

정보교육을 위한 파이썬

정보 탐색

Version 0.0.9-d2

저자: Charles Severance

번역: 이광춘, 한정수
(xwmooc)

Copyright © 2009- Charles Severance.

Printing history:

October 2013: Major revision to Chapters 13 and 14 to switch to JSON and use OAuth.
Added new chapter on Visualization.

September 2013: Published book on Amazon CreateSpace

January 2010: Published book using the University of Michigan Espresso Book machine.

December 2009: Major revision to chapters 2-10 from *Think Python: How to Think Like a Computer Scientist* and writing chapters 1 and 11-15 to produce *Python for Informatics: Exploring Information*

June 2008: Major revision, changed title to *Think Python: How to Think Like a Computer Scientist*.

August 2007: Major revision, changed title to *How to Think Like a (Python) Programmer*.

April 2002: First edition of *How to Think Like a Computer Scientist*.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. This license is available at creativecommons.org/licenses/by-nc-sa/3.0/. You can see what the author considers commercial and non-commercial uses of this material as well as license exemptions in the Appendix titled Copyright Detail.

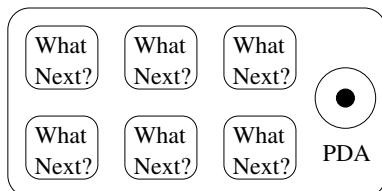
The L^AT_EX source for the *Think Python: How to Think Like a Computer Scientist* version of this book is available from <http://www.thinkpython.com>.

Chapter 1

왜 프로그래밍을 배워야 하는가?

컴퓨터 프로그램을 만드는 행위(프로그래밍)는 매우 창의적이며 향후 뿐만 아니라 이상으로 얻을 것이 많다. 프로그램을 만드는 이유는 어려운 자료분석의 문제를 해결하려는 것에서부터 다른 사람의 문제를 해결해주는 재미를 느끼는 것까지 다양한 이유가 있다. 이 책에서 모든 사람이 어떻게 프로그램을 만드는지를 알고, 프로그램을 만드는지를 알게되면, 새로 습득한 프로그래밍 기술로 원하는 것을 해결할 수 있는 것을 배우게 된다.

우리의 일상은 노트북부터 핸드폰까지 다양한 컴퓨터에 둘러싸여 있다. 이러한 컴퓨터가 개인비서로 우리를 위해서 많은 일을 대신해 준다고 생각한다. 일상생활에서 접하는 컴퓨터 하드웨어는 우리에게 ”다음에 무엇을 하면 좋겠습니까?”라는 질문을 지속적으로 물어보게 만들어 졌다.



프로그래머는 운영체제와 하드웨어에 응용 프로그램을 만들었고, 결국 많은 것들을 도와주는 PDA(Personal Digital Assistant)로 진화했다. 컴퓨터는 빠르며, 큰 저장소를 가지고 있어 우리가 컴퓨터에게 ”다음꺼 실행해(do next)를 컴퓨터가 이해할 수 있는 말로 지시를 하게되면 우리에게 매우 유용할 수 있다.

예를 들어, 다음의 세 문단을 보고 가장 많이 나오는 문단의 단어를 찾아보고 얼마나 나오는지를 알려주세요라고 컴퓨터에게 시킬 수 있다. 사람이 몇초만에 단어를 읽고 이해할 수는 있지만, 그 단어가 몇번 나오는지를 세는 것은 매우 고생스러운 과정이다. 왜냐하면 사람은 지루하고 반복되는 일의 문제를 해결하는데 적합하지 않기 때문이다. 컴퓨터는 정반대이다. 논문이나 책에서 텍스트를 읽고 이해하는 것은 컴퓨터에게 어렵다. 하지만 단어를 세고 가장 많이 사용되는 단어를 말해주는 것은 컴퓨터에게는 무척이나 쉽다.

python words.py

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

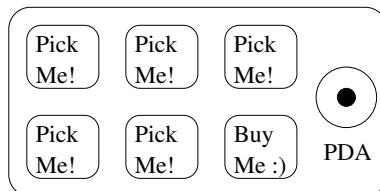
우리의 개인 정보분석 비서는 ”to”라는 단어가 가장 많이 사용되었고 16번 나왔다고 바로 답을 준다.

사람이 잘하지 못하는 점을 컴퓨터가 잘할 수 있다는 사실을 이해하면 왜 컴퓨터 언어로 컴퓨터와 대화해야 하는지를 알 수 있다. 컴퓨터와 대화할 수 있는 언어(Python)를 배우게되면 지루하고 반복되는 일을 컴퓨터가 처리하게 하면 더 많은 시간을 창의적이고, 직관적이며, 창조적인 시간을 컴퓨터와 함께 할 수 있다.

1.1 창의성과 동기

이책은 직업 프로그래머를 위해서 저작된 것은 아니지만, 직업적으로 프로그램을 만드는 작업은 개인적으로나 경제적인 면에서 꽤 매력적인 일이다. 특히, 유용하며, 심미적이고, 똑똑한 프로그램을 다른 사람이 사용할 수 있도록 만드는 것은 매우 창의적인 일이다. 컴퓨터는 다양한 그룹의 프로그래머들이 사용자의 관심과 시선을 빼았기 위해서 경쟁적으로 다양한 프로그램을 가지고 있다. 이렇게 개발된 프로그램은 사용자가 원하는 바를 충족시키고 훌륭한 사용자 경험을 주려고 노력한다. 이러한 상황에서 사용자가 소프트웨어를 고르게 될 때 고객의 선택에 대해서 프로그래머는 직접적으로 보상을 받게된다.

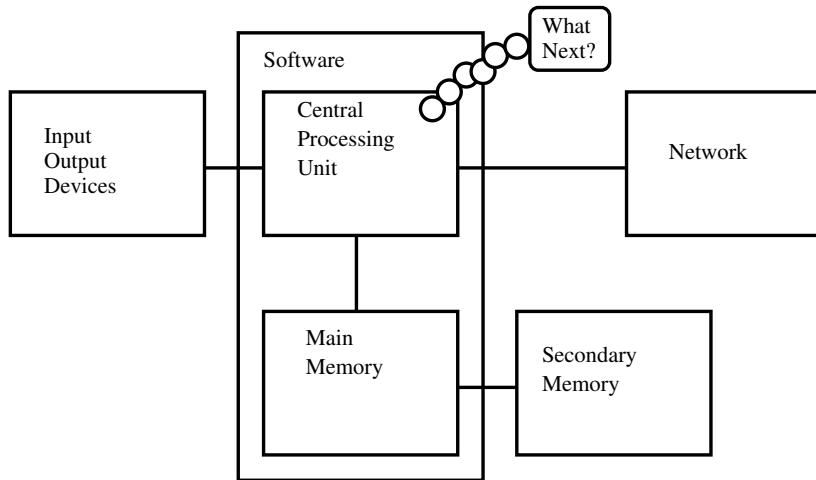
만약 프로그램을 프로그래머 집단의 창의적인 결과물로 바라본다면, 아마도 다음의 그림이 PDA 컴퓨터에 의미가 있을 듯 하다.



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

1.2 컴퓨터 하드웨어 아키텍처

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



주요 부품을 살펴보자.

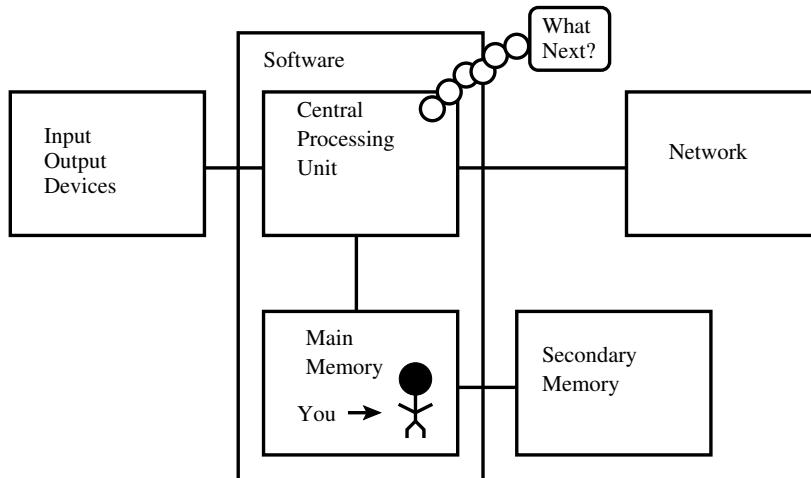
- **중앙처리장치(Central Processing Unit, CPU):** 다음은 무엇을 할까요? (“What is next?”) 명령어를 처리하는 주요 부분이다. 컴퓨터가 3.0 GHz라면 초당 명령어(다음은 무엇을 할까요? What is next?)를 삼백만번 처리할 수 있다고 계속 물을 수 있다. CPU의 처리속도를 따라서 컴퓨터와 빠르게 대화하는 것을 배울 것이다.
- **주 기억장치(Main Memory):** 주 기억장치는 중앙처리장치(CPU)가 급하게 명령어를 처리하기 위해 필요로 하는 정보를 저장하는 용도로 사용된다. 주 기억장치는 중앙처리장치만큼이나 빠르다. 그러나 주 기억장치에 저장된 정보는 컴퓨터가 꺼지면 자동으로 지워진다.

보조 기억장치 보조 기억장치는 정보를 저장하기 위해 사용되지만, 주 기억장치보다 속도가 느리다. 전기가 나갔을 때도 정보를 기억하는 것이 장점이다. 휴대용 USB 기억장치나 이동 MP3 플레이어에 사용되는 USB의 플래쉬 메모리나 디스크 드라이브가 여기에 속한다.

- **입출력장치(Input Output Devices):** 간단하게 화면, 키보드, 마우스, 마이크, 스피커, 터치패드가 포함된다. 컴퓨터와 사람이 상호작용하는 방식이다.
- **네트워크(Network):** 요즘 거의 모든 컴퓨터는 네트워크로 정보를 주고 받는 네트워크 커넥션(Network Connection) 하드웨어를 가지고 있다. 네트워크는 정보를 저장하는 느린 저장소로 혹은 때때로 원하는 정보를 가져오지 못하는 것으로 보조 기억장치(Secondary memory)로 생각할 수 있다.

어떻게 이러한 주요 부품들이 작동하는지에 대한 자세한 사항은 컴퓨터를 만드는 사람에게 달려있지만, 프로그램을 만들 때 컴퓨터 주요부품에 대해서 언급되어 컴퓨터 전문용어를 습득하고 이해하는 것은 도움이 된다.

프로그래머로서 여러분들은 사용자가 원하는 자료를 분석하고 문제를 풀 수 있는 컴퓨터 자원들을 사용하고 오케스트레이션하는 것이다.



프로그래머로 중앙처리장치(CPU)와 대화하며 ”다음은 무엇을 수행하세요”라고 지시할 것이다. 때때로 중앙처리장치(CPU)에게 주 기억장치, 보조 기억장치, 네트워크, 입출력 장치를 사용하라고 지시할 것이다.

프로그래머는 컴퓨터의 ”다음은 무엇을 수행할까요”에 대한 답을 하는 사람이기도 하다. 하지만, 컴퓨터에 답하기 위해서 5mm 크기로 컴퓨터에 프로그래머를 집어넣고 초당 30억개의 명령어로 답을 하는 것은 매우 불편할 것이다. 그래서, 미리 컴퓨터에게 수행할 명령문을 써놔야한다. 이렇게 미리 작성된 명령문의 집합을 **프로그램(Program)**이라고 하며, 명령어 집합을 작성하고 명령어 집합이 올바르게 작성될 수 있도록 하는 행위를 **프로그래밍(Programming)**이라고 부른다.

1.3 프로그래밍 이해하기

책의 나머지 장을 통해서 책을 읽고 있는 당신을 프로그래밍의 장인으로 인도할 것입니다. 중국에는 책을 읽고 있는 여러분은 **프로그래머**가 될 것입니다. 아마도 전문적인 프로그래머는 아닐지라도 적어도 자료/정보 분석 문제를 보고 그 문제를 풀수 있는 기술을 가지게는 될 것입니다.

이런 점에서 프로그래머가 되려면 두가지 기술이 필요로 합니다.

- 첫째, 파이썬같은 프로그래밍 언어 - 어휘와 문법을 알 필요가 있습니다. 단어를 새로운 언어에 맞추어 쓸 수 있어야 하며 새로운 언어를 잘 표현된 문장으로 구성하는지를 알아야 합니다.
- 둘째, 스토리(Story)를 말 할 수 있어야 합니다. 스토리를 만들 때, 독자에게 우리의 아이디어(idea)를 전달하기 위해서 단어와 문장을 조합합니다. 스토리를 만들 때 기술과 예술적인 면이 있으며, 기술과 예술적인 면은 여러번 쓰기 연습을 통하고 피드백을 받으므로써 향상됩니다. 프로그래밍에서, 우리가 만든 프로그램은 스토리이고, 풀려고 하는 문제는 ”아이디어”에 해당합니다.

파이썬과 같은 프로그래밍 언어를 배우게 되면, 자바스크립트나 C++ 같은 두 번째 언어를 배우는 것은 무척이나 쉽습니다. 새로운 프로그래밍 언어는 매우 다른 어휘와 문법을 가지지만, 문제푸는 기술을 배우기만 하면, 모든 프로그래밍 언어에서도 동일하게 접근할 수 있습니다.

파이썬의 어휘와 문단은 금방 배웁니다. 새로운 종류의 문제를 풀기 위해 논리적인 프로그램을 짜는 것은 오래 걸립니다. 여러분은 작문을 배우듯이 프로그래밍을 배우게 될 것입니다. 프로그래밍을 읽고 설명하는 것으로 시작해서 간단한 프로그램을 작성하고, 점차적으로 복잡한 프로그램을 작성하게 될 것입니다. 어느 순간에 명상에 잠기게 되고, 문제해결과 프로그램의 패턴을 보게되고, 좀더 자연스럽게 어떻게 문제를 받아들여 그 문제를 해결할 수 있는 프로그램을 작성하게 될 것입니다. 그리고, 그 순간에 도착하게 되면, 프로그래밍은 매우 즐겁고 창의적인 과정이 될 것입니다.

파이썬 프로그램의 어휘와 구조로 시작을 합니다. 간단한 예제가 언제 처음으로 프로그램을 읽기 시작했는지를 일깨워 주니 인내심을 가지세요.

1.4 단어와 문장

사람의 언어와 달리, 파이썬의 어휘는 실질적으로 매우 적다. 어휘를 예약어 (reserved words)로 부른다. 이들 단어는 파이썬에 매우 특별한 의미를 가진다. 파이썬 프로그램에서 파이썬이 이들 단어를 보게되면, 이들 단어는 파이썬에게 단 하나의 유일한 의미를 지니게 된다. 후에 여러분들이 프로그램을 작성할 때 여러분들이 만든 자신만의 단어를 작성하게 되는데 이를 **변수(Variable)**라고 합니다. 여러분의 변수의 이름을 지을 때 폭넓은 자유를 가질 수 있지만, 변수의 이름으로 파이썬의 예약어를 사용할 수는 없습니다.

이런 점에서 강아지를 훈련시킬 때 ”앉아”, ”기달려”, 가져와 같은 특별한 어휘를 사용합니다. 강아지에게 이런 특별한 예약어를 사용하지 않을 때, 강아지는 주인이 특별한 어휘를 사용하지 않을 때 주인을 물끄러미 쳐다보기만 합니다. 예를 들어, ”여러분이 더 많은 사람들이 건강을 전반적으로 향상하는 방향으로 가자 원한다”고 말하면, 강아지가 듣는 것은 ”뭐라 뭐라 뭐라 **walk** 뭐라” 이렇게 들릴 것이다. 왜냐하면 ”가자”가 강아지의 언어에는 예약어¹이기 때문이다. 이러한 사실은 개와 사람사이에는 예약어가 없다는 것을 의미할지도 모른다.

사람이 파이썬에게 말을 하는 예약어는 다음과 같은 것이 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	

¹<http://xkcd.com/231/>

강아지의 사례와는 다르게 파이썬은 이미 완벽하게 훈련이 되어 있다. 여러분이 “try”라고 말하면, 파이썬은 여러분이 매번 “try”라고 말할 때마다 실패 없이 시도를 할 것이다.

이러한 예약어를 배울 것이고 어떻게 잘 사용되는지도 함께 배울것이지만, 지금은 파이썬에 말하는 것에 집중할 것이다. 파이썬에게 말하는 것 중 좋은 것은 다음과 같은 메세지를 던지는 것으로도 파이썬에 말을 할 수 있다는 것이다.

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

이 간단한 문장이 파이썬의 구문(Syntax)론적으로 완벽하다. 위 문장은 예약어 ‘print’로 시작해서 출력하고자 하는 문자열을 작은 따옴표로 감싸안아 올바르게 파이썬에게 전달했다.

1.5 파이썬과 대화하기

파이썬으로 우리가 알고 있는 단어를 가지고 간단한 문장을 만들었고, 새로운 언어 기술을 시험하기 위해서 파이썬과 대화를 어떻게 하는지 알 필요가 있다.

파이썬과 대화를 시작하기 전에, 파이썬 소프트웨어를 컴퓨터에 설치하고 컴퓨터에서 파이썬을 어떻게 실행하는지를 배워야 한다. 이번장에서 다루기에는 너무 구체적이고 자세한 사항이기 때문에 [www.pythonguides.com](http://www.pythonguides.com/installing-python-on-mac-os-x/)을 참조하는 것을 권고한다. 자세한 설치 방법과 화면을 캡쳐하여 윈도우와 맥킨토쉬 시스템 및 실행하는 방법을 설명하였다. 설치가 마무리되고 터미널이나 명령어 실행창에서 **python**을 치게되면 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.
>>>
```

>>> 프롬프트는 파이썬 인터프리터가 여러분에게 요청하는 방식이다. ”다음에 파이썬이 무엇을 실행하기를 원합니까?” 파이썬은 여러분과 대화를 나눌 준비가 되었다. 이제 남은 것은 파이썬 언어로 어떻게 말하는 지를 여러분이 아는 것이고 여러분은 대화를 할 수 있다.

예를들어 여러분이 가장 간단한 파이썬 언어의 단어나 문장 조차도 알수가 없다고 가정해 봅시다. 우주 비행사가 저 멀리 떨어진 행성에 착륙할 때 사용하는 간단한 말을 사용하여 행성의 거주민에게 대화를 시도한다고 생각해 봅시다.

```
>>> 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 되는것 같지 않습니다. 뭔가 빨리 다른 생각을 하지 않는다면, 행성의 거주민은 창으로 찔르고, 기름에 잘 발라 불위에서 바베큐를 만들어 저녁으로 먹을 듯합니다.

운 좋게도 기나긴 우주 여행중 이책의 복사본을 가지고 와서 다음과 같이 빠르게 친다고 생각해봅시다.

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

훨씬 좋아보기고, 좀더 커뮤니케이션을 이어갈 수 있을 것으로 보입니다.

```
>>> 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 잠시동안 잘 진행되다가 여러분이 파이썬 언어로 말하다가 정말 사소한 실수를 저질러 파이썬이 오류를 뱉어낸다.

이번에 파이썬이 놀랍도록 복잡하고 강력하고 파이썬과 의사소통을 할때 사용하는 신택스(syntax)가 매우 까다롭다는 것은 알 수 있었다. 파이썬은 다른말로 똑똑(Intelligent)하지는 않다. 지금까지 여러분은 자신과 대화를 저절한 신택스(syntax)를 가지고 대화를 했습니다.

여러분이 다른사람이 작성한 프로그램을 사용한다는 것은 여러분과 파이썬을 사용하는 다른 프로그래머가 파이썬을 중간 매개체로 대화를 하는 것으로 볼 수 있습니다. 파이썬은 프로그램을 만든 저작자가 어떻게 대화가 진행되어져야 하는지를 표현하는 방식입니다. 다음 몇 장에 걸쳐서 여러분은 파이썬을 이용하여 여러분의 프로그램을 이용하는 다른 많은 프로그래머 중의 한명이 될 것입니다.

파이썬 인터프리터와 대화하는 첫장을 끝내기 전에, 파이썬 행성의 거주자에게 ”안녕히 계세요”를 말하는 적절한 방법을 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()
```

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

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

1.6 전문용어: 인터프리터와 컴파일러

파이썬은 사람이 읽고 쓸수 있고 컴퓨터도 읽고 쓸 수 있도록 고안된 **하이레벨(High-level)** 언어이다. 다른 하이레벨 언어는 자바, C++, PHP, 루비, 베이직, 펄, 자바스크립트 등 다수가 있다. 중앙처리장치(CPU)내에서 실제 하드웨어 수준에서 이런 하이레벨 언어를 이해하지 못한다.

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

```
01010001110100100101010000001111  
11100110000011101010010101101101  
...
```

0과 1로만 되어 있기 때문에 기계어가 간단해 보이지만, 시택스는 복잡하고 파이썬보다 훨씬 어렵다. 그래서 매우 소수의 프로그래머만이 기계어를 쓸수 있다. 대신에 파이썬과 자바스크립트 같은 하이레벨 언어로 프로그래머가 작성할 수 있도록 다양한 번역기(translator)를 만들었다. 이들 번역기는 프로그램을 중앙처리장치에 의해서 실제 실행이 가능한 기계어로 변환하여 준다.

기계어는 컴퓨터하드웨어에 묶여있기 때문에 기계어는 다른 형식의 하드웨어에 **이식(portable)**이 되지 않는다. 하이레벨 언어로 작성된 프로그램은 새로운 하드웨어위에 다른 인터프리터를 이용하여 옮겨 실행이 가능하고 다른 하드웨어에 사용할 수 있도록 프로그램을 다시 컴파일하여 사용할 수 있다.

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

인터프리터는 프로그래머에 의해서 쓰여진 소스코드를 읽고, 소스코드를 파싱하고, 즉석에서 명령어를 해석한다. 파이썬은 인터프리터다. 파이썬을 인터렉티브 모드로 실행할때, 파이썬 명령문을 쓰면, 파이썬이 즉성에서 처리하고, 다른 파이썬 명령어를 여러분으로부터 기다린다.

파이썬 명령어는 파이썬이 나중에 사용될 값을 기억하기를 바란다. 적당한 이름을 잡아서 그 값을 기억시키고, 나중에 그 이름을 호출하여 값을 사용할 수 있다. 이러한 목적으로 값을 저장하는 것을 **변수(variable)**라고 한다.

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

```
>>> print y  
42  
>>>
```

이 예제에서 파이썬이 `x`라는 라벨을 사용하여 6이라는 값을 저장하기를 바라고 나중에 사용코자 한다. `print` 예약어를 사용하여 파이썬이 잘 기억하고 있는지를 검증한다. 그리고 `x`를 반환하여 7을 곱하고 새로운 변수 `y`에 값을 집어 넣는다. 그리고 `y`에 현재 무엇이 저장되어 있는지 출력하라고 파이썬에게 요청한다.

파이썬에 한줄 한줄 명령어를 쳐 넣고 있지만, 파이썬은 앞쪽에 명령문에서 생성된 자료가 나중의 실행 명령문에서 사용될 수 있도록 정렬된 명령문으로 처리한다. 방금전 논리적이고 의미있는 순서로 4줄의 명령문을 가진 한 단락을 작성한 것이다.

위에서 본것처럼 파이썬과 대화를 주고받을 수 있는 것이 **인터프리터**의 본질이다. **컴파일러**는 완전한 프로그램을 하나의 파일에 담겨지고, 하이레벨 소스코드가 기계어로 번역되고, 컴파일러가 나중에 실행되도록 기계어를 파일에 담아놓는다.

윈도우를 사용한다면, 실행 가능한 기계어 프로그램의 확장자가 ”.exe”(executable), 혹은 ”.dll”(dynamically loadable library)임을 확인할 수 있다. 리눅스와 맥 키토쉬에서는 실행화일을 의미하는 확장자는 없다.

텍스트 편집기에서 실행파일을 열게되면 다음과 같은 읽을 수 없는 좀 괴상한 출력을 화면상에서 확인할 수 있다.

It is not easy to read or write machine language so it is nice that we have **interpreters** and **compilers** that allow us to write in a high-level language like Python or C.

Now at this point in our discussion of compilers and interpreters, you should be wondering a bit about the Python interpreter itself. What language is it written in? Is it written in a compiled language? When we type “python”, what exactly is happening?

The Python interpreter is written in a high level language called “C”. You can look at the actual source code for the Python interpreter by going to www.python.org and working your way to their source code. So Python is a program itself and it is compiled into machine code and when you installed Python on your computer (or the vendor installed it), you copied a machine-code copy of the translated Python program onto your system. In Windows the executable machine code for Python itself is likely in a file with a name like:

That is more than you really need to know to be a Python programmer, but sometimes it pays to answer those little nagging questions right at the beginning.

1.7 프로그램 작성하기

파이썬 인터프리터에 명령어를 치는 것은 파이썬의 주요기능을 알아볼 수 있는 좋은 방법이지만 좀더 복잡한 문제를 해결하기 위해서는 권해드리지는 않습니다.

프로그램을 작성할 때 **스크립트(script)**로 불리는 파일에 명령어 집합을 작성하기 위해서 텍스트 편집기를 주로 사용합니다. 파이썬 스크립트는 .py라는 확장자로 가집니다.

스크립트를 실행하기 위해서 파이썬 인터프리터에게 파일의 이름을 말해줍니다. 유니스나 윈도우 명령창에서 python hello.py를 치게 되면 다음과 같은 결과를 얻게 됩니다.

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

”csev\$”은 운영시스템의 명령어 프롬프트이고, ”cat hello.py”는 문자열을 출력하라는 한줄의 파이썬 프로그램을 표시하라는 명령어입니다.

인터랙트브 모드로 파이썬 코드를 보여달라는 방식 대신에 파이썬 인터프리터를 호출하고 ”hello.py” 파일에서 소스코드를 읽으라고 말하는 것입니다.

이 새로운 방식은 파이썬 프로그램을 끝내기 위해 **quit()**을 사용할 필요가 없는 것이 좋은 점입니다. 파이썬이 파일에서 소스코드를 읽을 때, 파일 끝까지 읽게되면 파이썬은 자동으로 끝낼 줄을 알고 있습니다.

1.8 프로그램이란 무엇인가?

프로그램(Program)의 가장 기본적인 정의는 어떠한 일을 할 수 있도록 조작된 일련의 파이썬 명령문의 집합이다. 가장 간단한 **hello.py** 스크립트도 프로그램이다. 한줄의 프로그램으로 특별히 유익하고 쓸모가 있는 것은 아니지만 엄격한 의미에서 파이썬 프로그램이 맞다.

프로그램을 이해하는 가장 쉬운 방법은 프로그램이 어떠한 문제를 풀려고 만들 어졌는지 문제를 먼저 생각하는 것이다. 그리고 그 문제를 풀려고 작성된 프로그램을 살펴보는 것이다.

예를 들어 여러분이 페이스북의 일련의 게시된 글에 가장 자주 사용된 단어에 관심을 가지고 소셜 컴퓨팅 연구를 한다고 생각해 봅시다. 페이스북에 게시된

글들을 출력하고 가장 흔한 단어를 찾으로 열심히 들여다 볼 것이지만 매우 오래 걸리고 실수하기 쉽습니다. 하지만 파이썬 프로그램을 이용하면, 빨리 정확하게 작업을 할 수 있는 파이썬 프로그램을 작성해서 주말에 뭔가 재미있는 일로 보낼 수 있습니다.

예를 들어 다음의 자동차(car)와 광대(clown)를 보고, 가장 많이 나오는 단어는 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, 몇백만줄의 텍스트를 보고서 동일한 일을 있다고 상상해 보세요. 솔직히 수작업으로 단어를 세는 것보다 파이썬을 배워 프로그램을 배우는 것이 훨씬 빠를 것입니다.

더 좋은 소식은 이미 텍스트 파일에서 가장 자주 나오는 단어를 찾아내는 간단한 프로그램을 제시했고, 작성했고, 시험까지 했다. 이걸 가지고 여러분들은 바로 사용을 할 수 있기 때문에 수고를 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
```

이 프로그램을 사용하려고 파이썬을 알 필요도 없다. 10장에 걸쳐서 멋진 파이썬 프로그램을 만드는 방법을 배우게 될 것입니다. 지금 여러분은 사용자로 단순히 프로그램을 사용하지만 이 프로그램이 얼마나 많은 수작업을 줄일 수 있는지 보여주고 있다. 코드를 작성하고 **words.py**로 저장하여 실행을 하거나, <http://www.pythonguides.com/code/>에서 소스코드를 다운로드 받아 실행하면 된다.

파이썬과 파이썬 언어가 중간의 중개자로서 여러분(사용자)과 저자(프로그래머)사이에서 중개자의 역할을 훌륭히 하고 있는 것을 보여주고 있습니다. 파이썬이 설치된 컴퓨터에서 누구나 사용할 수 있는 공통의 언어로 유용한 명령 순서(즉, 프로그램)를 주고받는지 보여준다. 그래서 파이썬과 직접 의사소통하지 않고 파이썬을 통해서 서로 의사소통할 수 있다.

1.9 프로그램 구성요소

다음의 몇장에 걸쳐서 파이썬 어휘, 문장구조, 문단구조에 대해서 배울 것이다. 파이썬의 강력한 점에 대해서 배울 것이고, 유용한 프로그램을 작성하기 위해서 파이썬의 역량을 조합하는 법을 배울 것이다.

프로그램을 작성하기 위해서 사용하는 로우레벨(low-level) 패턴이 몇가지 있다. 파이썬을 위해서 만들어졌다기 보다는 기계어부터 하이레벨 언어에 이르기까지 모든 언어에 공통된 사항이다.

입력: 컴퓨터 바깥세계에서 데이터를 가져온다. 파일로부터 데이터를 읽을 수도 있고, 마이크나 GPS 같은 센서에서 데이터를 입력받을 수도 있다. 위 프로그램에서 사용자의 키보드로 데이터를 입력받아 입력값으로 사용된 사례이다.

출력: 화면에 프로그램의 결과값을 보여주거나 파일에 저장한다. 혹은 음악을 연주하거나 텍스트를 읽도록 스피커 같은 장치에 데이터를 보낸다.

순차 실행: 스크립트에 작성된 순서에 맞추어 한줄 한줄 실행된다.

조건 실행: 조건을 확인하고 명령문을 실행하거나 건너뛴다.

반복 실행: 반복적으로 명령문을 실행한다. 대체로 반복 실행시 변화를 수반 한다.

재사용: 명령어를 한번 작성하고 이름을 주어 저장하고 프로그램의 필요에 따라 이름을 불러 몇차례 다시 사용한다.

너무나 간단하게 들리지만, 전혀 간단하지는 않다. 걸음거리를 간단히 ”한 다리를 다를 다리 앞에 놓으세요”라고 말하는 것 같다. 프로그램을 짜는 ”예술”은 이러한 기본 요소를 조합하고 엮어 사용자에게 유용한 무언가를 만드는 것이다.

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

1.10 프로그램이 잘못되면?

파이썬과 처음에 대화를 할때, 파이썬 코드를 명확하게 작성하여 의사소통을 해야한다. 작은 차이 혹은 실수는 파이썬이 여러분이 작성한 프로그램을 보다가 포기하게 만듭니다.

초보 파이썬 프로그래머는 파이썬이 오류에 대해서는 인정사정볼 것 없다고 생각합니다. 파이썬이 모든 사람을 좋아하는 것 같지만, 파이썬은 사적으로 사람들을 알고 있고, 뒤끝이 있습니다. 이러한 사실로 인해서 파이썬은 완벽하게 작성된 프로그램만을 가지게 되고 ”잘못 작성되었어요”라고 뱉어내고 여러분에게 고통을 줍니다.

```

>>> print 'Hello world!'
      File "<stdin>", line 1
          print 'Hello world!'
                  ^
SyntaxError: invalid syntax
>>> print 'Hello world'
      File "<stdin>", line 1
          print '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 다뤄봐야 얻을 것은 없어요. 파이썬은 도구고 감정이 없습니다. 여러분이 필요로 할 때마다 여러분에게 봉사하고 기쁨을 주기 위해서 존재할 뿐입니다. 오류 메세지가 심하게 들릴지는 모르지만 단지 파이썬이 도와달라고 하는 요청일 뿐입니다. 여러분이 입력한 것을 쭉 읽어보고 여러분이 입력한 것을 이해할 수 없다고만 말할 뿐입니다.

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

여러분이 작성하는 프로그램이 점점 유용해지고 복잡해짐에 따라 3가지 유형의 오류를 마주치게 된다.

구문 오류(Syntax Error): 첫번째 마주치는 오류로 고치기 가장 쉽습니다. 구문 오류는 파이썬의 문법에 맞지 않는다는 것을 의미합니다. 파이썬은 구문 오류가 발생한 줄을 찾아 정확한 위치를 알려줍니다. 하지만, 파이썬이 제시하는 오류가 그 이전 프로그램 부문에서 발생했을 수도 있기 때문에 파이썬이 알려주는 곳 뿐만 아니라 그 앞쪽도 살펴볼 필요가 있다. 따라서 파이썬이 제시하는 구문 오류의 경우 오류가 난 곳은 오류를 고치기 위한 시작점으로 의미가 있다.

논리 오류(Logic Error): 논리 오류는 프로그램의 구문은 완벽하지만 명령문의 실행에 혹은 다른 명령어부분과 관련된 부문에서 실수가 있는 것이다. 논리 오류의 예를 들어보자. "물병에서 한모금 마시고, 가방에 넣고, 도서관으로 걸어가서, 물병을 닫는다"

시맨틱 오류(Semantic Error): 시맨틱 오류는 구문론적으로 완벽하고 올바른 순서로 프로그램의 명령문이 작성되었지만 프로그램에 오류가 숨어있다.

프로그램은 완벽하게 작동하지만 여러분이 의도한 바를 수행하지는 못합니다. 간단한 예로 여러분이 식당으로 가는 방향을 알려주고 있습니다.” “… 주유소 사거리에 도착했을 때, 왼쪽으로 돌아 1.6km 쭉 가면 왼쪽편에 빨간색 빌딩에 식당이 있습니다.” 친구가 매우 늦어 전화로 지금 농장에 있고 혀간으로 걸어가고 있는데 식당을 발견할 수 없다고 전화를 합니다. 그러면 여러분은 ”주유소 왼쪽 혹은 오른쪽을 돋거야” 말하면, 그 친구는 ”말한대로 완벽하게 따라서 했고, 말한대로 적기까지 했는데, 왼쪽으로 돌아 1.6km에 주요소가 있다고 했어”, 그러면 여러분은 ”미안해, 내가 가지고 있는 건 구문론적으로는 완벽한데, 사소한 시맨틱 오류가 있네!”라고 말할 것이다.

위 세 종류의 오류에 대해서 파이썬은 여러분이 요청한 것을 충실히 수행하기 위해서 최선을 다합니다.

1.11 학습으로의 여정

여러분이 책을 읽으면서 개념들이 처음에 잘 와 닿지 않는다고 기죽을 필요는 없어요. 어렵누이 말하는 것을 배울 때, 처음 몇년동안 웅얼거리는 것은 문제가 아닙니다. 간단한 어휘에서 간단한 문장으로 옮겨가고, 문장에서 문단으로 옮겨가는데 6개월이 걸려도 괜찮습니다. 흥미로운 완벽한 짧은 스토리를 자신의 언어로 작성하는데 몇 년이 걸립니다.

여러분이 파이썬을 빨리 배울수 있도록 다음의 몇장에 걸쳐서 정보를 제공해 드릴 것입니다. 새로운 언어를 습득하는 것과 같아서 자연스럽게 느껴지기까지 흡수하고 이해하기까지 시간이 걸립니다. 큰 그림(Big Picture)을 이루는 작은 조각을 정의하면서 여러분을 큰 그림을 볼 수 있도록 여러 주제를 찾고, 또 찾으면서 혼란이 생길 수 있다. 이책이 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 배운것을 다시 리뷰하고 앞의 연습 문제를 다시 하면서 처음에 난공불락이라 여겼던 어려운 주제도 더 잘 배우고 이해할 수 있다는 것을 알게될 것이다.

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

만약 어떤 것이 특별히 힘들다면, 밤새도록 앉아서 지켜보고 노력하는 것은 별로 의미가 없다. 잠시 쉬고, 낮잠을 자고, 간식을 먹고 다른사람이나 강아지에게 문제를 설명하고 자문을 구한 후에 깨끗한 정신과 눈으로 돌아와서 다시 시도해 보라. 단연컨데 이책의 프로그래밍 개념을 배우자마자 정말 쉽고 멋지다는 것을 돌이켜 보면 알게될 것이다. 프로그래밍 언어는 정말 배울 가치가 있다.

1.12 용어사전

버그(bug): 프로그램 오류

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

컴파일러(compile): 하이레벨 언어로 작성된 프로그램을 로우레벨 언어로 즉시 혹은 나중에 사용하도록 번역하는 번역기

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

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

인터프리트(interpret): 하이레벨 언어의 프로그램을 한번에 한줄씩 번형해서 실행하는 것

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

기계어 코드(machine code): 중앙처리장치에 의해서 바로 실행될수 있는 가장 낮은 수준의 언어로된 소프트웨어

주메모리(main memory): 프로그램과 데이터를 저장한다. 전기가 나가게 되면 주메모리에 저장된 정보는 사라진다.

파싱(parse): 프로그램을 검사하고 구문론적 구조를 분석하는 것

이식(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:) 컴퓨터이션(Computation)을 명세하는 명령어의 집합

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

보조 기억장치 전기가 나갔을 때도 정보를 기억하고 프로그램을 저장하는 저장소. 일반적으로 주메모리보다 속도가 느린다. USB의 플래쉬 메모리나 디스크 드라이브가 여기에 속한다.

시맨틱(semantics): 프로그램의 의미

시맨틱 오류(semantic error): 프로그래머가 의도한 것과 다른 행동을 하는 프로그램의 오류

소스코드(source code): 하이레벨 언어로 기술된 프로그램

1.13 연습문

Exercise 1.1 컴퓨터 보조기억장치의 기능은 무엇입니까?

- a) 모든 연산과 프로그램의 로직을 수행한다.
- b) 인터넷의 웹페이지를 불러온다.
- c) 파워가 없을 때도 장시간 정보를 저장한다.
- d) 사용자로부터 입력정보를 받는다.

Exercise 1.2 프로그램은 무엇입니까?

Exercise 1.3 컴파일러와 인터프리터의 차이점을 설명하세요.

Exercise 1.4 기계어 코드는 다음 중 어느 것입니까?

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

Exercise 1.5 다음 코드에서 잘못된 점을 설명하세요.

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

Exercise 1.6 다음의 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 능력부위에 해당하는지 예로하여 다음을 설명하세요. (1) 중앙처리장치, (2) 주메모리, (3) 보조메모리, (4) 입력장치 (5) 출력장치 중앙처리장치에 상응하는 사람의 몸 부위는 어디입니까?

Exercise 1.9 구문오류("Syntax Error")는 어떻게 고칩니다?

Chapter 2

변수, 표현식, 스테이트먼트 (Statement)

2.1 값(Value)과 형식(Type)

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

이들 값들은 다른 형식에 속해 있는데, 2는 정수, 'Hello, World!' ('안녕 세상!') 는 문자열(String)에 속해 있다. 문자(Letter)의 열에 있어서 문자열이라고 부른다. 여러분과 인터프리터는 문자열을 확인할 수 있는데 이유는 인용부호 따옴표에 쌓여 있기 때문이다.

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

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

값의 형식을 확신을 못한다면, 인터프리터가 알려준다.

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

놀랍지도 않게, strings은 str 형식이고, 정수는 int 형식이다. 소수점을 가진 숫자는 float 형식이다. 왜냐하면 이들 숫자가 부동소수점 형식으로 표현되기 때문이다.

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

'17', '3.2' 같은 값은 어떨까? 문자처럼 보이지만 문자처럼 따옴표안에 쌓여 있다.

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

'17', '3.2' 은 문자열이다.

아주 큰 정수를 입력할, 1,000,000 처럼 세자리 숫자마다 콤마(,)를 입력한다. 하지만, 파이썬에서 적법한 정수는 아니지만 적법하다.

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

하지만, 파이썬이 뺄은 값은 우리가 기대했던 것이 아니다. 파이썬은 1,000,000 을 콤마로 구분된 정수로 인식한다. 따라서 사이 사이 공백을 넣어 출력했다.

이 사례가 여러분이 처음 경험하게 되는 시맨틱 오류이다. 코드가 예리 메세지 없이 실행이되지만, ”올바르게(right)” 작동을 하는 것은 아니다.

2.2 변수(Variable)

프로그래밍 언어의 가장 강력한 기능중의 하나는 변수를 다룰 수 있는 능력이다. **변수(Variable)**는 값을 참조할 수 있는 이름이다.

할당 스테이스먼트(Assignment statement)는 새로운 변수를 생성하고 값을 준다.

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

위의 예제는 세개의 할당을 보여준다. 첫번째 할당의 예제는 message 변수에 문자열을 할당한다. 두번째 예제는 변수 n에 정수 17을 할당한다. 세번째 예제는 pi 변수에 π 근사값을 할당하는 경우이다.

변수의 값을 출력하기 위해서 print 스테이트먼트를 사용한다.

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

변수의 형식은 변수가 참조하는 값의 형식이다.

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

2.3 변수명(Variable name)과 예약어(keywords)

대체로 프로그래머는 의미 있는 변수명을 고른다. 프로그래머는 변수가 어디에 사용되는지를 문서화도 한다.

변수명은 임의로 길 수 있다. 변수명은 문자와 숫자를 포함할 수 있지만, 통상 문자로 시작한다. 대문자를 사용하는 것도 적합하지만 소문자로 시작하는 변수명으로 시작하는 것도 좋은 생각이다. (후에 왜 그런지 보게될 것이다.)

변수명에 밑줄(underscore character, _)이 들어갈 수 있다. 밑줄은 `my_name` 혹은 `airspeed_of_unladen_swallow`처럼 여러 단어와 함께 사용된다.

적합하지 못한 변수명을 사용하면, 구문 오류를 보게된다.

```
>>> 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@`는 특수 문자 (@)를 변수명에 포함해서 적합하지 않다. `class` 변수명은 뭐가 잘못된 것일까?

`class`는 파이썬의 예약어 중의 하나이다. 인터프리터는 예약어를 프로그램 구조를 파악하기 위해서 사용하고 변수명으로는 사용할 수 없다.

파이썬의 31개의 키워드¹를 예약어로 이미 가지고 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	

여러분은 예약어 목록을 잘 가지고 다니고 싶을 것입니다. 인터프리터가 변수명 중에서 불평하고 이유를 모를 경우 예약어 목록에 있는지 확인해 보세요.

2.4 스테이트먼트(Statement)

스테이트먼트(statement)는 파이썬 인터프리터가 실행하는 코드의 단위입니다. `print`, `assignment` 두 종류의 스테이트먼트를 봤습니다.

인터랙트브 모드에서 스테이트먼트를 입력할 때, 만약 한줄이면 인터프리터는 스테이트먼트를 실행하고 결과를 출력합니다.

¹In Python 3.0, `exec` is no longer a keyword, but `nonlocal` is.

스크립트는 보통 여러줄의 스테이트먼트로 구성됩니다. 하나 이상의 스테이트먼트가 있다면, 스크립트가 실행되며 결과가 한번에 나타납니다.

예를 들어, 다음의 스크립트를 생각해 봅시다.

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

위 스크립트는 다음의 결과를 출력합니다.

```
1
2
```

할당 스테이트먼트 ($x=2$)는 결과를 출력하지 않습니다.

2.5 연산자(Operator)와 피연산자(Operands)

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

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

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

나눗셈 연산자는 여러분이 기대하는 행동을 하지 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 부동 소수점 나눗셈을 하기 때문입니다.

두 개의 피연산자가 정수이면, 결과도 정수입니다. **부동 소수점 나눗셈²** 은 소수점 이하를 절사해서 이 예제에서는 소수점이하 잘라버려 0 이 됩니다.

```
>>> minute/60.0
0.9833333333333328
```

2.6 (표현)식(Expression)

(표현)식 (expression)은 값, 변수, 연산자의 조합입니다. 값은 자체로 표현식이고, 변수도 동일합니다. 따라서 다음의 표현식은 모두 적합합니다. (변수 x에 사전에 어떤 값이 할당되었다고 가정하고)

²파이썬 3.0에서 이 나눗셈의 값은 소수점입니다. 파이썬 3.0에서 새로운 연산자 //는 정수 나눗셈을 수행합니다.

```
17
x
x + 17
```

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

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

하지만, 스크립트에서는 표현식 자체로 어떠한 것도 수행하지는 않습니다. 초심자에게 혼란스러운 점입니다.

Exercise 2.1 파이썬 인터프리터에 다음의 스테이트먼트를 입력하고 결과를 보세요.

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

2.7 연산자 적용 우선순위 (Order of Operations)

1개 이상의 연산자가 표현식에 등장할 때 연산자 실행의 순서는 순위 규칙(rules of precedence)에 따른다. 수학 연산자에 대해서 파이썬은 수학의 관례를 동일하게 따른다. 영어 두문어 PEMDAS는 기억하기 좋은 방식이다.

- **괄호(Parentheses)**는 가장 높은 순위를 가지고 여러분이 원하는 순위에 맞춰 실행할 때 사용한다. 괄호안에 있는 식이 먼저 실행되기 때문에 $2 * (3-1)$ 은 4가 정답이고, $(1+1)**(5-2)$ 는 8이다. 괄호를 표현식을 읽기 쉽게 하려고 사용하기도 한다. $(\text{minute} * 100) / 60$ 는 실행순서가 결과값에 영향을 주지 않지만 가독성이 상대적으로 좋다.
- **멱승(Exponentiation)**이 다음으로 높은 우선순위를 가진다. 그래서 $2**1+1$ 은 4가 아니라 3이고, $3*1**3$ 은 27이 아니고 3이다.
- **곱셈(Multiplication)**과 **나눗셈(Division)**은 덧셈(Addition), 뺄셈(Subtraction)보다 높은 우선 순위를 가지고 같은 실행의 순위를 갖는다. $2*3-1$ 은 4가 아니고 5이고, $6+4/2$ 는 5가 아니라 8이다.
- 같은 실행의 순위를 갖는 연산자는 왼쪽에서 오른쪽으로 실행된다. $5-3-1$ 표현식은 3이 아니고 1이다. 왜냐하면 5-3이 먼저 실행되고 나서 2에서 1을 빼기 때문이다.

여러분이 의도한 순서대로 연산이 일어날 수 있도록 좀 의심스러운 경우는 괄호를 사용하세요.

2.8 나머지 연산자 (Modulus Operator)

나머지 연산자(**modulus operator**)는 정수에 사용하며, 첫번째 피연산자가 두번 째 피연산자로 나눌 때 나머지 값을 만들어 냅니다. 파이썬에서 나머지 연산자는 퍼센트 기호(%)입니다. 구문은 다른 연산자와 동일합니다.

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

7을 3으로 나누면 2가 되고 1을 나머지로 갖게됩니다.

나머지 연산자가 놀랍도록 유용합니다. 예를 들어 한 숫자가 다른 숫자로 나눌 수 있는지 없는지를 확인할 수도 있습니다. $x \% y$ 가 0이라면 x 는 y 로 나눌 수 있습니다.

또한, 숫자로부터 가장 오른쪽의 숫자를 분리하는데 사용할 수도 있습니다. 예를 들어 $x \% 10$ 은 x 의 10진수인 경우 가장 오른쪽 숫자를 뽑아낼 수 있고, 동일하게 $x \% 100$ 은 가장 오른쪽 2개 숫자를 뽑아낼 수도 있습니다.

2.9 문자열 연산자 (String Operator)

+ 연산자는 문자열에도 사용할 수 있지만, 수학에서의 덧셈의 의미는 아닙니다. 대신에 문자열의 끝과 끝을 연결하여 **접합(concatenation)**을 수행합니다. 예를 들어

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

이 프로그램의 출력은 100150입니다.

2.10 사용자에게서 입력값 받기

때때로 키보드를 통해서 사용자에게서 변수에 대한 값을 받고 싶을 때가 있습니다. 키보드³로부터 입력값을 받을 수 있는 `raw_input`이라는 빌트인(built-in) 함수를 파이썬은 제공합니다. 이 함수가 실행될 때 파이썬은 실행을 멈추고 사용자로부터 입력 받기를 기다립니다. 사용자가 Return 혹은 엔터를 누르게되면

³파이썬 3.0에서 이 함수는 `input`으로 명명되었습니다.

프로그램은 다시 실행되고 raw_input은 사용자가 입력한 값을 문자열로 반환합니다.

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

사용자로부터 입력을 받기 전에 프롬프트에서 사용자에게 어떤 값을 입력해야하는지 출력하는 것도 좋은 생각이다. 입력값을 위해 잠시 멈춰있기 전에 raw_input 함수에 출력될 문자열에 대한 정보를 사용자에게 전달하는 것이다.

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

프롬프트의 끝에 \n 은 새줄(newline)을 의미합니다. 새줄은 줄을 바꾸게 하는 특수 문자입니다. 이런 이유 때문에 사용자의 입력이 프롬프트 밑에 출력이 됩니다.

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

```
>>> 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 발생합니다.

```
>>> 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()
```

이런 종류의 오류를 나중에 더 만나게 될 것입니다.

2.11 주석(Comment)

프로그램이 커지고 복잡해짐에 따라 읽기는 점점 어려워집니다. 형식언어로 촘촘하고 코드의 일부분을 읽기 어렵고 무슨 역할을 왜 수행하는지 이해하기 어렵습니다.

이런 이유로 프로그램이 무엇을 하는지를 일반적인 언어로 프로그램에 노트를 달아놓는게 좋은 습관입니다. 이러한 노트를 주석(**Comments**)라고 하고 # 기호로 시작합니다.

```
# 경과한 시간의 퍼센트를 계산
percentage = (minute * 100) / 60
```

이 경우, 주석 자체가 한줄이다. 주석을 프로그램의 뒤에 놓을 수도 있다.

```
percentage = (minute * 100) / 60      # 경과한 시간의 퍼센트를 계산
```

뒤의 모든 것은 무시되기 때문에 프로그램에는 아무런 영향이 없습니다. 주석은 코드의 명확하지 않은 점을 문서화할 때 가장 유용합니다. 프로그램을 읽는 사람이 코드가 무엇을 하는지 이해할 수 있다고 가정하는 것은 그럴 듯합니다. 왜 그런지를 설명하는 것은 더더욱 유용합니다.

다음의 주석은 코드와 중복으로 쓸모가 없습니다.

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

다음의 주석은 코드에 없는 유용한 정보가 있습니다.

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

좋은 변수명은 주석을 할 필요를 없게 만들지만, 지나치게 긴 변수명은 읽기 어려운 복잡한 표현식이 될 수도 있기 때문에 상충관계(trade-off)가 존재합니다.

2.12 기억하기 쉬운 변수명 만들기

변수를 이름 짓는 간단한 규칙을 따르고, 예약어를 피하기만 하면 변수를 이름지 을 수 있는 무척이나 많은 경우의 수가 존재합니다. 처음에 이러한 선택의 폭이 프로그램을 읽는 사람이나 프로그램을 작성하는 사람 모두에게 혼란스러울 수 있습니다. 예를 들어, 다음의 3개 프로그램은 수행하는 것이 동일하다는 점에서 동일하지만 여러분이 읽고 이해하는데는 많은 차이점이 있습니다.

```
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 이들 3개 프로그램을 정확하게 동일하게 바라보지만, 사람은 이들 프로그램을 매우 다르게 바라보고 이해하게 됩니다. 사람은 가장 빨리

두번째 프로그램의 의도를 알아차립니다. 왜냐하면 프로그래머가 변수에 저장되는 값에 관계없이 프로그래머의 의도를 나타내는 변수명을 사용했기 때문입니다.

현명하게 선택된 변수명을 기억하기 쉬운 변수명(*"mnemonic variable name"*)이라고 합니다. 기억하기 좋은 영어 단어 "mnemonic"⁴은 기억을 돋는다는 뜻입니다. 왜 변수를 생성했는지 기억하기 좋게 하기 위해서 기억하기 좋은 변수명을 선택합니다.

매우 훌륭하게 들리고, 기억하기 좋은 변수명을 만드는게 좋은 아이디어 같지만, 기억하기 좋은 변수명은 초보 프로그래머가 코드를 파싱하고 이해하는데 걸림돌이 되기도 한다. 왜냐하면 31개의 예약어도 기억하지 못하고 때때로 너무 서술적인 이름의 변수가 마치 우리가 일반적으로 사용하는 언어처럼 보여 잘 선택된 변수명처럼 보이지 않기 때문이다.

어떤 데이터를 반복하는 다음의 파이썬 코드를 살펴보자. 여러분은 곧 반복 루프를 보지만 이것이 무엇을 의미하는지 알기 위해서 퍼즐을 풀기 시작할 것이다.

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

무엇이 일어나고 있는 것일까요? for, word, in 등등 어느 것이 예약어일까요? 변수명은 무엇일까요? 파이썬이 기본적으로 단어의 개념을 이해할까요? 초보 프로그래머는 어떤 부분의 코드가 이 예제와 동일해야 하는지와 단지 프로그래머의 선택에 의한 부분이 코드의 어느부분인지 분간하는데 애를 먹는다.

다음의 코드는 위의 코드와 동일하다.

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

초보 프로그래머가 이 코드를 보고 어떤 부분이 파이썬의 예약어이고 어느 부분이 프로그래머가 선택한 변수명인지 알 수 있다. 파이썬이 피자와 피자조각에 대한 근본적인 이해가 없고 피자는 하나 혹은 여러 조각으로 구성된다는 근본적인 사실을 알지 못한다.

하지만, 여러분의 프로그램이 데이터를 읽고 데이터의 단어를 찾는다면 피자(pizza)와 피자조각(slice)은 기억하기 좋은 변수명이 아니다. 이를 변수명은 프로그램의 의미를 왜곡시킬 수 있다.

좀 시간을 보낸 후에 흔한 예약어에 대해서 알게될 것이고 이들 예약어가 여러분에게 눈에 띄게 될 것이다.

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

for, in, print, :은 파이썬에서 정의된 예약어로 굵게 표시되어 있고, 프로그래머가 생성한 word, words는 굵게 표시되어 있지 않다. 많은 텍스트 에디터는 파이썬의 구문을 알고 있고 파이썬 예약어와 프로그래머의 변수를 구분하기 위해서 다른 색깔로 구분지어준다. 잠시 후에 여러분은 이제 파이썬을 읽고 변수와 예약어에 대해서 빨리 구분할 수 있을 것이다.

⁴<http://en.wikipedia.org/wiki/Mnemonic>.

2.13 디버깅(Debugging)

이 지점에서 여러분이 만들 것 같은 구문 오류는 `odd~ job`, `US$` 같은 특수문자 를 포함한 잘못된 변수명과 `class`, `yield` 같은 예약어를 변수명으로 사용하는 것이다.

변수명에 공백을 넣는다면, 파이썬은 연산자 없이 두 개의 피연산자로 생각합니다.

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

구문 오류에 대해서, 구문오류 메세지는 그다지 도움이 되지 못합니다. 가장 흔한 오류 메세지는 `SyntaxError: invalid syntax`, `SyntaxError: invalid token`인데 둘다 그다지 도움이 되지 못합니다.

여러분이 많이 만드는 실행 오류는 정의전에 사용("use before def")하는 것으로 변수에 값을 할당하기 전에 변수를 사용할 경우 발생합니다. 여러분이 변수명을 쓸 때도 발생할 수 있습니다.

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

변수명은 대소문자를 구분합니다. `LaTeX`는 `latex`와 같지 않습니다.

이 지점에서 여러분이 저지르기 쉬운 구문 오류는 연산자의 우선 순위일 것입니다. 예를 들어 $\frac{1}{2\pi}$ 를 계산하기 위해서 다음과 같이 프로그램을 작성하게 되면

...

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

나눗셈이 먼저 일어나서 $\pi/2$ 를 같은 것이 아닙니다. 파이썬이 여러분이 쓴 의도를 알게할 수는 없습니다. 그래서 이런 경우 오류 메세지를 얻지는 않지만 잘못된 답을 여러분은 얻게 될 것입니다.

2.14 용어 설명

할당(assignment): 변수에 값을 할당하는 스테이트먼트

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

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

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

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

부동 소수점(floating-point): 분수를 가진 숫자를 표현하는 방식

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

정수(integer): 완전수를 나타내는 형식

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

니모닉(mnemonic): 기억 보조, 변수에 저장된 것을 기억하기 좋게 변수에 니모닉 이름을 부여한다.

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

피연산자(operand): 연산자가 연산을 수행하는 값의 하나

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

순위 규칙(rules of precedence): 여러 개의 연산자와 피연산자를 포함한 표현식이 평가되는 실행 순서를 정한 규칙 집합

스테이트먼트(statement): 명령이나 액션을 나타내는 코드 부문. 지금까지 assignment, print 스테이트먼트를 보았습니다.

문자열(string): 일련의 문자를 나타내는 형식

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

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

변수(variable): 값을 참조하는 이름

2.15 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 원한다면, 파이썬의 빌트인 `round` 함수를 사용하여 소수점 아래 두자리까지 반올림하여 작성할 수 있다.

Exercise 2.4 다음 할당 스테이트먼트를 실행한다고 합시다.

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

다음 표현식에 대해서 표현식의 값과 표현식의 값의 형을 작성하세요.

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

정답을 확인하기 위해서 파이썬 인터프리터를 사용하세요.

Exercise 2.5 사용자에게서 섭시 온도를 입력받아 화씨온도를 변환하고 변환된 온도를 출력하는 프로그램을 작성하세요.

Chapter 3

조건부 실행

3.1 불 연산식(Boolean expressions)

불 연산식(boolean expression)은 참(True) 혹은 거짓(False)를 가진 연산 표현식이다. 다음 예제는 == 연산자를 사용하는데 두개의 피연산자를 비교하여 값이 동일하면 참(True), 그렇지 않으면 거짓(False)을 출력한다.

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

참(True)과 거짓(False)은 불(bool) 형식에 속하는 특별한 값들이고, 문자열은 아니다.

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

==연산자는 비교(comparison operators) 연산자 중의 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.

여러분에게 이들 연산자가 친숙할지 모르지만, 파이썬 기호는 수학 기호와는 다르다. 일반적인 오류에는 비교의 같다의 의미로 == 연산자 대신에 =를 사용하는 것이다. = 연산자는 할당 연산자이고, ==연산자는 비교 연산자이다. =<, => 비교 연산자는 파이썬에는 없다.

3.2 논리 연산자

3개의 논리 연산자(logical operators): and, or, not가 있다. 논리 연산자 의미는 영어 의미와 유사하다. 예를 들어,

`x > 0 and x < 10`

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

`n%2 == 0 or n%3 == 0`은 두 조건문 중의 하나만 참이되면, 즉, 숫자가 2 혹은(or) 3으로 나누어진다면 참이다.

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

엄밀히 말해서, 논리 연산자의 두 피연산자는 모두 불 연산 표현이여야 하지만, 파이썬은 그렇게 엄격하지는 않는다. 어떤 0이 아닌 숫자 모두 ”true”로 해석된다.

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

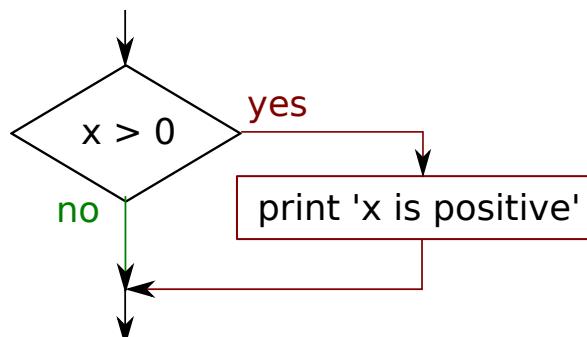
이러한 유연함이 유용할 수 있으나, 혼란을 줄 수도 있으니 유의해서 사용해야 됩니다. 무슨 일을 하고 있는지 정확하게 알지 못한다면 피하는게 좋습니다.

3.3 조건문 실행

유용한 프로그램을 작성하기 위해서는 조건을 확인하고 조건에 따라 프로그램의 실행을 바꿀 수 있어야 한다. 조건문(Conditional statements)은 그럴 수 있는 능력을 부여한다. 가장 간단한 형태는 if 문이다.

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

if문 뒤에 불 연산 표현문을 조건(condition)이라고 한다.



만약 조건문이 참이면, 들여쓰기 된 스테이트먼트가 실행된다. 만약 조건문이 거짓이면, 들여쓰기 된 스테이트먼트의 실행을 생략한다.

if문은 함수 정의나 for 반복문과 동일한 구조를 가진다. if문은 콜론(:)으로 끝나는 헤더 머리부문과 들여쓰기 블록으로 구성된다. if문과 같은 구문을 한 줄 이상에 걸쳐 작성되기 때문에 **복합문(compound statements)**이라고 한다.

if문 본문에 실행 명령문의 제한은 없으나 최소한 한 줄은 있어야 한다. 때때로, 본문에 하나의 실행명령문이 없는 경우가 있다. 아직 코드를 작성하지 않고 자리를 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 바뀌어 본문을 작성중에 있다는 것을 다음과 같이 보여줍니다.

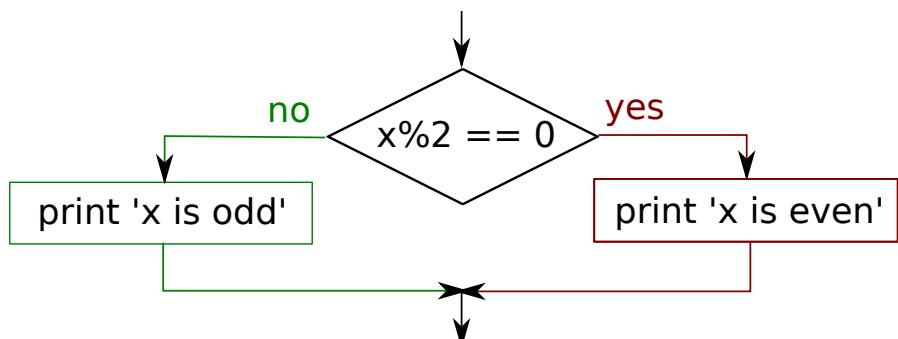
```
>>> x = 3  
>>> if x < 10:  
...     print 'Small'  
...  
Small  
>>>
```

3.4 대안 실행

if문의 두 번째 형태는 **대안 실행(alternative execution)**으로 두 가지 경우의 수가 존재하고, 조건이 어느 방향인지를 결정한다. 구문(Syntax)은 아래와 같다.

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

x가 2로 나누었을 때, 0이 되면, x는 짝수이고, 프로그램은 짝수 결과 메시지를 출력한다. 만약 조건이 거짓이라면, 두 번째 명령문 블록이 실행된다.



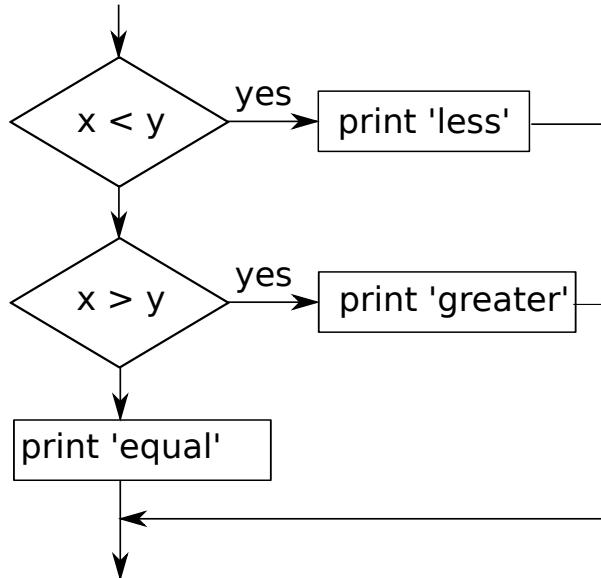
조건은 참 혹은 거짓이서, 대안 중 하나만 정확하게 실행될 것이다. 대안을 **분기(Branch)**라고도 하는데 이유는 실행 흐름의 분기가 되기 때문이다.

3.5 연쇄 조건문

때때로, 두 가지 이상의 경우의 수가 있으며, 두 가지 이상의 분기가 필요하다. 이 같은 연산을 표현하는 방법이 **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'
```

`elif`는 ”`else if`”의 축약어이다. 이번에도 단 한번의 분기만 실행된다.



`elif`문의 갯수에 제한은 없다. `else`문이 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 주에 하나의 조건이 참이면, 해당 분기가 수행되고, `if`문 전체는 종료된다. 설사 하나 이상의 조건이 참이라고 하더라도, 첫 번째 참 분기만 수행된다.

3.6 중첩 조건문

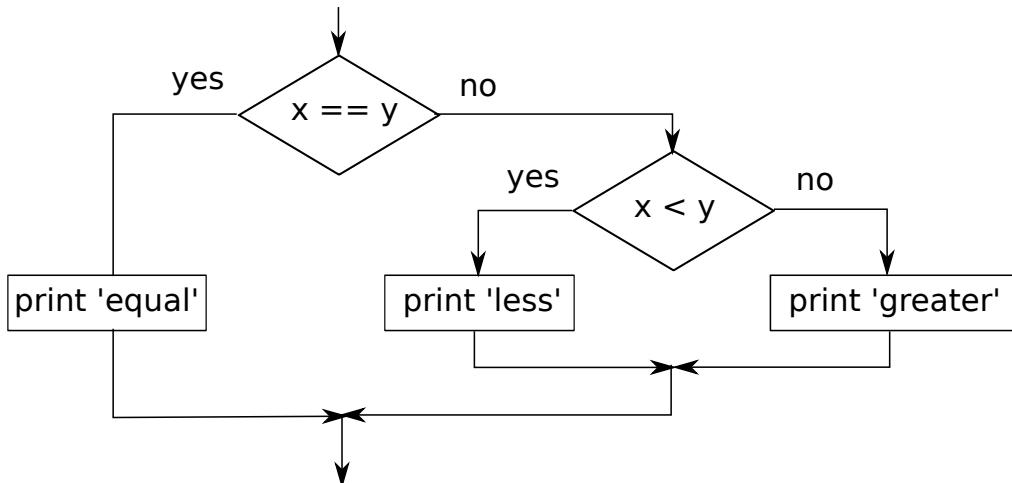
하나의 조건문이 조건문 내부에 중첩될 수도 있다. 다음처럼 삼분 예제를 작성할 수 있다.

```

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 조건문에 두 개의 분기가 있다. 첫 분기는 간단한 명령 실행문을 담고 있다. 두 번째 분기는 자체가 두 개의 분기를 가지고 있는 또 다른 if문을 담고 있다. 자체로 둘다 조건문이지만, 두 분기 모두 간단한 실행 명령문이다.



명령문 블록을 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 다시 작성할 수 있다.

```

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

```

print문은 두개의 조건문이 통과될 때만 실행돼서, and 연산자와 동일한 효과를 거둘 수 있다.

```

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

```

3.7 try와 catch를 활용한 예외 처리

앞에서 사용자가 타이핑한 것을 읽어 정수로 파싱하기 위해서 함수 raw_input 와 int을 사용한 프로그램 코드를 살펴 보았다. 또한 이렇게 하는 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 명령문을 실행할 때, ”이런” 잠시 있다가 다음 명령 실행문으로 넘어가는 새로운 명령 프롬프트를 보게 된다.

하지만, 코드가 파이썬 스크립트로 실행이 되면 오류가 발생하고 즉시, 그 지점에서 멈추게 된다. 다음의 명령 실행문은 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 실행하고 적절하지 않은 입력값을 타이핑하게되면, 다소 불친절한 오류 메시지와 함께 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 예측되거나, 예측 못한 오류를 다루는 “try / except”로 불리는 조건 실행 구조가 파이썬에 내장되어 있다. try와 except의 기본 사항은 일부 명령 문이 문제를 가 있다는 것을 사전에 알고 있고, 만약 그 때문에 오류가 발생하게 된다면 대신 실행할 명령문을 프로그램에 추가하는 것이다. except 블록의 명령문은 오류가 없다면 실행되지 않는다.

파이썬의 try, except 기능을 프로그램 코드의 실행에 보험을 든다고 생각할 수 있다.

온도 변환기 프로그램을 다음과 같이 다시 작성할 수 있다.

```
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 블록의 명령문을 우선 실행합니다. 만약 모든 것이 순조롭다면, except 블록을 건너뛰고, 다음 코드를 실행합니다.

except이 try 블록에서 발생하면, 파이썬은 try블록을 건너뛰고 except블록의 명령문을 수행합니다.

```
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 부릅니다. 예제에서는 except 절에서는 단순히 오류 메시지를 출력만 합니다. 대체로, 예외 처리는 오류를 고치거나, 다시 시작하거나, 최소한 프로그램이 정상적으로 종료될 수 있게 합니다.

3.8 논리 연산식의 단락(Short circuit) 평가

파이썬이 $x \geq 2 \text{ and } (x/y) > 2$ 와 같은 논리 연산식을 처리할 때, 왼쪽에서부터 오른쪽으로 연산식을 평가한다. and의 정의 때문에 x이 2보다 작다면, $x \geq 2$ 은 거짓(False)이 되서, 전체는 $(x/y) > 2$ 이 참(True) 혹은 거짓(False)에 관계없이 거짓(False)이 된다. 파이썬이 논리 연산식의 나머지 부분을 평가해도 나아지는 것이 없다고 탐지할 때, 평가를 멈추고 나머지 논리 연산식에 대한 연산도 중지한다. 최종 논리 연산식의 값이 이미 알려졌기 때문에 더 이상의 연산을 멈출 때, 이를 단락(Short-circuiting) 평가라고 한다.

이것이 세심해 보일 수 있지만, 단락 행동이 가디언 패턴(guardian pattern)으로 불리는 좀 더 똑똑한 기술로 연결된다. 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
>>>
```

파이썬이 (x/y) 연산을 평가할 때 y이 0이어서 실행오류 발생시켜서, 세번째 연산은 수행되지 않습니다. 하지만, 두 번째 예제의 경우 $x \geq 2$ 이 거짓(False)이어서 전체가 거짓(False)이 되어 (x/y) 평가가 실행되지 않고, 단락(Short-circuiting) 평가 규칙에 의해서 오류도 발생하지 않습니다.

오류가 발생할 것 같은 평가식 앞에 **가디언(gardian)** 평가식을 전략적으로 배치해서 논리 평가식을 아래와 같이 구성합니다.

```
>>> 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 $x \geq 2$ 이 거짓(False)이라 and에서 멈춥니다. 두 번째 논리 표현식은 $x \geq 2$ 이 참(True), $y \neq 0$ 은 거짓(False)이어서 (x/y) 까지 갈 필요는 없습니다. 세 번째 논리 표현식은 (x/y) 연산이 끝난 후에 $y \neq 0$ 이 수행되어서 오류가 발생합니다.

두 번째 논리 표현식에서 $y \neq 0$ 이 '0'이 아니어만 (x/y) 이 실행될 수 있도록 **가디언(gardian)** 역할을 수행한다고 할 수 있다.

3.9 디버깅(Debugging)

오류가 발생했을 때, 파이썬이 화면에 출력하는 트레이스백(traceback)은 상당한 정보를 담고 있지만, 특히 스택에 많은 프레임이 있는 경우 엄청나게 보일 수도 있다. 대체로 유용한 정보는 다음과 같다.

- 어떤 종류의 오류인가.
- 어디서 발생했는가.

구문 오류는 대체로 발견하기 쉽지만, 몇 가지는 애매합니다. 공백 오류가 대표적인데, 공백(space)과 탭(tab)은 구별이 되지 않고, 통상 무시하고 넘어가기 쉽기 때문입니다.

```
>>> x = 5
>>> y = 6
  File "<stdin>", line 1
    y = 6
    ^
SyntaxError: invalid syntax
```

이 예제의 문제는 두 번째 줄이 한 칸 공백으로 들여써서 발생하는 것입니다. 하지만, y 에 오류 메시지가 있는데 프로그래머를 잘못된 곳으로 인도합니다. 대체로 오류 메시지는 문제가 어디에서 발견되었는지를 지칭하지만, 실제 오류는 코드 앞에서 종종 선행하는 줄에 있을 수 있습니다.

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

```
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 메시지¹가 발생합니다.

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

오류 메시지가 5번째 줄에 있지만, 잘못된 것은 없습니다. 실제 오류를 발견하기 위해서, 출력값은 0인 ratio의 값을 print문을 사용해서 출력해 보는 것이 도움이 됩니다. 문제는 4번째 줄에 있는데, 두 정수를 나누기를 소수점 연산을 했기 때문입니다. signal_power 와 noise_power 를 부동 소수점값으로 표현하는게 해결책입니다.

대체로, 오류 메시지는 어디에 문제가 발견되었는지를 말해주지만, 종종 어디서 원인이 발생했는지는 말해주지 않습니다.

3.10 용어 정의

몸통부문(body): 복합문 내부에 열련의 명령문 실행을 기술한 부문

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

분기(branch): 조건문에서 대안 실행 명령문의 한 흐름

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

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

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

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

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

¹파이썬 3.0에서는 오류 메시지가 발생하지 않습니다. 정수 피연산자인 경우에도 나눗셈 연산자가 부동 소수점 나눗셈을 수행합니다.

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.

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

단락(short circuit): 왜냐하면 파이썬이 나머지 조건 표현식 평가를 할 필요 없이 최종 결과를 알기 때문에, 파이썬이 논리 표현식 평가를 일부 진행하고, 더 이상의 평가를 멈출 때.

3.11 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, except를 사용하여 봉급계산 프로그램을 다시 작성하세요. 숫자가 아닌 입력값을 잘 처리해서 메시지를 출력하고 프로그램을 종료하게 됩니다. 다음은 프로그램의 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

입력값으로 다양한 다른 값을 출력하도록 반복적으로 보이는 것처럼 프로그램을 실행하세요.

Chapter 4

함수

4.1 함수 호출

프로그래밍 문맥에서, **함수(function)**는 연산을 수행하는 일련의 명명된 명령문이다. 함수를 정의할 때, 이름과 일련의 명령문을 명시한다. 후에, 함수를 이름으로 ”호출(call)”할 수 있다. 이미 **함수 호출(function call)**의 예제를 살펴 보았다.

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

함수명은 `type`이다. 괄호안의 표현식을 함수의 **인수(argument)**라고 한다. 인수는 함수의 입력값으로 함수 내부로 전달되는 값이나 변수이다. 앞의 `type` 함수의 결과값은 인수의 형(type)이다.

통상 함수가 인수를 받아 결과를 돌려준다고 한다. 결과를 **결과값(return value)**이라고 부른다.

4.2 내장(Built-in) 함수

파이썬에는 함수를 정의할 필요없이 사용할 수 있는 많은 중요 내장함수가 있다. 파이썬을 처음 만든 사람이 공통의 문제를 해결할 수 있는 함수를 작성해서 여러분이 사용할 수 있도록 파이썬에 포함을 했습니다.

`max`와 `min` 함수는 리스트의 최소값과 최대값을 각기 계산해서 여러분에게 보여 줍니다.

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

`max` 함수는 문자열의 ”가장 큰 문자”, 상기 예제에서는 ”w”, `min`함수는 최소 문자를, 상기 예제에서는 공백, 출력합니다.

또 다른 매우 자주 사용되는 내장 함수는 얼마나 많은 항목이 있는지 출력하는 `len` 함수가 있습니다. 만약 `len` 함수의 인수가 문자열이면 문자열의 문자 갯수를 반환합니다.

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

이들 함수는 문자열에만 한정된 것이 아니라, 뒷장에서 보듯이 다양하게 다양한 자료형에 사용될 수 있습니다.

내장함수의 이름은 사전에 점유된 예약어로 다뤄야 하고, 예를 들어 "max"를 변수명으로 사용을 피해야 합니다.

4.3 형 변환 함수

파이썬은 A형(type)에서 B형(type)으로 값을 변환하는 내장 함수를 제공합니다. `int` 함수는 임의의 값을 입력 받아 변환이 가능하면 정수형으로 변환하고, 그렇지 않으면 불평을 합니다.

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

`int`는 부동 소수점 값을 정수로 변환할 수 있지만 소수점 이하를 절사합니다.

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

`float`는 정수와 문자열을 부동 소수점으로 변환합니다.

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

마지막으로, `str`은 인수를 문자열로 변환합니다.

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

4.4 난수(Random numbers)

동일한 입력을 받을 때, 대부분의 컴퓨터는 매번 동일한 출력값을 생성하기 때문에 결정적(**deterministic**)이라고 합니다. 결정론이 대체로 좋은데 왜냐하면,

동일한 계산은 동일한 결과를 기대할 수 있기 때문입니다. 하지만, 어떤 어플리케이션에서는 컴퓨터가 예측불가능하길 바랍니다. 게임이 좋은 예이고, 더 많은 예를 찾을 수 있습니다. 프로그램을 온전하게 비결정론적으로 만드는 것은 쉽지 않은 것으로 밝혀졌지만, 비결정론적인 것처럼 보이게 하는 방법은 있습니다. 의사 난수(**pseudorandom numbers**)를 생성하는 알고리즘을 사용하는 방법이 그 중 하나입니다. 의사 난수는 이미 결정된 연산에 의해서 생성된다는 점에서 진정한 의미의 난수는 아니지만, 이렇게 생성된 숫자를 진정한 난수와 구별하기는 불가능에 가깝다.

`random` 모듈은 의사 난수를 생성하는 함수를 제공한다. (이하 의사 난수 대신 “난수(`random`)”로 간략히 부르기로 한다.)

`random` 함수는 0.0과 1.0 사이의 부동 소수점 난수를 반환한다. `random` 함수는 0.0은 생성하지만 1.0은 생성하지 않는다. 매번 `random` 함수를 호출할 때마다 이미 생성된 난수열에서 하나씩 하나씩 뽑아 쓰게 됩니다. 샘플로 다음 반복문을 실행해 봅시다.

```
import random

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

프로그램이 0.0과 1.0을 포함하지 않는 최대 1.0에서 10개의 난수 리스트를 생성합니다.

```
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 단지 많은 함수 중의 하나입니다. `randint` 함수는 최저(`low`)와 최고(`high`) 매개 변수를 입력받아 최저값(`low`)과 최고값(`high`)을 포함하는 두 사이의 정수를 반환합니다.

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

무작위로 배열로부터 하나의 숫자를 뽑아내기 위해서, `choice`를 사용합니다.

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

random 모듈은 정규분포, 지수분포, 감마분포 및 몇가지 추가된 연속형 분포에서 난수를 생성할 수 있는 함수도 제공합니다.

4.5 수학 함수

파이썬은 가장 친숙한 수학 함수를 제공하는 수학 모듈이 있습니다. 수학 모듈을 사용하기 전에, 수학 모듈 가져오기를 실행합니다.

```
>>> import math
```

이 명령문은 math 모듈 개체를 생성한다. 모듈 개체를 출력하면, 모듈 개체에 대한 정보를 얻을 수 있다.

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

모듈 개체는 모듈에 정의된 함수와 변수를 담고 있다. 함수 중에 하나에 접근하기 위해서, 점으로 구분되는 모듈의 이름과 함수의 이름을 명시해야 한다. 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 로그 지수 10으로 신호 대비 소음 비율을 계산한다. 수학 모듈은 자연로그를 log함수를 호출해서 사용할 수 있도록 제공한다.

두 번째 예제는 라디안의 사인값을 찾는 것이다. 변수의 이름이 힌트로 sin과 다른 삼각 함수(cos, tan 등)는 라디안을 인수로 받는다. 도수에서 라디안으로 변환하기 위해서 360으로 나누고 2π 를 곱한다.

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

math.pi 표현문은 수학 모듈에서 pi 변수를 얻고, π 와 근사적으로 동일하고 15 자리수까지 정확하다.

삼각함수를 배웠다면, 앞의 연산 결과를 2를 루트를 씌우고 2로 나누어서 비교한다.

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

4.6 신규 함수 추가

지금까지 파이썬에 딸려 있는 함수를 사용했지만 새로운 함수를 추가하는 것도 가능하다. **함수 정의(function definition)**는 신규 함수명과 함수가 호출될 때 실행할 일련의 명령문을 명세한다. 함수를 신규로 정의하면, 프로그램 내내 반복해서 함수를 재사용할 수 있다.

여기 예제가 있다.

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

`def`는 이것이 함수 정의를 나타내는 키워드입니다. 함수명은 `print_lyrics`입니다. 함수명을 명명 규칙은 변수명과 동일합니다. 문자, 숫자, 그리고 몇몇 문장 부호는 사용할 수 있지만, 첫 문자가 숫자는 될 수 없다. 함수명으로 예약어 키워드를 사용할 수도 없고, 동일한 변수명과 함수명은 피하는 것이 좋다.

함수명 뒤에 빈 팔호는 이 함수가 어떠한 인수도 갖지 않는다는 것을 나타낸다. 나중에, 입력값으로 인수를 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 들여쓰기를 해야 한다. 파이썬 관례로 들여쓰기는 항상 4칸의 공백이다. 몸통 부문은 제약 없이 명령문을 작성할 수 있다.

`print`문의 문자열은 이중 인용부호로 감싸진다. 단일 인용부호나, 이중 인용부호나 차이가 없다. 대부분의 경우 단일 인용부호를 사용하고, 단일 인용부호가 문자열에 나타나는 경우, 이중 인용부호를 사용하여 단일 인용부호가 출력되게 감싼다.

함수 정의를 인터랙티브 모드에서 타이핑을 한다면, 함수 정의가 끝나지 않았다는 것을 알 수 있도록 생략부호(...)를 출력한다.

```
>>> 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 위해서는 엔터(Enter)키를 눌러 빈 줄을 삽입한다. (스크립트에서는 반드시 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`의 값은 'function' 형을 가지는 **함수 개체(function object)**이다.

신규 함수를 호출하는 구문은 내장 함수의 경우와 동일합니다.

```
>>> 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 가능합니다. 예를 들어, 이전의 후렴구를 반복하기 위해 repeat_lyrics 함수를 작성할 수 있습니다.

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

그리고 나서, repeat_lyrics 함수를 호출합니다.

```
>>> 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, 이 방법이 실제 노래가 불려지는 방법은 아닙니다.

4.7 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 프로그램은 두개의 함수(print_lyrics, repeat_lyrics)를 담고 있다. 함수정의는 다른 명령문과 동일하게 수행되지만, 함수 자체를 생성한다는 점에서 차이가 있다. 함수 내부의 명령문은 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 실행해 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?

4.8 실행 흐름

함수가 첫 사용전에 정의되는 것을 확인하기 위해서, 명령문의 실행 순서를 파악해야 하는데 이를 **실행 흐름(flow of execution)**이라고 한다.

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

함수 정의(definitions)가 프로그램의 실행 순서를 바꾸지는 않는다. 하지만, 함수 내부의 명령문은 함수가 호출될 때까지 실행이 되지 않는 것을 기억합니다.

함수 호출은 프로그램의 실행 흐름을 우회하는 것과 같습니다. 다음 실행 명령문으로 가기 전에 실행 흐름은 함수의 몸통 부분으로 건너 뛰어 실행하고는 다시 건너 뛰기를 시작한 지점으로 다시 돌아온다.

하나의 함수가 또 다른 함수를 호출한다는 것을 기억할 때까지는 매우 간단하게 들립니다. 함수 중간에서 프로그램이 또 다른 함수의 명령문을 수행할지도 모릅니다. 하지만, 새로운 함수가 실행되는 중간에 프로그램이 또 다른 함수를 실행할지도 모릅니다!

다행스럽게도, 파이썬은 프로그램의 실행 위치를 정확히 추적합니다. 그래서, 함수가 실행을 완료할 때마다, 프로그램이 함수를 호출해서 떠난 지점으로 정확히 되돌려 놓습니다. 프로그램의 마지막에 도달했을 때, 프로그램은 종료합니다.

조금 복잡한 이야기의 교훈은 무엇일까요? 프로그램을 읽을 때, 위에서부터 아래로 읽을 필요는 없습니다. 때때로, 실행 흐름을 따르는 것이 좀더 이치에 맞습니다.

4.9 매개 변수(parameter)와 인수(argument)

지금까지 살펴본 몇몇 내장 함수는 인수를 요구합니다. 예를 들어, `math.sin` 함수를 호출할 때, 숫자를 인수로 넘겨야 합니다. 어떤 함수는 2개 이상의 인수를 받습니다. `math.pow`는 밑과 지수 2개의 인수가 필요합니다.

인수는 함수 내부에서 **매개 변수(parameters)**로 불리는 변수로 할당됩니다. 하나의 인수를 받는 사용자 정의 함수가 예제로 있습니다.

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

사용자 정의 함수는 인수를 받아 `bruce` 매개변수에 할당한다. 함수가 호출될 때, 매개변수의 값(무엇이든지 관계 없이)을 두번 출력합니다.

사용자 정의 함수는 출력 가능한 임의의 값에 작동합니다.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

내장함수에 적용되는 동일한 조합 규칙이 사용자 정의 함수에도 적용되어서, `print_twice` 함수의 인수로 어떤 종류의 표현식도 가능합니다.

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

인수는 함수가 호출되기 전에 평가가 완료되어서, 예제에서 '`Spam` '*4과 `math.cos(math.pi)`은 단지 1회만 평가 됩니다.

인수로 변수도 사용이 가능합니다.

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

인수로 넘기는 변수명(`michael`)은 매개 변수명(`bruce`)과 아무런 연관이 없다. 무슨 값이 호출된든 호출하는 측에서는 상관이 없다. 여기 `print_twice` 함수에서 누구나 `bruce`라고 부른다.

4.10 열매 함수(fruitful function)와 빈 함수(void function)

수학 함수와 같은 몇몇 함수는 결과를 만들어 낸다. 좀더 좋은 이름이 없어서, 결과를 만들어 내는 함수를 **열매함수(fruitful functions)**라고 명명한다. `print_twice`와 같은 명령을 수행하지만, 결과를 만들어 내지 않는 함수를 **빈 함수(void functions)**라고 부른다.

열매 함수를 호출할 때는 결과 값을 가지고 뭔가를 하려고 한다. 예를 들어, 결과값을 변수에 할당하거나, 표현식의 일부로 재사용할 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 열매함수를 호출하고 변수에 결과값을 저장하지 않으면 반환되는 결과값은 안개속에 사라져간다!

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

이 스크립트는 제곱근 5의 값을 연산하지만, 변수에 결과값을 저장하거나, 화면에 출력하지 않아서 그 다지 유용하지는 않다.

빈 함수(Void functions)는 화면에 뭔가 출력하거나 뭔가 다른 효과를 가지지만, 반환값은 없다. 빈 함수를 사용하여 결과에 변수를 할당하면, `None`으로 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, 함수내부에 `return`문을 사용한다. 예를 들어, 두 숫자를 더해서 결과를 반환하는 `addtwo`라는 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`, `b`는 각각 3, 5이다. `addtwo` 함수는 두 숫자의 덧셈을 수행하고 `added`라는 로컬 변수에 저장하고, `return`문을 사용해서 덧셈 결과를 반환하고, `x`라는 변수에 할당하여 출력한다.

4.11 왜 함수를 사용하는가?

프로그램을 함수로 나누는 고생을 할 가치가 왜 있는지 불명확할지 모릅니다. 여기 몇 가지 이유가 있습니다.

- 명령문을 그룹으로 만들어 새로운 함수로 명명하는 것은 프로그램을 읽고, 이해하고, 디버그하기 좋게 합니다.
- 함수는 반복 코드를 제거해서 프로그램을 작고 콤팩트하게 만듭니다. 후에 프로그램에 수정사항이 생기면, 단지 한 곳에서만 수정을 하면 됩니다.

- 긴 프로그램을 함수로 나누어 작성하는 것은 작은 부분에서 버그를 수정할 수 있게 하고 이를 조합해서 전체 온전한 프로그램을 만들 수 있습니다.
- 잘 설계된 함수는 종종 많은 프로그램에 유용하게 사용됩니다. 잘 설계된 프로그램을 작성하고 디버그를 해서 오류가 없이 만들게 되면, 나중에 재사용이 용이합니다.

책의 나머지 부분에서 이 개념을 설명하는 함수 정의를 종종 사용할 것입니다. 함수를 만들고 사용하는 기술의 일부는 ”리스트에서 가장 작은 값을 찾아내는 것”과 같은 생각을 적절하게 추상화하여 함수를 작성하는 것입니다. 나중에, 리스트에서 가장 작은 값을 찾아내는 코드를 보여 줄 것입니다. 리스트를 인수로 받아 가장 작은 값을 반환하는 `min` 함수를 작성해서 여러분에게 보여드릴 것입니다.

4.12 디버깅

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

탭과 공백은 통상 눈에 보이지 않기 때문에 디버그를 어렵게 합니다. 자동으로 들여쓰기를 해주는 에디터를 프로그램 작성 시 사용하세요.

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

동일하지만 잘못된 프로그램을 반복적으로 실행한다면, 디버깅은 오래 걸릴 수 있습니다.

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

4.13 용어정의

알고리즘(algorithm): 특정 범주의 문제를 해결하는 일반적인 프로세스

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

몸통 부문(body): 함수 정의문 내부에 일련의 명령문

복합(composition): 좀더 큰 표현식의 일부분으로 표현식을 사용하거나, 좀더 큰 명령문의 일부로서의 명령문.

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.

열매 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 인수 리스트로 구성된다.

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

함수 개체(function object): 함수 정의로 생성되는 값. 함수명은 함수 개체를 참조하는 변수다.

머리 부분(header): 함수 정의의 첫번째 줄

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

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

매개 변수(parameter): 인수로 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, 반환 값은 표현식의 값이 된다.

빈 함수(void function): 반환값을 갖지 않는 함수

4.14 연습문제

Exercise 4.4 파이썬의 ”def” 키워드의 목적은 무엇입니까?

What is the purpose of the ”def” keyword in Python?

- a) ”다음의 코드는 정말 좋다”라는 의미를 가진 속어
- b) 함수의 시작을 표시한다.
- c) 다음의 들여쓰기 코드 부문은 나중을 위해 저장되어 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 가지는 함수 computepay을 생성하여, 초과근무에 대해서는 50% 초과 근무 수당을 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

입력값으로 다양한 다른 값을 테스트하기 위해서 반복적으로 프로그램을 실행하세요.

Chapter 5

반복(Iteration)

5.1 변수 갱신

할당문의 일반적인 패턴은 변수를 갱신하는 할당문이다. 변수의 새로운 값은 예전 값에 의존하게 된다.

```
x = x+1
```

상기 예제는 “현재의 값 x에 1을 더해서 x를 새로운 값으로 갱신한다.”

존재하지 않는 변수를 갱신하려면, 오류가 발생한다. 왜냐하면 x에 값을 할당하기 전에 오른쪽을 파이썬이 평가해야 하기 때문이다.

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

변수를 갱신하기 전에 간단한 변수 할당으로 통상 **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.

5.2 while문

반복적인 작업을 자동화하기 위해서 종종 컴퓨터를 사용한다. 동일하거나 비슷한 작업을 오류 없이 반복하는 것은 컴퓨터가 사람보다 잘하는 것이다. 반복이 흔한 일이어서, 파이썬은 반복 작업을 쉽게 하도록 몇 가지 언어적인 기능을 제공합니다.

파이썬에서 반복의 한 형태가 **while문**입니다. 5에서부터 거꾸로 세어서 마지막에 “Blastoff!”를 출력하는 간단한 프로그램이 있습니다.

```

n = 5
while n > 0:
    print n
    n = n-1
print 'Blastoff!'

```

마치 영어를 읽듯이 while을 읽어 내려갈 수 있습니다. n이 0보다 큰 동안에 n의 값을 출력하고 n값을 1만큼 줄입니다. 0에 도달했을 때, while문을 빠져나가 Blastoff!”를 화면에 출력합니다.

좀 더 형식적으로 정리하면, while문의 실행 흐름이 다음에 있습니다.

1. 참 (True) 혹은 거짓 (False)를 산출하는 조건을 평가한다.
2. 만약 조건이 거짓이면, while문을 빠져나가 다음 명령문을 계속 실행 한다.
3. 만약 조건이 참이면, 몸통 부문의 명령은을 실행하고 다시 처음 1번 단계로 돌아간다.

3번째 단계에서 처음으로 다시 돌아가는 반복을 하기 때문에 이런 종류의 흐름을 **루프(loop)**이라고 한다. 매번 루프의 몸통부문을 실행할 때마다, 이것을 **반복(iteration)**이라고 한다. 상기 루프에 대해서 “5번 반복했다고 말한다.” 즉, 루프의 몸통부문이 5번 수행되었다.

루프의 몸통부문은 필히 하나 혹은 그 이상의 변수값을 바꾸어서 결국 조건식이 거짓이 되어 루프가 종류가 되게 만들어야 한다. 매번 루프가 실행될 때마다 변경되고 루프가 끝나는 것을 관리하는 변수를 **반복 변수(iteration variable)**라고 한다.

만약 반복 변수가 없다면, 루프는 영원히 돌 것이고, 결국 **무한 루프(infinite loop)**에 빠질 것이다.

5.3 무한 루프

프로그래머에게 무한한 즐거움의 원천은 아마도 ”거품내고, 헹구고, 반복” 적혀 있는 샴프 사용법 문구가 무한루프라는 것을 알아차리는 것입니다. 왜냐하면, 루프를 얼마나 많이 실행해야 하는지 말해주는 **반복 변수(iteration variable)**가 없기서 무한 반복하기 때문입니다.

숫자를 꺼꾸로 세는 (countdown) 예제는 루프가 끝나는 것을 증명할 수 있다. 왜냐하면 n값이 유한하고, n이 매번 루프를 돌 때마다 작아져서 결국 0에 도달 할 것이기 때문이다. 다른 경우 반복 변수가 전혀 없기 때문에 루프가 명백하게 무한으로 돈다.

5.4 무한 반복과 break

종종 봄통 부문을 절반 진행할 때까지 루프를 종료해야하는 시점인지 확신을 못합니다. 이런 경우 의도적으로 무한 루프를 작성하고 break 문을 사용하여 루프를 빠져나옵니다.

while문 논리 표현식이 단순히 논리 상수 참(True)으로 되어 있어 이 루프는 명백하게 **무한 루프(infinite loop)**이다.

```
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 것입니다.

연산식의 상수 값이 참(True)이라는 사실로 루프 상단의 논리 연산식이 항상 참값이여서 프로그램을 영원히 혹은 배터리가 모두 소진될 때까지 실행될 것이다.

이것이 역기능 무한 루프는 사실이지만, 유용한 루프를 작성하기 위해 이 패턴을 여전히 이용할 것입니다. 단, 루프 봄통 부문에 break문을 사용하여 루프를 빠져나가는 조건에 도달했을 때, 루프를 명시적으로 빠져나갈 수 있도록 주의깊게 코드를 추가해야 합니다.

예를 들어, 사용자가 done을 치기 전까지 사용자로부터 입력받을 받기 원하다고 가정해서 프로그램 코드를 아래와 같이 작성합니다.

For example, suppose you want to take input from the user until they type done. You could write:

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

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

매번 프로그램이 꺼쇠 괄호로 사용자에게 명령문을 치도록 재촉합니다. 사용자가 done을 타이핑하면, break문이 실행되어 루프를 빠져나오게 됩니다. 그렇지 않은 경우 프로그램은 사용자가 무엇을 입력하든 메아리처럼 입력한 그대로 출력하고 다시 루프 처음으로 되돌아 갑니다. 여기 예제로 실행한 결과가 있습니다.

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

while 루프를 이와 같은 방식으로 작성하는 것은 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.

5.5 continue로 반복 종료

때때로 루프 반복 중간에 현재 반복을 끝내고, 다음 반복으로 즉시 점프하고 싶을 때가 있습니다. 현재 반복의 루프 몸통 부분을 끝내지 않고 다음 반복으로 건너뛰기 위해서 continue문을 사용합니다.

사용자가 ”done”을 입력할 때까지 입력값을 그대로 출력하는 루프 예제가 있다. 하지만 파일 주석문처럼 해쉬(#)로 시작하는 줄을 출력되지 않는 줄로 다루고 있다.

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

continue문이 추가된 신규 프로그램을 실행한 예제가 있다.

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

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

5.6 for문을 사용한 명확한 루프

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

while문을 불명확한 루프(*indefinite loop*)라고 하는데, 왜냐하면 어떤 조건이 거짓(False)가 될 때까지 단지 루프를 돌기 때문이다. 하지만, for루프는 알고 있는 항목의 집합만큼 루프를 돌게되어서 집합에 항목이 있는 만큼만 실행이 된다.

for루프의 구문은 while루프의 구문과 비슷하다. for문이 있고, 루프 몸통 부문으로 구성된다.

```
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 3개의 문자열 각각에 대해 몸통 부문을 실행하여 다음의 결과를 얻게된다.

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

for 루프를 영어로 번역하는 것은 while문을 번역하는 것과 같이 직접적이지는 않다. 하지만, 만약 friends를 집합(set)으로 생각한다면 다음과 같다. friends라고 명명된 집한에서 각 friend에 대해서 한번씩 for 루프의 몸통 부문의 명령문을 실행하세요.

for 루프를 살펴보면, **for**와 **in**은 파이썬 키워드로 예약어이고 friend와 friends는 변수이다.

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

특히, friend는 for 루프의 **반복 변수(iteration variable)**입니다. friend 변수는 루프의 각 반복에 대해서 변하게 되고, 언제 for 루프가 끝나는지 통제합니다. **반복 변수**는 friend 변수에 저장된 3개의 문자열을 순차적으로 지나간다.

5.7 루프 패턴

종종 for와 while문을 사용하여, 항목의 리스트, 파일 콘텐츠를 훑어 자료의 가장 큰 값이나 작은 값 같은 것을 찾습니다.

for나 while 루프는 일반적으로 다음과 같이 구축됩니다.

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

루프 패턴의 개념과 작성을 시연하기 위해서 숫자 리스트를 사용할 것입니다.

5.7.1 계산과 합산 루프

예를 들어, 리스트의 항목의 숫자를 계산하기 위해서 다음 for 루프를 작성합니다.

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

루프가 시작하기 전에 변수 count를 0으로 놓고, 숫자 목록을 실행하기 위해 for 루프를 작성합니다. 반복(iteration) 변수는 itervar라고 하고, itervar은 루프에서 사용되지 않지만, itervar는 루프를 통제하고 루프 몸통 부문이 목록의 각 값에 대해서 한번 실행되게 합니다.

루프 몸통부문에 목록의 각 값에 대해서 변수 count 값에 1을 더합니다. 루프가 실행될 때, count 값은 지금까지 살펴본 목록 값의 횟수가 됩니다.

루프가 종료되면, count 값은 총 목록 숫자가 됩니다. 총 숫자는 루프 끝에 얻어 졌다. 루프를 구성해서, 루프가 끝났을 때 원하는 바를 얻었다.

숫자의 총계를 계산하는 또 다른 비슷한 루프는 다음과 같다.

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

이 루프에서, 반복 변수(iteration variable)가 사용되었다. 앞선 루프에서처럼 변수 count에 1을 단순히 더하는 대신에, 실제 숫자 (3, 41, 12, 등)를 각 루프 반복을 수행하는 동안 작업중인 합계에 더하게 하였다. 변수 total을 생각해보면, total은 지금까지 값의 총계다. 루프가 시작하기 전에 total은 어떤 값도 살펴 본 적이 없어서 0이다. 루프가 도는 중간에는 total은 작업중인 총계가 된다. 루프의 마지막 단계에서 total은 항목의 모든 값의 총계가 된다.

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

계산 루프나 합산 루프나 특히 실무에서는 유용하지는 않다. 왜냐하면 리스트에서 항목의 개수와 총계를 계산하는 len()과 sum()가 각각 내장 함수로 있기 때문이다.

5.7.2 최대값과 최소값 루프

리스트나 순서에서 가장 큰 값을 찾기 위해서, 다음과 같이 루프를 작성합니다.

```
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 `largest`는 ”지금까지 본 가장 큰 수”로 생각할 수 있다. 루프 시작 앞에 `largest` 값이 상수 `None`이다. `None`은 ”빈” 변수를 표기하기 위해서 변수에 저장하는 특별한 상수 값이다.

루프 시작 전에 지금까지 본 가장 큰수는 `None`이다. 왜냐하면 아직 어떤 값도 보지 않았기 때문이다. 루프가 실행되는 동안에, `largest` 값이 `None`이면, 첫 번째 본 값이 지금까지 본 가장 큰 값이 된다. 첫번째 반복에서 `itervar`는 3이 되는데 `largest` 값이 `None`이여서 즉시, `largest`값을 3으로 놓는다.

첫번째 반복 후에 `largest`는 더 이상 `None`이 아니어서, ”지금까지 본” 값보다 더 큰 값을 찾게 될 때 작동하는 복합 논리 표현식의 두 번째 `itervar > largest`인지를 확인하는 부분이 작동된다. ”더 큰” 값을 찾게 되면 변수 `largest`에 새로운 값으로 대체한다. `largest`가 3에서 41, 41에서 74로 변경되어 출력되는 것을 확인할 수 있다.

루프의 끝에서 모든 값을 훑어서 변수 `largest`는 리스트의 가장 큰 값을 담고 있다.

최소값을 계산하기 위해서는 코드가 매우 유사하지만 작은 변화가 필요하다.

```
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 `smallest`는 루프 실행 전에, 중에, 완료 후에 ”지금까지 본 가장 작은” 값이 된다. 루프 실행이 완료되면, `smallest`는 리스트의 최소 값을 담게 된다.

계산과 합산에서와 마찬가지로 파이썬 내장함수 `max()`와 `min()`이 이렇게 루프 문을 작성하는 것을 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
```

최소 코드 함수 버전에서 파이썬에 이미 내장된 `min` 함수와 동등하게 만들기 위해서 모든 `print`문을 삭제했다.

5.8 디버깅

좀 더 큰 프로그램을 작성함에 따라, 좀 더 많은 시간을 디버깅에 보내는 자신을 발견할 것이다. 좀 더 많은 코드는 버그가 숨을 수 있는 좀 더 많은 장소와 좀 더 많은 오류가 발생할 기회가 있다는 것을 의미한다.

디버깅 시간을 줄이는 한 방법은 ”양분에 의한 디버깅(debugging by bisection)” 기법이다. 예를 들어, 프로그램에 100 줄이 있고 한번에 하나씩 확인한다면, 100 번의 단계가 필요하다.

대신에 프로그램을 반으로 나눕니다. 프로그램의 정확히 중간이나, 중간부분에서 중간값을 확인합니다. `print`문이나, 검증 효과를 갖는 상응하는 대용물을 넣고 프로그램을 실행합니다.

중간지점확인이 부정확하면 문제는 양분한 프로그램의 앞부분에 있음에 틀림 없다. 만약 정확하다면, 문제는 프로그램 뒷부분에 있다.

이와 같은 방식으로 확인을 수행하게 되면, 검토해야하는 코드의 줄수를 절반으로 계속 줄일 수 있다. 100번의 단계가 걸리는 것에 비해 6번의 단계 후에 이론적으로 1 혹은 2 줄의 코드로 범위를 좁힐 수 있다.

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

5.9 용어정의

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

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

감소(decrement): 변수의 값을 감소하여 개선

초기화(initialize): 개선될 변수의 값을 초기 값으로 할당

증가(increment): 변수 값을 증가시켜 개선 (통상 1씩) (often by one).

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

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

5.10 Exercises

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, try와 except를 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 목록의 최대값과 최소값을 출력하세요.

Chapter 6

문자열

6.1 문자열은 순열이다.

문자열은 문자의 순열이다. 대괄호 연산자로 한번에 하나씩 문자에 접근할 수 있다.

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

두 번째 명령문은 변수 fruit에서 1번 위치의 문자를 추출하여 변수 letter에 할당한다. 대괄호의 표현식을 **인덱스(index)**라고 부른다. 인덱스는 순열의 무슨 문자를 사용자가 원하는지 표시한다.

하지만, 여려분이 기대한 것을 얻지 못합니다.

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

대부분의 사람에게 'banana'의 첫 문자는 a가 아니라 b입니다. 하지만, 파이썬에서 인덱스는 문자열의 처음부터 값이 시작됩니다. 첫 글자의 시작 값(오프셋, offset)은 0입니다.

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

b가 'banana'의 0번째 문자가 되고 a가 첫번째, n이 두번째 문자가 됩니다.

b	a	n	a	n	a
[0]	[1]	[2]	[3]	[4]	[5]

인덱스로 문자와 연산자를 포함하는 어떤 표현식도 사용 가능지만, 인덱스의 값은 정수여야 합니다. 정수가 아닌 경우 다음과 같은 결과를 얻게 됩니다.

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

6.2 len함수를 사용하여 문자열의 길이 구하기

`len`은 문자열의 문자 갯수를 반환하는 내장함수다.

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

문자열의 가장 마지막 문자를 얻기 위해서, 아래와 같이 시도하려고 할 것입니다.

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

`IndexError`의 이유는 '`banana`'에 6번 인덱스의 문자가 없기 때문입니다. 0에서부터 시작했기 때문에 6개의 문자는 0 5로 번호가 매겨졌습니다. 마지막 문자를 얻기 위해서 `length`에서 1을 빼야 합니다.

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

대안으로 문자의 끝에서 역으로 수를 세는 음의 인덱스를 사용할 수 있습니다.

`fruit[-1]`은 마지막 문자를 `fruit[-2]`는 끝에서 두 번째 등등 활용할 수 있습니다.

6.3 루프를 사용한 문자열 운행법

많은 연산의 경우 문자열을 한번에 한 문자씩 처리하는 합니다. 종종 처음에서 시작해서, 차례로 각 문자를 선택하고, 선택된 문자에 임의의 연산을 수행하고, 끝까지 계속합니다. 이런 처리 패턴을 **운행법(traversal)**라고 합니다. 운행법을 작성하는 한 방법은 `while` 루프입니다.

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

`while` 루프가 문자열을 운행하여 문자열을 한줄에 한자씩 화면에 출력합니다. 루프 조건이 `index < len(fruit)`이여서, `index`가 문자열의 길과 같을 때, 조건은 거짓이 되고, 루프의 몸통 부문은 실행이 되지 않습니다. 파이썬이 접근한 마지막 문자는 문자열의 마지막 문자인 `len(fruit)-1` 인덱스의 문자입니다.

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

운행법을 작성하는 다른 방법은 `for` 루프입니다.

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

루프를 매번 반복할 때, 문자열의 다음 문자가 변수 `char`에 할당됩니다. 루프는 더 이상 남겨진 문자가 없을 때까지 계속 실행됩니다.

6.4 문자열 조각

문자열의 일부분을 **문자열 조각(slice)**이라고 합니다. 문자열 조각을 선택하는 것은 문자를 선택하는 것과 유사합니다.

```
>>> 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 반환합니다.

콜론 앞의 첫 인덱스를 생략하면, 문자열 조각은 문자열의 처음부터 시작합니다. 두 번째 인덱스를 생략하면, 문자열 조각은 문자열의 끝까지 갑니다.

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

만약 첫번째 인덱스가 두번째보다 크거나 같은 경우 결과는 두 인용부호로 표현되는 빈 문자열(empty string)이 됩니다.

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

빈 문자열은 어떠한 문자도 포함하고 있지 않아서 길이가 0 이지만, 이외에 다른 문자열과 동일합니다.

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

6.5 문자열은 불변이다.

문자열의 문자를 변경하려는 의도로 할당문의 왼쪽편에 `[]` 연산자를 사용하고 싶은 유혹이 있을 것입니다. 예를 들어 다음과 같습니다.

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

이 경우 ”개체”는 문자열이고, 할당하고자 하는 문자는 ”항목”이다. 지금 당장은 개체는 값과 같은 것이지만, 후에 좀더 정의를 상세화할 것입니다. 항목은 순열의 값중의 하나입니다.

오류의 이유는 문자열이 불변(immutable)이기 때문입니다. 따라서 존재하는 문자열을 바꿀 수 없다는 의미입니다. 최선은 원래 문자열에 변화를 준 새로운 문자열을 생성하는 것입니다.

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

새로운 첫 문자에 greeting 문자열 조각을 연결해서, 원래 문자열에는 어떤한 영향도 주지 않는 새로운 문자열을 만들었습니다.

6.6 루프 돌기(looping)와 세기(counting)

다음 프로그램은 문자열에 문자 a가 나타나는 횟수를 셹니다.

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

상기 프로그램은 카운터(counter)라고 부르는 또다른 연산 패턴을 보여줍니다. 변수 count는 0으로 초기화 되고, 매번 a를 찾을 때마다 증가합니다. 루프를 빠져나갔을 때, count는 결과 값, a가 나타난 총 횟수를 담고 있습니다.

Exercise 6.3 상기 코드를 캡슐화(encapsulation)하여 문자열과 문자를 인수로 받는 count라는 함수를 작성해서 일반화하세요.

6.7 in 연산자

연산자 in은 불연산자로 두개의 문자열을 받아, 첫 번째 문자열이 두 번째 문자열의 일부이면 참(True)을 반환한다.

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

6.8 문자열 비교

비교 연산자도 문자열에서 동작합니다. 두 문자열이 같은지를 살펴봅시다.

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

다른 비교 연산자는 단어를 알파벳 순으로 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 않습니다. 모든 대문자는 소문자 앞에 위치합니다.

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

이러한 문제를 다루는 일반적인 방식은 비교연산을 수행하기 전에 문자열을 표준 포맷으로 예를 들어 모두 소문자, 변환하는 것입니다. ”Pineapple”로 무장한 사람들로부터 여러분을 보호하는 경우를 명심하세요.

6.9 string 메쏘드

문자열은 파이썬 **개체(objects)**의 한 예이다. 개체는 데이터(실제 문자열 자체)와 **메쏘드(methods)**를 담고 있다. 메쏘드는 개체내부에 내장되고 개체의 어떤 **인스턴스(instance)**에도 사용될 수 있는 효과적인 함수다.

파이썬은 개체에 대해서 이용가능한 메쏘드를 보여주는 `dir` 함수가 있다. `type` 함수는 개체의 형(type)을 보여주고, `dir`은 개체의 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 메쏘드를 보여주고, 메쏘드에 대한 간단한 문서를 `help`를 사용할 수 있지만, 문자열 메쏘드에 대한 좀 더 좋은 문서 정보는 docs.python.org/library/string.html에서 찾을 수 있다.

인수를 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 변경된 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 인수를 갖지 않는 것을 나타낸다.

메쏘드를 부르는 것을 **호출(invocation)**이라고 부른다. 상기의 경우, `word`에 `upper` 메쏘드를 호출한다고 말한다.

예를 들어, 문자열안에 한 문자의 위치를 찾는 `find`라는 문자열 메쏘드가 있다.

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

상기 예제에서, `word` 문자열의 `find` 메쏘드를 호출하여 매개 변수로 찾고 있는 문자를 넘긴다.

`find` 메쏘드는 문자뿐만 아니라 부분 문자열도 찾을 수 있다.

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

검색 시작 위치를 지정하는 두 번째 인덱스를 인수로 갖을 수도 있다.

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

한가지 자주 있는 작업은 `strip` 메쏘드를 사용해서 문자열의 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 값을 반환한다.

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

startswith가 대소문자를 구별하는 것을 요구하기 때문에 `lower` 메쏘드를 사용해서 임의의 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 결과 문자열이 문자 ”p”로 시작하는지를 검증하기 위해서, `lower` 메쏘드가 호출되고 나서 바로 `startswith` 메쏘드를 사용한다. 순서만 주의깊게 다룬다면, 한줄에 여러개의 메쏘드를 호출할 수 있다.

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

6.10 문자열 파싱(Parsing)

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

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

각 라인의 뒤쪽 전자우편 주소(uct.ac.za)만 뽑아내고 싶을 것이다. `find` 메쏘드와 문자열 조각(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 조각을 사용한다.

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

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

`find` 메쏘드에 대한 문서는 docs.python.org/library/string.html에서 참조 가능하다.

6.11 형식 연산자

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

첫 피연산자는 **형식 문자열 format string**로 두번째 피연산자가 어떤 형식으로 표현되는지를 명세하는 하나 혹은 그 이상의 **형식 열 format sequence**을 담고 있다. 결과값은 문자열이다.

예를 들어, 형식 열 '%d'의 의미는 두번째 피연산자가 정수 형식으로 표현됨을 뜻한다. (d는 “decimal”를 나타낸다.)

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

결과는 '42' 문자열로 정수 42와 혼동하면 안 된다.

형식 열은 문자열 어디에서도 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 문자열의 형식 열이 하나 이상이라면, 두번째 인수는 튜플(tuple)이 된다. 형식 열 각각은 튜플의 요소와 순서대로 일치된다.

다음 예제는 정수 형식을 표현하기 위해서 '%d', 부동 소수점 형식을 표현하기 위해서 '%g', 문자열 형식을 표현하기 위해서 '%s'을 사용한 것을 보여준다. 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 문자열의 형식 열의 숫자와 일치해야 하고, 요소의 형도 형식 열과 일치해야 한다.

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

첫 예제는 충분한 요소 개수가 않돼고, 두 번째 예제는 형식이 맞지 않는다.

형식 연산자는 강력하지만, 사용하기가 어렵다. 더 많은 정보는 docs.python.org/lib/typesseq-strings.html에서 찾을 수 있다.

6.12 디버깅

프로그램을 작성하면서 배양해야 하는 기술은 항상 자신에게 질문을 하는 것이다. ”여기서 무엇이 잘못 될 수 있을까?” 혹은 ”사용자가 내가 작성한 완벽한 프로그램을 망가뜨리기 위해 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. 0번째 문자가 없어서 트레이스백(traceback)이 발생한다. 입력줄이 비어있을 때, 코드 3번째 줄을 ”안전”하게 만드는 두 가지 방법이 있다.

`startswith` 메쏘드를 사용해서 빈 문자열이면 거짓 (False) 을 반환한다.

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

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

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

6.13 용어정의

카운터(counter): 무언가를 세기 위해서 사용되는 변수로 일반적으로 0으로 초기화 되고 증가한다.

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

형식 연산자(format operator): 형식 문자열과 튜플을 받아, 형식 문자열에 지정된 형식으로 튜플 요소를 포함하는 문자열을 생성하는 연산자, %.

행식 열(format sequence): %d처럼 어떻게 값이 형식적으로 표현되어야 하는지를 명세하는 ”형식 문자열”의 문자의 열.

형식 문자열(format string): 형식 열을 포함하는 형식 연산자와 함께 사용되는 문자열.

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

호출(invocation): 메쏘드를 호출하는 명령문.

불변(immutable): 열의 항목에 할당할 수 없는 특성.

인덱스(index): 문자열의 문자처럼 열의 항목을 선택하기 위해 사용되는 정수 값.

항목(item): 열에 있는 값의 하나.

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

개체(object): 변수가 참조하는 무엇. 지금에서는 ”개체”와 ”값”을 구별없이 사용한다.

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

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

조각(slice): 인덱스의 범위로 지정된 문자열 부분.

운행법(traverse): 열의 항목을 반복적으로 훑기, 각각에 대해서는 동일한 연산을 수행.

6.14 연습문제

Exercise 6.5 아래 문자열을 파이썬 코드를 작성하세요.

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

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

Exercise 6.6 <https://docs.python.org/2.7/library/stdtypes.html#string-methods>에서 문자열 메쏘드 문서를 읽어보세요. 어떻게 동작하는가를 이해도를 확인하기 위해서 몇개를 골라 실험을 해보세요. `strip`과 `replace`가 특히 유용합니다.

문서는 문서는 좀 혼동스러울 수 있는 구문을 사용합니다. 예를 들어, `find(sub[, start[, end]])`의 꺪쇠기호는 선택(옵션) 인수를 나타냅니다. 그래서, `sub`은 필수지만, `start`은 선택 사항이고, 만약 `start`가 인수로 포함된다면, `end`는 선택이 된다.

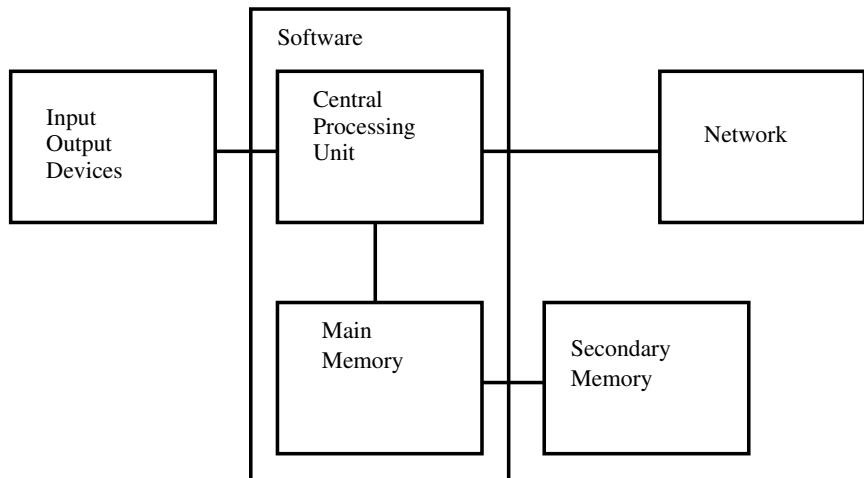
Chapter 7

파일

7.1 영속성(Persistence)

지금까지, 프로그램을 어떻게 작성하고 조건문 실행, 합수, 반복을 사용하여 중앙처리장치(CPU, Central Processing Unit)에 프로그래머의 의도를 커뮤니케이션하는 것을 학습했다. 주기억장치(Main Memory)에 어떻게 자료구조를 생성하고 사용하는지를 배웠다. CPU와 주기억장치는 소프트웨어가 동작하고 실행하는 곳이고 모든 ”생각(thinking)”이 일어나는 곳이다.

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



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

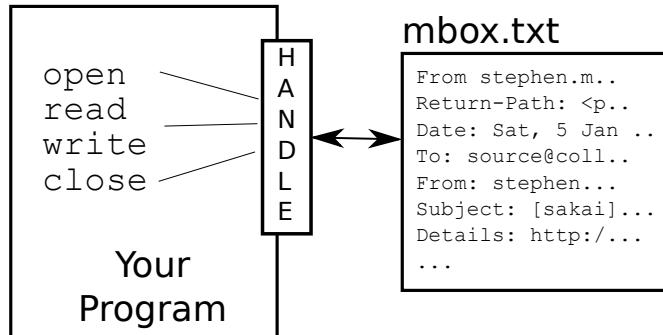
우선 텍스트 편집기로 작성한 텍스트 파일을 읽고 쓰는 것에 초점을 맞출 것이다. 나중에 데이터베이스 소프트웨어로 읽고 쓰도록 설계된 바이너리 파일인 데이터베이스와 어떻게 작업하는지를 보게 될 것이다.

7.2 파일 열기

하드 디스크의 파일을 읽거나 쓸려고 할 때, 파일을 **열여야** 한다. 파일을 여는 것은 운영체제와 커뮤니케이션하는데 운영체제는 각 파일의 데이터가 어디에 저장되었는지를 알고 있다. 여러분이 파일을 열 때, 운영체제는 파일이 존재하는 확인하고 이름으로 파일을 찾을 수 있게 요청한다. 이번 예제에서, 파일을 시작한 동일한 폴더에 저장된 `mbox.txt` 파일을 열 것이다. www.py4inf.com/code/mbox.txt에서 파일은 다운로드할 수 있다.

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

`open()` 성공하면, 운영체제는 **파일 핸들러(file handle)**을 반환한다. **파일 핸들러(file handle)**은 파일에 담겨있는 실제 데이터가 아니고, 대신에 데이터를 읽을 수 있도록 ”핸들(handle)”을 사용할 수 있도록 한다. 요청한 파일이 존재하고, 파일을 읽을 수 있는 적절한 권한이 있다면 이제 핸들이 여러분에게 주어졌다.



파일이 존재하지 않는다면, `open`은 트레이스백(traceback) 오류로 파일 열기를 실패하고, 파일에 존재하는 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 좀 더 우아하게 처리할 것이다.

7.3 텍스트 파일과 라인

파이썬 문자열이 일련의 문자의 열로 생각하는 것과 마찬가지로 텍스트 파일은 일련의 라인으로 생각할 수 있다. 예를 들어, 다음은 오픈 소스 프로젝트 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.txt에서 가능하고, 간략한 버전의 파일은 www.py4inf.com/code/mbox.txt에서 얻을 수 있다. 이들 파일은 여러개의 전자우편 메시지를 담고 있는 파일로 표준 표맷으로 되어 있다. "From"으로 시작하는 라인은 메시지 본문을 구별하고 "From:"으로 시작하는 줄은 본문 메시지의 일부다. 더 자세한 정보는 en.wikipedia.org/wiki/Mbox에서 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-\n으로 **새줄(newline)** 문자를 표현한다. 두 문자처럼 보이지만, 사실은 단일 문자다. 인터프리터에 "stuff"을 입력한 변수를 보면, 문자열에 \n가 있다. 하지만, print문을 사용하여 문자열을 출력하면, 문자열이 새줄 문자에 의해서 두줄로 나누어지는 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 문자다. 왜냐하면 새줄(newline) 문자는 한 문자이기 때문이다.

그래서, 파일의 라인을 볼 때, 라인의 끝을 표시하는 새줄(newline)로 불리는 눈에 보이지 않는 특수 문자가 각 줄의 끝에 있다고 상상할 필요가 있다.

그래서, 새줄(newline) 문자는 파일의 문자를 라인으로 분리한다.

7.4 파일 읽어오기

파일 핸들러(file handle)가 파일의 자료를 담고 있지 않지만, 파일의 각 라인을 읽고 라인수를 세기 위해서 for 루프를 사용하여 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 for문 열에 사용할 수 있다. for문 단순하게 파일의 라인 수를 세고 총 라인수를 출력한다. for문을 대략 일반어로 풀어 말하면, ”파일 핸들로 표현되는 파일의 각 라인마다, count 변수에 1을 다하세요”

open 함수가 전체 파일을 바로 읽지 않는 이유는 파일이 수기가 바이트 파일 크기를 가질 수도 있기 때문이다. open 문은 파일 크기에 관계없이 파일을 여는데 동일한 시간이 걸린다. for문이 파일로부터 자료를 읽어오는 역할을 한다.

파일을 이 같은 방식의 for문을 사용해서 읽어올 때, 파이썬은 새줄(newline) 문자를 사용해서 파일의 자료를 라인으로 쪼갠다. 파이썬은 새줄(newline) 문자 구분되는 각 라인 단위로 읽어이고, for 루프가 매번 반복할 때마다 마지막 문자로 새줄(newline)을 line 변수에 추가한다.

for 문은 데이터를 한번에 한줄씩 읽어오기 때문에 데이터를 저장하기 위해서 주기억장치의 저장공간 부족없이 매우 큰 파일을 효과적으로 읽어서 라인을 셀 수 있다.

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

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

이 예제에서, mbox-short.txt 전체 파일 내용(94,626 문자)이 변수 inp에 바로 읽혀졌다. inp에 저장된 문자열 자료의 첫 20 문자를 출력하기 위해서 문자열 쪼개기를 사용했다.

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

주기억장치가 감당해 낼 수 없는 매우 큰 크기의 파일이라면, for나 while 루프를 사용해서 파일을 쪼개서 읽는 프로그램을 작성해야 한다.

7.5 파일을 통한 검색

파일의 데이터를 검색할 때, 파일을 읽고, 대부분의 줄은 넘어가고, 특정한 조건을 만족하는 라인만 처리하는 것이 흔한 패턴이다. 간단한 검색 메카니즘을 구현하기 위해서 파일을 읽어 들이는 패턴과 문자열(methods)를 조합하여 사용한다.

예를 들어, 파일을 읽고, “From:”으로 시작하는 라인만 출력하고자 한다면, 원하는 접두사로 시작하는 라인만을 선택하기 위해서 **startswith** 문자열 메소드를 사용한다.

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

이 프로그램이 실행될 때, 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, 추가적으로 왜 빈 라인이 보이는 걸까? 왜냐하면 눈에 보이지 않는 새줄(newline) 문자 때문이다. 각 라인이 새줄(newline)으로 끝나서 print문은 새줄(newline)을 포함하는 변수 **line**의 문자열을 출력한다. 그리고 나서 **print**문이 추가로 새줄(newline)을 추가해서 결국 우리가 보기에는 두 줄 효과가 나타난다.

마지막 문자를 제외한 라인을 출력하기 위해서 라인 쪼개기(slicing)를 할 수 있지만, 좀 더 간단한 접근법은 다음과 같이 문자열 오른쪽 끝에서부터 공백을 벗겨내는 **rstrip** 메소드를 사용하는 것이다.

```
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 검색 루프(search loop)를 `continue`를 사용해서 구조화할 필요가 있다. 검색 루프의 기본 생각은 ”흥미로운” 라인을 집중적으로 찾고, ”흥미롭지 않은” 라인은 효과적으로 건너뛰는 것이다. 그리고 나서 흥미로운 라인을 찾게되면, 그 라인에서 무슨 연산을 수행하는 것이다.

흥미롭지 않은 라인은 건너 뛰는 패턴을 따르는 루프를 다음과 같이 구성할 수 있다.

```
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 시작하는 라인이어서 `continue`문을 사용해서 건너뛴다. ”흥미로운” 라인 (즉, ”From:”으로 시작하는 라인)에 대해서는 연산처리를 수행한다.

`find` 문자열 메쏘드를 사용해서 검색문자열이 어는 라인에 있는지를 찾아주는 텍스트 편집기의 검색기능을 흉내낼 수 있다. `find` 메쏘드는 다른 문자열 내부에 찾는 문자열이 있는지 찾고, 문자열의 위치를 반환하거나, 만약 문자열이 없다면 -1을 반환하기 때문에, ”@uct.ac.za”(남아프리카 케이프 타운 대학으로 부터 왔다) 문자열을 포함하는 라인을 보이기 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
...
```

7.6 사용자가 파일명을 고르게 만들기

다른 파일을 처리할 때마다 파일 코드를 편집하기를 원치 않는다. 매번 프로그램이 실행될 때마다, 파일명을 사용자가 입력하도록 요청하는 것이 좀더 유

용할 것이다. 그래서 파이썬 코드를 바꾸지 않고, 다른 파일에 대해서도 동일한 프로그램을 사용할 수 있다.

아래와 같이 `raw_input`을 사용해서 사용자로부터 파일명을 읽어서 수행하는 것이 간단하다.

```
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 일고 변수명으로 `fname`에 저장하고, 그 파일을 연다. 이제 다른 파일에 대해서 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 보세요. ”여기서 어디가 잘못될 수 있는가?” 혹은 ”우리의 친절한 사용자가 멋진 작은 프로그램을 트레이스백(traceback)을 남기고 바로 끝날 수 있게 만들어 사용자의 눈에는 뭐 좋지 않은 프로그램이라는 인상을 남기기 위해서 무엇을 할 수 있을까?

7.7 try, except, open 사용하기

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

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

품질보증 조직은 소프트웨어를 제품으로 구매하거나 작성한 프로그램에 대해 월급을 지급하는 사용자에게 프로그램이 전달되기 전까지 프로그램의 오류를 발견하는 것에 책임이 있다. 그래서 품질보증 조직은 프로그래머의 최고의 친구다.

프로그램의 오류를 찾았기 때문에, try/except 구조를 사용해서 오류를 우아하게 고쳐봅시다. open 호출이 잘못될 수 있다고 가정하고, open 호출이 실패할 때 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 함수는 프로그램을 끝낸다. 결코 돌아오지 않는 함수를 호출한 것이다. 이제 사용자 혹은 품질 보증 조직에서 옳지 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
```

open 호출을 보호하는 것은 파이썬 프로그램을 작성할 때 try, except의 적절한 사용 예제가 된다. 파이썬 방식(“Python way”)으로 무언가를 작성할 때, “파이썬스럽다(Pythonic)”이라는 용어를 사용한다. 상기 파일을 여는 것은 파이썬스러운 방식의 좋은 예가 된다고 말한다.

파이썬에 좀더 자신감이 불게되면, 다른 파이썬 프로그래머와 동일한 문제에 대해서 두 가지 동치하는 해답을 가지고 있는 것이 좀더 “파이썬스러운지”에 대한 현답을 찾는데 관여하게 된다.

“좀더 파이썬스럽게” 되는 이유는 프로그램ming이 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.

7.8 파일에 쓰기

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

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

파일이 이미 존재한다면, 쓰기 모드에서 여는 것은 과거 데이터를 모두 지워버리고, 파일이 깨끗한 다시 시작되니 주의가 필요합니다. 만약 파일이 존재하지 않는다면, 새로운 파일이 생성됩니다.

파일 핸들 개체의 write 메소드는 데이터를 파일에 저장합니다.

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

다시, 파일 개체는 어디에 마지막 포인터가 있는지 위치를 추적해서, 만약 write 메소드를 다시 호출하게 되면, 새로운 데이터를 파일 끝에 추가합니다.

라인을 끝내고 싶을 때 명시적으로 새줄(newline) 문자를 삽입해서 파일에 쓰도록 라인 끝을 관리하도록 꼭 확인해야 합니다.

print문은 자동적으로 새줄(newline)을 추가하지만, write 메소드는 자동적으로 새줄(newline)을 추가하지는 않습니다.

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

파일에 쓰기가 끝났을 때, 파일을 필히 닫아야 합니다. 파일을 닫는 것은 데이터의 마지막 비트까지 디스크에 물리적으로 쓰여져서, 전원이 나가더라도 자료가 유실되지 않도록 하는 역할을 합니다.

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

파일 읽기로 연 파일을 닫을 수 있지만, 몇개의 파일을 열어논다면 약간 단정치 못한 것으로 끝날 수 있습니다. 왜냐하면 프로그램이 종료될 때 열린 모든 파일을 닫도록 파일이 자동으로 확인합니다. 파일에 쓰기를 할 때는, 파일을 명시적으로 닫아서 다른 일들이 발생할 여지를 없애야 합니다.

7.9 디버깅

파일을 읽고 쓸 때, 공백으로 문제에 종종 봉착합니다. 이런 종류의 오류는 공백, 탭, 새줄(newline)이 눈에 보이지 않기 때문에 디버그하기가 쉽지 않습니다.

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

내장함수 repr이 도움이 될 수 있습니다. 인수로 임의의 개체를 잡아 개체의 문자열 표현으로 반환합니다. 문자열의 공백문자는 역슬래쉬 열로 나타냅니다.

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

디버깅에 도움이 될 수 있습니다.

여러분이 봉착하는 또 다른 문제는 다른 시스템은 라인의 끝을 표기하기 위해서 다른 문자를 사용합니다. 어떤 시스템은 \n으로 새줄(newline)을 표기하고, 다른 시스템은 \r으로 반환 문자(return character)를 사용합니다. 둘다 사용하는 시스템도 있습니다. 파일을 다른 시스템으로 옮긴다면, 이러한 불일치가 문제를 야기합니다.

대부분의 시스템에는 A 포맷에서 B 포맷으로 변환하는 응용프로그램이 있습니다. wikipedia.org/wiki/Newline에서 응용프로그램을 찾을 수 있고, 좀 더 많은 것을 읽을 수 있다. 물론, 여러분이 직접 프로그램을 작성할 수도 있다.

7.10 용어정의

잡기(catch): try와 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.

7.11 연습문제

Exercise 7.1 파일을 읽고 한줄씩 파일의 내용을 모두 대문자로 출력하는 프로그램을 작성하세요. 프로그램을 실행하면 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
```

You can download the file from www.py4inf.com/code/mbox-short.txt

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. 라인의 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.txt과 mbox-short.txt 파일에 프로그램을 시험하세요.

Exercise 7.3 때때로, 프로그래머가 지루하거나, 약간의 재미를 목적으로, 프로그램에 무해한 **부활절 달걀(Easter Egg)**를 넣습니다. ([en.wikipedia.org/wiki/Easter_egg_\(media\)](https://en.wikipedia.org/wiki/Easter_egg_(media))) 사용자가 파일명을 입력하는 프로그램을 변형시켜, ’na na boo boo’로 파일명을 정확하게 입력했을 때, 재미난 메시지를 출력하는 프로그램을 작성하세요. 파일이 존재하거나, 존재하지 않는 모든 파일에 대해서 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.txt
File cannot be opened: missing.txt
```

```
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.

Chapter 8

리스트

8.1 리스트는 열이다.

문자열처럼, **리스트(list)**는 일련의 값이다. 문자열에서, 같은 문자지만, 리스트에서는 임의의 형(type)이 될 수 있다. 리스트의 값은 **요소(elements)**나 때때로 **항목(items)**으로 불린다.

신규 리스트를 생성하는 방법은 여러가지다. 가장 간단한 방법은 꺼쇠 괄호([와])로 요소를 감싸는 것이다.

```
[10, 20, 30, 40]  
['crunchy frog', 'ram bladder', 'lark vomit']
```

첫번째 예제는 네 개의 정수 리스트다. 두번째 예제는 3개의 문자열 리스트다. 문자열의 요소는 같은 형(type)일 필요는 없다. 다음의 리스트는 문자열, 부동 소수점 숫자, 정수, (아!) 또 다른 리스트를 담고 있다.

```
['spam', 2.0, 5, [10, 20]]
```

또 다른 리스트 내부에 리스트는 **중첩(nested)**되어 있다.

어떤 요소도 담고 있지 않는 리스트는 **빈 리스트(empty list)**라고 부르고, 빈 꺼쇠 괄호("[]")로 생성할 수 있다.

예상했듯이, 리스트 값을 변수에 할당할 수 있다.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']  
>>> numbers = [17, 123]  
>>> empty = []  
>>> print cheeses, numbers, empty  
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

8.2 리스트는 변경가능하다.

리스트의 요소에 접근하는 구문은 문자열의 문자에 접근하는 것과 동일한 꺼쇠 괄호 연산자다. 꺼쇠 괄호 내부의 표현식은 인덱스를 명시한다. 인덱스는 0에서부터 시작한다.

```
>>> print cheeses[0]
Cheddar
```

문자열과 달리, 리스트의 항목의 순서를 바꾸거나, 리스트에 새로운 항목을 재할당할 수 있기 때문에 리스트는 변경가능하다. 꺾쇠 괄호 연산자가 할당문의 왼쪽편에 나타날 때, 새로 할당될 리스트의 요소를 나타낸다.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

리스트 numbers의 1번째 요소는 123 값을 가지고 있으나, 이제는 5 값을 가진다.

리스트를 인덱스와 요소의 관계로 생각할 수 있다. 이 관계를 **매핑(mapping)**이라고 부른다. 각각의 인덱스는 요소중의 하나에 대응("maps to")된다.

리스트 인덱스는 문자열 인덱스와 같은 방식으로 동작한다.

- 임의의 정수 표현식은 인덱스로 사용될 수 있다.
- 존재하지 않는 요소를 읽거나 쓰려고 하면, IndexError가 발생한다.
- 인덱스가 음의 값이면, 리스트의 끝에서부터 역으로 센다.

in 연산자도 또한 리스트에서 동작한다.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

8.3 리스트 운행법

리스트의 요소를 운행하는 가장 흔한 방법은 for문을 사용하는 것이다. 구문은 문자열에서 사용한 것과 동일하다.

```
for cheese in cheeses:
    print cheese
```

리스트의 요소를 읽기만 한다면 이것만으로 잘 동작한다. 하지만, 리스트의 요소를 쓰거나, 갱신하는 경우, 인덱스가 필요하다. 리스트의 요소를 쓰거나 갱신하는 흔한 방법은 range와 len 함수를 조합하는 것이다.

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

상기 루프는 리스트를 운행하고 각 요소를 갱신한다. len함수는 리스트의 요소의 갯수를 반환한다. range 함수는 0에서 $n - 1$ 까지 리스트 인덱스를 반환한다.

여기서, n 은 리스트의 길이다. 매번 루프가 반복될 때마다, i 는 다음 요소의 인덱스를 얻는다. 몸통 부분의 할당문은 i 를 사용해서 요소의 옛값을 일고 새값을 할당한다.

빈 리스트의 `for`문은 결코 몸통부분을 실행하지 않는다.

```
for x in empty:
    print 'This never happens.'
```

리스트가 또 다른 리스트를 담을 수 있지만, 중첩된 리스트는 여전히 하나의 요소로 센다. 다음 리스트의 길이는 4이다.

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

8.4 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 횟수 만큼 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 반복 한다.

8.5 리스트 쪼개기(List slices)

쪼개는 연산자는 리스트에도 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, 쪼개기는 처음부터 시작한다. 두 번째 인덱스를 생략하면, 쪼개기는 끝까지 간다. 그래서 양쪽의 인덱스를 생략하면, 쪼개기는 전체 리스트를 복사한다.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

리스트는 변경이 가능하기 때문에 리스트를 접고, 돌리고, 훼손하는 연 \times 나을 수행하기 전에 사본을 만드는 것이 유용하다.

할당문의 왼편의 쪼개기 연산자는 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']
```

8.6 리스트 메쏘드

파이썬은 리스트에 연산하는 메쏘드를 제공한다. 예를 들어, append 메쏘드는 리스트 끝에 신규 요소를 추가한다.

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print t
['a', 'b', 'c', 'd']
```

extend 메쏘드는 인수로 리스트를 받아 모든 요소를 리스트에 추가한다.

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print t1
['a', 'b', 'c', 'd', 'e']
```

상기 예제는 t2 리스트를 변경없이 놓아둔다.

sort 메쏘드는 낮음에서 높음으로 리스트의 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 None을 반환한다. 우연히 t = t.sort() 이렇게 작성한다면, 결과에 실망할 것이다.

8.7 요소 삭제

리스트에서 요소를 삭제하는 몇 가지 방법이 있다. 리스트 요소 인덱스를 알고 있다면, pop 메쏘드를 사용한다.

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print t
['a', 'c']
>>> print x
b
```

`pop` 메소드는 리스트를 변경하여 제거된 요소를 반환한다. 인덱스를 주지 않으면, 마지막 요소를 지우고 반환한다.

요소에서 제거된 값이 필요없다면, `del` 연산자를 사용할 수 있다.

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
['a', 'c']
```

인덱스가 아닌 제거할 요소를 알고 있다면, `remove` 메소드를 사용할 수 있다.

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print t
['a', 'c']
```

`remove` 메소드의 반환값은 `None`이다.

하나 이상의 요소를 제거하기 위해서, 쪼개기 인덱스(slice index)와 `del`을 사용한다.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print t
['a', 'f']
```

마찬가지로, 쪼개기는 두 번째 인덱스를 포함하지 않는 두 번째 인덱스까지의 모든 요소를 선택한다.

8.8 리스트와 함수

루프를 작성하지 않고 리스트를 빠를게 살펴볼 수 있도록 리스트에 적용할 수 있는 많은 내장함수가 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 때, `sum()` 함수는 동작한다. `max()`, `len()`, 등등의 함수는 문자열 리스트나, 비교 가능한 다른 형(type)의 리스트에 사용할 수 있다.

리스트를 사용해서, 사용자가 입력한 숫자 목록의 평균을 계산하는 앞서 작성한 프로그램을 다시 작성할 수 있다.

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
```

이 프로그램에서, `count` 와 `sum` 변수를 사용해서 반복적으로 사용자가 숫자를 입력하면 값을 저장하고, 지금까지 사용자가 입력한 누적 합계를 계산하는 것이다.

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 전에 빈 리스트를 생성하고, 매번 숫자를 입력할 때, 숫자를 리스트에 추가한다. 프로그램 마지막에 간단하게 리스트의 합계를 계산하고, 평균을 출력하기 위해서 입력한 숫자 개수로 나누었다.

8.9 리스트와 문자열

문자열은 일련의 문자이고, 리스트는 일련의 값이다. 하지만 리스트의 문자는 문자열과 같지는 않다. 문자열에서 리스트의 문자로 변환하기 위해서, `list`를 사용한다.

```
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

list는 내장함수의 이름이기 때문에, 변수명으로 사용하는 것을 피해야 한다. 1의 사용을 1처럼 보이기 때문에 피한다. 그래서, t를 사용하였다.

list 함수는 문자열을 각각의 문자로 쪼갠다. 문자열을 단어로 쪼개려면, split 메쏘드를 사용할 수 있다.

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
>>> print t[2]
the
```

split 메쏘드를 사용해서 문자열을 리스트의 토큰으로 쪼개기만 하면, 인덱스 연산자('[]')를 사용하여 리스트의 특정한 단어를 볼 수 있다.

구분자(**delimiter**)가 단어의 경계로 어느 문자를 사용할지를 지정하는데, split 메쏘드를 호출할 때 두 번째 선택 인수로 사용할 수 있다. 다음 예제는 구분자로 하이픈('-')을 사용한다.

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

join 메쏘드는 split 메쏘드의 역이다. 문자열 리스트를 받아 리스트 요소를 결합한다. join은 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 join 메쏘드는 단어 사이에 공백을 넣는다. 공백없이 문자열을 결합하기 위해서, 구분자로 빈 문자열 ''을 사용한다.

8.10 라인을 파싱하기

파일을 읽을 때 통상, 단지 전체 라인을 출력하는 것 말고 뭔가 다른 것을 하고자 한다. 종종 ”흥미로운 라인을” 찾아서 라인을 파싱하여 흥미로운 부분(parse)을 찾고자 한다. “From ”으로 시작하는 라인에서 요일을 찾고자 하면 어떨까?

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

split 메쏘드는 이런 종류의 문제에 직면했을 때, 매우 효과적이다. ”From ”으로 시작하는 라인을 찾고 split 메쏘드로 파싱하고 라인의 흥미로운 부분을 출력하는 작은 프로그램을 작성할 수 있다.

```

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 형태로 continue 문을 if문과 동일한 라인에 놓았다. if 문의 축약 형태는 continue 문을 다음 라인에 들여쓰기를 한 것과 동일하다.

프로그램은 다음을 출력한다.

```

Sat
Fri
Fri
Fri
...

```

나중에, 작업할 라인을 선택하고, 탐색하는 정확한 비트(bit) 수준의 정보를 찾아내기 위해서 어떻게 해당 라인에서 뽑아내는 정교한 기술에 대해서 배울 것이다.

8.11 개체와 값(value)

아래 할당문을 실행한다.

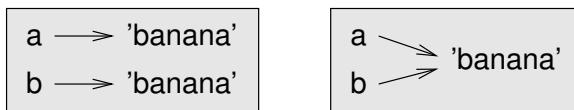
```

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 개체를 참조하는 것이다. 두 번째 경우는 같은 개체를 참조하는 것이다.

두 변수가 동일한 개체를 참조하는지를 확인하기 위해서, is 연산자가 사용된다.

```

>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True

```

이 경우, 파이썬은 하나의 문자열 개체를 생성하고 a 와 b 모두 동일한 개체를 참조한다.

하지만, 두개의 리스트를 생성할 때, 두 개의 개체를 얻게된다.

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

상기의 경우, 두개의 리스트는 동등하다고 말할 수 있다. 왜냐하면 동일한 요소를 가지고 있기 때문이다. 하지만, 같은 개체는 아니기 때문에 동일하지는 않다. 두개의 개체가 동일하다면, 두 개체는 또한 등등하다. 하지만, 동등하다고 해서 반드시 동일하지는 않다.

지금까지 ”개체”와 ”값(value)”를 구분없이 사용했지만, 개체가 값을 가진다고 말하는 것에는 좀더 정확하다. `a = [1, 2, 3]` 을 실행하면, `a` 는 리스트 개체로 특별한 일련의 요소값을 가진다. 만약 또 다른 리스트가 동일한 요소를 가진다면, 그 리스트는 같은 값을 가진다고 말한다.

8.12 에일리어싱(Aliasing)

`a`가 개체를 참조하고, `b = a` 할당하다면, 두 변수는 동일한 개체를 참조한다.

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

개체 변수의 연관을 참조(reference)라고 한다. 상기의 경우 동일한 개체를 두 개의 참조가 있다.

하나 이상의 참조를 가진 개체는 한개 이상의 이름을 가져서 개체가 **에일리어스 (aliased)** 되었다고 한다.

만약 에일리어스된 개체가 변경 가능하면, 변화의 여파는 다른 개체에도 파급된다.

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

이런 행동이 유용하기도 하지만, 오류를 발생하기도 쉽다. 일반적으로, 변경 가능한 개체(mutable object)로 작업할 때 에일리어싱을 피하는 것이 안전하다.

문자열 같은 변경 불가능한 개체에 에일리어싱은 그다지 문제가 되지 않는다.

```
a = 'banana'
b = 'banana'
```

상기 예제에서, `a` 와 `b`가 동일한 문자열을 참조하든 참조하지 않든 거의 차이가 없다.

8.13 리스트 인수

리스트를 함수에 인수로 전달할 때, 함수는 리스트의 참조를 얻는다. 만약 함수가 리스트 매개 변수를 변경한다면, 호출자는 변화를 보게된다. 예를 들어, `delete_head`는 리스트로부터 첫 요소를 제거한다.

```
def delete_head(t):
    del t[0]
```

여기 어떻게 `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 개체에 대한 애일리어스(alienes)다.

리스트를 변경하는 연산자와 신규 리스트를 생성하는 연산자를 구별하는 것은 중요하다. 예를 들어, `append` 메소드는 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, 다음의 함수는 리스트의 처음(head)을 삭제하지 않는다.

```
def bad_delete_head(t):
    t = t[1:]          # 틀림 (WRONG) !
```

슬라이스 연산자는 새로운 리스트를 생성하지만, 인수로 전달된 리스트에는 어떠한 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. 여기 어떻게 사용되었는지 예시가 있다.

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print rest
['b', 'c']
```

Exercise 8.1 리스트를 인수로 받아 리스트를 변경하여, 첫번째 요소와 마지막 요소를 제거하고 None을 반환하는 chop 함수를 작성하게요.

그리고 나서, 리스트를 인수로 받아 처음과 마지막 요소를 제외한 나머지 요소를 새로운 리스트로 반환하는 middle 함수를 작성하세요.

8.14 디버깅

부주의한 리스트의 사용이나 다른 변경가능한 재체를 사용하는 경우 오랜 시간의 디버깅으로 이끌 수 있다. 몇몇 일반적인 함정과 회피하는 방법을 소개한다.

1. 대부분의 리스트 메쏘드는 인수를 변경하고, None을 반환한다. 이는 새로운 문자열을 반환하고 원 문자열은 그대로 두는 문자열과 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) !
```

sort 메쏘드는 None을 반환하기 때문에, 리스트 t에 수행한 다음 연산은 수행되지 않는다.

리스트를 사용한 메쏘드와 연산자를 사용하기 전에, 문서를 주의깊게 읽고, 인터랙티브 모드에서 시험하는 것을 권한다. 리스트가 문자열과 같은 다른 열과 공유하는 메쏘드와 연산자는 docs.python.org/lib/typesseq.html에 문서화되어 있다. 변경가능한 열에만 적용되는 메쏘드와 연산자는 docs.python.org/lib/typesseq-mutable.html에 문서화되어 있다.

2. 관용구를 선택하고 고수하라.

리스트와 관련된 문제의 일부는 리스트를 가지고 할 수 있는 것이 너무 많다는 것이다. 예를 들어, 리스트에서 요소를 제거하기 위해서, pop, remove, del, 혹은 쪼개기 할당(slice assignment)도 사용할 수 있다. 요소를 추가하기 위해서 append 메쏘드나 + 연산자를 사용할 수 있다. 하지만 다음이 맞다는 것을 잊지 마세요.

```
t.append(x)
t = t + [x]
```

하지만, 다음은 잘못됐다.

And these are wrong:

```
t.append([x])          # 틀림 (WRONG) !
t = t.append(x)        # 틀림 (WRONG) !
t + [x]                # 틀림 (WRONG) !
t = t + x              # 틀림 (WRONG) !
```

인터랙티브 모드에서 각각을 연습해 보해서 제대로 이해하고 있는지 확인해 보세요. 마지막 두개만이 실행 오류를 하고, 다른 세가지는 모두 작동하지만, 잘못된 것을 수행함을 주목하세요.

3. 에일리어싱을 피하기 위해서 사본 만들기.

인수를 변경하는 `sort` 같은 메쏘드를 사용하지만, 원 리스트도 보관하길 원한다면, 사본을 만들 수 있다.

```
orig = t[:]
t.sort()
```

상기 예제에 원 리스트는 그대로 둔 상태로 새로 정렬된 리스트를 반환하는 내장함수 `sorted`를 사용할 수 있다. 하지만 이 경우에는, 변수명으로 `sorted`를 사용하는 것을 피해야 한다.

4. 리스트, split, 파일

파일을 읽고 파싱할 때, 프로그램을 중단할 수 있는 입력값을 마주칠 수 있는 수많은 기회가 있다. 그래서 파일을 훑어 ”건초더미에서 바늘”을 찾는 프로그램을 작성할 때, **가디언 패턴(guardian pattern)**을 다시 살펴보는 것은 좋은 생각이다. 파일의 라인에서 요일을 찾는 프로그램을 다시 살펴보자.

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

각 라인을 단어로 나누었기 때문에, `startswith`를 사용하지 않고, 라인에 관심있는 단어가 있는지 살펴보기 위해서 단순히 각 라인의 첫 단어를 살펴본다. 다음과 같이 ”From”이 없는 라인을 건너 뛰기 위해서 `continue`문을 사용한다.

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    if words[0] != 'From' : continue
    print words[2]
```

프로그램이 훨씬 간단하고, 파일 끝의 새줄(newline)을 제거하기 위해서 `rstrip`을 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 Sat를 출력하고 나서 트레이스백 오류(traceback error)로 프로그램이 정상 동작에 실패한다. 무엇이 잘못되었는가? 어딘가 엉망이 된 데이터가 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 앞에 적절하고, 프로그램 실패를 야기하는 데이터를 출력한다.

이 접근법은 많은 라인을 출력하지만, 최소한 프로그램에 손에 잡히는 단서를 최소한 준다. 그래서 words를 출력하는 출력문을 5번째 라인 앞에 추가한다. "Debug:"를 접두어로 라인에 추가하여, 정상적인 출력과 디버그 출력과 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 트레이스백(traceback)을 보고 트레이스백 바로 앞에서 무엇이 생겼는지를 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 라인을 split 쪼개서 단어로 만들 때 얻는 리스트 단어를 출력한다. 프로그램이 실패할 때, 리스트의 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 그것을 생각하지 못했을까? 첫 단어(word[0])가 "From"과 일치하는지를 코드가 살펴볼 때, "index out of range"가 발생한다.

물론, 첫 단어가 없다면 첫 단어 점검을 회피하는 **가디언 코드(guardian code)**를 넣기 최적의 장소이기는 하다. 코드를 보호하는 방법은 많다. 첫 단어를 살펴보기 전에 단어의 갯수를 확인하는 방법을 여기서는 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 필요가 있는 경우를 대비해서, `print`문을 제거하는 대신에 `print`문을 주석 처리한다.

단어가 '0'인지를 살펴보고 만약에 '0'이면 파일의 다음 라인으로 건너뛰도록 `continue`문을 사용하는 **가디언 문(guardian statement)**을 추가한다.

두 개의 `continue`문이 ”흥미롭고” 데이터를 처리할 라인의 집합에 좀 더 정제하도록 돋는 것으로 생각할 수 있다.

단어가 없는 라인은 ”흥미없어서” 다음 라인으로 건너뛴다. 첫 단어에 ”From”이 없는 라인도 ”흥미없어서” 건너뛴다.

변경된 프로그램은 성공적으로 실행되어서, 아마도 올바를게 작성된 것으로 보인다. **가디언 문(guardian statement)**이 `words[0]`의 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 적절하게 보호되지 않은지를 생각해 보세요. 프로그램이 실패하도록 텍스트 파일을 구성할 수 있는지 살펴보세요. 그리고 나서, 라인이 적절하게 보호되고 프로그램을 변경하세요. 그리고, 새로운 텍스트 파일을 잘 다룰 수 있도록 시험을 하세요.

Exercise 8.3 두 `if`문 없도록 상기 예제의 가디언 코드 (guardian code)를 다시 작성하세요. 대신에 단일 `if`문과 `and` 논리 연산자를 사용하는 복합 논리 표현식을 사용하세요.

8.15 용어정의

에일리어싱(aliasing): 하나 혹은 그 이상의 변수가 동일한 객체를 참조하는 상황.

구분자(delimiter): 문자열이 어디서 쪼개져야 할지를 표기하기 위해서 사용되는 문자나 문자열.

요소(element): 리스트나 혹은 다른 열의 값의 하나로 항목(item)이라고도 한다.

동등한(equivalent): 같은 값을 가짐.

인덱스(index): 리스트의 요소를 지칭하는 정수 값.

동일한(identical): 동등을 함축하는 같은 개체임.

리스트(list): 일련의 값.

리스트 운행법(list traversal): 리스트의 각 요소를 순차적으로 접근함.

중첩 리스트(nested list): 또 다른 리스트의 요소인 리스트.

개체(object): 변수가 참조할 수 있는 무엇. 개체는 형(type)과 값(value)을 가진다.

참조(reference): 변수와 값의 연관.

8.16 연습문제

Exercise 8.4 www.py4inf.com/code/romeo.txt에서 파일 사본을 다운로드 받으세요.

romeo.txt 파일을 열어, 한 줄씩 읽어들이는 프로그램을 작성하세요. 각 라인마다 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 우편함 데이터를 읽어 들이는 프로그램을 작성하세요. "From"으로 시작하는 라인을 발견했을 때, 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" 라인의 갯수를 세고, 끝에 갯수를 출력한다.

여기 몇 줄을 삭제한 좋은 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 끝나는 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
```

Chapter 9

딕셔너리(Dictionaries)

딕셔너리(dictionary)는 리스트 같지만 좀 더 일반적이다. 리스트에서 위치(인덱스)는 정수이지만, 딕셔너리에서는 인덱스가 임의의 형(type)이 될 수 있다.

딕셔너리를 키(keys)라고 불리는 인덱스 집합에서 값(value) 집합으로 사상하는 것으로 생각할 수 있다. 각 키는 값에 대응한다. 키와 값의 연관은 키-밸류 페어(key-value pair)라고 부르고, 종종 항목(item)으로도 부른다.

한 예로서, 영어에서 스페인 단어에 대응하는 사전을 만들 것이다. 키와 값은 모두 문자열이다.

dict 함수는 항목이 전혀 없는 사전을 새로이 생성한다. dict는 내장함수명이어서, 변수명으로 사용하는 것을 피해야 한다.

```
>>> eng2sp = dict()  
>>> print eng2sp  
{}
```

구불구불한 괄호 {}는 빈 딕셔너리를 나타낸다. 딕셔너리에 항목을 추가하기 위해서 꺪쇠 괄호를 사용한다.

```
>>> eng2sp['one'] = 'uno'
```

상기 라인은 키 'one'에서 값 'uno'로 대응하는 항목을 생성한다. 딕셔너리를 다시 출력하면, 키와 값 사이에 콜론(:)을 가진 키-밸류 페어(key-value pair)를 볼 수 있다.

```
>>> print eng2sp  
{'one': 'uno'}
```

출력 형식이 또한 입력 형식이다. 예를 들어, 세 개 항목을 가진 신규 딕셔너리를 생성할 수 있다.

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

eng2sp을 출력하면, 놀랄 것이다.

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

키-밸류 페어(key-value pair)의 순서가 같지 않다. 사실 동일한 사례를 여러분의 컴퓨터에 입력하면, 다른 결과를 얻게 된다. 일반적으로, 딕셔너리의 항목의 순서는 예측 가능하지 않다.

딕셔너리의 요소가 결코 정수 인덱스로 색인되지 않아서 문제는 아니다. 대신에, 키를 사용해서 상응하는 값을 찾을 수 있다.

```
>>> print eng2sp['two']
'dos'
```

'two' 키는 항상 값 'dos'에 상응되어서 항목의 순서는 문제가 되지 않는다.

만약 키가 딕셔너리에 존재하지 않으면, 예외 오류가 발생한다.

```
>>> print eng2sp['four']
KeyError: 'four'
```

`len`함수를 딕셔너리에 사용해서, 키-밸류 페어(key-value pair)의 개수를 반환한다.

```
>>> len(eng2sp)
3
```

`in` 연산자가 딕셔너리에 작동되는데, 딕셔너리에 키(key)로 무언가 있는지 알려준다. (값으로 나타나는 것은 충분히 좋지는 않다.)

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

딕셔너리에 값으로 무언가 있는지 알기 위해서, 리스트로 값을 반환하고 나서 `in` 연산자를 사용하는 `values` 메쏘드를 사용한다.

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

`in` 연산자는 리스트와 딕셔너리에 대해 다른 알고리즘을 사용한다. 리스트는 선형 검색 알고리즘을 사용한다. 리스트가 길어짐에 따라 검색 시간은 리스트의 길이에 비례하여 길어지게 된다. 딕셔너리에 대해서 파이썬은 해쉬 테이블(**hash table**)로 불리는 놀라운 특성을 가진 알고리즘을 사용한다. `in` 연산자는 얼마나 많은 항목이 딕셔너리에 있는지에 관계없이 대략 동일한 시간이 소요된다. 왜 해쉬 함수가 마술 같은지에 대해서는 설명하지 않지만, [wikipedia.org/wiki/Hash_table](https://en.wikipedia.org/wiki/Hash_table)에 좀더 많은 것을 읽을 수 있다.

Exercise 9.1 `words.txt`의 단어를 읽어서 딕셔너리에 키로 저장하는 프로그램을 작성하세요. 값이 무엇이든지 상관없습니다. 딕셔너리에 문자열을 확인하는 가장 빠른 방법으로 `in` 연산자를 사용할 수 있습니다.

9.1 카운터 집합으로의 딕셔너리

문자열이 주어진 상태에서, 각 문자가 얼마나 나타나는지를 센다고 가정합시다. 몇 가지 방법이 아래에 있습니다.

1. 26개 변수를 알파벳 문자 각각에 대해 생성합니다. 그리고 나서 문자열을 훑고 아마도 연쇄 조건문을 사용하여 해당하는 카운터를 하나씩 증가합니다.
2. 26개 요소를 가진 리스트를 생성합니다. 내장함수 `ord`를 사용해서 각 문자를 숫자로 변환합니다. 리스트안에 인덱스로서 숫자를 사용하고 카운터를 증가합니다.
3. 키로 문자, 카운터로 해당 값을 가지는 딕셔너리를 생성합니다. 처음 문자를 본다면, 딕셔너리에 항목으로 추가합니다. 추가한 후에 존재하는 항목의 값을 증가합니다.

상기 3개의 선택사항은 동일한 연산을 수행하지만, 각각은 다른 방식으로 연산을 구현합니다.

구현(implementation)은 연산(computation)을 수행하는 방법이다. 어떤 구현방법이 다른 것보다 좋다. 예를 들어, 딕셔너리 구현의 장점은 사전에 어느 문자가 문자열에 나타날지를 알지 못하고, 나타날 문자에 대한 공간만 준비하면 된다는 것이다.

여기 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
```

카운터 혹은 빈도에 대한 통계 용어인 **히스토그램(histogram)**를 효과적으로 연산한다.

`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.

여기 프로그램의 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`'은 2회 등등 나타냄을 보여준다.

딕셔너리에는 키와 디폴트 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
```

카운팅 루프를 단순화하는 get메쏘드를 사용하는 것은 파이썬에서 흔히 사용되는 ”속어(idiom)”가 되고, 책의 끝까지 많이 사용할 것이다. if문을 가진 프로그램과 get메쏘드를 사용한 루프를 가진 in 연산자 프로그램을 시간을 가지고 비교해 보세요. 동일한 연산을 수행하지만, 하나는 더 간결합니다.

9.2 딕셔너리와 파일

딕셔너리의 흔한 사용법 중의 하나는 파일에 단어의 빈도수를 세는 것이다. http://shakespeare.mit.edu/Tragedy/romeoandjuliet/romeo_juliet.2.2.html 사이트에서 로미오와 줄리엣(*Romeo and Juliet*) 텍스트 파일에서 시작합시다.

처음 연습으로 구두점이 없는 짧고 간략한 텍스트 버전을 사용합니다. 나중에 구두점이 포함된 전체 텍스트로 작업을 할 것입니다.

```
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 작성합니다.

두개의 for 루프를 사용합니다. 외곽 루프는 파일의 라인을 읽고, 내부 루프는 라인의 각 단어에 대해 반복합니다. 하나의 루프는 외곽 루프가 되고, 또 다른 루프는 내부 루프가 되어서 **중첩루프(nested loops)**라고 불리는 패턴의 예입니다.

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 있습니다.

두 중첩 루프의 조합이 입력 파일의 모든 라인의 모든 단어의 빈도수를 세는 것을 확인합니다.

```

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 모든 단어의 빈도수를 해쉬 순으로 출력합니다. romeo.txt 파일은 www.py4inf.com/code/romeo.txt에서 다운로드 가능합니다.

```

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 것은 약간 불편해서, 좀더 도움이 되는 출력을 만드는 파이썬 코드 추가가 필요하다.

9.3 반복과 딕셔너리

for문에 열로서 딕셔너리를 사용한다면, 딕셔너리의 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.

앞서 설명한 다양한 루프 숙어를 구현하기 위해서 이 패턴을 사용한다. 예를 들어 딕셔너리에 10보다 큰 값을 가진 항목을 모두 찾아내기를 원한다면, 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 딕셔너리의 키(keys) 반복해서, 해당하는 키에 상응하는 값(value)을 구해내기 위해 인덱스 연산자를 사용해야 한다. 여기 출력값이 있다.

```

jan 100
annie 42

```

10 이상 값만 가진 항목만 볼 수 있다.

알파벳 순으로 키를 출력하고자 한다면, 딕셔너리 개체의 keys 메쏘드를 사용해서 딕셔너리 키 리스트를 생성한다. 그리고 나서 리스트를 정렬하고, 정렬된 리스트를 훑고, 아래와 같이 정렬된 순서로 키/밸류 페어를 출력하도록 각 키를 조회한다.

```

counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = counts.keys()
print lst
lst.sort()
for key in lst:
    print key, counts[key]

```

여기 출력결과가 있다.

```

['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100

```

keys 메쏘드로부터 얻은 정렬되지 않은 키 리스트가 있고, for 루프로 정렬된 키/밸류 페어가 있다.

9.4 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 처리해서, “soft!”, “soft”는 다른 단어가 되고 각 단어에 대해서 구별되는 딕셔너리 항목을 생성한다.

파일에 대문자가 있어서, “who”와 “Who”를 다른 단어, 다른 빈도수를 가진 것으로 처리한다.

lower, punctuation, translate 문자열 메쏘드를 사용해서 이러한 문제를 해결할 수 있다. translate 메쏘드가 가장 적합하다. translate 메쏘드에 대한 문서는 다음과 같다.

```
string.translate(s, table[, deletechars])
```

Delete all characters from s that are in deletechars (if present), and then translate the characters using table, which must be a 256-character string giving the translation for each character value, indexed by its ordinal. If table is None, then only the character deletion step is performed.

table을 명세하지는 않을 것이고, deletechars 매개변수를 사용해서 모든 구두점을 삭제할 것이다. 파이썬이 ”구두점”으로 간주하는 문자 리스트를 출력하게 할 것이다.

```
>>> import string
>>> string.punctuation
'!"#$%&\ \ ()*+,.-./:;<=>?@[\\\]^_`{|}`'
```

프로그램에 다음과 같은 수정을 했습니다.

```
import string # New Code

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) # New Code
    line = line.lower() # New Code
    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 수정했습니다. 나머지 프로그램은 변경된게 없습니다. 파이썬 2.5 이전 버전에는 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 많이 혼한 자료 분석 문제에 대해서 내장 기능을 가지고 있는 것을 깨닫는 것이다. 시간이 지남에 따라, 충분한 예제 코드를 보고 충분한 문서를 읽어서 여러분의 작업을 편하게 할 수 있는 다른 사람이

이미 작성한 코드가 존재하는지를 살펴보기 위해서 어디를 찾아봐야 하는지를 알게 될 것이다.

다음은 출력결과의 축약 버전이다.

```
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 찾고 자는 하는 것을 찾았으나 파이썬 **튜플(tuples)**에 대해서 학습할 필요가 있다. 튜플을 학습하기 위해서 다시 이 예제를 살펴볼 것이다.

9.5 디버깅

점점 더 큰 데이터로 작업함에 따라, 수작업으로 데이터를 확인하거나 출력을 통해서 디버깅을 하는 것이 어려울 수 있다. 큰 데이터를 디버깅하는 몇 가지 제안이 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, n을 오류를 재현하는 가장 작은 값으로 줄여라. 오류를 찾고 수정해 나감에 따라 점진적으로 늘려나가라.

요약값과 형을 확인하라(Check summaries and types): 전체 데이터를 출력하고 검증하는 대신에 데이터의 요약하여 출력하는 것을 생각하라. 예를 들어, 딕셔너리의 항목의 숫자 혹은 리스트 숫자의 총계

실행 오류(runtime errors)의 일반적인 원인은 올바른 형(right type)이 아니기 때문이다. 이런 종류의 오류를 디버깅하기 위해서, 값의 형을 출력하는 것으로 종종 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 두 가지 다른 연산의 결과를 비교해서 일치하는지를 살펴보는 것이다. ”일치성 검사(consistency check)”라고 부른다.

고급 출력(Pretty print the output): 디버깅 출력을 서식화하는 것은 오류를 발견하는 것을 용이하게 한다.

다시 한번, 발판(scaffolding)을 만드는데 들인 시간은 디버깅에 소비되는 시간을 줄일 수 있다.

9.6 용어정의

딕셔너리(dictionary): 키(key)에서 해당 값으로 맵핑(mapping)

해쉬테이블(hashtable): 파일 썬 딕셔너리를 구현하기 위해 사용된 알고리즘

해쉬 함수(hash function): 키에 대한 위치를 계산하기 위해서 해쉬테이블에서 사용되는 함수

히스토그램(histogram): 카운터 집합.

구현(implementation): 연산(computation)을 수행하는 방법

항목(item): 키-밸류 페어에 대한 또 다른 이름.

키(key): 키-밸류 페어의 첫번째 부분으로 딕셔너리에 나타나는 개체.

키-밸류 페어(key-value pair): 키에서 값으로 맵핑을 표현.

룩업(lookup): 키를 가지고 해당 값을 찾는 딕셔너리 연산.

중첩 루프(nested loops): 또 다른 루프 ”내부”에 하나 혹은 그 이상의 루프가 있음. 외곽 루프가 1회 실행될 때, 내부 루프는 전체 반복을 완료함.

값(value): 키-밸류 페어의 두번째 부분으로 딕셔너리에 나타나는 개체. 앞에서 사용한 ”값(value)” 보다 더 구체적이다.

9.7 연습문제

Exercise 9.2 커밋(commit)이 무슨 요일에 수행되었는지에 따라 전자우편 메세지를 구분하는 프로그램을 작성하세요. ”From”으로 시작하는 라인을 찾고, 3 번째 단어를 찾아서 요일 횟수를 세서 저장하세요. 프로그램을 끝에 딕셔너리의 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 세고 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 가지는지를 알아내는 코드를 추가하세요.

결국, 모든 데이터를 읽고, 딕셔너리를 생성해서 최대 루프를 사용해서 딕셔너리르 훑어서 누가 가장 많은 전자우편 메시지를 갖고, 그 사람이 얼마나 많은 메시지를 가지는지를 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 명을 기록한다. 누가 메일을 보냈는지 대신(즉, 전체 전자우편 주소)에 메시지가 어디에서부터 왔는지를 기록한다. 프로그램 마지막에 딕셔너리의 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}
```

Chapter 10

튜플(Tuples)

10.1 튜플은 불변하다.

튜플(tuple)¹은 리스트와 마찬가지로 일련의 값이다. 튜플에 저장된 값은 임의의 형이 될 수 있고, 정수로 색인이 되어 있다. 중요한 차이점은 튜플은 **불변(immutable)**하다는 것이다. 튜플은 또한 **비교 가능(comparable)**하고 **해쉬 가능(hashable)**해서, 리스트의 값을 정렬할 수 있고, 파이썬 딕셔너리의 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, 튜플을 빼를게 알아볼 수 있도록 팔호로 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 파이썬은 ('a')을 문자열로 평가된느 팔호를 가진 문자열 표현으로 간주한다.

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

튜플을 생성하는 다른 방법은 내장함수 `tuple`을 사용하는 것이다. 인수가 없는 경우, 빈 튜플을 생성한다.

¹재미난 사실: 단어 ”튜플(tuple)”은 다양한 길이 (한배, 두배, 세배, 네배, 다섯배, 여섯배, 일곱배 등) 숫자열에 주어진 이름에서 유래한다.

```
>>> t = tuple()
>>> print t
()
```

만약 인수가 문자열, 리스트 혹은 튜플 같은 열인 경우, tuple에 호출한 결과는 요소열을 가진 투플이 된다.

```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

튜플(tuple)이 생성자 이름이 때문에 변수명으로 튜플을 사용을 피해야 한다. 대부분의 리스트 연산자는 튜플에서도 사용 가능하다. 꺭쇠 연산자는 요소를 색인한다.

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

그리고, 슬라이스 연산자(slice operator)는 범위의 요소를 선택한다.

```
>>> print t[1:3]
('b', 'c')
```

하지만, 튜플 요소중의 하나를 변경하고 하면, 오류가 발생한다.

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

튜플의 요소를 변경할 수는 없지만, 튜플을 다른 튜플로 교체는 할 수 있다.

```
>>> t = ('A',) + t[1:]
>>> print t
('A', 'b', 'c', 'd', 'e')
```

10.2 튜플 비교하기

비교연산자는 튜플과 다른 열(sequence)에도 동작한다. 파이썬은 각 열로부터 첫 요소를 비교하는 것에서부터 비교를 시작한다. 만약 두 요소가 같다면, 다음 요소로 비교를 진행하여 다른 요소를 찾을 때까지 계속한다. 후속 요소는 얼마나 큰 값인지에 관계없이 비교 고려대상은 아니다.

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

sort 함수도 동일한 방식으로 작동한다. 첫 요소를 먼저 정렬하지만, 동일한 경우 두 번째 요소를 정렬하고, 그 후속 요소를 동일한 방식으로 정렬한다. 이 기능은 다음 **DSU**라고 불리는 패턴에 적용된다.

데코레이트(Decorate) 열로부터 요소를 선행하는 하나 혹은 그 이상의 키를 가진 튜플 리스트를 구축하는 열

정렬(Sort) 파이썬 내장 함수 sort를 사용한 투플 리스트

언데코레이트(Undecorate) 열의 정렬된 요소를 추출.

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
```

첫 루프는 투플 리스트를 생성하고, 각 투플은 선행하여 길이를 가진 단어다.

sort가 첫 요소, 길이를 우선 비교하고, 동률일 경우 두 번째 요소를 고려한다.
sort 함수의 인수 reverse=True는 내림차순으로 정렬한다.

두 번째 루프는 투플 리스트를 훑고, 내림차순 길이 순으로 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 내림차순 길이 순으로 정렬된 문장은 시적인
의미를 많이 잃어버렸다.

10.3 투플 할당

파이썬 언어의 독특한 구문론적인 기능중의 하나는 할당문의 왼편에 투플을 놓을 수 있는 것이다. 왼쪽 편이 열인 경우 한번에 하나 이상의 변수를 할당할 수 있게 해준다.

다음 예제에서, 열인 두개 요소 리스트가 있고 하나의 명령문으로 변수 x와 y에
열의 첫번째와 두번째 요소를 할당한다.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

마술이 아니다. 파이썬은 대략 튜플 할당 구문을 다음과 같이 해석한다.²

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

문체적 할당문의 왼편에 튜플을 사용할 때, 괄호를 생략한다. 하지만 다음은 동일하게 적합한 구문이다.

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

튜플 할당문을 사용하는 특히 똑똑한 응용사례는 하나의 명령문으로 두 변수의 값을 교체(swap)하는 것이다.

```
>>> a, b = b, a
```

양쪽 모두 튜플이지만, 왼편은 튜플의 변수이고 오른편은 튜플의 표현식이다. 오른편의 값이 왼편의 해당하는 변수에 할당된다. 오른편의 모든 표현식은 할당이 이루어지기 이전에 평가된다. 왼편의 변수의 숫자와 오른편의 값의 숫자는 동일해야 한다.

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

좀 더 일반적으로 오른 편은 임의의 열(문자열, 리스트 혹은 튜플)이 될 수 있다. 예를 들어, 전자우편 주소를 사용자 이름과 도메인으로 쪼개기 위해서 다음과 같이 프로그램을 작성할 수 있다.

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

split 함수로부터 반환값은 두개의 요소를 가진 리스트다. 첫번째 요소는 uname에 두번째 요소는 domain에 할당된다.

```
>>> print uname
monty
>>> print domain
python.org
```

²파이썬은 구문을 문자그대로 해석하지는 않는다. 예를 들어, 동일한 것을 덕셔너리로 작성한다면, 예상한 것처럼 작동하지는 않는다.

10.4 딕셔너리와 튜플

딕셔너리는 튜플의 리스트를 반환하는 `items` 메쏘드가 있다. 각 튜플은 키-밸류 페어다.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 방법은 키로 정렬된 딕셔너리 내용을 출력하는 것이다.

```
>>> 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.

10.5 딕셔너리로 다중 할당

`items` 함수, 튜플 할당문, `for`문을 조합해서, 하나의 루프로 딕셔너리의 키와 값을 훑는 멋진 코드 패턴을 만들 수 있다.

```
for key, val in d.items():
    print val, key
```

이 루프는 두개의 반복 변수(**iteration variables**)를 가진다. `items` 함수는 튜플 리스트를 반환하고, `key`, `val`는 딕셔너리의 키-밸류 페어 각각을 성공적으로 반복하는 튜플 할당을 수행한다.

매번 루프를 반복할 때마다, `key`와 `value`는 여전히 해쉬 순으로 되어 있는 딕셔너리의 다음 키-밸류 페어로 진행한다.

루프의 출력은 다음과 같다.

```
10 a
22 c
1 b
```

다시한번 해쉬 키 순서다. 즉, 특별한 순서가 없다.

두 기술을 조합하면, 딕셔너리 내용을 키-밸류 페어에 저장된 값의 순서로 정렬하여 출력할 수 있다.

³파이썬 3.0으로 가면서 살짝 달라졌다.

이것을 수행하기 위해서, 각 튜플이 (value, key) 인 튜플 리스트를 작성한다. items 메쏘드를 사용하여 리스트 (key, value) 튜플을 만든다. 하지만 이번에는 키가 아닌 값으로 정렬한다. 키-밸류 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 튜플의 첫 요소로 값을 가지는 튜플 리스트를 생성함으로서 튜플 리스트를 정렬하여 값으로 정렬된 딕셔너리를 얻었다.

10.6 가장 빈도수가 높은 단어

로미오와 줄리엣 2장 2막 텍스트 파일로 다시 돌아와서, 텍스트에 가장 빈도수가 높은 단어를 10개를 출력하기 위해서 이 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)하는 딕셔너리를 계산하는 프로그램 첫 부분은 바뀌지 않는다. 하지만, counts 를 단순히 출력하는 대신에 (val, key) 튜플 리스트를 생성하고 역순으로 리스트를 정렬한다.

값이 처음이기 때문에, 비교를 위해서 값이 사용되고, 만약 동일한 값을 가진 튜플이 하나이상 존재한다면, 두번째 요소(키)를 살펴보게 되어서 값이 동일한 경우 키의 알파벳 순으로 추가적으로 정렬이 된다.

마지막에 다중 할당 반복을 수행하는 멋진 `for` 루프를 작성하고 리스트 쪼개기 (`lst[:10]`)를 통해 가장 빈도수가 높은 상위 10개 단어를 출력한다.

이제 마지막 출력문은 단어 빈도 분석에서 원하는 것을 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 19줄의 파이썬 프로그램으로 수행된 사실이 왜 파이썬이 정보 탐색의 언어로서 좋은 선택인지 보여준다.

10.7 딕셔너리 키로 튜플 사용하기

튜플은 **해쉬가능(hashable)**하고, 리스트는 그렇지 못하기 때문에, 딕셔너리에 사용할 **복합(composite)** 키를 생성하려면, 키로 튜플을 사용해야 한다.

만약 성(last-name)과 이름(first-name) 짝을 가지고 전화번호에 매핑(사상)하는 전화번호부를 생성하려고 하면, 복합키를 마주친다. 변수 `last`, `first`, `number` 을 정의했다고 가정하면, 다음과 같이 딕셔너리 할당문을 작성할 수 있다.

```
directory[last, first] = number
```

꺾쇠 괄호의 표현은 튜플이다. 딕셔너리를 훑기 위해서 `for` 루프에 튜플 할당을 사용한다.

```
for last, first in directory:
    print first, last, directory[last, first]
```

튜플인 `directory`에 키를 루프가 훑는다. 각 튜플 요소를 `last`, `first`에 할당하고, 이름과 해당 전화번호를 출력한다.

10.8 열 : 문자열, 리스트, 튜플

여기서 튜플 리스트에 초점을 맞추었지만, 거의 모든 예제가 또한 리스트의 리스트, 튜플의 튜플, 튜플의 리스트에도 동작한다. 가능한 조합을 열거하는 것을 피하기 위해서 시퀀스의 시퀀스 (sequences of sequences)에 대해서 논의하는 것이 때로는 편리하다.

대부분의 맵락에서 다른 종류의 시퀀스(문자열, 리스트, 튜플)는 상호 호환해서 사용될 수 있다. 그래서 왜 어떻게 다른 것보다 이것을 선택해야 될까요?

명확하게 시작하기 위해서, 문자열은 요소가 문자여야 하기 때문에 다른 시퀀스 보다 더 제약된다. 문자열은 또한 불변(immutable)이다. 새로운 문자열을 생성하는 것과 반대로, 문자열의 문자를 변경하고자 한다면, 대신에 문자 리스트를 사용할 필요가 있다.

리스트는 좀 더 튜플보다 일반적이다. 부분적으로는 변경가능(mutable)하기 때문이다. 하지만, 튜플을 좀 더 선호해야 하는 몇 가지 경우가 있다.

1. 어떤 맵락에서 `return`문처럼, 리스트보다 튜플을 생성하는 것이 구문론적으로 간략하다. 다른 맵락에서는 리스트가 더 선호될 수 있다.
2. 딕셔너리 키로서 시퀀스를 사용하려면, 튜플이나 문자열 같은 불변형(im-mutable type)을 사용해야 한다.
3. 함수에 인자로 시퀀스를 전달하려면, 튜플을 사용하는 것이 에일리어싱(aliasing)으로 생기는 예기치 못한 행동에 대한 가능성을 줄인다.

튜플은 불변(immutable)이어서, 현재 리스트를 변경하는 `sort`, `reverse` 같은 메쏘드를 제공하지는 않는다. 하지만, 파이썬은 내장함수 `sorted`, `reversed`를 제공해서, 매개 변수로 임의의 시퀀스를 받아 같은 요소를 다른 순서로 된 새로운 리스트를 반환한다.

10.9 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, **모양 오류(shape errors)**라고 불리는 오류에 노출되어 있다. 즉, 자료 구조가 잘못된 형(type), 크기, 구성일 경우 오류가 발생한다. 혹은 코드를 작성하고, 자료의 모양을 잊게 되면 오류가 발생한다.

예를 들어, 정수 하나인 리스트를 기대하고, 리스트가 아닌 일반 정수를 준다면, 작동하지 않을 것이다.

프로그램을 디버깅할 때, 정말 어려운 버그에 작업을 한다면, 다음 네 가지를 시도할 수 있다.

코드 읽기(reading): 코드를 면밀히 조사하고, 반복적으로 읽고, 의도한 해도 프로그램이 작성되었는지를 확인하라.

실행(running): 변경하고, 다른 버전을 실행해서 실험하라. 종종, 프로그램의 적절한 장소에 적절한 것을 배치해서, 문제가 명확하지만, 때때로, 발판(scaffolding)을 만들기 위해서 많은 시간을 쓰기도 한다.

반추(ruminating): 생각의 시간을 가지세요. 어떤 종류의 오류인가? 구문, 실행, 시맨틱(의미론). 오류 메시지로부터 혹은 프로그램 출력으로부터 무슨 정보를 얻을 수 있는가? 어떤 종류의 오류가 지금 보고 있는 문제를 만들었을까? 문제가 나타나기 전에 마지막으로 바꾼 것은 무엇인가?

퇴각(retreating): 어느 시점에선가, 최선은 물러서서, 최근의 변경을 다시 원복하는 것이다. 동작하고 이해하는 프로그램으로 다시 돌아가서, 다시 프로그램을 작성하는 것이다.

초보 프로그래머는 종종 이를 활동중 하나에 사로잡혀 다른 것을 잊곤 한다. 각 활동은 그 자신만의 실패 양태가 있다.

예를 들어, 프로그램을 정독하는 것은 문제가 인쇄상의 오류에 있다면 도움이 되지만, 문제가 개념상 오해에 뿌리를 두고 있다면 그렇지는 못할 것이다. 만약 여러분이 작성한 프로그램을 이해하지 못한다면, 100번 읽을 수는 있지만, 오류를 발견할 수는 없다. 왜냐하면, 오류는 여러분의 손에 있기 때문입니다.

실험을 수행하는 것은 특히 작고 간단한 테스트를 진행한다면 도움이 될 수 있다. 하지만, 코드를 읽거나, 생각없이 실험을 수행한다면, 프로그램이 작동될 때 까지 랜덤 변경을 개발하는 ”랜덤 워크 프로그램(random walk programming)” 패턴에 빠질 수 있다. 말할 필요없이 랜덤 워크 프로그래밍은 시간이 오래 걸린다.

생각의 시간을 가져야 한다. 디버깅은 실험 과학 같은 것이다. 문제가 무엇인지에 대한 최소한 한가지 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 너무 많고, 복잡하다면 최고의 디버깅 기술도 무용지물이다. 가끔, 최선의 선택은 퇴각하는 것이다. 작동하고 이해하는 곳까지 후퇴해서 프로그램을 간략화하라.

초보 프로그래머는 종종 퇴각하기를 꺼려한다. 왜냐하면, 설사 틀렸지만, 몇 라인의 코드를 지울 수 없기 때문이다. 삭제하지 않는 것이 기분이 좋다면, 프로그램을 다시 작성하기 전에 프로그램을 다른 파일에 복사하라. 그리고 나서, 한번에 조금씩 붙여넣어라.

정말 어려운 버그(hard bug)를 발견하고 고치는 것은 코드 일기, 실행, 반추, 때때로 퇴각을 요구한다. 만약 이를 활동 중 하나에 빠져있다면, 다른 것들을 시도해 보세요.

10.10 용어정의

비교가능한(comparable): 동일한 형의 다른 값과 비교하여 큰지, 작은지, 혹은 같은지를 확인하기 위해서 확인할 수 있는 형(type). 비교가능한(comparable) 형은 리스트에 넣어서 정렬할 수 있다.

자료 구조(data structure): 연관된 값의 집합, 종종 리스트, 딕셔너리, 튜플 등으로 조직화된다.

DSU: “decorate-sort-undecorate,”의 약어로 튜플 리스트를 생성, 정렬, 결과의 일부를 추출을 포함하는 패턴.

개더(gather): 변수-길이 인수 튜플을 조합하는 연산.

해쉬형의(hashable): 해쉬 함수를 가진 형(type). 정수, 소수점, 문자열 같은 불변형은 해쉬형이다. 리스트나 딕셔너리처럼 변경가능한 형은 해쉬형이 아니다.

스캐터(scatter): 시퀀스를 리스트 인수로 다루는 연산.

(자료 구조의) 모양(shape of a data structure): 자료 구조의 형(type), 크기, 구성의 요약.

싱글톤.singleton: 단일 요소를 가진 리스트 (혹은 다른 시퀀스).

튜플(tuple): 요소들의 불변 시퀀스.

튜플 할당(tuple assignment): 오른편에 스퀸스, 왼편에 튜플 변수를 가진 할당문. 오른편이 평가되고나서 각 요소들은 왼편의 변수에 할당된다.

10.11 연습문제

Exercise 10.1 앞서 작성한 프로그램을 다음과 같이 수정하세요. ”From”라인을 읽고 파싱하여 라인에서 주소를 뽑아내세요. 딕셔너리를 사용하여 각 사람으로부터 메시지 숫자를 셹니다.

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 셹니다. ”From” 라인으로부터 시간 문자열을 찾고 콜론(:) 문자를 사용하여 문자열을 쪼개서 시간을 추출합니다. 각 시간에 대해서 카운트를 누적하고 아래에 보여지듯이 시간단위로 정렬하여 한 라인에 한시간씩 갯수를 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, 빈도수를 내림차순으로 문자(*letters*)를 출력하는 프로그램을 작성하세요. 작성한 프로그램은 모든 입력을 소문자로 변환하고 a-z 문자만 셹니다. 공백, 숫자, 문장기호 a-z를 제외한 다른 어떤 것도 세지 않습니다. 몇몇 다른 언어의 텍스트 샘플을 구해서 언어마다 문자 빈도가 어떻게 바뀌는지 살펴보세요. 결과를 [wikipedia.org/wiki/Letter_frequencies](https://en.wikipedia.org/wiki/Letter_frequencies)의 표와 비교하세요.

Chapter 11

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 몇 권 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 프로그램에 가져오기(`import`)하여야 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, 매 라인을 루프를 반복해서 정규 표현식 `search()` 메쏘드를 호출하여 문자열 "From"이 포함된 라인만 출력한다. 상기 프로그램은 정규 표현식의 진정 강력한 기능을 사용하지 않았다. 왜냐하면, `line.find()` 메쏘드를 가지고 동일한 결과를 쉽게 구현할 수 있기 때문이다.

정규 표현식의 강력한 기능은 문자열에 해당하는 라인을 좀 더 정확하게 제어하기 위해서 검색 문자열에 특수문자를 추가할 때 확인할 수 있다. 매우 적은 코드를 작성해도 정규 표현식에 특수 문자를 추가하는 것 만으로도 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 일치하는 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 시작하는 라인만 일치할 수 있다. 여전히 매우 간단한 프로그램으로 문자열 라이브러리에서 `startswith()` 메쏘드로 동일하게 수행할 수 있다. 하지만, 무엇이 정규 표현식을 매칭하는가에 대해서 좀 더 많은 제어를 할 수 있게 하는 특수 액션 문자를 담고 있는 정규 표현식의 개념을 소개하기에는 충분하다.

11.1 정규 표현식의 문자 매칭

좀 더 강력한 정규 표현식을 작성할 수 있는 다른 특수문자 많이 있다. 가장 자주 사용되는 특수 문자는 임의의 문자를 매칭하는 마침표다.

다음 예제에서 정규 표현식 ”F..m:”은 ”From:”, ”Fxxm:”, ”F12m:”, ”F!@m:” 같은 임의의 문자열을 매칭한다. 왜냐하면 정규 표현식의 마침표 문자가 임의의 문자와 매칭되기 때문이다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^.m:', line) :
        print line
```

정규 표현식에 “*”, “+” 문자를 사용하여 문자가 원하는만큼 반복을 나타내는 기능과 결합되었을 때, 더욱 강력해진다. “*”, “+” 특수 문자는 검색 문자열에 하나의 문자만을 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 좀 더 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, ”.+” @ 기호로 되는 하나 혹은 그 이상의 문자들의 라인을 성공적으로 매칭한다. 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 문자령리 마지막 @ 기호를 매칭한다.

From: stephen.marquard@uct.ac.za, csev@umich.edu, and cwen@iupui.edu

추가로 다른 특수문자를 추가함으로서 별표나 더하기 기호가 너무 ”탐욕(greedy)”스럽지 않게 할 수 있다. 와일드 카드 특수문자의 탐욕스러운 기능을 끄는 것에 대해서는 자세한 정보를 참조바란다.

11.2 정규 표현식 사용 데이터 추출

파이썬에서 문자열에서 데이터를 추출하려면, `findall()` 메쏘드를 사용해서 정규 표현식과 매칭되는 모든 부속 문자열을 추출할 수 있다. 형식에 관계없이 임의의 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 라인의 각각의 형식에 맞는 코드를 작성하고는 싶지 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 인수 문자열을 찾아서 전자우편 주소처럼 보이는 모든 문자열을 리스트로 반환한다. 공백이 아닌 문자 (\S)와 매칭되는 두 문자 스퀀스를 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, “\S+” 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, 전자우편처럼 보이는 부속 문자열을 적어도 하나 찾아서 출력하기 위해서 반환 리스트 요소 숫자가 0 보다 큰가를 만을 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 수조 몇몇은 “<”, “;” 같은 잘못된 문자가 앞과 뒤에 붙어있다. 문자나 숫자로 시작하고 끝나는 문자열 부분만 관심 있다고 하자.

그러기 위해서, 정규 표현식의 또 다른 기능을 사용한다. 매칭하려는 다중 허용 문자 집합을 표기하기 위해서 꺭쇠 팔호를 사용한다. 그런 의미에서 “\S”은 공백이 아닌 문자 집합을 매칭하게 한다. 이제 매칭하려는 문자에 대해서 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(“\S*”)로 하나의 소문자, 대문자 혹은 숫자(“[a-zA-Z0-9]”)를 가지며, @ 다음에 0 혹은 그

이상의 공백이 아닌 문자("\S*")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 원편에 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 임의의 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.

11.3 검색과 추출 조합하기

다음과 같은 "X-" 문자열로 시작하는 라인의 숫자를 찾고자 한다면,

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

임의의 라인에서 임의의 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 그 이상의 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 있어야 한다.

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 합니다. `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 추가할 때, 문자열이 매칭될 때, 무시된다. 하지만, `findall()`을 사용할 때, 매칭할 전체 정규 표현식을 원할지라도, 정규 표현식을 매칭하는 부속 문자열의 부분만을 뽑아내다는 것을 괄호가 표시한다.

그래서, 프로그램에 다음과 같이 수정한다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^\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 한다면, 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, 라인의 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 행동이 왜 각 숫자로 모든 5자리 숫자를 얻은 이유다. 정규 표현식 라이브러리는 양방향으로 파일 처음이나 끝에 숫자가 아닌 것을 마주칠 때 까지 뺀어 나간다.

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 split를 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 라인의 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 라인의 정의다.

`findall()`을 사용해서 단지 시간만 뽑아내기 위해서, 두 숫자를 팔호를 다음과 같이 추가한다.

```
^From .* ([0-9][0-9]):
```

작업 결과는 다음과 같이 프로그램에 나타난다.

```
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']
...
```

11.4 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(ˆ) 같은 실제 문자를 매칭하고자 한다.

역슬래쉬(＼)를 가진 문자를 앞에 덮붙여서 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 덮붙여서, 실제로 ”라인 끝(end of line)” 매칭 대신에 입력 문자열의 달러 기호화 매칭한다. 정규 표현식의 나머지 부분은 하나 혹은 그 이상의 숫자 혹은 소수점 문자를 매칭한다. 주목: 꺪쇠 팔호 내부에 문자는 ”특수 문자”가 아니다. 그래서 “[0-9.]”은 실제 숫자 혹은 점을 의미한다. 꺪쇠 팔호 외부에 점은 ”와일드 카드(wild-card)” 문자이고 임의의 문자와 매칭한다. 꺪쇠 팔호 내부에서 점은 점일 뿐이다.

11.5 요약

지금까지 정규 표현식의 표면을 짚은 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)

\s

공백 문자를 매칭.

\S

공백이 아닌 문자를 매칭.(\s의 반대).

*

바로 앞선 문자에 적용되고 0 혹은 그 이상의 앞선문자와 매칭을 표기함.

*?

바로 앞선 문자에 적용되고 0 혹은 그 이상의 앞선문자와 매칭을 ”탐욕적이지 않은(non-greedy) 방식”으로 표기함.

+

바로 앞선 문자에 적용되고 1 혹은 그 이상의 앞선문자와 매칭을 표기함.

+?

바로 앞선 문자에 적용되고 1 혹은 그 이상의 앞선문자와 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-zA-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 임의의 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.

\b

빈 문자열을 매칭하지만, 단어의 시작과 끝에만 사용된다.

\B

빈 문자열을 매칭하지만, 단어의 시작과 끝이 아닌 곳에 사용된다.

\d

임의의 숫자와 매칭하여 [0-9] 집합에 상응함.

\D

임의의 숫자가 아닌 문자와 매칭하여 [^0-9] 집합에 상응함.

11.6 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
```

grep을 사용하여 mbox-short.txt 파일 내부에 ”From:” 문자열로 시작하는 라인을 보여준다. grep 명령어를 가지고 약간 실험을 하고 grep에 대한 문서를 읽는다면, 파이썬에서 지원하는 정규 표현식과 grep에서 지원되는 정규 표현식의 차이를 발견할 것이다. 예를 들어, grep 공백이 아닌 문자 “\S”을 지원하지 않는다. 그래서 약간 더 복잡한 집합 표기 “[^]”을 사용해야 한다. “[^]”은 간단히 정리하면, 공복을 제외한 임의의 문자와 매칭한다.

11.7 디버깅

만약 특정 메소드의 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.

11.8 용어정의

부서지기 쉬운 코드(brittle code): 입력 데이터가 특정한 형식일 경우에만 작동하는 코드. 하지만 올바른 형식에서 약간이도 벗어나게 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): 좀 더 복잡한 검색 문자열을 표현하는 언어. 정규 표현식은 특수 문자를 포함해서 검색은 라인의 처음 혹은 끝만 매칭하거나 많은 다른 비슷한 것만 매칭한다.

와일드 카드(wild card): 임의의 문자를 매칭하는 특수 문자. 정규 표현식에서 와일드 카드 문자는 마침표 문자다.

11.9 연습 문제

Exercise 11.1 유닉스의 grep 명령어를 모사하는 간단한 프로그램을 작성하세요. 사용자가 정규 표현식을 입력하고 정규 표현식에 매칭되는 라인수를 셉하는 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
```