "Leah Culver",
        },
        {
            "id": 40426722,
            "followers_count": 2635,
            "status": {
                "text": "RT @WSJ: Big employers like Google ...",
                "created_at": "Sat Sep 28 19:36:37 +0000 2013",
            },
            "location": "Victoria Canada",
```

```
            "screen_name": "_valeriei",
            "name": "Valerie Irvine",
    ],
    "next_cursor_str": "1444171224491980205"
}
leahculver
   @jazzychad I just bought one .__.
_valeriei
   RT @WSJ: Big employers like Google, AT&amp;amp;T are h
ericbollens
   RT @lukew: sneak peek: my LONG take on the good &a
halherzog
   Learning Objects is 10. We had a cake with the LO,
scweeker
   @DeviceLabDC love it! Now where so I get that "etc

Enter Twitter Account:
```

출력 마지막은 drchuck 트위터 계정에서 가장 최근 친구 5명을 `for` 루프로 읽고 친구의 가장 마지막 상태 정보를 출력한다. 반환된 JSON에는 이용가능한 더 많은 데이터가 있다. 또한, 프로그램 출력을 보게 되면, 특정 계정의 "find the friends"가 일정 기간동안 실행 가능한 타임라인 질의 숫자와 다른 사용량에 제한을 두고 있음을 볼 수 있다.

이와 같은 보안 API키는 누가 트위터 API를 사용하고 어느 정도 수준으로 트위터를 사용하는지에 대해서 트위트가 확고한 신뢰를 갖게 한다. 사용량에 한계를 두고 서비스를 제공하는 방식은 단순히 개인적인 목적으로 데이터 검색을 할 수는 있지만, 하루에 수백만 API 호출로 데이터를 추출하여 제품을 개발 못하게 제한하는 기능도 동시에 한다.

제 9 절 용어정의

API: 응용 프로그램 인터페이스(Application Program Interface) - 두 응용 프로그램 컴포넌트 간에 상호작용하는 패턴을 정의하는 응용 프로그램 간의 계약.

ElementTree: XML데이터를 파싱하는데 사용되는 파이썬 내장 라이브러리.

JSON: JavaScript Object Notation- 자바스크립트 객체(JavaScript Objects) 구문을 기반으로 구조화된 데이터 마크업(markup)을 허용하는 형식.

REST: REpresentational State Transfer - HTTP 프로토콜을 사용하여 응용 프로그램 내부에 자원에 접근을 제공하는 일종의 웹서비스 스타일.

SOA: 서비스 지향 아키텍처(Service Oriented Architecture) - 응용 프로그램이 네트워크에 연결된 컴포넌트로 구성될 때.

XML: 확장 마크업 언어(eXtensible Markup Language) - 구조화된 데이터의 마크업을 허용하는 형식.

제 10 절 Exercises

Exercise 13.1 데이터를 가져와서 두 문자 국가 코드를 출력하도록 `www.py4inf.com/code/geojson.py` 혹은 `www.py4inf.com/code/geoxml.py`을 수정하세요. 오류 검사 기능을 추가하여 국가 코드가 없더라도 프로그램이 역추적(traceback)이 생성하지 않도록 하세요. 프로그램이 정상 작동하면, "Atlantic Ocean"을 검색하고 어느 국가에도 속하지 않는 지역을 처리할 수 있는지 확인하세요.

14 장

데이터베이스와 SQL(Structured Query Language) 사용하기

제 1 절　데이터베이스가 뭔가요?

데이터베이스(database)는 데이터를 저장하기 위한 목적으로 조직된 파일이다. 대부분의 데이터베이스는 키(key)와 값(value)를 매핑한다는 의미에서 딕셔너리처럼 조직되었다. 가장 큰 차이점은 데이터베이스는 디스크(혹은 다른 영구 저장소)에 위치하고 있어서, 프로그램 종료 후에도 정보가 계속 저장된다. 데이터베이스가 영구 저장소에 저장되어서, 컴퓨터 주기억장치(memory) 크기에 제한받는 딕셔너리보다 훨씬 더 많은 정보를 저장할 수 있다.

딕셔너리처럼, 데이터베이스 소프트웨어는 엄청난 양의 데이터 조차도 매우 빠르게 삽입하고 접근하도록 설계되었다. 컴퓨터가 특정 항목으로 빠르게 찾아갈 수 있도록 데이터베이스에 인덱스(indexes)를 추가한다. 데이터베이스 소프트웨어는 인덱스를 구축하여 성능을 보장한다.

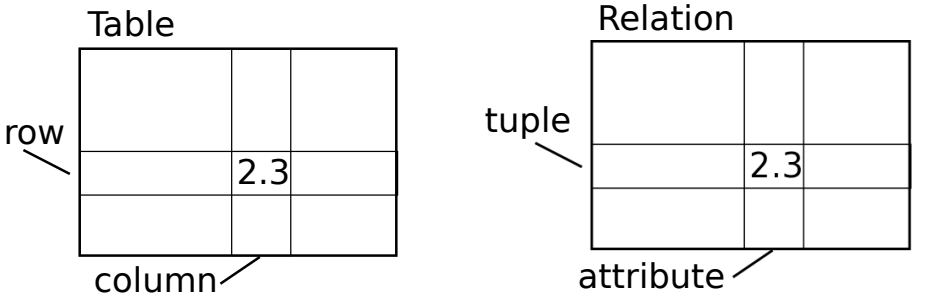
다양한 목적에 맞춰 서로 다른 많은 데이터베이스 시스템이 개발되어 사용되고 있다. Oracle, MySQL, Microsoft SQL Server, PostgreSQL, SQLite이 여기에 포함된다. 이 책에서는 SQLite를 집중해서 살펴볼 것이다. 왜냐하면 매우 일반적인 데이터베이스이며 파이썬에 이미 내장되어 있기 때문이다. 응용프로그램 내부에서 데이터베이스 기능을 제공하도록 SQLite가 다른 응용프로그램 내부에 *내장(embedded)*되도록 설계되었다. 예를 들어, 다른 많은 소프트웨어 제품이 그렇듯이, 파이어폭스 브라우져도 SQLite를 사용한다.

`http://sqlite.org/`

이번 장에서 기술하는 트위터 스파이더링 응용프로그램처럼 정보과학(Informatics)에서 마주치는 몇몇 데이터 조작 문제에 SQLite가 적합하다.

제 2 절 데이터베이스 개념

처음 데이터베이스를 볼때 드는 생각은 마치 엑셀같은 다중 시트를 지닌 스프레드쉬트(spreadsheet)같다는 것이다. 데이터베이스에서 주요 데이터 구조물은 테이블(tables), 행(rows), and 열(columns)이 된다.



관계형 데이터베이스의 기술적인 면을 설명하면 테이블, 행, 열의 개념은 관계(relation), 튜플(tuple), and 속성(attribute) 각각 형식적으로 매칭된다. 이번 장에서는 조금 덜 형식 용어를 사용한다.

제 3 절 파이어폭스 애드온 SQLite 매니저

SQLite 데이터베이스 파일에 있는 데이터를 다루기 위해서 이번장에서 주로 파이썬 사용에 집중을 하지만, 다음 웹사이트에서 무료로 이용 가능한 SQLite 데이터베이스 매니저(SQLite Database Manager)로 불리는 파이어폭스 애드온(add-on)을 사용해서 좀더 쉽게 많은 작업을 수행할 수 있다.

`https://addons.mozilla.org/en-us/firefox/addon/sqlite-manager/`

브라우져를 사용해서 쉽게 테이블을 생성하고, 데이터를 삽입, 편집하고 데이터베이스 데이터에 대해 간단한 SQL 질의를 실행할 수 있다.

이러한 점에서 데이터베이스 매니저는 텍스트 파일을 작업할 때 사용하는 텍스트 편집기와 유사하다. 텍스트 파일에 하나 혹은 몇개 작업만 수행하고자 하면, 텍스트 편집기에서 파일을 열어 필요한 수정작업을 하고 닫으면 된다. 텍스트 파일에 작업할 사항이 많은 경우는 종종 간단한 파이썬 프로그램을 작성하여 수행한다. 데이터베이스로 작업할 때도 동일한 패턴이 발견된다. 간단한 작업은 데이터베이스 매니저를 통해서 수행하고, 좀더 복잡한 작업은 파이썬으로 수행하는 것이 더 편리하다.

제 4 절 데이터베이스 테이블 생성하기

데이터베이스는 파이썬 리스트 혹은 딕셔너리보다 좀더 명확히 정의된 구조를 요구한다.[1].

[1]실질적으로 SQLite는 열에 저장되는 데이터 형식에 대해서 좀더 많은 유연성을 부여하지만, 이번 장에서는 데이터 형식을 엄격하게 적용해서 MySQL 같은 다른 관계형 데이터베이스

데이터베이스에 테이블(table)을 생성할 때, 열(column)의 명칭과 각 열(column)에 저장하는 데이터 형식을 사전에 정의해야 한다. 데이터베이스 소프트웨어가 각 열의 데이터 형식을 인식하게 되면, 데이터 형식에 따라 데이터를 저장하고 찾아오는 방법을 가장 효율적인 방식을 선택할 수 있다.

다음 url에서 SQLite에서 지원되는 다양한 데이터 형식을 살펴볼 수 있다.

http://www.sqlite.org/datatypes.html

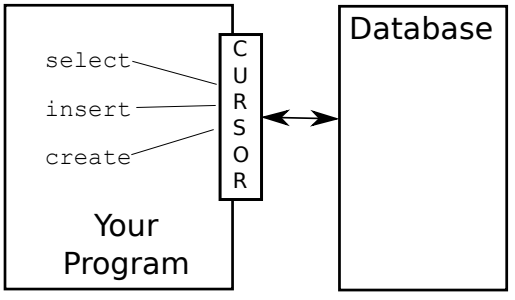
처음에는 데이터 구조를 사전에 정의하는 것이 불편하게 보이지만, 대용량의 데이터가 데이터베이스에 포함되더라도 데이터의 빠른 접근을 보장하는 잇점이 있다.

데이터베이스 파일과 데이터베이스에 두개의 열을 가진 `Tracks` 이름의 테이블을 생성하는 코드는 다음과 같다.

```
import sqlite3

conn = sqlite3.connect('music.sqlite3')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks ')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

conn.close()
```

연결 (`connect`) 연산은 현재 디렉토리 `music.sqlite3` 파일에 저장된 데이터베이스에 "연결(connection)"한다. 파일이 존재하지 않으면, 자동 생성된다. "연결(connection)"이라고 부르는 이유는 때때로 데이터베이스가 응용프로그램이 실행되는 서버로부터 분리된 "데이터베이스 서버(database server)"에 저장되기 때문이다. 지금 간단한 예제 파일의 경우에 데이터베이스가 로컬 파일 형태로 파이썬 코드 마찬가지로 동일한 디렉토리에 있다.

파일을 다루는 파일 핸들(file handle)처럼 데이터베이스에 저장된 파일에 연산을 수행하기 위해서 커서(cursor)를 사용한다. `cursor()`를 호출하는 것은 개념적으로 텍스트 파일을 다룰 때 `open()`을 호출하는 것과 개념적으로 매우 유사하다.



커서가 생성되면, `execute()` 메쏘드를 사용하여 데이터베이스 콘텐츠에 명령어 실행을 할 수 있다.

시스템에도 동일한 개념이 적용되게 한다.

데이터베이스 명령어는 특별한 언어로 표현된다. 단일 데이터베이스 언어를 학습하도록 서로 다른 많은 데이터베이스 업체 사이에서 표준화되었다.

데이터베이스 언어를 SQL(Structured Query Language 구조적 질의 언어)로 부른다.

http://en.wikipedia.org/wiki/SQL

상기 예제에서, 데이터베이스에 두개의 SQL 명령어를 실행했다. 관습적으로 데이터베이스 키워드는 대문자로 표기한다. 테이블명이나 열의 명칭처럼 사용자가 추가한 명령어 부분은 소문자로 표기한다.

첫 SQL 명령어는 만약 존재한다면 데이터베이스에서 `Tracks` 테이블을 삭제한다. 동일한 프로그램을 실행해서 오류 없이 반복적으로 `Tracks` 테이블을 생성하도록하는 패턴이다. `DROP TABLE` 명령어는 데이터베이스 테이블 및 테이블 콘텐츠 전부를 삭제하니 주의한다. (즉, "실행취소(undo)"가 없다.)

```
cur.execute('DROP TABLE IF EXISTS Tracks ')
```

두번째 명령어는 `title` 문자형 열과 `plays` 정수형 열을 가진 `Tracks`으로 명명된 테이블을 생성한다.

```
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')
```

이제 `Tracks`으로 명명된 테이블을 생성했으니, SQL `INSERT` 연산을 통해 테이블에 데이터를 넣을 수 있다. 다시 한번, 데이터베이스에 연결하여 커서(`cursor`)를 얻어 작업을 시작한다. 그리고 나서 커서를 사용해서 SQL 명령어를 수행한다.

SQL `INSERT` 명령어는 어느 테이블을 사용할지 특정한다. 그리고 나서 `(title, plays)` 포함할 필드 목록과 테이블 새로운 행에 저장될 `VALUES` 나열해서 신규 행을 정의를 마친다. 실제 값이 `execute()` 호출의 두번째 매개변수로 튜플`( 'My Way', 15 )` 로 넘겨는 것을 표기하기 위해서 값을 물음표 `(?, ?)`로 명기한다.

```
import sqlite3

conn = sqlite3.connect('music.sqlite3')
cur = conn.cursor()

cur.execute('INSERT INTO Tracks (title, plays) VALUES ( ?, ? )',
    ( 'Thunderstruck', 20 ) )
cur.execute('INSERT INTO Tracks (title, plays) VALUES ( ?, ? )',
    ( 'My Way', 15 ) )
conn.commit()

print 'Tracks:'
cur.execute('SELECT title, plays FROM Tracks')
for row in cur :
   print row

cur.execute('DELETE FROM Tracks WHERE plays < 100')
conn.commit()
```

```
cur.close()
```

먼저 테이블에 두개 열을 삽입(INSERT)하고 commit() 명령어를 사용하여 데이터가 데이터베이스에 저장되도록 했다.

Tracks

title	plays
Thunderstruck	20
My Way	15

그리고 나서, SELECT 명령어를 사용하여 테이블에 방금 전에 삽입한 행을 불러왔다. SELECT 명령어에서 데이터를 어느 열(title, plays)에서, 어느 테이블Tracks에서 을 가져올지 명세한다. SELECT 명령문을 수행한 후에, 커서는 for문 반복을 수행하는 것과 같다. 효율성을 위해서, 커서가 SELECT 명령문을 수행할 때 데이터베이스에서 모든 데이터를 읽지 않는다. 대신에 for문에서 행을 반복해서 가져하듯이 요청시에만 데이터를 읽어온다.

프로그램 실행결과는 다음과 같다.

```
Tracks:
(u'Thunderstruck', 20)
(u'My Way', 15)
```

for 루프가 행을 두개를 읽어왔다. 각각의 행은 title로 첫번째 값을,plays로 두번째 값을 갖는 파이썬 튜플이다. title 문자열이 u'로 시작한다고 걱정하지 마라. 해당 문자열은 라틴 문자가 아닌 다국어를 저장할 수 있는 유니코드(Unicode) 문자열을 나타내는 것이다.

프로그램 마지막에 SQL 명령어를 실행 사용해서 방금전에 생성한 행을 모두 삭제(DELETE) 했기 때문에 프로그램을 반복해서 실행할 수 있다. 삭제(DELETE) 명령어는 WHERE 문을 사용하여 선택 조건을 표현할 수 있다. 따라서 명령문에 조건을 충족하는 행에만 명령문을 적용한다. 이번 예제에서 기준이 모든 행에 적용되어 테이블에 아무 것도 없게 된다. 따라서 프로그램을 반복적으로 실행할 수 있다. 삭제(DELETE)를 실행한 후에 commit()을 호출하여 데이터베이스에서 데이터를 완전히 제거했다.

제 5 절 SQL(Structured Query Language) 요약

지금까지, 파이썬 예제를 통해서 SQL(Structured Query Language)을 사용했고, SQL 명령어에 대한 기본을 다루었다. 이번 장에서는 SQL 언어를 보고 SQL 구문 개요를 살펴본다.

대단히 많은 데이터베이스 업체가 존재하기 때문에 호환성의 문제로 SQL(Structured Query Language)이 표준화되었다. 그래서, 여러 업체가 개발한 데이터베이스 시스템 사이에 호환하는 방식으로 커뮤니케이션 가능하다.

관계형 데이터베이스는 테이블, 행과 열로 구성된다. 열(column)은 일반적으로 텍스트, 숫자, 혹은 날짜 자료형을 갖는다. 테이블을 생성할 때, 열의 명칭과 자료형을 지정한다.

```
CREATE TABLE Tracks (title TEXT, plays INTEGER)
```

테이블에 행을 삽입하기 위해서 SQL INSERT 명령어를 사용한다.

```
INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)
```

INSERT 문장은 테이블 이름을 명기한다. 그리고 나서 새로운 행에 놓고자 하는 열/필드 리스트를 명시한다. 그리고 나서 키워드 VALUES와 각 필드 별로 해당하는 값을 넣는다.

SQL SELECT 명령어는 데이터베이스에서 행과 열을 가져오기 위해 사용된다. SELECT 명령문은 가져오고자 하는 행과 WHERE절을 사용하여 어느 행을 가져올지 지정한다. 선택 사항으로 ORDER BY 절을 이용하여 반환되는 행을 정렬할 수도 있다.

```
SELECT * FROM Tracks WHERE title = 'My Way'
```

* 을 사용하여 WHERE 절에 매칭되는 각 행의 모든 열을 데이터베이스에서 가져온다.

주목할 점은 파이썬과 달리 SQL WHERE 절은 등식을 시험하기 위해서 두개의 등치 기호 대신에 단일 등치 기호를 사용한다. WHERE에서 인정되는 다른 논리 연산자는 <, >, <=, >=, != 이고, 논리 표현식을 생성하는데 AND, OR, 괄호를 사용한다.

다음과 같이 반환되는 행이 필드값 중 하나에 따라 정렬할 수도 있다.

```
SELECT title,plays FROM Tracks ORDER BY title
```

행을 제거하기 위해서, SQL DELETE 문장에 WHERE 절이 필요하다. WHERE 절이 어느 행을 삭제할지 결정한다.

```
DELETE FROM Tracks WHERE title = 'My Way'
```

다음과 같이 SQL UPDATE 문장을 사용해서 테이블에 하나 이상의 행 내에 있는 하나 이상의 열을 갱신(UPDATE)할 수 있다.

```
UPDATE Tracks SET plays = 16 WHERE title = 'My Way'
```

UPDATE 문장은 먼저 테이블을 명시한다. 그리고 나서, SET 키워드 다음에 변경할 필드 리스트 와 값을 명시한다. 그리고 선택사항으로 갱신될 행을 WHERE절에 지정한다. 단일 UPDATE 문장은 WHERE절에서 매칭되는 모든 행을 갱신한다. 혹은 만약 WHERE절이 지정되지 않으면,테이블 모든 행에 대해서 갱신(UPDATE)을 한다.

네가지 기본 SQL 명령문(INSERT, SELECT, UPDATE, DELETE)은 데이터를 생성하고 유지 관리하는데 필요한 기본적인 4가지 작업을 가능케 한다.

제 6 절 데이터베이스를 사용한 트위터 스파이더링(Spidering)

이번장에서 트위커 계정을 조사하고 데이터베이스를 생성하는 간단한 스파이더링 프로그램을 작성합니다. *주의: 프로그램을 실행할 때 매우 주의하세요. 여러분의 트위터 계정 접속이 차단될 정도로 너무 많은 데이터를 가져오거나 장시간 프로그램을 실행하지 마세요.*

임의 스파이더링 프로그램이 가지는 문제점 중의 하나는 종종 중단되거나 여러번 재시작할 필요가 생긴다는 것이다. 이로 사유로 지금까지 가져온 데이터를 잃을 수도 있다는 것이다. 데이터 가져오기온 처음 시점에서 항상 다시 시작하고 싶지는 않다. 그래서 데이터를 가져오면 저장하길 원한다. 프로그램이 자동으로 백업작업을 수행해서 중단된 곳에서부터 다시 가져오는 작업을 했으면 한다.

출발은 한 사람의 트위터 친구와 상태 정보를 가져오는 것에서 시작한다. 그리고, 친구 리스트를 반복하고, 향후에 가져올 수 있도록 친구 각각을 데이터베이스에 추가한다. 한 사람의 트위터 친구를 처리한 후에, 데이터베이스를 확인하고 친구의 친구 한명을 가져온다. 이것을 반복적으로 수행하고, "방문하지 않는(unvisited)" 친구를 선택하고,친구의 리스트를 가져온다. 그리고 향후 방문을 위해서 현재 리스트에서 보지 않은 친구를 추가한다.

"인기도(popularity)"를 측정하도록 데이터베이스에 특정 친구를 얼마나 자주 봤는지를 기록한다.

알고 있는 계정 리스트를 저장함으로써, 혹은 계정을 가져왔는 혹은 그렇지 않은지, 그리고 계정이 컴퓨터 하드디스크 데이터베이스에서 얼마나 인기있는지에 따라 원하는 만큼 프로그램을 멈추거나 다시 시작할 수 있다.

프로그램이 약간 복잡하다. 트위터 API를 사용한 책의 앞선 예제에서 가져온 코드에 기반하여 작성되었다.

다음이 트위터 스파이더링 응용프로그램 소스코드다.

```
import urllib
import twurl
import json
import sqlite3

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('spider.sqlite3')
cur = conn.cursor()

cur.execute('''
CREATE TABLE IF NOT EXISTS Twitter
(name TEXT, retrieved INTEGER, friends INTEGER)''')

while True:
    acct = raw_input('Enter a Twitter account, or quit: ')
    if ( acct == 'quit' ) : break
    if ( len(acct) < 1 ) :
```

```
        cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
        try:
            acct = cur.fetchone()[0]
        except:
            print 'No unretrieved Twitter accounts found'
            continue

    url = twurl.augment(TWITTER_URL,
              {'screen_name': acct, 'count': '20'} )
    print 'Retrieving', url
    connection = urllib.urlopen(url)
    data = connection.read()
    headers = connection.info().dict
    # print 'Remaining', headers['x-rate-limit-remaining']
    js = json.loads(data)
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ) )

    countnew = 0
    countold = 0
    for u in js['users'] :
        friend = u['screen_name']
        print friend
        cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
            (friend, ) )
        try:
            count = cur.fetchone()[0]
            cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
                (count+1, friend) )
            countold = countold + 1
        except:
            cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
                VALUES ( ?, 0, 1 )''', ( friend, ) )
            countnew = countnew + 1
    print 'New accounts=',countnew,' revisited=',countold
    conn.commit()

cur.close()
```

데이터베이스는 `spider.sqlite3` 파일에 저장되어 있다. 테이블 이름은 `Twitter`다. `Twitter` 테이블은 계정 이름에 대한 열, 계정 친구 정보를 가져왔는지 여부를 나타내는 열, 그리고 계정이 얼마나 많이 "친구추가(friended)" 되었는가 나타내는 열로 구성되었다.

프로그램의 메인 루프에서, 사용자가 트위터 계정 이름을 입력하거나 프로그램에서 나가기 위해 "끝내기(quit)"를 입력한다. 사용자가 트위터 계정을 입력하면, 친구 리스트와 상태정보도 가져온다. 만약 데이터베이스에 존재하지 않다면 데이터베이스에 친구로 추가한다. 만약 친구가 이미 리스트에 존재한다면, 데이터베이스 행으로 `friends` 필드에 추가한다.

만약 사용자가 엔터키를 누르면, 아직 가져오지 않은 다음 트위터 계정에 대해서 데이터베이스 정보를 살펴본다. 그 계정의 친구와 상태 정보를 가져오고, 데이터베이스에 추가하거나 갱신하고, `friends count`를 증가한다.

친구 리스트와 상태정보를 가져왔으면, 반환된 JSON 형식 user 항목을 반복돌려 각 사용자의 screen_name을 가져온다. 그리고 나서 SELECT 문을 사용하여 데이터베이스에 screen_name이 저장되었는지, 레코드가 존재하면 친구 숫자(friends)를 확인한다.

```
countnew = 0
countold = 0
for u in js['users'] :
    friend = u['screen_name']
    print friend
    cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
        (friend, ) )
    try:
        count = cur.fetchone()[0]
        cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
            (count+1, friend) )
        countold = countold + 1
    except:
        cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
            VALUES ( ?, 0, 1 )''', ( friend, ) )
        countnew = countnew + 1
print 'New accounts=',countnew,' revisited=',countold
conn.commit()
```

커서가 SELECT문을 수행하면 행을 가져온다. for문으로 동일한 작업을 할 수 있지만, 단지 하나의 행(LIMIT 1)만을 가져오기 때문에, SELECT 처리 결과에서 첫번째만 가져오는 fetchone() 메쏘드를 사용한다. fetchone()은 설사 하나의 필드만 있더라도 행을 튜플(tuple)로 반환하기 때문에, [0]을 사용해서 튜플로부터 첫번째 값을 얻어 count 변수에 현재 친구 숫자를 구한다.

정상적으로 데이터를 가져오면, SQL WHERE절을 가진 UPDATE문을 사용하여 친구의 계정에 매칭되는 행에 대해서 friends 열에 추가한다. SQL에 두개의 플레이스홀더(placeholder, 물음표)가 있고, execute()의 두 매개변수가 물음표 자리에 SQL 안으로 치환될 값을 가진 두-요소 튜플이 된다.

만약 try 블록에서 코드가 작동하지 않는다면, 아마도 SELECT 문의 WHERE name = ? 절에서 매칭되는 레코드가 없기 때문이다. 그래서, except 블록에서, SQL INSERT문을 사용하여 screen_name을 가져온 적이 없고 친구 숫자를 0 으로 설정해서 친구의 screen_name을 테이블에 추가한다.

처음 프로그램을 실행하고 트위터 계정을 입력하면, 프로그램이 다음과 같이 실행된다.

```
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit: quit
```

프로그램을 처음으로 실행하여서, 데이터베이스는 비여 있다. spider.sqlite3 파일에 데이터베이스를 생성하고, Twitter 이름의 테이블을 추가한다. 그리고 나서 친구 몇명을 가져온다. 데이터베이스가 비여있기 때문에 모든 친구를 추가한다.

이 지점에서 spider.sqlite3 파일에 무엇이 있는지를 살펴보기 위해서 간단한 데이터베이스 덤퍼(dumper)를 작성한다.

```
import sqlite3

conn = sqlite3.connect('spider.sqlite3')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur :
   print row
   count = count + 1
print count, 'rows.'
cur.close()
```

상기 프로그램은 데이터베이스를 열고 Twitter 테이블의 모든 행과 열을 선택하고 루프를 모든 행에 대해 돌려 행별로 출력한다.

앞서 작성한 트위터 스파이더를 실행한 후에 이 프로그램을 실행하면, 출력 결과는 다음과 같다.

```
(u'opencontent', 0, 1)
(u'lhawthorn', 0, 1)
(u'steve_coppin', 0, 1)
(u'davidkocher', 0, 1)
(u'hrheingold', 0, 1)
...
20 rows.
```

각 screen_name에 대해 한 행만 있다. screen_name 데이터를 가져오지 않아서 데이터베이스에 있는 모든 사람은 친구가 한명 뿐이다.

이제 데이터베이스가 트위터 계정 (drchuck)에서 친구 정보를 가져온 것을 확인했다. 프로그램을 반복적으로 실행해서, 다음과 같이 트위터 계정을 입력하는 대신에 엔터키를 눌러 "처리되지 않은" 다음 계정 친구정보를 가져오게 한다.

```
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 18  revisited= 2
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

엔터키를 누를 때(즉, 트위터 계정을 명시하지 않았을 때), 다음 코드가 수행된다.

```
    if ( len(acct) < 1 ) :
        cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
        try:
            acct = cur.fetchone()[0]
        except:
            print 'No unretrieved twitter accounts found'
            continue
```

SQL SELECT**문을 사용해서 첫 사용자(**LIMIT 1**) 이름을 가져온다. "사용자를 가져왔는가"의 값은 여전히 0으로 설정되어 있다. try/except 블록 내부에** fetchone()[0] **패턴을 사용하여 가져온 데이터에서** screen_name**을 추출하던가 혹은 오류 메시지를 출력하고 다시 돌아간다.**

처리되지 않은 screen_name**을 성공적으로 가져오면, 다음과 같이 데이터를 가져온다.**

```
url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '20'} )
print 'Retrieving', url
connection = urllib.urlopen(url)
data = connection.read()
js = json.loads(data)

cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ) )
```

데이터를 성공적으로 가져오면 UPDATE**문을 사용하여 이 계정의 친구 가져오기를 완료했는지 표기하기 위해서** retrieved **열에 1 을 표시한다. 이렇게 함으로써 반복적으로 동일한 데이터를 가져오지 않게 하고 트위터 친구 네트워크를 타고 앞으로 나갈 수 있게 한다.**

친구 프로그램을 실행하고 다음 방문하지 않은 친구의 친구 정보를 가져오기 위해서 두번 엔터를 누르고, 결과값을 확인하는 프로그램을 실행하면, 다음 출력값을 얻게 된다.

```
(u'opencontent', 1, 1)
(u'lhawthorn', 1, 1)
(u'steve_coppin', 0, 1)
(u'davidkocher', 0, 1)
(u'hrheingold', 0, 1)
...
(u'cnxorg', 0, 2)
(u'knoop', 0, 1)
(u'kthanos', 0, 2)
(u'LectureTools', 0, 1)
...
55 rows.
```

lhawthorn**과** opencontent**을 방문한 이력이 잘 기록됨을 볼 수 있다.** cnxorg **과** kthanos **계정은 이미 두 명의 팔로워(follower)가 있다. 친구 세명(**drchuck**,** opencontent**,** lhawthorn**)을 가져와서, 테이블은 55 친구 행이 생겼다.**

매번 프로그램을 실행하고 엔터키를 누를 때마다, 다음 방문하지 않은(예, 다음 계정은 steve_coppin**) 계정을 선택해서, 친구 목록을 가져오고, 가져온 것으로 표기하고,** steve_coppin **친구 각각을 데이터베이스 끝에 추가하고 데이터베이스에 이미 추가되어 있으면 친구 숫자를 갱신한다.**

프로그램의 데이터가 모두 데이터베이스 디스크에 저장되어서, 스파이더링을 잠시 보류할 수 있ek. 데이터 손실 없이 원하는만큼 다시 시작할 수 있다.

제 7 절 데이터 모델링 기초

관계형 데이터베이스의 진정한 힘은 다중 테이블과 테이블 사이의 관계를 생성할 때 생긴다. 응용프로그램 데이터를 쪼개서 다중 테이블과 두 테이블 간에 관계를 설정하는 것을 데이터 모델링(data modeling)이라고 한다. 테이블 정보와 테이블 관계를 표현하는 설계 문서를 데이터 모델(data model)이라고 한다.

데이터 모델링(data modeling)은 상대적으로 고급 기술이여서 이번 장에서는 관계형 데이터 모델링의 가장 기본적인 개념만을 소개한다. 데이터 모델링에 대한 좀더 자세한 사항은 다음 링크에서 시작해 볼 수 있다.

http://en.wikipedia.org/wiki/Relational_model

트위터 스파이더 응용프로그램으로 단순히 한 사람의 친구가 몇명인지 세는 대신에, 모든 관계 리스트를 가지고서 특정 계정에 팔로잉하는 모든 사람을 찾는다.

모두 팔로잉하는 계정을 많이 가지고 있어서, 트위터(Twitter) 테이블에 단순히 하나의 열만을 추가해서는 해결할 수 없다. 그래서 친구를 짝으로 추적할 수 있는 새로운 테이블을 생성한다. 다음이 간단하게 상기 테이블을 생성하는 방식이다.

```
CREATE TABLE Pals (from_friend TEXT, to_friend TEXT)
```

drchuck을 팔로잉하는 사람을 마주칠 때마다, 다음과 같은 형식의 행을 삽입한다.

```
INSERT INTO Pals (from_friend,to_friend) VALUES ('drchuck', 'lhawthorn')
```

drchuck 트위터 피드에서 친구 20명을 처리하면서, "drchuck"을 첫 매개변수로 가지는 20개 레코드를 삽입해서 데이터베이스에 중복되는 많은 문자열이 생길 것이다.

문자열 데이터 중복은 데이터베이스 정규화(database normalization) 모범 사례(berst practice)를 위반하게 만든다. 기본적으로 데이터베이스 정규화는 데이터베이스에 결코 한번 이상 동일한 문자열을 저장하지 않는다. 만약 한번 이상 데이터가 필요하다면, 그 특정 데이터에 대한 숫자 키(key)를 생성하고, 그 키를 사용하여 실제 데이터를 참조한다.

실무에서, 문자열이 컴퓨터 주기억장치나 디스크에 저장되는 정수형 자료보다 훨씬 많은 공간을 차지하고 더 많은 처리시간이 비교나 정렬에 소요된다. 항목이 단지 수백개라면, 저장소나 처리 시간이 그다지 문제되지 않는다. 하지만, 데이터베이스에 수백만명의 사람 정보와 1억건 이상의 링크가 있다면, 가능한 빨리 데이터를 스캔하는 것이 정말 중요하다.

앞선 예제에서 사용된 Twitter 테이블 대신에 People로 명명된 테이블에 트위커 계정을 저장한다. People 테이블은 트위터 사용자에 대한 행과 관련된 숫자 키를 저장하는 추가 열(column)이 있다. SQLite는 데이터 열의 특별한 자료형(INTEGER PRIMARY KEY)을 이용하여 테이블에 삽입할 임의 행에 대해서 자동적으로 키값을 추가하는 기능이 있다.

다음과 같이 추가적인 id 열을 가진 People 테이블을 생성할 수 있다.

```
CREATE TABLE People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)
```

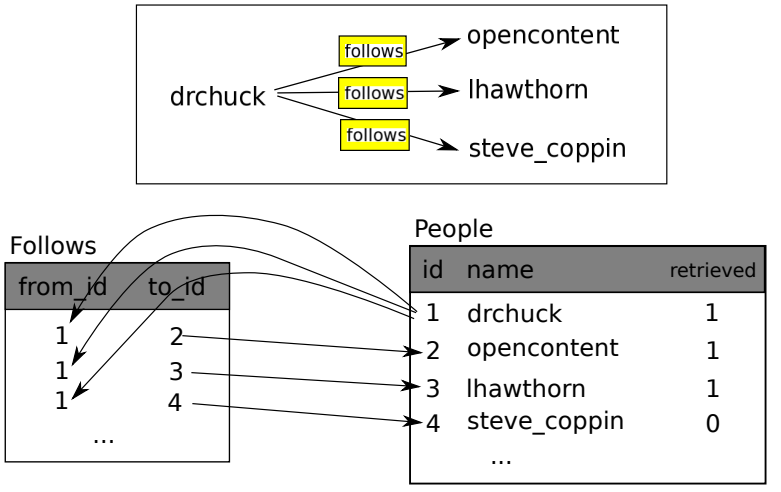
People 테이블의 각 행에서 친구 숫자를 더 이상 유지관리하고 있지 않음을 주목하세요. id 열 자료형으로 INTEGER PRIMARY KEY 선택할 때 함축되는 의미는 다음과 같다., 사용자가 삽입하는 각 행에 대해서 SQLite가 자동으로 유일한 숫자 키를 할당하고 관리하게 한다. UNIQUE 키워드를 추가해서 SQLite에 name에 동일한 값을 가진 두 행을 삽입하지 못하게 한다.

상기 Pals 테이블을 생성하는 대신에, 데이터베이스에 from_id, to_id 두 정수 자료형 열을 지닌 Follows 테이블을 생성한다. Follows 테이블은 from_id과 to_id의 *조합*으로 테이블이 유일하다는 제약사항도 가진다. (즉, 중복된 행을 삽입할 수 없다.)

```
CREATE TABLE Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id) )
```

테이블에 UNIQUE절을 추가한다는 의미는 레코드를 삽입할 때 데이터베이스에서 지켜야하는 규칙 집합을 의사소통하는 것이다. 잠시 후에 보겠지만, 프로그램상에 편리하게 이러한 규칙을 생성한다. 이러한 규칙 집합은 실수를 방지하게 하고 코드를 작성을 간결하게 한다.

본질적으로 Follows 테이블을 생성할 때, "관계(relationship)"를 모델링하여 한 사람이 다른 사람을 "팔로우(follow)"하고 이것을 (a) 사람이 연결되어 있고, (b) 관계을 방향성이 나타나도록 숫자를 짝지어 표현한다.



제 8 절 다중 테이블을 가지고 프로그래밍

테이블 두개, 주키(primary key)와 앞서 설명된 참조 키를 사용하여 트위터 스파이더링 프로그램을 다시 작성한다. 다음은 새로운 버젼 프로그램 코드다.

```
import urllib
import twurl
import json
import sqlite3

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('friends.sqlitesqlite3')
cur = conn.cursor()

cur.execute('''CREATE TABLE IF NOT EXISTS People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')

while True:
    acct = raw_input('Enter a Twitter account, or quit: ')
    if ( acct == 'quit' ) : break
    if ( len(acct) < 1 ) :
        cur.execute('''SELECT id, name FROM People
            WHERE retrieved = 0 LIMIT 1''')
        try:
            (id, acct) = cur.fetchone()
        except:
            print 'No unretrieved Twitter accounts found'
            continue
    else:
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
            (acct, ) )
        try:
            id = cur.fetchone()[0]
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                VALUES ( ?, 0)''', ( acct, ) )
            conn.commit()
            if cur.rowcount != 1 :
                print 'Error inserting account:',acct
                continue
            id = cur.lastrowid

    url = twurl.augment(TWITTER_URL,
       {'screen_name': acct, 'count': '20'} )
    print 'Retrieving account', acct
    connection = urllib.urlopen(url)
    data = connection.read()
    headers = connection.info().dict
    print 'Remaining', headers['x-rate-limit-remaining']

    js = json.loads(data)
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE People SET retrieved=1 WHERE name = ?', (acct, ) )

    countnew = 0
    countold = 0
```

```
    for u in js['users'] :
        friend = u['screen_name']
        print friend
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
            (friend, ) )
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                VALUES ( ?, 0)''', ( friend, ) )
            conn.commit()
            if cur.rowcount != 1 :
                print 'Error inserting account:',friend
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
        cur.execute('''INSERT OR IGNORE INTO Follows (from_id, to_id)
            VALUES (?, ?)''', (id, friend_id) )
    print 'New accounts=',countnew,' revisited=',countold
    conn.commit()

cur.close()
```

프로그램이 다소 복잡해 보인다. 하지만, 테이블을 연결하기 위해서 정수형 키를 사용하는 패턴을 보여준다. 기본적인 패턴은 다음과 같다.

1. **주키(primary key)와 제약 사항을 가진 테이블을 생성한다.**
2. **사람(즉, 계정 이름)에 대한 논리 키가 필요할 때 사람에 대한** `id` **값이 필요하다. 사람 정보가 이미** `People` **테이블에 존재하는지에 따라, (1)** `People` **테이블에 사람을 찾아서 그 사람에 대한** `id` **값을 가져오거나, (2) 사람을** `People` **테이블에 추가하고 신규로 추가된 행의** `id` **값을 가져온다.**
3. **"팔로우(follow)" 관계를 잡아내는 행을 추가한다.**

이들 각각을 순서대로 다룰 것이다.

8.1 데이터베이스 테이블의 제약사항

테이블 구조를 설계할 때, 데이터베이스 시스템에 몇 가지 규칙을 설정할 수 있다. 이러한 규칙은 실수를 방지하고 잘못된 데이터가 테이블에 들어가는 것을 막는다. 테이블을 생성할 때:

```
cur.execute('''CREATE TABLE IF NOT EXISTS People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')
```

`People` **테이블에** `name` **칼럼이 유일**(`UNIQUE`)**함을 나타낸다.** `Follows` **테이블의 각 행에서 두 숫자 조합은 유일하다는 것도 나타낸다. 하나 이상의 동일한 관계를 추가하는 것 같은 실수를 이러한 제약 사항을 통해서 방지한다.**

다음 코드에서 이런 제약사항의 장점을 확인할 수 있다.

```
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
    VALUES ( ?, 0)''', ( friend, ) )
```

INSERT 문에 OR IGNORE 절을 추가해서 만약 특정 INSERT가 "name이 유일(unique)해야 한다"를 위반하게 되면, 데이터베이스 시스템은 INSERT를 무시한다. 데이터베이스 제약 사항을 안전망으로 사용해서 무언가가 우연히 잘못되지 않게 방지한다.

마찬가지로, 다음 코드는 정확히 동일 Follows관계를 두번 추가하지 않는다.

```
cur.execute('''INSERT OR IGNORE INTO Follows
    (from_id, to_id) VALUES (?, ?)''', (id, friend_id) )
```

다시 한번, Follows 행에 대해 지정한 유일한 제약사항을 위반하게 되면 INSERT 시도를 무시하도록 데이터베이스에 지시한다.

8.2 레코드를 가져오거나 삽입하기

사용자가 트위터 계정을 입력할 때, 만약 계정이 존재한다면, id 값을 찾아야 한다. 만약 People 테이블에 계정이 존재하지 않는다면, 레코드를 삽입하고 삽입된 행에서 id 값을 얻어와야 한다.

이것은 매우 일반적인 패턴이고, 상기 프로그램에서 두번 수행되었다. 가져온 트위터 JSON 사용자(user) 노드에서 screen_name을 추출할 때, 친구 계정의 id를 어떻게 찾는지 코드가 보여준다.

시간이 지남에 따라 점점 더 많은 계정이 데이터베이스에 존재할 것 같기 때문에, SELECT문을 사용해서 People 레코드가 존재하는지 먼저 확인한다.

try 구문 내에서 모든 것이 정상적으로 잘 작동하면[2], fetchone()을 사용하여 레코드를 가져와서, 반환된 튜플의 첫번째 요소만 읽어오고 friend_id에 저장한다.

만약 SELECT가 실패하면, fetchone()[0] 코드도 실패하고 제어권은 except 블록으로 이관된다.

```
        friend = u['screen_name']
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
            (friend, ) )
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                VALUES ( ?, 0)''', ( friend, ) )
            conn.commit()
            if cur.rowcount != 1 :
```

[2] 일반적으로, 문장이 "만약 모든 것이 잘 된다면"으로 시작하면, 코드는 필히 try/except를 필요로 한다.

```
            print 'Error inserting account:',friend
            continue
        friend_id = cur.lastrowid
        countnew = countnew + 1
```

except 코드에서 끝나게 되면, 행을 찾지 못해서 행을 삽입해야 한다는 의미가 된다. INSERT OR IGNORE를 사용해서 오류를 피하고 데이터베이스에 진정으로 갱신하기 위해서 commit()을 호출한다. 데이터베이스에 쓰기가 수행된 후에, 얼마나 많은 행이 영향을 받았는지 확인하기 위해서 cur.rowcount로 확인한다. 단일 행을 삽입하려고 했는데, 영향을 받은 행의 숫자가 1과 다르다면, 그것은 오류다.

삽입(INSERT)이 성공하면, cur.lastrowid를 살펴보고 신규로 생성된 행의 id 칼럼에 무슨 값이 대입되었는지 알 수 있다.

8.3 친구관계 저장하기

트위터 사용자와 JSON 친구에 대한 키값을 알게되면, 다음 코드로 두 개의 숫자를 Follows 테이블에 삽입하는 것은 간단하다.

```
cur.execute('INSERT OR IGNORE INTO Follows (from_id, to_id) VALUES (?, ?)',
    (id, friend_id) )
```

데이터베이스를 생성할 때 유일(unique)한 제약조건과 INSERT문에 OR IGNORE을 추가함으로써 데이터베이스 스스로 관계를 두번 삽입하는 것을 방지하도록 한 것에 주목한다.

다음에 프로그램의 샘플 실행 결과가 있다.

```
Enter a Twitter account, or quit:
No unretrieved Twitter accounts found
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

drchuck 계정으로 시작해서, 프로그램이 자동적으로 다음 두개의 계정을 선택해서 데이터베이스에 추가한다.

다음은 프로그램 수행을 완료한 후에 People과 Follows 테이블에 첫 몇개의 행이다.

```
People:
(1, u'drchuck', 1)
(2, u'opencontent', 1)
(3, u'lhawthorn', 1)
```

```
(4, u'steve_coppin', 0)
(5, u'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
```

People 테이블의 id, name, visited 필드와 Follows 테이블 양 끝에 관계 숫자를 볼 수 있다. People 테이블에서, 사람 첫 세명을 방문해서, 데이터를 가져온 것을 볼 수 있다. Follows 테이블의 데이터는 drchuck(사용자 1)이 첫 다섯개 행에 보여진 모든 사람에 대해 친구임을 나타낸다. 이것은 당연한데 왜냐하면 처음 가져와서 저장한 데이터가 drchuck의 트위터 친구들이기 때문이다. Follows 테이블에서 좀더 많은 행을 출력하면, 사용자 2, 3 혹은 그 이상의 친구를 볼 수 있다.

제 9 절 세 종류의 키

지금까지 데이터를 다중 연결된 테이블에 넣고 키(keys)를 사용하여 행을 연결하는 방식으로 데이터 모델을 생성했는데, 키와 관련된 몇몇 용어를 살펴볼 필요가 있다. 일반적으로 데이터베이스 모델에서 세가지 종류의 키가 사용된다.

- 논리 키(logical key)는 "실제 세상"이 행을 찾기 위해서 사용하는 키다. 데이터 모델 예제에서, name 필드는 논리키다. 사용자에 대해서 screen_name이고, name 필드를 사용하여 프로그램에서 여러번 사용자 행을 찾을 수 있다. 논리 키에 UNIQUE 제약 사항을 추가하는 것이 의미있다는 것을 종종 이해하게 된다. 논리 키는 어떻게 바깥 세상에서 행을 찾는지 다루기 때문에, 테이블에 동일한 값을 가진 다중 행이 존재한다는 것은 의미가 없다.

- 주키(primary key)는 통상적으로 데이터베이스에서 자동 대입되는 숫자다. 프로그램 밖에서는 일반적으로 의미가 없고, 단지 서로 다른 테이블에서 행을 열결할 때만 사용된다. 테이블에 행을 찾을 때, 통상적으로 주키를 사용해서 행을 찾는 것이 가장 빠르게 행을 찾는 방법이다. 주키는 정수형이어서, 매우 적은 저장공간을 차지하고 매우 빨리 비교 혹은 정렬할 수 있다. 이번에 사용된 데이터 모델에서 id 필드가 주키의 한 예가 된다.

- 외부 키(foreign key)는 일반적으로 다른 테이블에 연관된 행의 주키를 가리키는 숫자다. 이번에 사용된 데이터 모델의 외부 키의 사례는 from_id다.

주키 id필드명을 호출하고, 항상 외부키에 임의 필드명에 접미사로 _id 붙이는 명명규칙을 사용한다.

제 10 절 JOIN을 사용하여 데이터 가져오기

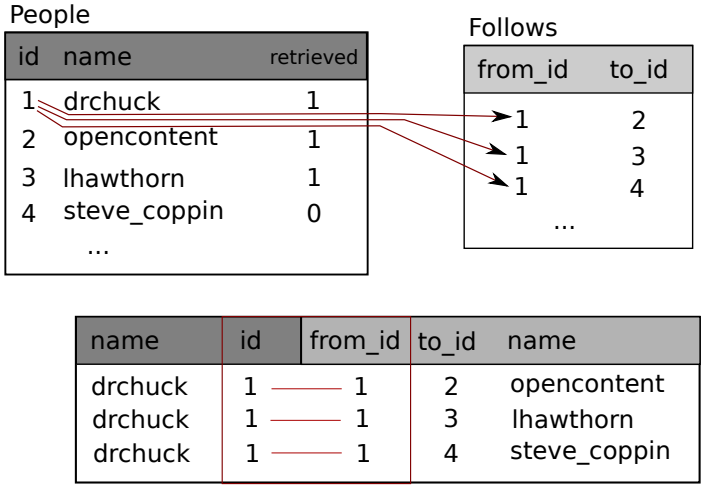
데이터 정규화 규칙을 따라서, 데이터를 주키와 외부키로 연결된 두개의 테이블로 나누어서, 테이블 데이터를 다시 합치기 위해서 SELECT를 작성할 필요가 있다.

SQL은 JOIN절을 사용해서 테이블을 다시 연결한다. JOIN절에서 테이블 사이의 행을 다시 연결할 필드를 지정한다.

다음은 JOIN절을 가진 SELECT 예제이다.

```
SELECT * FROM Follows JOIN People
    ON Follows.from_id = People.id WHERE People.id = 1
```

JOIN절은 Follows와 People 테이블에서 선택하는 필드를 나타낸다. ON절은 어떻게 두 테이블이 합쳐지는지를 나타낸다. Follows에서 행을 선택하고 People에서 행을 추가하는데, Follows 테이블의 from_id와 People 테이블의 id 값은 동일하다.



JOIN 결과는 People 테이블 필드와 Follows 테이블에서 매칭되는 필드를 가진 "메타-행(meta-row)"을 생성한다. People 테이블 id 필드와 Follows 테이블 from_id 사이에 하나 이상의 매칭이 존재한다면, JOIN은 필요하면 데이터를 중복하면서, 행에 매칭되는 짝 각각에 대해 메타-행을 생성한다.

다중 테이블 트위터 프로그램을 수차례 수행한 후에 데이터베이스에 있는 데이터를 가지고 다음 코드가 시연한다.

```
import sqlite3

conn = sqlite3.connect('spider.sqlite3')
cur = conn.cursor()

cur.execute('SELECT * FROM People')
count = 0
print 'People:'
for row in cur :
```

```
   if count < 5: print row
   count = count + 1
print count, 'rows.'

cur.execute('SELECT * FROM Follows')
count = 0
print 'Follows:'
for row in cur :
   if count < 5: print row
   count = count + 1
print count, 'rows.'

cur.execute('''SELECT * FROM Follows JOIN People
    ON Follows.from_id = People.id WHERE People.id = 2''')
count = 0
print 'Connections for id=2:'
for row in cur :
   if count < 5: print row
   count = count + 1
print count, 'rows.'

cur.close()
```

이 프로그램에서 People**과** Follows **테이블을 먼저 보여주고, 함께 JOIN된 데이터 일부분을 보여준다.**

다음이 프로그램 출력이다.

```
python twjoin.py
People:
(1, u'drchuck', 1)
(2, u'opencontent', 1)
(3, u'lhawthorn', 1)
(4, u'steve_coppin', 0)
(5, u'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
Connections for id=2:
(2, 1, 1, u'drchuck', 1)
(2, 28, 28, u'cnxorg', 0)
(2, 30, 30, u'kthanos', 0)
(2, 102, 102, u'SomethingGirl', 0)
(2, 103, 103, u'ja_Pac', 0)
20 rows.
```

People**과** Follows **테이블에서 칼럼을 볼 수 있고, 마지막 행의 집합은** JOIN **절을 가진** SELECT**문의 결과다.**

마지막 SELECT**문에서, "opencontent" (즉** People.id=2**)를 친구로 가진 계정을 찾는다.**

마지막 SELECT "메타-행"의 각각에서 첫 두 열은 Follows 테이블에서, 3번째부터 5번째 열은 People 테이블에서 가져왔다. 두번째 열(Follows.to_id)과 세번째 열(People.id)은 join된 "메타-열"에서 매칭됨을 볼 수 있다.

제 11 절 요약

이번 장은 파이썬에서 데이터베이스 사용 기본적인 개요에 대해 폭넓게 다루었다. 데이터를 저장하기 위해서 파이썬 딕셔너리나 일반적인 파일보다 데이터베이스를 사용하여 코드를 작성하는 것이 훨씬 복잡하다. 그래서, 만약 작성하는 응용프로그램이 실질적으로 데이터베이스 역량을 필요하지 않는다면 굳이 데이터베이스를 사용할 이유는 없다. 데이터베이스가 특히 유용한 상황은 (1) 큰 데이터셋에서 작은 임의적인 갱신이 많이 필요한 응용프로그램을 작성할 때 (2) 데이터가 너무 커서 딕셔너리에 담을 수 없고 반복적으로 정보를 검색할 때, (3) 한번 실행에서 다음 실행 때까지 데이터를 보관하고, 멈추고, 재시작하는데 매우 긴 실행 프로세스를 갖는 경우다.

많은 응용프로그램 요구사항을 충족시키기 위해서 단일 테이블로 간단한 데이터베이스를 구축할 수 있다. 하지만, 대부분의 문제는 몇개의 테이블과 서로 다른 테이블간에 행이 연결된 관계를 요구한다. 테이블 사이 연결을 만들 때, 좀더 사려깊은 설계와 데이터베이스의 역량을 가장 잘 사용할 수 있는 데이터베이스 정규화 규칙을 따르는 것이 중요하다. 데이터베이스를 사용하는 주요 동기는 처리할 데이터의 양이 많기 때문에, 데이터를 효과적으로 모델링해서 프로그램이 가능하면 빠르게 실행되게 만드는 것이 중요하다.

제 12 절 디버깅

SQLite 데이터베이스에 연결하는 파이썬 프로그램을 개발할 때 하나의 일반적인 패턴은 파이썬 프로그램을 실행하고 SQLite 데이터베이스 브라우저를 통해서 결과를 확인하는 것이다. 브라우저를 통해서 빠르게 프로그램이 정상적으로 작동하는지를 확인할 수 있다.

SQLite에서 두 프로그램이 동시에 동일한 데이터를 변경하지 못하기 때문에 주의가 필요하다. 예를 들어, 브라우저에서 데이터베이스를 열고 데이터베이스에 변경을 하고 "저장(save)"버튼을 누르지 않는다면, 브라우져는 데이터베이스 파일에 "락(lock)"을 걸구, 다른 프로그램이 파일에 접근하는 것을 막는다. 특히, 파일이 잠겨져 있으면 작성하고 있는 파이썬 프로그램이 파일에 접근할 수 없다.

해결책은 데이터베이스가 잠겨져 있어서 파이썬 코드가 작동하지 않는 문제를 피하도록 파이썬에서 데이터베이스에 접근하려 시도하기 전에 데이터베이스 브라우져를 닫거나 혹은 File 메뉴를 사용해서 브라우져 데이터베이스를 닫는 것이다.

제 13 절 용어정의

속성(attribute): 튜플 내부에 값의 하나. 좀더 일반적으로 "열", "칼럼", "필드"로 불린다.

제약(constraint): 데이터베이스가 테이블의 필드나 행에 규칙을 강제하는 것. 일반적인 제약은 특정 필드에 중복된 값이 없도록 하는 것(즉, 모든 값이 유일해야 한다.)

커서(cursor): 커서를 사용해서 데이터베이스에서 SQL 명령어를 수행하고 데이터베이스에서 데이터를 가져온다. 커서는 네트워크 연결을 위한 소켓이나 파일의 파일 핸들러와 유사하다.

데이터베이스 브라우져(database browser): 프로그램을 작성하지 않고 직접적으로 데이터베이스에 연결하거나 데이터베이스를 조작할 수 있는 소프트웨어.

외부 키(foreign key): 다른 테이블에 있는 행의 주키를 가리키는 숫자 키. 외부키는 다른 테이블에 저장된 행사이에 관계를 설정한다.

인텍스(index): 테이블에 행이 추가될 때 정보 검색하는 것을 빠르게 하기 위해서 데이터베이스 소프트웨어가 유지관리하는 추가 데이터.

논리 키(logical key): "외부 세계"가 특정 행의 정보를 찾기 위해서 사용하는 키. 사용자 계정 테이블의 예로, 사람의 전자우편 주소는 사용자 데이터에 대한 논리 키의 좋은 후보자가 될 수 있다.

정규화(normalization): 어떠한 데이터도 중복이 없도록 데이터 모델을 설계하는 것. 데이터베이스 한 장소에 데이터 각 항목 정보를 저장하고 외부키를 이용하여 다른 곳에서 참조한다.

주키(primary key): 다른 테이블에서 테이블의 한 행을 참조하기 위해서 각 행에 대입되는 숫자 키. 종종 데이터베이스는 행이 삽입될 때 주키를 자동 삽입하도록 설정되었다.

관계(relation): 튜플과 속성을 담고 있는 데이터베이스 내부 영역. 좀더 일반적으로 "테이블(table)"이라고 한다.

튜플(tuple): 데이터베이스 테이블에 단일 항목으로 속성 집합이다. 좀더 일반적으로 "행(row)"이라고 한다.

15 장

데이터 시각화

지금까지 파이썬 언어 자체를 학습했고, 데이터를 다루기 위해서 파이썬, 네트워크, 그리고 데이터베이스를 어떻게 활용하는가도 학습했다.

이번장에서는 학습한 모두를 모아 완전한 응용프로그램 세개를 작성하여 데이터 관리와 시각화 할 것이다. 실제 문제를 해결하는데 시작할 수 있는 샘플 코드로도 응용프로그램을 사용할 수도 있다.

각 응용프로그램은 ZIP 파일로 압축되어서 다운로드 받아 로컬 컴퓨터에 압축을 풀고 실행한다.

제 1 절　지리정보 데이터로 구글맵 생성하기

이번 프로젝트에서 구글 지오코딩(geocoding) API를 사용해서 사용자가 입력한 대학교 이름 지리 정보를 정리하고 구글 지도에 데이터를 표시한다.

시작하기 위해서 다음 url에서 응용프로그램을 다운로드 한다.

`www.py4inf.com/code/geodata.zip`

해결할 첫번째 문제는 무료 구글 지오코딩 API가 일일 요청횟수에 제한이 있다는 것이다. 그래서, 만약 데이터가 많다면, 여러번 중지, 재시작하는 검색 프로세스가 요구된다. 그래서, 문제를 두 단계로 나누었다.

첫번째 단계에서, where.data 파일에 "설문(survey)" 데이터를 받는다. 한번에 한줄씩 읽고 구글에서 지리정보를 자져와 geodata.sqlite 데이터베이스에 저장한다. 각 사용자가 입력한 위치에 대한 지오코딩 API를 사용하기 전에, 특정 입력 라인에 데이터가 있는지를 확인하기 위해서 간단히 점검한다. 데이터베이스는 지오 코딩 데이터의 로컬 "캐쉬(cache)"처럼 동작해서 두번 동일 데이터에 대해서는 구글에 요청하지 않도록 한다.

geodata.sqlite 파일을 제거함으로써 언제라도 프로세스를 재시작할 수 있다.

geoload.py 프로그램을 실행한다. geoload.py 프로그램은 where.data에 입력 라인을 읽고, 각 라인별로 데이터베이스 존재유무를 확인한다. 만약 위치에 대한 데이터가 없다면, 지오코딩 API를 호출해서 데이터를 가져오고 데이터베이스에 저장한다.

약간의 데이터가 데이터베이스에 이미 존재한 것을 확인한 후에 샘플로 실행한 결과가 다음에 있다.

```
Found in database  Northeastern University
Found in database  University of Hong Kong, ...
Found in database  Technion
Found in database  Viswakarma Institute, Pune, India
Found in database  UMD
Found in database  Tufts University

Resolving Monash University
Retrieving http://maps.googleapis.com/maps/api/
    geocode/json?sensor=false&address=Monash+University
Retrieved 2063 characters {    "results" : [
{u'status': u'OK', u'results': ... }

Resolving Kokshetau Institute of Economics and Management
Retrieving http://maps.googleapis.com/maps/api/
    geocode/json?sensor=false&address=Kokshetau+Inst ...
Retrieved 1749 characters {    "results" : [
{u'status': u'OK', u'results': ... }
...
```

첫 다섯 지점은 이미 데이터베이스에 있으므로 건너뛴다. 프로그램은 가져오지 못한 위치 정보를 찾아 스캔하고 정보를 가져온다.

geoload.py는 언제라도 멈춰 정지시킬 수 있다. 실행할 때마다 지오코딩 API 호출 횟수를 제한하기 위한 카운터가 있다. where.data에 수백개의 데이터 항목만 있다면, 하루 사용량 한계를 넘지 않을 것을 것이다. 하지만, 만약 더 많은 데이터가 있다면, 모든 입력 데이터에 대한 모든 지리정보 데이터로 구성된 데이터베이스를 만들기 위해서 몇일에 걸쳐서 여러번 실행할지도 모른다.

데이터를 geodata.sqlite에 적재하면, geodump.py 프로그램을 사용하여 데이터를 시각화할 수 있다. 프로그램이 데이터베이스를 읽고 위치, 위도, 경도를 실행가능한 자바스크립 코드로 where.js에 쓴다.

geodump.py 프로그램 실행결과는 다음과 같다.

```
Northeastern University, ... Boston, MA 02115, USA 42.3396998 -71.08975
Bradley University, 1501 ... Peoria, IL 61625, USA 40.6963857 -89.6160811
...
Technion, Viazman 87, Kesalsaba, 32000, Israel 32.7775 35.0216667
Monash University Clayton ... VIC 3800, Australia -37.9152113 145.134682
Kokshetau, Kazakhstan 53.2833333 69.3833333
...
12 records written to where.js
Open where.html to view the data in a browser
```

where.html 파일은 HTML과 자바스크립트로 구성되어서 구글 맵을 시각화한다. where.js에 가장 최신 데이터를 읽어서 시각화할 데이터로 사용한다. where.js 파일 형식은 다음과 같다.

```
myData = [
[42.3396998,-71.08975, 'Northeastern Uni ... Boston, MA 02115'],
[40.6963857,-89.6160811, 'Bradley University, ... Peoria, IL 61625, USA'],
[32.7775,35.0216667, 'Technion, Viazman 87, Kesalsaba, 32000, Israel'],
   ...
];
```

리스트의 리스트를 담고 있는 자바스크립트다. 자바스크립트 리스트 상수에 대한 구문은 파이썬과 매우 유사하여, 구문이 여러분에게 매우 친숙해야 한다.

위치를 보기 위해서 브라워져에서 where.html을 연다. 각 맵 핀을 여기저기 돌아다니면서 지오코딩 API가 사용자가 입력한 것에 대해서 반환한 위치를 찾는다. where.html 파일을 열었을 때 어떤 데이터도 볼 수 없다면, 자바스크립트나 브라우저 개발자 콘솔을 확인한다.

제 2 절 네트워크와 상호 연결 시각화

이번 응용프로그램에서 검색엔진 기능 일부를 수행한다. 먼저 웹의 일부분을 스파이더링하고 어느 페이지가 가장 많이 연결되었는지 결정하기 위해서 간략한 구글 페이지 랭크 알고리즘을 실행한다. 스파이더링한 작은 웹 조각에 대해서 연결성과 페이지 랭크를 시각화한다. 시각화 산출물을 만들기 위해서 D3 자바스크립트 시각화 라이브러리 `http://d3js.org/`를 사용한다.

응용프로그램을 다음 url에서 다운로드 받아 압축을 푼다.

`www.py4inf.com/code/pagerank.zip`

첫 프로그램(spider.py)은 웹사이트를 크롤(craw)하고 일련의 페이지를 뽑아내서 데이터베이스(spider.sqlite)에 넣어 페이지 간에 링크를 기록한다. 언제라도 spider.sqlite 파일을 제거하고 spider.py를 재실행함으로써 프로세스를 다시 시작할 수 있다.

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:2
1 http://www.dr-chuck.com/ 12
2 http://www.dr-chuck.com/csev-blog/ 57
How many pages:
```

샘플 실행결과, 웹사이트를 크롤해서, 웹페이지 두개를 가져왔다. 프로그램을 재실행해서 좀더 많은 웹페이지를 크롤하게 한다면, 데이터베이스에 이미 등록된 웹페이지는 다시 크롤하지는 않는다. 재시작할 때 무작위로 크롤하지 않은 페이지로 이동해서 그곳에서 크롤링을 시작한다. 그래서 연속적으로 spider.py을 실행하는 것은 추가적이된다.

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:3
3 http://www.dr-chuck.com/csev-blog 57
4 http://www.dr-chuck.com/dr-chuck/resume/speaking.htm 1
5 http://www.dr-chuck.com/dr-chuck/resume/index.htm 13
How many pages:
```

같은 데이터베이스에서 다중 시작 지점을 가질 수 있다. 프로그램 내부에서 이를 "webs"라고 부른다. 스파이더는 모든 웹사이트에서 방문하지 않는 링크 중에 무작위로 선택하여 방문할 다음 웹 페이지를 선정한다.

spider.sqlite 파일에 들어있는 내용을 살펴보고자 한다면, 다음과 같이 spdump.py을 실행한다.

```
(5, None, 1.0, 3, u'http://www.dr-chuck.com/csev-blog')
(3, None, 1.0, 4, u'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, None, 1.0, 2, u'http://www.dr-chuck.com/csev-blog/')
(1, None, 1.0, 5, u'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

인입 링크 수, 이전 페이지 랭크, 신규 페이지 랭크, 페이지 id, 페이지 url을 보여준다. spdump.py 프로그램은 최소한 하나의 인입 링크가 있는 페이지만을 보여준다.

데이터베이스에 몇개의 페이지가 있다면, sprank.py 프로그램을 실행하여 페이지에 대한 페이지 랭크를 실행한다. 단순하게 얼마나 많이 페이지 랭크 반복을 수행할지 지정한다.

```
How many iterations:2
1 0.546848992536
2 0.226714939664
[(1, 0.559), (2, 0.659), (3, 0.985), (4, 2.135), (5, 0.659)]
```

데이터베이스를 다시 열어서 페이지 랭크가 갱신되었는지 확인한다.

```
(5, 1.0, 0.985, 3, u'http://www.dr-chuck.com/csev-blog')
(3, 1.0, 2.135, 4, u'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, 1.0, 0.659, 2, u'http://www.dr-chuck.com/csev-blog/')
(1, 1.0, 0.659, 5, u'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

sprank.py를 원하는 만큼 실행한다. 실행할 때마다 페이지 랭크를 정교화한다. sprank.py을 몇번 실행하고 나서 spider.py으로 좀더 많은 페이지를 스파이더링한다. 그리고 나서, sprank.py을 실행하여 페이지 랭크 값을 다시 수렴하여 갱신한다. 검색엔진은 일반적으로 동시에 크롤링과 랭킹 프로그램을 실행한다.

웹페이지를 다시 스파이더링하지 않고, 페이지 랭크 계산을 재시작하고자 한다면, spreset.py을 사용하고 나서 sprank.py를 다시 시작한다.

```
How many iterations:50
1 0.546848992536
2 0.226714939664
3 0.0659516187242
4 0.0244199333
5 0.0102096489546
6 0.00610244329379
...
42 0.000109076928206
43 9.91987599002e-05
44 9.02151706798e-05
45 8.20451504471e-05
46 7.46150183837e-05
47 6.7857770908e-05
48 6.17124694224e-05
49 5.61236959327e-05
50 5.10410499467e-05
[(512, 0.0296), (1, 12.79), (2, 28.93), (3, 6.808), (4, 13.46)]
```

페이지 랭크 알고리즘을 매번 반복하면, 페이지 마다 페이지 랭크 평균 변화를 출력한다. 초기에 네트워크는 매우 불균형 상태여서 각각의 페이지 랭크 값은 반복할 때마다 요동친다. 하지만, 짧게 몇번 반복한 다음에 페이지 랭크는 수렴한다. prank.py을 충분히 오랜동안 실행해서 페이지 랭크 값을 수렴하게 만든다.

페이지 랭크에 대해서 현재 최상위 페이지를 시각화하고자 한다면, spjson.py을 실행하여 데이터베이스를 읽고 웹브라우져에서 볼 수 있는 JSON 형식으로 가장 많이 연결된 페이지에 대해서 데이터를 저장한다.

```
Creating JSON output on spider.json...
How many nodes? 30
Open force.html in a browser to view the visualization
```

웹브라우져에 force.html 파일을 열어 작업 데이터 결과를 볼 수 있다. 노드와 링크의 자동 레이아웃을 보여준다. 임의 노드를 클릭하고 끌 수도 있고, 노드를 더블 클릭해서 노드로 표현된 URL을 확인할 수 있다.

만약 다른 유틸리티를 재실행하고자 하면, spjson.py을 다시 실행하고 브라우져의 새로 고치기를 눌러서 spider.json에서 새로운 데이터를 가져온다.

제 3 절 전자우편 데이터 시각화

책을 이 지점까지 읽어오면서 mbox-short.txt과 mbox.txt 파일에 매우 친숙해졌을 것이다. 이제는 전자우편 데이터 분석 수준을 다음 단계로 옮겨갈 때다.

실무에서 종종 서버에서 전자우편 데이터를 가져오는 것은 시간이 많이 걸리고, 가져온 데이터는 오류가 많고, 일관되지 못하고, 많은 보정과 정비가 필요하다. 이번 장에서, 지금까지 작성한 가장 복잡한 응용프로그램을 가지고 거의 1GB 데이터를 추출하여 시각화 작업을 한다.

응용프로그램을 다음 url에서 다운로드한다.

`www.py4inf.com/code/gmane.zip`

`www.gmane.org`의 무료 전자우편 보관 서비스 데이터를 사용한다. 전자우편 활동에 대한 멋진 보관 서비스를 검색기능과 함께 제공하기 때문에, 오픈 소스 프로젝트에서는 매우 인기가 높다. API를 통한 데이터 접근에 대해서도 매우

자유로운 정책을 유지한다. 사용량에 대한 제한은 없지만, 자율적으로 서비스에 부하를 너무 많이 주지 말고 필요한 데이터만 가져가도록 요청한다. gmane의 사용조건에 대해서는 다음 페이지를 참조한다.

`http://gmane.org/export.php`

서비스 접근에 지연을 추가하거나 오랜 기간에 걸쳐서 장시간이 소요되는 작업을 분산함으로써 gmane.org 데이터를 책임감있게 사용하는 것은 매우 중요하다. 다른 사용자를 위해서 무료 서비스를 오용하지 말고, 서비스를 망치지 말아주세요.

이 프로그램을 사용하여 Sakai 전자우편 주소 데이터를 스파이더링하면, 거의 1 GB의 데이터를 생성하고 몇일에 걸쳐서 수차례 실행을 해야한다. ZIP 압축파일 내부에 README.txt이 어떻게 대부분의 Sakai 전자우편 코퍼스(corpus)를 다운로드할 수 있는지에 대한 안내정보를 담고 있다. 그래서 단지 프로그램을 실행하기 위해서 5일 동안 스파이더링을 할 필요는 없다. 이미 스파이더링된 콘텐츠를 다운로드했다면, 좀더 최근의 메시지를 가져오기 위해서 스파이더링 프로세스를 다시 실행해야 한다.

첫 단계는 gmane 저장소를 스파이더링하는 것이다. 기본 URL은 gmane.py 파일과 Sakai 개발자 리스트에 하드코딩 되어 있다. 기본 url을 변경함으로써 또 다른 저장소를 스파이더링 할 수 있다. 기본 url을 변경하는 경우 content.sqlite 파일을 필히 삭제하세요.

gmane.py 파일은 책임감있는 캐쉬 스파이더로서 작동한다. 천천히 실행되며 1초에 한개의 전자우편 메시지를 가져와서 gmane에 의해서 작동 못하게 되는 것을 피한다. 데이터베이스 모든 데이터를 저장한다. 필요하면 자주 중단하고 재시작한다. 모든 데이터를 가져오는데 몇 시간이 걸릴 수 있다. 그래서 여러번 재시작하는 것이 필요하다.

Sakai 개발자 리스트에 있는 마지막 5개 메시지를 가져오는 프로그램 gmane.py을 실행한 결과다.

```
How many messages:10
http://download.gmane.org/gmane.comp.cms.sakai.devel/51410/51411 9460
    nealcaidin@sakaifoundation.org 2013-04-05 re: [building ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51411/51412 3379
    samuelgutierrezjimenez@gmail.com 2013-04-06 re: [building ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51412/51413 9903
    da1@vt.edu 2013-04-05 [building sakai] melete 2.9 oracle ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51413/51414 349265
    m.shedid@elraed-it.com 2013-04-07 [building sakai] ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51414/51415 3481
    samuelgutierrezjimenez@gmail.com 2013-04-07 re: ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51415/51416 0

Does not start with From
```

프로그램은 1번 부터 이미 스파이더링되지 않는 첫 메시지까지 content.sqlite을 스캔하고 그 메시지에서 스파이더링을 시작한다. 원하는 메시지 숫자를 스파이더링할 때까지 계속하거나 잘못된 형식의 메시지가 나오는 페이지에 도달할 때까지 스파이더링을 계속한다.

때때로 gmane.org도 메시지를 잃어버린다. 아마도 관리자가 메시지를 삭제하거나 아마도 분실했을 것이다. 만약 스파이더가 멈추고 잃어버린 메시지를 만나게 되면 SQLite 매니저로 가서 다른 모든 행을 공백으로 구성하고 잃어버린 id를 가진 행을 추가하여 gmane.py을 재시작한다. 이렇게 함으로써 스파이더링 프로세스를 재시작하고 계속 진행할 수 있다. 빈 메시지는 다음번 프로세스에서는 무시된다.

모든 메시지를 스파이더링하고 content.sqlite 파일에 넣게 되면, 리스트에 메시지가 보내질 때 gmane.py을 다시 시작해서 새로운 메시지를 얻게 된다.

content.sqlite 데이터는 원데이터 형식이고 비효율적인 데이터 모델을 가지고 있으며 압축되어 있지 않다. 의도적으로 그렇게 함으로써 SQLite 매니저가 content.sqlite 파일을 직접 살펴보고, 스파이더링 프로세스에서 문제를 디버그할 수 있도록 한다. 매우 느린 상태에서 데이터베이스에 질의를 실행하는 것은 좋은 생각은 아니다.

두번째 프로세스는 gmodel.py 프로그램을 실행하는 것이다. content.sqlite로부터 가다듬지 않은 원데이터를 읽어서 index.sqlite 파일에 깨끗하게 잘 모델링된 형식 데이터로 저장한다. index.sqlite 파일은 content.sqlite 보다 10배이상 크기가 적다. 왜냐하면 헤더와 본문 텍스트를 압축하기 때문이다.

gmodel.py을 실행할 때마다, index.sqlite를 삭제하고 다시 생성한다. 그래서 함으로써 데이터 정비 프로세스를 약간 바꾸는데 매개변수를 조정하고 content.sqlite의 매핑 테이블을 편집할 수 있게 한다. 다음은 gmodel.py을 샘플로 실행시킨 것이다. 매번 250개 전자우편 메시지가 처리될 때마다 한 라인을 출력해서 거의 1GB 전자우편 데이터를 처리하는 동안 진행사항을 지켜볼 수 있다.

```
Loaded allsenders 1588 and mapping 28 dns mapping 1
1 2005-12-08T23:34:30-06:00 ggolden22@mac.com
251 2005-12-22T10:03:20-08:00 tpamsler@ucdavis.edu
501 2006-01-12T11:17:34-05:00 lance@indiana.edu
751 2006-01-24T11:13:28-08:00 vrajgopalan@ucmerced.edu
...
```

gmodel.py 프로그램은 많은 데이터 정비 작업을 처리한다.

도메인 이름이 .com, .org, .edu, .net에 대해서는 2 단계로 다른 도메인 이름은 3 단계로 잘랐다. 그래서 si.umich.edu은 umich.edu이 되고, caret.cam.ac.uk은 cam.ac.uk이 된다. 또한 전자우편 주소는 소문자가 되게 하고, 다음과 같은 @gmane.org 주소는 메시지 코퍼스 어딘가 있는 실제 전자우편 주소와 매칭하여 실제 주소로 변환한다.

```
arwhyte-63aXycvo3TyHXe+LvDLADg@public.gmane.org
```

content.sqlite 데이터베이스를 살펴보면, 도메인 이름과 전자우편 주소가 시간에 따라 변경되는 것을 보정하려고 개인 전자우편 주소를 매핑하는 테이블이 두개 있다. 예를 들어, Sakai 개발자 리스트 개발기간에 걸쳐 직업을 바꿈에 따라 Steve Githens는 다음 전자우편 주소를 사용했다.

```
s-githens@northwestern.edu
sgithens@cam.ac.uk
swgithen@mtu.edu
```

content.sqlite의 매핑(Mapping) 테이블에 항목을 두개를 추가해서 gmodel.py 프로그램이 3개 주소를 하나로 매핑한다.

```
s-githens@northwestern.edu ->  swgithen@mtu.edu
sgithens@cam.ac.uk -> swgithen@mtu.edu
```

다중 DNS 이름을 하나의 DNS 이름으로 매핑하려면 DNSMapping 테이블에 비슷한 항목을 생성할 수 있다. 다음 매핑이 Sakai 데이터에 추가된다.

```
iupui.edu -> indiana.edu
```

그래서, 다양한 인디애나 대학교 캠퍼스의 모든 계정이 함께 추적된다.

데이터를 볼 때마다, gmodel.py를 반복적으로 실행하고 데이터를 좀더 깨끗하게 만들기 위해서 매핑을 추가한다. 작업이 마쳐지면, index.sqlite에 인덱스 잘된 버젼의 전자우편 데이터가 있다. 자료 분석을 위해 이 파일을 사용한다. 이 파일로 데이터분석은 정말 빠르게 수행된다.

첫번째, 가장 간단한 자료 분석은 "누가 가장 많은 전자우편을 보냈는가?"와 "어느 조직에서 가장 많은 전자우편을 보냈는가?"를 알아보는 것이다. gbasic.py을 사용해서 수행할 수 있다.

```
How many to dump? 5
Loaded messages= 51330 subjects= 25033 senders= 1584

Top 5 Email list participants
steve.swinsburg@gmail.com 2657
azeckoski@unicon.net 1742
ieb@tfd.co.uk 1591
csev@umich.edu 1304
david.horwitz@uct.ac.za 1184

Top 5 Email list organizations
gmail.com 7339
umich.edu 6243
uct.ac.za 2451
indiana.edu 2258
unicon.net 2055
```

gmane.py이나 혹은 gmodel.py과 비교하여 gbasic.py이 얼마나 빨리 실행되는지 주목하세요. 모두 동일한 데이터에서 작업을 수행하지만, gbasic.py은 index.sqlite에 저장된 압축되고 정규화된 데이터를 사용한다. 처리할 데이터가 많다면, 응용프로그램과 같이 다중-단계 프로세스를 개발하는데 시간이 약간 더 걸리지만, 데이터를 탐색하고 시각화할 때는 많은 시간을 절약해 준다.

gword.py 파일에 주제 라인의 단어 빈도를 간략히 시각화할 수 있다.

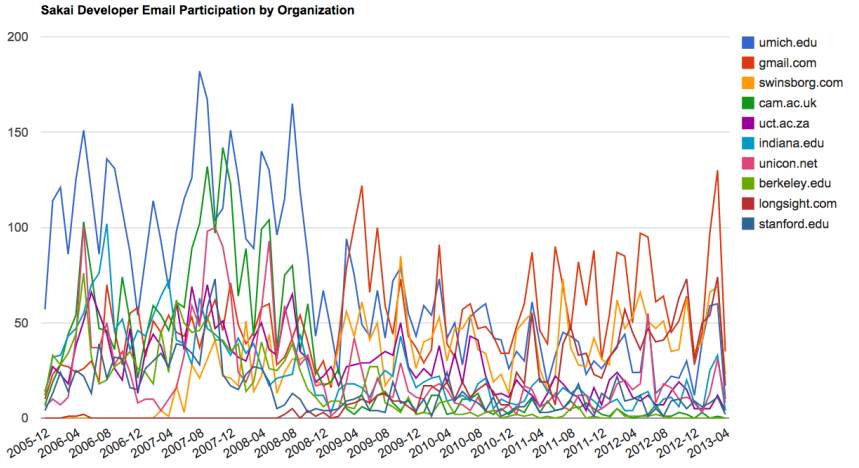
```
Range of counts: 33229 129
Output written to gword.js
```

이번 장 처음의 것과 유사한 워드 클라우드를 생성하기 위해서 gword.htm를 사용하여 시각화할 gword.js 파일을 생성한다.

두번째 시각화는 gline.py으로 생성된다. 시간에 따른 조직별 전자우편 참여를 연산한다.

```
Loaded messages= 51330 subjects= 25033 senders= 1584
Top 10 Oranizations
['gmail.com', 'umich.edu', 'uct.ac.za', 'indiana.edu',
'unicon.net', 'tfd.co.uk', 'berkeley.edu', 'longsight.com',
'stanford.edu', 'ox.ac.uk']
Output written to gline.js
```

출력값은 gline.htm을 사용하여 시각화된 gline.js에 쓰여진다.



작성한 프로그램은 상대적으로 복잡하고 정교한 응용프로그램으로 실제 데이터를 불러오고, 정비하고, 시각화하는 기능을 갖고 있다.

16 장

컴퓨터의 일반적인 작업 자동화

파일, 네트워크, 서비스, 그리고 데이터베이스에서 데이터를 읽어왔다. 파이썬은 또한 여러분의 로컬 컴퓨터 디렉토리와 폴더를 훑어서 파일도 읽어온다.

이번 장에서, 여러분의 로컬 컴퓨터를 스캔하고 각 파일에 대해서 연산을 수행하는 프로그램을 작성한다. 파일은 디렉토리(또한 "폴더"라고도 부른다.)에 정렬되어 보관된다. 간단한 파이썬 스크립트로 전체 로컬 컴퓨터나 디렉토리 여기저기 뒤져야 찾아지는 수백 수천개 파일에 대한 단순한 작업을 짧게 수행한다.

트리상의 디렉토리나 파일을 여기저기 돌아다니기 위해서 `os.walk`과 `for` 루프를 사용한다. `open`이 파일 콘텐츠를 읽는 루프를 작성하는 것과 비슷하게, `socket`은 네트워크 연결된 콘텐츠를 읽는 루프를 작성하고, `urllib`는 웹문서를 열어 콘텐츠를 루프를 통해서 읽어오게 한다.

제 1 절 파일 이름과 경로

모든 실행 프로그램은 "현재 디렉토리(current directory)"를 가지고 있는데 작업 대부분을 수행하는 디폴트 디렉토리가 된다. 예를 들어, 읽기 위해서 파일을 연다면, 파이썬은 현재 디렉토리에서 파일을 찾는다.

`os` 모듈(os는 "운영체제(operating system)"의 약자)은 파일과 디렉토리를 작업하는 함수를 제공한다. `os.getcwd`은 현재 디렉토리 이름을 반환한다.

```
>>> import os
>>> cwd = os.getcwd()
>>> print cwd
/Users/csev
```

`cwd` 는 **current working directory**의 약자로 현재 작업 디렉토리다. 예제의 결과는 `/Users/csev`인데 `csev` 사용자의 홈 디렉토리가 된다.

파일을 식별하는 `cwd` 같은 문자열을 경로(path)라고 부른다. 상대경로(relative path)는 현재 디렉토리에서 시작하고, 절대경로(absolute path)는 파일 시스템의 가장 최상단의 디렉토리에서 시작한다.

지금까지 살펴본 경로는 간단한 파일 이름이여서, 현재 디렉토리에서 상대적이다. 파일의 절대 경로를 알아내기 위해서 os.path.abspath을 사용한다.

```
>>> os.path.abspath('memo.txt')
'/Users/csev/memo.txt'
```

os.path.exists은 파일이나 디렉토리가 존재하는지 검사한다.

```
>>> os.path.exists('memo.txt')
True
```

만약 존재하면, os.path.isdir이 디렉토리인지 검사한다.

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

마찬가지로 os.path.isfile은 파일인지를 검사한다.

os.listdir은 주어진 디렉토리에 파일 리스트(그리고 다른 디렉토리)를 반환한다.

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

제 2 절 예제: 사진 디렉토리 정리하기

얼마 전에 핸드폰에서 사진을 받아서 서버에 저장하는 플릭커(Flickr)와 유사한 소프트웨어를 개발했다. 플릭커가 존재하기 전에 작성해고, 플릭커가 생긴 이후에도 계속해서 사용했다. 왜냐하면 영원히 원본을 보관하고 싶어서다.

문자메시지를 사용해서 한줄 텍스트 문자나 전자우편 주소 제목줄도 보낼 수 있다. 사진 파일과 마찬가지로 텍스프 파일형식 메시지를 동일한 디렉토리에 저장했다. 사진을 찍은 월, 년, 일, 그리고 시간에 기초한 디렉토리 구조다. 다음은 사진 한장과 설명 텍스트를 가진 예제다.

```
./2006/03/24-03-06_2018002.jpg
./2006/03/24-03-06_2018002.txt
```

7년이 지난 후에, 정말 많은 사진과 짧은 설명문이 생겼다. 시간이 지남에 따라 핸드폰을 바꿨다. 메시지에서 짧은 설명문을 뽑아내는 코드가 잘 동작하지 않고 짧은 설명문 대신에 쓸모없는 데이터를 서버에 추가했다.

파일을 훑어서 어느 텍스트 파일이 정말 짧은 설명문이고, 어는 것이 쓰레기인지 찾아서 잘못된 파일은 삭제하고 싶었다. 첫번째 할 일은 다음 프로그램을 사용하여 폴더에 전체 텍스트 파일 목록을 얻는 것이다.

```
import os
count = 0
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
```

```
        if filename.endswith('.txt') :
            count = count + 1
print 'Files:', count

python txtcount.py
Files: 1917
```

이것을 가능하게 하는 가장 중요한 코드는 파이썬 os.walk 라이브러리다. os.walk을 호출하고 시작 디렉토리를 지정해 주면, 재귀적으로 모든 디렉토리와 하위 디렉토리를 훑는다. "." 문자열은 현재 디렉토리에서 시작해서 하위 디렉토리로 훑는 것을 표시한다. 디렉토리 각각에 도착하면, for 루프 몸통부분에 있는 튜플에서 값 세개를 얻는다. 첫번째 값은 현재 디렉토리 이름, 두번째 값은 현재 디렉토리의 하위 디렉토리 리스트, 그리고 세번째 값은 현재 디렉토리 파일 리스트다.

명시적으로 하위 디렉토리 각각을 살펴보지 않는다. 왜냐하면, os.walk가 자동으로 모든 폴더를 방문할 것이기 때문이다. 하지만, 각 파일을 살펴보고 싶기 때문에, 간단한 for 루프를 작성해서 현재 디렉토리에 파일 각각을 조사한다. ".txt"로 끝나는 파일이 있는지 확인한다. 전체 디렉토리 트리를 훑어서 접미사 ".txt"로 끝나는 파일 숫자를 카운트한다.

얼마나 많은 파일이 ".txt" 확장자로 끝나는지 감을 잡았으면, 다음 작업은 자동적으로 어느 파일이 정상이고, 어느 파일이 문제가 있는지를 파이썬이 결정하도록 하는 것이다. 간단한 프로그램을 작성해서 파일과 파일의 크기를 출력한다.

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           thefile = os.path.join(dirname,filename)
           print os.path.getsize(thefile), thefile
```

파일을 단순히 카운트하는 대신에 os.path.join을 사용하여 디렉토리 안에서 디렉토리 이름과 파일 이름을 합쳐서 파일 이름을 생성한다. 문자열 연결 대신에 os.path.join을 사용하는 것이 중요한데 왜냐하면 파일 경로를 나타내기 위해서 윈도우에서는 파일 경로를 생성하기 위해서 역슬래쉬(\)를 사용하고, 리눅스나 애플에서는 슬래쉬 (/)를 사용하기 때문이다. os.path.join은 이러한 차이를 알고 어느 운영체제에서 동작하는지 인지하고 시스템에 따라 적절한 합치기 작업을 수행한다. 그래서 동일한 파이썬 코드가 윈도우나 유닉스계열 시스템에도 실행된다.

디렉토리 경로를 가진 전체 파일 이름을 갖게 되면, os.path.getsize 유틸리티를 사용해서 크기를 얻고 출력해서 다음 결과값을 만들어 낸다.

```
python txtsize.py
...
18 ./2006/03/24-03-06_2303002.txt
22 ./2006/03/25-03-06_1340001.txt
22 ./2006/03/25-03-06_2034001.txt
...
```

```
2565 ./2005/09/28-09-05_1043004.txt
2565 ./2005/09/28-09-05_1141002.txt
...
2578 ./2006/03/27-03-06_1618001.txt
2578 ./2006/03/28-03-06_2109001.txt
2578 ./2006/03/29-03-06_1355001.txt
...
```

출력값을 스캔하면, 몇몇 파일은 매우 짧고, 다른 많은 파일은 매우 큰데 동일한 크기(2578, 2565)임을 볼 수 있다. 수작업으로 몇개의 큰 파일을 살펴보면, T-Mobile 핸드폰에서 보내지는 전자우편에 함께 오는 일반적인 동일한 HTML을 가진 것임을 알 수 있다.

```
<html>
        <head>
                <title>T-Mobile</title>
...
```

파일을 대충 살펴보면, 파일에 그다지 유용한 정보가 없으므로 삭제한다.

하지만, 파일을 삭제하기 전에, 한 줄 이상인 파일을 찾고 내용을 출력하는 프로그램을 작성한다. 2578 혹은 2565 문자길이를 가진 파일을 보여주지는 않는다. 왜냐하면, 파일에 더 이상 유용한 정보가 없음을 알기 때문이다.

그래서 다음과 같이 프로그램을 작성한다.

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           thefile = os.path.join(dirname,filename)
           size = os.path.getsize(thefile)
           if size == 2578 or size == 2565:
               continue
           fhand = open(thefile,'r')
           lines = list()
           for line in fhand:
               lines.append(line)
           fhand.close()
           if len(lines) > 1:
                print len(lines), thefile
                print lines[:4]
```

`continue`**를 사용하여 두 "잘못된 크기(bad sizes)" 파일을 건너뛰고, 나머지 파일을 열고, 파이썬 리스트에 파일 라인을 읽는다. 만약 파일이 하나 이상의 라인이면, 파일에 얼마나 많은 라인이 있는지 출력하고, 첫 세줄을 출력한다.**

두개의 잘못된 파일 크기를 제외하고, 모든 한줄짜리 파일이 정상적이라고 가정하고나면 깨끗하게 정리된 파일을 생성된다.

```
python txtcheck.py
3 ./2004/03/22-03-04_2015.txt
['Little horse rider\r\n', '\r\n', '\r']
```

```
2 ./2004/11/30-11-04_1834001.txt
['Testing 123.\n', '\n']
3 ./2007/09/15-09-07_074202_03.txt
['\r\n', '\r\n', 'Sent from my iPhone\r\n']
3 ./2007/09/19-09-07_124857_01.txt
['\r\n', '\r\n', 'Sent from my iPhone\r\n']
3 ./2007/09/20-09-07_115617_01.txt
...
```

하지만, 한가지 더 성가신 패턴의 파일이 있다. 두 공백 라인과 "Sent from my iPhone"으로 구성된 3줄짜리 파일이다. 프로그램을 다음과 같이 변경하여 이러한 파일도 처리하게 한다.

```
lines = list()
for line in fhand:
    lines.append(line)
if len(lines) == 3 and lines[2].startswith('Sent from my iPhone'):
    continue
if len(lines) > 1:
     print len(lines), thefile
     print lines[:4]
```

단순하게 3줄짜리 파일이 있는지 검사하고 만약 지정된 텍스트로 세번째 라인이 시작한다면, 건너뛴다.

이제 프로그램을 실행하면, 4개의 다중 라인 파일만을 보게되고 모든 파일이 잘 처리된 것으로 보인다.

```
python txtcheck2.py
3 ./2004/03/22-03-04_2015.txt
['Little horse rider\r\n', '\r\n', '\r']
2 ./2004/11/30-11-04_1834001.txt
['Testing 123.\n', '\n']
2 ./2006/03/17-03-06_1806001.txt
['On the road again...\r\n', '\r\n']
2 ./2006/03/24-03-06_1740001.txt
['On the road again...\r\n', '\r\n']
```

프로그램의 전반적인 패턴을 살펴보면, 연속적으로 파일을 어떻게 승인할지와 거절할지를 정교화했고, "잘못된" 패턴을 발견하면 `continue`를 사용해서 잘못된 파일을 건너뛰게 했다. 그래서 잘못된 더 많은 파일 패턴이 탐지되도록 코드를 정교화했다.

이제 파일을 삭제할 준비가 되었다. 로직을 바꿔서, 나머지 올바른 파일을 출력하는 대신에, 삭제할 "잘못된" 파일만을 출력한다.

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           thefile = os.path.join(dirname,filename)
           size = os.path.getsize(thefile)
           if size == 2578 or size == 2565:
```

```
            print 'T-Mobile:',thefile
            continue
        fhand = open(thefile,'r')
        lines = list()
        for line in fhand:
            lines.append(line)
        fhand.close()
        if len(lines) == 3 and lines[2].startswith('Sent from my iPhone'):
            print 'iPhone:', thefile
            continue
```

이제 삭제할 대상 목록과 왜 이 파일이 삭제 대상으로 나왔는지를 볼 수 있다. 프로그램은 다음 출력을 생성한다.

```
python txtcheck3.py
...
T-Mobile: ./2006/05/31-05-06_1540001.txt
T-Mobile: ./2006/05/31-05-06_1648001.txt
iPhone: ./2007/09/15-09-07_074202_03.txt
iPhone: ./2007/09/15-09-07_144641_01.txt
iPhone: ./2007/09/19-09-07_124857_01.txt
...
```

무작위로 파일을 검사해서 프로그램에 버그가 우연하게 들어가 있는지 혹은 원치 않는 파일이 작성한 프로그램 로직에 끌려들어 갔는지 확인할 수 있다.

결과값에 만족하고, 다음이 삭제할 파일 목록임으로, 프로그램에 다음과 같은 변경을 한다.

```
        if size == 2578 or size == 2565:
            print 'T-Mobile:',thefile
            os.remove(thefile)
            continue
...
        if len(lines) == 3 and lines[2].startswith('Sent from my iPhone'):
            print 'iPhone:', thefile
            os.remove(thefile)
            continue
```

이번 버젼의 프로그램에서 파일을 출력하고 `os.remove`**을 사용하여 잘못된 파일을 삭제한다.**

```
python txtdelete.py
T-Mobile: ./2005/01/02-01-05_1356001.txt
T-Mobile: ./2005/01/02-01-05_1858001.txt
...
```

재미로, 프로그램을 두번 실행하게 되면 모든 잘못된 파일이 삭제되어서 출력값이 없다.

`txtcount.py`**을 다시 실행하면, 899 잘못된 파일이 삭제되었음을 알 수 있다.**

```
python txtcount.py
Files: 1018
```

이번 장에서 일련의 절차에 따라 파이썬을 사용하여 디렉토리와 파일을 검색하는 패턴을 살펴봤다. 천천히 파이썬을 사용해서 디렉토리를 정리하기 위해서 무엇을 할지를 결정했다. 어느 파일이 좋고 어느 파일이 유용하지 않는지 파악한 후에 파이썬을 사용해서 파일을 삭제하고 파일 정리를 수행했다.

해결하려고 하는 문제가 무척 간단하여 파일 이름만을 살펴보는 것으로 충분히 처리할 수 있다. 혹은 모든 파일을 읽고 파일 내부에 패턴을 찾을 필요가 있을 수도 있다. 때때로, 모든 파일을 읽고, 파일의 일부에 변경이 필요할지도 모른다. `os.walk`와 다른 `os` 유틸리티가 어떻게 사용되는지 이해하기만 하면 이 모든 것은 매우 명확한다.

제 3 절 명령 줄 인자

앞선 장에서 `raw_input`을 사용하여 파일명을 사용자로부터 입력받고, 파일에서 데이터를 읽어 다음과 같이 처리했다.

```
name = raw_input('Enter file:')
handle = open(name, 'r')
text = handle.read()
...
```

파이썬을 시작할 때, 명령 라인에서 파일 이름을 입력 받음으써 프로그램을 간략화 할 수 있다. 지금까지 파이썬 프로그램을 단순하게 실행하고 다음과 같이 명령어 프롬프트에 응답했다.

```
python words.py
Enter file: mbox-short.txt
...
```

파이썬 파일 다음에 추가 문자열을 배치하고 파이썬 프로그램에서 명령 줄 인수(command line arguments)에 접근한다. 명령 줄에서 인자를 읽는 것을 시연하는 간단한 프로그램이 다음에 있다.

```
import sys
print 'Count:', len(sys.argv)
print 'Type:', type(sys.argv)
for arg in sys.argv:
   print 'Argument:', arg
```

`sys.argv`의 콘텐츠는 문자열 리스트로 첫 문자열은 파이썬 프로그램 이름, 그리고 나머지 문자열은 파이썬 파일 다음의 명령 줄의 인자들이다.

다음은 명령 줄에서 명령 줄 인자 몇개를 읽어들이는 프로그램이다.

```
python argtest.py hello there
Count: 3
Type: <type 'list'>
Argument: argtest.py
Argument: hello
Argument: there
```

세가지 요소 리스트로 프로그램에 넘겨지는 인자가 3개 있다. 리스트의 첫 요소는 파일 이름 (argtest.py) 그리고 나머지는 파일 이름 뒤에 2 명령 줄 인자다.

파일을 읽어 오는 프로그램을 다시 작성할 수 있다. 다음과 같이 명령 줄 인자로 파일 명을 받는다.

```
import sys

name = sys.argv[1]
handle = open(name, 'r')
text = handle.read()
print name, 'is', len(text), 'bytes'
```

두번째 명령 줄 인자를 파일 이름으로 취한다. (`[0]` 항목 프로그램 이름은 생략한다.) 파일을 열고 다음과 같이 콘텐츠를 읽는다.

```
python argfile.py mbox-short.txt
mbox-short.txt is 94626 bytes
```

입력값으로 명령문 인자를 사용하는 것은 파이썬 프로그램을 재사용하기 쉽게 하고, 특히 하나 혹은 두개의 문자열을 입력받을 때 유용한다.

제 4 절 파이프(Pipes)

대부분의 운영 시스템은 쉘(shell)로 알려진 명령어 기반 인터페이스를 지원한다. 일반적으로 쉘은 파일 시스템을 탐색하거나 응용 프로그램을 실행하는 명령어를 지원한다. 예를 들어, 유닉스에서 `cd` 명령어로 디렉토리를 변경하고 `ls` 명령어로 디렉토리의 콘텐츠를 보여주고, `firefox`를 타이핑해서 웹브라우져를 실행한다.

쉘에서 실행시킬 수 있는 어떤 프로그램이나 파이프(pipe)를 사용하여 파이썬에서도 실행시킬 수 있다. 파이프는 작동중인 프로세스를 표현하는 객체다.

예를 들어, 유닉스 명령어[1] `ls -l`는 정상적으로 현재 디렉토리의 콘텐츠(긴 형식으로)를 보여준다. `os.popen`를 가지고 `ls` 를 실행시킬 수 있다.

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

인자는 쉘 명령어를 포함하는 문자열이다. 반환 값은 파일 포인터로 열린 파일처럼 동작한다. `readline`으로 한번에 한 라인씩 `ls` 프로세스로부터 출력을 읽거나 `read`로 한번에 전체를 가져올 수 있다.

```
>>> res = fp.read()
```

모두 완료되면, 파일처럼 파이프를 닫는다.

[1] 파이프를 사용하여 `ls` 같은 운영 시스템 명령어로 대화할 때, 무슨 운영 시스템을 사용하는지 알고 운영 시스템에서 지원되는 명령어로 파이프를 열수 있다는 것이 중요하다.

```
>>> stat = fp.close()
>>> print stat
None
```

반환되는 값은 `ls` 프로세스의 최종 상태값이다. `None`은 (오류 없이) 정상적으로 종료됨을 의미한다.

제 5 절 용어정의

절대경로(absolute path): 파일이나 디렉토리가 어디에 저장되어 있는지를 저장하는 문자열로 "최상단의 디렉토리"에서 시작해서, 현재 작업 디렉토리에 관계없이 파일이나 디렉토리를 접근하는데 사용할 수 있다.

체크썸(checksum): 해싱(hashing)을 참조하세요. "체크썸(checksum)"단어는 네트워크로 데이터가 보내지거나 백업 매체에 쓰여지고 다시 읽어올 때, 데이터가 왜곡되었는지를 검증하는 필요에서 생겨났다. 데이터가 쓰여지거나 보내질 때, 송신 시스템은 체크썸을 계산하고 또한 체크썸도 보낸다. 데이터가 읽혀지거나 받았을 때, 수신 시스템을 수신된 데이터의 체크썸을 다시 계산하고 받은 체크썸과 비교한다. 만약 체크썸이 매칭되지 않으면, 전송 시에 데이터가 왜곡된 것으로 판단해야 한다.

명령 줄 인자(command line argument): 파이썬 파일 이름 뒤에 명령 줄에 매개변수.

현재 작업 디렉토리(current working directory): 여러분이 "작업하고 있는" 현재 디렉토리. 명령-줄 인터페이스에서 대부분의 시스템에 `cd` 명령어를 사용하여 작업 디렉토리를 변경할 수 있다. 경로 정보 없이 파일만을 사용하여 파이썬에서 파일을 열게 될 때, 파일은 프로그램을 실행하고 있는현재 작업 디렉토리에 있어야 한다.

해싱(hashing): 가능한 큰 데이터를 읽고 그 데이터에 대해서 유일한 체크썸을 생성하는 것. 최고의 해쉬 함수는 거의 "충돌(collision)"을 만들지 않는다. 여기서 충돌은 서로 다른 두 데이터 스트림에 해쉬 함수를 줄 때 동일한 해쉬값을 돌려받는 것이다. **MD5, SHA1, SHA256** 는 가장 많이 사용되는 해쉬 함수의 사례다.

파이프(pipe): 파이프는 실행하는 프로그램에 연결이다. 파이프를 사용해서, 데이터를 다른 프로그램에 보내거나 그 프로그램에서 데이터를 받는 프로그램을 작성할 수 있다. 파이프는 소켓(socket)과 매우 유사하다. 차이점은 파이프는 동일한 컴퓨터에서 실행되는 프로그램을 연결하는데만 사용된다는 것이다. (즉, 네트워크를 통해서는 사용할 수 없다.)

상대경로(relative path): 파일 혹은 디렉토리가 어디에 저장되었는지를 현재 작업 디렉토리에 상대적으로 표현하는 문자열.

쉘(shell): 운영 시스템에 명령줄 인터페이스. 다른 시스템에서는 또한 "터미널 프로그램(terminal program)"이라고 부른다. 이런 인터페이스에서 라인에 명령어와 매개 변수를 타입하고 명령을 실행하기 위해서 "엔터(enter)"를 누른다.

워크(walk): 모든 디렉토리를 방문할 때까지 디렉토리, 하위 디렉토리, 하위의 하위 디렉토리 전체 트리를 방문하는 개념을 나타내기 위해서 사용된 용어. 여기서 이것을 "디렉토리 트리를 워크"한다고 부른다.

제 6 절 연습문제

Exercise 16.1 MP3 파일이 대규모로 수집되어 있은 곳에는 같은 노래의 복사본 하나 이상이 다른 디렉토리 혹은 다른 파일 이름으로 저장되어 있을 수 있다. 이번 연습문제의 목표는 이런 중복 파일을 찾는 것이다.

1. .mp3같은 확장자를 가진 파일을 모든 디렉토리와 하위 디렉토리를 검색해서 동일한 크기를 가진 파일짝을 목록으로 보여주는 프로그램을 작성하세요. 힌트: 딕셔너리를 사용하세요. 딕셔너리의 키는 `os.path.getsize`에서 파일의 크기가 되고, 딕셔너리의 값(value)는 파일 이름과 결합된 경로명이 된다. 파일을 매번 마주칠 때마다 현재 파일과 동일한 크기를 가진 파일이 이미 존재하는지를 검사한다. 만약 그렇다면, 중복된 크기 파일이 있고 파일 크기와 두 파일 이름을 출력한다. (해쉬에서 한 파일, 찾고 있는 곳에서 또다른 파일)

2. **체크썸(checksum)** 알고리즘이나 해싱을 사용하여 중복 콘텐츠를 가진 파일을 찾는 이전의 프로그램을 개작하세요. 예를 들어, MD5 (Message-Digest algorithm 5)는 임의적으로 긴 "메시지"를 가지고 128비트 "체크썸"을 반환한다. 다른 콘텐츠를 가진 두 파일이 같은 체크썸을 반환할 확률은 매우 적다.

   `wikipedia.org/wiki/Md5`에서 MD5에 대해서 더 배울 수 있다. 다음 코드 조각은 파일을 열고, 읽고, 체크썸을 계산한다.

```
import hashlib
...
            fhand = open(thefile,'r')
            data = fhand.read()
            fhand.close()
            checksum = hashlib.md5(data).hexdigest()
```

   딕셔너리를 생성해서 체크썸이 키가 되고 파일 이름이 값(value)이 된다. 체크섬을 계산하고 키로 이미 딕셔너리에 있게 되면, 중복 콘텐츠인 두 파일 있어서 딕셔너리에 파일과 방금전에 읽은 파일을 출력한다. 사진 파일 폴더에서 실행한 샘플 출력물이 다음에 있다.

```
./2004/11/15-11-04_0923001.jpg ./2004/11/15-11-04_1016001.jpg
./2005/06/28-06-05_1500001.jpg ./2005/06/28-06-05_1502001.jpg
./2006/08/11-08-06_205948_01.jpg ./2006/08/12-08-06_155318_02.jpg
```

   명백하게 때때로 같은 사진을 한번 이상 보내거나 원본을 삭제하지 않고 종종 사진 사본을 만든다.

부록 A

윈도우 상에서 파이썬 프로그래밍

이번 부록에서 일련의 단계를 거쳐서 윈도우상에서 파이썬을 실행한다.

파이썬 프로그램을 편집하고 실행하는데취할 수 있는 서로 다른 많은 접근방법이 있다. 이것은 간단한 접근 방법중의 하나다.

먼저, 프로그래머 편집기를 설치할 필요가 있다. Notepad나 윈도우 워드를 가지고 파이썬 프로그램을 편집할 필요는 없다. 프로그램은 "일반 텍스트(flat-text)" 파일이여서 텍스트 파일을 편집하는데 좋은 에디터만 필요하다.

윈도우 시스템에 추천하고 싶은 편집기는 다음에서 다운받아 설치할 수 있는 Notepad++다.

http://sourceforge.net/projects/notepad-plus/files/

www.python.org 웹사이트에서 파이썬 2를 다운로드한다.

http://www.python.org/download/releases/2.7.5/

파이썬을 설치하면, 컴퓨터에 `C:\Python27` 같은 새로운 폴더가 생긴다.

파이썬 프로그램을 생성하기 위해서 시작 메뉴에서 NotePad++를 실행하고 확장자가 ".py"인 파일로 저장한다. 연습으로 `py4inf` 이름의 폴더를 바탕화면에 생성한다. 폴더명을 짧게 하고 폴더명과 파일명에 어떤 공백도 넣지 않는 것이 좋다.

첫 파이썬 프로그램을 다음과 같이 작성한다.

```
print 'Hello Chuck'
```

여러분의 이름으로 바꾸는 것을 제외하고, 파일을 바탕화면`\py4inf\prog1.py` 에 저장한다.

명령-줄(command line) 실행은 윈도우 버젼마다 다른다.

- 윈도우 비스타, 윈도우 7: 시작(Start)을 누루고, 명령어 실행 윈도우에서 단어 `command`를 입력하고 엔터를 친다.

- **윈도우-XP: 시작(Start)을 누루고,실행(Run)을 누루고, 대화창에서 단어** `cmd`**를 입력하고 확인(OK)을 누른다.**

지금 어느 폴더에 있는지를 말해주는 프롬프트 텍스트 윈도우에서 현재 위치를 확인할 수 있다.

Windows Vista and Windows-7: `C:\Users\csev`
Windows XP: `C:\Documents and Settings\csev`

이것이 "홈 디렉토리"다. 이제 다음 명령어를 사용해서 작성한 파이썬 프로그램을 저장한 폴더로 이동한다.

```
C:\Users\csev\> cd Desktop
C:\Users\csev\Desktop> cd py4inf
```

그리고 다음을 타이핑한다.

```
C:\Users\csev\Desktop\py4inf> dir
```

작성한 파일 목록을 보기 위해서 `dir` **명령어를 타이핑 할때,** `prog1.py`**이 보여야 한다.**

프로그램을 실행하기 위해서, 단순히 명령 프롬프트에서 파일 이름을 타이핑하고 엔터를 친다.

```
C:\Users\csev\Desktop\py4inf> prog1.py
Hello Chuck
C:\Users\csev\Desktop\py4inf>
```

NotePad++에서 파일을 편집하고, 저장하고, 명령줄로 돌아온다. 다시 명령줄 프롬프트에서 파일명을 타이핑해서 프로그램을 실행한다.

만약 명령줄 윈도우에서 혼동이 생기면, 단순하게 닫고 새로 시작한다.

힌트: 스크롤 백해서 이전에 입력한 명령을 다시 실행하기 위해서 "위쪽 화살표"를 명령줄에서 누른다.

NotePad++에서 환경설정 선호(preference)를 살펴보고 탭 문자가 공백 4개로 되도록 설정한다. 이 단순한 설정이 들여쓰기 오류를 찾는 수고를 많이 경감시켜 준다.

`www.py4inf.com`에서 파이썬 프로그램을 편집하고 실행하는 좀더 많은 정보를 얻을 수 있다.

부록 B

매킨토쉬 상에서 파이썬 프로그래밍

이번 부록에서 일련의 단계를 거쳐서 매킨토쉬상에서 파이썬을 실행한다. 파이썬이 이미 매킨토쉬 운영시스템에 포함되어 있어서, 어떻게 파이썬 파일을 편집하는지와 터미널 윈도우에서 파이썬 프로그램을 실행하는 것을 학습할 필요가 있다.

파이썬 프로그램을 편집하고 실행하는데 취할 수 있는 서로 다른 많은 접근방법이 있다. 이것은 간단한 접근 방법중의 하나다.

먼저, 프로그래머 편집기를 설치할 필요가 있다. Notepad나 윈도우 워드를 가지고 파이썬 프로그램을 편집할 필요는 없다. 프로그램은 "일반 텍스트(flat-text)" 파일이여서 텍스트 파일을 편집하는데 좋은 에디터만 필요하다.

매킨토쉬 시스템에 추천하고 싶은 편집기는 다음에서 다운받아 설치할 수 있는 TextWrangler다.

http://www.barebones.com/products/TextWrangler/

파이썬 프로그램을 생성하기 위해서 Applications 폴더 TextWrangler를 실행한다.

첫 파이썬 프로그램을 다음과 같이 작성한다.

```
print 'Hello Chuck'
```

여러분의 이름으로 바꾸는 것을 제외하고, 파일을

바탕화면\py4inf\prog1.py에 저장한다.

명령-줄(command line) 실행은 윈도우 버젼마다 다른다. py4inf로 데스크탑(Desktop) 폴더에 파일을 저장한다. 폴더명을 짧게 하고 폴더명과 파일명에 어떤 공백도 넣지 않는 것이 좋다. 폴더를 만들고 파일을 Desktop\py4inf\prog1.py 으로 저장한다.

터미널(Terminal) 프로그램을 실행. 가장 쉬운 방법은 화면 우측 상단의 Spotlight 아이콘(돋보기)를 누르고, "terminal" 엔터치고 응용프로그램을 실행한다.

"홈 디렉토리"에서 시작한다. 터미널 윈도우에서 `pwd` 명령어를 타이핑해서 현재 디렉토리를 확인한다.

```
67-194-80-15:~ csev$ pwd
/Users/csev
67-194-80-15:~ csev$
```

프로그램을 실행하기 위해서 파이썬 프로그램을 담고 있는 폴더에 있어야 한다. `cd` 명령어를 사용해서 새 폴더로 이동하고 `ls` 명령어로 폴더의 파일 목록을 화면에 출력한다.

```
67-194-80-15:~ csev$ cd Desktop
67-194-80-15:Desktop csev$ cd py4inf
67-194-80-15:py4inf csev$ ls
prog1.py
67-194-80-15:py4inf csev$
```

프로그램을 실행하기 위해서, 단순히 명령 프롬프트에서 `python` 명령어와 파일이름을 타이핑하고 엔터를 친다.

```
67-194-80-15:py4inf csev$ python prog1.py
Hello Chuck
67-194-80-15:py4inf csev$
```

TextWrangler에서 파일을 편집하고, 저장하고, 명령줄로 돌아온다. 다시 명령줄 프롬프트에서 파일명을 타이핑해서 프로그램을 실행한다.

만약 명령줄 윈도우에서 혼동이 생기면, 단순하게 닫고 새로 시작한다.

힌트: 스크롤 백해서 이전에 입력한 명령을 다시 실행하기 위해서 "위쪽 화살표"를 명령줄에서 누른다.

TextWrangler에서 환경설정 선호(preference)를 살펴보고 탭 문자가 공백 4개로 되도록 설정한다. 이 단순한 설정이 들여쓰기 오류를 찾는 수고를 많이 경감시켜 준다.

`www.py4inf.com`에서 파이썬 프로그램을 편집하고 실행하는 좀더 많은 정보를 얻을 수 있다.

부록 C

공헌(contribution)

”정보교육을 위한 파이썬”에 공헌하신 분 목록

Bruce Shields 초기 초안을 편집, Sarah Hegge, Steven Cherry, Sarah Kathleen Barbarow, Andrea Parker, Radaphat Chongthammakun, Megan Hixon, Kirby Urner, Sarah Kathleen Barbrow, Katie Kujala, Noah Botimer, Emily Alinder, Mark Thompson-Kular, James Perry, Eric Hofer, Eytan Adar, Peter Robinson, Deborah J. Nelson, Jonathan C. Anthony, Eden Rassette, Jeannette Schroeder, Justin Feezell, Chuanqi Li, Gerald Gordinier, Gavin Thomas Strassel, Ryan Clement, Alissa Talley, Caitlin Holman, Yong-Mi Kim, Karen Stover, Cherie Edmonds, Maria Seiferle, Romer Kristi D. Aranas (RK), Grant Boyer, Hedemarrie Dussan,

“Think Python” 서문

“Think Python” 특이한 역사

(Allen B. Downey)

1999년 1월 자바로 프로그램 기초 과목 수업을 준비하고 있었다. 세번에 걸쳐서 수업을 했지만 좌절하고 있었다. 학급에서 낙제비율이 무척이나 높았고, 정상적으로 이수한 학생의 전반적인 성취도가 무척 낮았다.

목격한 여러 문제점 중의 하는 책이다. 자바 책이 너무 방대하고 불필요하게 상세한 정보가 너무 많았고 어떻게 프로그램을 작성하는지에 대한 높은 수준의 지침은 부족했다. 학생들 모두 뚜껑문 효과(trapdoor effect)로 고생했다. 즉, 쉽게 시작하고, 점진적으로 나아가지고나서 5장 주변에 하위권 학생이 떨어져 나가는 것이다. 너무 많은 새로운 교재를 너무 빨리 학습해야했고, 저자는 학기의 나머지를 수습하는데 사용했다.

첫 수업시작하기 2주일 전에 직접 책을 쓰기로 마음먹었다. 목표는 다음과 같다.

- 짧게 한다. 읽지 않는 50 페이지보다 읽히는 10 페이지가 더 낫다.
- 용어에 주의한다. 전문용어를 최소하하고 첫번째 사용에 각 용어를 정의한다.
- 점진적으로 구축해 나간다. 뚜껑문 효과를 피하기 위해서, 가장 어려운 주제를 잡고 일련의 작은 단계로 쪼갠다.
- 프로그래밍 언어보다 프로그래밍에 집중한다. 최소한의 유용한 자바 부분을 포함하고 나머지는 생략한다.

제목이 필요했고, 즉흥적으로 *어떻게 컴퓨터 과학자처럼 생각하기 (How to Think Like a Computer Scientist)*로 정했다.

첫 버젼은 엉성했지만 실질적으로 작동했다. 학생들은 읽고, 충분히 이해해서 수업시간을 어렵고, 흥미로운 주제에 좀더 시간을 쓸 수 있었고, 가장 중요한 것은 학생들이 실습을 하게된 것이다.

GNU 공개 문서 라이센스(GNU Free Documentation License)로 책을 배포해서 누구나 복사, 편집, 배포할 수 있게 했다.

다음에 멋진 일이 생겼다. 버지니아 고등학교 교사인 Jeff Elkner 선생님이 책을 채용해서 파이썬으로 번역했다. Jeff Elkner 선생님은 번역본을 보내왔고, 책을 읽고서 파이썬을 공부하게 되는 좀 특이한 경험을 했다.

Jeff와 저자는 책을 개작했고 Chris Meyers가 사례 연구를 추가했다. 2001년 GNU 공개 문서 라이센스로 *How to Think Like a Computer Scientist: Learning with Python* 제목으로 배포했다. Green Tea Press 출판사에서 책을 출판해서 Amazon.com과 대학 서점을 통해서 제본된 책을 팔기 시작했다. Green Tea Press 출판사의 다른 책들은 `greenteapress.com`에서 살펴볼 수 있다.

2003년 Olin College에서 수업을 시작해서 처음으로 파이썬을 가르치게 되었다. 자바와 비교하여 놀라웠다. 학생들이 덜 고생하고, 더 많이 배우고, 좀더 흥미로운 프로젝트를 수행하고, 전반적으로 훨씬 재미있게 되었다.

지난 5년 동안 책을 계속적으로 개발하고, 오류를 수정하고, 예제을 향상하고, 교재, 특히 연습문제를 추가했다. 2008년 대대적인 수정 작업을 시작했는데 동시에 차기 개정판에 관심을 보인 Cambridge University Press 편집자와 계약했다. 좋은 시점이다.

이 책을 즐기고, 컴퓨터 과학자처럼 적어도 약간은 프로그램하고 생각하는 것을 배우는데 도움이 되기를 바랍니다.

"Think Python" 감사의 글

(Allen B. Downey) 처음으로 가장 중요하게, Jeff Elkner에게 감사드린다. 자바 책을 파이썬으로 번역해서 이 프로젝트가 시작되게 했고 가장 좋아하는 언어가 된 파이썬을 소개해 주었다.

Chris Meyers에게도 감사의 말씀을 드린다. *How to Think Like a Computer Scientist* 책의 몇개 부분에 기여해 주셨다.

Jeff와 Chris와 협동작업을 할수 있게 만든 GNU 공개 문서 라이센스를 개발한 자유 소프트웨어 재단(Free Software Foundation)에 감사한다.

또한, *How to Think Like a Computer Scientist* 책을 작업하신 Lulu의 편집자에게 감사한다.

책의 초기 버젼을 함께 작업한 모든 학생들과 수정과 제안을 보내주신 부론에 나온 모든 기여자에게 감사한다.

그리고, 집사람 Lisa, Green Tea Press 그리고 다른 모든 것에도 감사한다.

Allen B. Downey
Needham MA

Allen Downey 컴퓨터 과학 부교수 Franklin W. Olin College of Engineering

"Think Python" 공헌자 목록

(Allen B. Downey)

날카로운 눈과 사려깊은 100명 이상의 독자가 지난 몇년동안 수정사항과 제안을 보내주었다. 이 프로젝트의 공헌과 열정은 매우 큰 도움이었다.

이들 참여자로부터의 각각의 공헌의 본질에 대한 자세한 사항은 "Think Python" 본문에서 확인하세요.

Lloyd Hugh Allen, Yvon Boulianne, Fred Bremmer, Jonah Cohen, Michael Conlon, Benoit Girard, Courtney Gleason and Katherine Smith, Lee Harr, James Kaylin, David Kershaw, Eddie Lam, Man-Yong Lee, David Mayo, Chris McAloon, Matthew J. Moelter, Simon Dicon Montford, John Ouzts, Kevin Parks, David Pool, Michael Schmitt, Robin Shaw, Paul Sleigh, Craig T. Snydal, Ian Thomas, Keith Verheyden, Peter Winstanley, Chris Wrobel, Moshe Zadka, Christoph Zwerschke, James Mayer, Hayden McAfee, Angel Arnal, Tauhidul Hoque and Lex Berezhny, Dr. Michele Alzetta, Andy Mitchell, Kalin Harvey, Christopher P. Smith, David Hutchins, Gregor Lingl, Julie Peters, Florin Oprina, D. J. Webre, Ken, Ivo Wever, Curtis Yanko, Ben Logan, Jason Armstrong, Louis Cordier, Brian Cain, Rob Black, Jean-Philippe Rey at Ecole Centrale Paris, Jason Mader at George Washington University made a number Jan Gundtofte-Bruun, Abel David and Alexis Dinno, Charles Thayer, Roger Sperberg, Sam Bull, Andrew Cheung, C. Corey Capel, Alessandra, Wim Champagne, Douglas Wright, Jared Spindor, Lin Peiheng, Ray Hagtvedt, Torsten Hübsch, Inga Petuhhov, Arne Babenhauserheide, Mark E. Casida, Scott Tyler, Gordon Shephard, Andrew Turner, Adam Hobart, Daryl Hammond and Sarah Zimmerman, George

Sass, Brian Bingham, Leah Engelbert-Fenton, Joe Funke, Chao-chao Chen, Jeff Paine, Lubos Pintes, Gregg Lind and Abigail Heithoff, Max Hailperin, Chotipat Pornavalai, Stanislaw Antol, Eric Pashman, Miguel Azevedo, Jianhua Liu, Nick King, Martin Zuther, Adam Zimmerman, Ratnakar Tiwari, Anurag Goel, Kelli Kratzer, Mark Griffiths, Roydan Ongie, Patryk Wolowiec, Mark Chonofsky, Russell Coleman, Wei Huang, Karen Barber, Nam Nguyen, Stéphane Morin, and Paul Stoop.

부록 D

저작권 세부정보

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
2013년 9월 9일

찾아보기