정보교육을 위한 파이썬

정보 탐색

Version 0.0.9-d2

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

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*.

The LaTeX 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

리스트

1.1 리스트는 열이다.

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

신규 리스틀 생성하는 방법은 여러가지다. 가장 간단한 방법은 꺾쇠 괄호(`[` 와 `]`)로 요소를 감싸는 것이다.

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

첫번째 예제는 네 개의 정수 리스트다. 드번째 예제는 3개의 문자열 리스트다. 문자열의 요소는 같은 형(type)일 필요는 없다. 다음의 리스트는 문자열, 부동소수점 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 꺾쇠 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] []
```

1.2 리스트는 변경가능하다.

리스트의 요소에 접근하는 구문은 문자열의 문자에 접근하는 것과 동일한 꺾쇠 괄호 연산자다. 꺽쇠 괄호 내부의 표현식은 인덱스를 명세한다. 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 인텍스와 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
```

1.3 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, 인텍스가 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 0 에서 $n-1$ 까지 리스트 인텍스를 반환한다.

여기서, *n*은 리스트의 길이다. 매번 루프가 반복될 때마다, i는 다음 요소의 인덱스를 얻는다. 몸통 부문의 할당문은 i를 사용해서 요소의 옛값을 일고 새값을 할당한다.

빈 리스트의 for문은 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]]
```

1.4 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 반복한다.

1.5 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 연ㅅ나을 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']
```

1.6 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 `None`을 반환한다. 우연히 `t = t.sort()` 이렇게 작성한다면, 결과에 실망할 것이다.

1.7 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 메쏘드는 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 메쏘드를 사용할 수 있다.

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

remove 메쏘드의 반환값은 None이다.

하나 이상의 요소를 제거하기 위해서, 쪼개기 인덱스(slice index)와 del을 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.

1.8 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, 비교가능한 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.

1.9 리스트와 문자열

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

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

list는 내장함수의 이름이기 때문에, 변수명으로 사용하는 것을 피해야 한다. l의 사용을 1 처럼 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.

1.10 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.

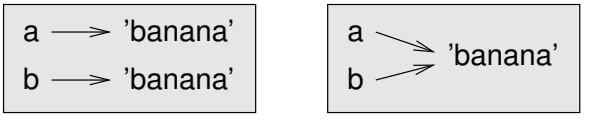
1.11 개체와 값(value)

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. 두번째 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 반듯이 동일하지는 않다.

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

1.12 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 두개의 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, 변경가능한 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.

1.13 리스트 인수

리스트를 함수에 인수로 전달할 때, 함수는 리스트의 참조를 얻는다. 만약 함수가 리스트 매개 변수를 변경한다면, 호출자는 변화를 보게된다. 예를 들어, delete_head는 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 t와 변수 letters는 동일한 개체에 대한 에일리어스(aliases)다.

리스트를 변경하는 연산자와 신규 리스트를 생성하는 연산자를 구별하는 것은 중요하다. 예를 들어, append 메쏘드는 리스트를 변경하지만, + 연산자는 신규 리스트를 생성한다.

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1
[1, 2, 3]
>>> print t2
None

>>> t3 = t1 + [3]
>>> print t3
[1, 2, 3]
>>> t2 is t3
False
```

리스트를 변경하는 함수를 작성할 때, 이 차이는 매우 중요하다. 예를 들어, 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, tail은 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 1.1 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.

1.14 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.html 에 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 sort같은 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 sorted를 사용할 수 있다. 하지만 이 경우에는, 변수명으로 sorted를 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, startswith를 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 rstrip을 사용할 필요도 없다. 하지만, 더 좋아졌는가?

```
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

작동하는 것 같지만, 첫줄에 Sat 를 출력하고 나서 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 `print`문을 추가하는 것이다. `print`문을 넣는 가장 좋은 장소는 프로그램이 동작하지 않는 라인 앞이 적절하고, 프로그램 실패를 야기하는 데이터를 출력한다.

이 접근법은 많은 라인을 출력하지만, 최소한 프로그램에 손에 잡히는 단서를 최소한 준다. 그래서 `words`를 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, print문을 제거하는 대신에 print문을 주석 처리한다.

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

두 개의 continue문이 "흥미롭고" 데이터를 처리할 라인의 집합에 좀더 정제하도록 돕는 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 1.2 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 1.3 두 if문 없도록 상기 예제의 가디언 코드(guardian code)를 다시 작성하세요. 대신에 단일 if문과 and 논리 연산자를 사용하는 복합 논리 표현식을 사용하세요.

1.15 용어정의

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

구분자(delimiter): 문자열이 어디서 쪼개져야할지를 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.

1.16 연습문제

Exercise 1.4 `www.py4inf.com/code/romeo.txt`에서 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 리스트웨 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 1.5 우편함 데이터를 읽어 들이는 프로그램을 작성하세요. "From"으로 시작하는 라인을 발견했을 때, `split` 함수를 사용하여 라인을 단어로 쪼게세요. "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 1.6 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
```