제 8 장

리스트 (List)

제 1 절 리스트는 순서(sequence)다.

문자열처럼, 리스트(list)는 값의 순서(sequence)다. 문자열에서, 값은 문자지만, 리스트에서는 임의 자료형(type)도 될 수 있다. 리스트 값은 요소(elements)나 때때로 항목(items)으로 불린다.

신규 리스틀 생성하는 방법은 여러 가지가 있다. 가장 간단한 방법은 꺾쇠 괄호 ([와])로 요소를 감싸는 것이다.

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

첫번째 예제는 4개 정수 리스트다. 두번째 예제는 3개 문자열 리스트다. 문자열 요소가 동일한 자료형(type)일 필요는 없다. 다음 리스트는 문자열, 부동 소수점 숫자, 정수, (아!) 또 다른 리스트를 담고 있다.

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

또 다른 리스트 내부에 리스트가 중첩(nested)되어 있다.

어떤 요소도 담고 있지 않는 리스트를 빈 리스트(empty list)라고 부르고, 빈 꺾쇠 괄호("[]")로 생성한다.

예상했듯이, 리스트 값을 변수에 대입할 수 있다.

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

제 2 절 리스트는 변경가능하다.

리스트 요소에 접근하는 구문은 문자열 문자에 접근하는 것과 동일한 꺾쇠 괄호 연산자다. 꺽쇠 괄호 내부 표현식은 인덱스를 명세한다. 기억할 것은 인덱스는

0 에서부터 시작한다는 것이다.

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

문자열과 달리, 리스트 항목 순서를 바꾸거나, 리스트에 새로운 항목을 다시 대입할 수 있기 때문에 리스트는 변경가능하다. 꺾쇠 괄호 연산자가 대입문 왼쪽 편에 나타날 때, 새로 대입될 리스트 요소를 나타낸다.

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

리스트 numbers 첫번째 요소는 123 값을 가지고 있었으나, 이제 5 값을 가진다.

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

리스트 인덱스는 문자열 인덱스와 동일한 방식으로 동작한다.

- 어떠한 정수 표현식도 인덱스로 사용할 수 있다.
- 존재하지 않는 요소를 읽거나 쓰려고 하면, 인덱스 오류 (IndexError)가 발생한다.
- 인덱스가 음의 값이면, 리스트 끝에서부터 역으로 센다.

in 연산자도 또한 리스트에서 동작하니 사용할 수 있다.

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

제 3 절 리스트 운행법

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

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

리스트 요소를 읽기만 한다면 이것만으로도 잘 동작한다. 하지만, 리스트 요소를 쓰거나, 갱신하는 경우, 인덱스가 필요하다. 리스트 요소를 쓰거나 갱신하는 일반적인 방법은 range와 len 함수를 조합하는 것이다.

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

상기 루프는 리스트를 운행하고 각 요소를 갱신한다. `len` 함수는 리스트 요소 갯수를 반환한다. `range` 함수는 0 에서 $n-1$ 까지 리스트 인덱스를 반환한다. 여기서, n은 리스트 길이다. 매번 루프가 반복될 때마다, `i`는 다음 요소 인덱스를 얻는다. 몸통 부문 대입문은 `i`를 사용해서 요소의 이전 값을 읽고 새 값을 대입한다.

빈 리스트에 대해서 `for`문은 결코 몸통 부문을 실행하지 않는다.

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

리스트가 또 다른 리스트를 담을 수 있지만, 중첩된 리스트는 여전히 요소 하나로 간주된다. 다음 리스트 길이는 4 이다.

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

제 4 절 리스트 연산자

`+` 연산자는 리스트를 결합한다.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

유사하게 `*` 연산자는 주어진 횟수만큼 리스트를 반복한다.

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

첫 예제는 `[0]`을 4회 반복한다. 두 번째 예제는 `[1, 2, 3]` 리스트를 3회 반복한다.

제 5 절 리스트 슬라이스(List slices)

슬라이스 연산자는 리스트에도 또한 동작한다.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

첫 번째 인덱스를 생략하면, 슬라이스는 처음부터 시작한다. 두 번째 인덱스를 생략하면, 슬라이스는 끝까지 간다. 그래서 양쪽의 인덱스를 생략하면, 슬라이스 결과는 전체 리스트를 복사한 것이 된다.

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

리스트는 변경이 가능하기 때문에 리스트를 접고, 돌리고, 훼손하는 연산을 수행하기 전에 복사본을 만들어 두는 것이 유용하다.

대입문 왼편의 슬라이스 연산자로 복수의 요소를 갱신할 수 있다.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print t
['a', 'x', 'y', 'd', 'e', 'f']
```

제 6 절 리스트 메쏘드

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

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

확장 (extend) **메쏘드는 인자로 리스트를 받아 모든 요소를 리스트에 덧붙인다.**

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

상기 예제는 t2 **리스트를 변경없이 그냥 둔다.**

정렬 (sort) **메쏘드는 낮음에서 높음으로 리스트 요소를 정렬한다.**

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

대부분의 리스트 메쏘드는 보이드(void)여서, 리스트를 변경하고 None**을 반환한다. 우연히** t = t.sort() **이렇게 작성한다면, 결과에 실망할 것이다.**

제 7 절 요소 삭제

리스트 요소를 삭제하는 방법이 몇 가지 있다. 리스트 요소 인덱스를 알고 있다면, 팝 (pop) **메쏘드를 사용한다.**

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

팝(pop) 메쏘드는 리스트를 변경하여 제거된 요소를 반환한다. 인덱스를 주지 않으면, 마지막 요소를 지우고 반환한다.

요소에서 제거된 값이 필요없다면, `del` **연산자를 사용한다.**

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

(인덱스가 아닌) 제거할 요소값을 알고 있다면, 제거 (remove) **메쏘드를 사용한다.**

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

제거 (remove) **메쏘드의 반환값은** None**이다.**

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

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

마찬가지로, 슬라이스는 두 번째 인덱스를 포함하지 않는 두 번째 인덱스까지 모든 요소를 선택한다.

제 8 절 리스트와 함수

루프를 작성하지 않고도 리스트를 빠르게 살펴볼 수 있는 리스트에 적용할 수 있는 내장함수가 많이 있다.

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print len(nums)
6
>>> print max(nums)
74
>>> print min(nums)
3
>>> print sum(nums)
154
>>> print sum(nums)/len(nums)
25
```

리스트 요소가 숫자일 때만, `sum()` 함수는 동작한다. `max()`, `len()`, 등등 다른 함수는 문자열 리스트나, 비교 가능한 다른 자료형(type) 리스트에 사용될 수 있다.

리스트를 사용해서, 앞서 작성한 프로그램을 다시 작성해서 사용자가 입력한 숫자 목록 평균을 계산한다.

우선 리스트 없이 평균을 계산하는 프로그램:

```
total = 0
count = 0
while ( True ) :
    inp = raw_input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print 'Average:', average
```

상기 프로그램에서, `count` 와 `sum` 변수를 사용해서 반복적으로 사용자가 숫자를 입력하면 값을 저장하고, 지금까지 사용자가 입력한 누적 합계를 계산한다.

단순하게, 사용자가 입력한 각 숫자를 기억하고 내장함수를 사용해서 프로그램 마지막에 합계와 갯수를 계산한다.

```
numlist = list()
while ( True ) :
    inp = raw_input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print 'Average:', average
```

루프가 시작되기 전 빈 리스트를 생성하고, 매번 숫자를 입력할 때마다 숫자를 리스트에 추가한다. 프로그램 마지막에 간단하게 리스트 총합을 계산하고, 평균을 산출하기 위해서 입력한 숫자 개수로 나눈다.

제 9 절 리스트와 문자열

문자열은 문자 순서(sequence)이고, 리스트는 값 순서(sequence)이다. 하지만 리스트 문자는 문자열과 같지는 않다. 문자열에서 리스트 문자로 변환하기 위해서, `list`를 사용한다.

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

list는 내장함수 이름이기 때문에, 변수명으로 사용하는 것을 피해야 한다. l을 사용하면 1 처럼 보이기 때문에 피한다. 그래서, t를 사용했다.

list 함수는 문자열을 각각의 문자로 쪼갠다. 문자열 단어로 쪼개려면, 분할 (split) 메쏘드를 사용할 수 있다.

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

분할 (split) 메쏘드를 사용해서 문자열을 리스트 토큰으로 쪼개면, 인덱스 연산자('[]')를 사용하여 리스트의 특정 단어를 볼 수 있다.

옵션 인자로 단어 경계로 어떤 문자를 사용할 것인지 지정하는데 사용되는 구분자 (delimiter)를 활용하여 분할 (split) 메쏘드를 호출한다. 다음 예제는 구분자로 하이픈('-')을 사용한 사례다.

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

합병 (join) 메쏘드는 분할 (split) 메쏘드의 역이다. 문자열 리스트를 받아 리스트 요소를 연결한다. 합병 (join)은 문자열 메쏘드여서, 구분자를 호출하여 매개 변수로 넘길 수 있다.

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

상기의 경우, 구분자가 공백 문자여서 결합 (join) 메쏘드가 단어 사이에 공백을 넣는다. 공백없이 문자열을 결합하기 위해서, 구분자로 빈 문자열 ''을 사용한다.

제 10 절 라인 파싱하기(Parsing)

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

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

이런 종류의 문제에 직면했을 때, 분할 (split) 메쏘드가 매우 효과적이다. 작은 프로그램을 작성하여 "From "으로 시작하는 라인을 찾고 분할 (split) 메쏘드로 파싱하고 라인의 흥미로운 부분을 출력한다.

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From ') : continue
    words = line.split()
    print words[2]
```

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

프로그램은 다음을 출력한다.

```
Sat
Fri
Fri
Fri
    ...
```

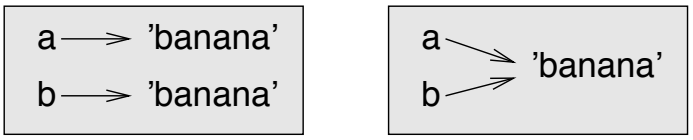
나중에, 매우 정교한 기술에 대해서 학습해서 정확하게 검색하는 비트(bit) 수준 정보를 찾아 내기 위해서 작업할 라인을 선택하고, 어떻게 해당 라인을 뽑아낼 것이다.

제 11 절 객체와 값(value)

다음 대입문을 실행하면,

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

a 와 b 모두 문자열을 참조하지만, 두 변수가 *동일한* 문자열을 참조하는지 알 수 없다. 두 가지 가능한 상태가 있다.



한 가지 경우는 a 와 b가 같은 값을 가지는 다른 두 객체를 참조하는 것이다. 두 번째 경우는 같은 객체를 참조하는 것이다.

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

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

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

하지만, 리스트 두 개를 생성할 때, 객체가 두 개다.

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

상기의 경우, 두 개의 리스트는 동등하다고 말할 수 있다. 왜냐하면 동일한 요소를 가지고 있기 때문이다. 하지만, 같은 객체는 아니기 때문에 동일하지는 않다. 두 개의 객체가 동일하다면, 두 객체는 또한 등등하다. 하지만, 동등하다고 해서 반듯이 동일하지는 않다.

지금까지 "객체(object)"와 "값(value)"을 구분 없이 사용했지만, 객체가 값을 가진다라고 말하는 것이 좀더 정확하다. `a = [1,2,3]` 을 실행하면, `a` 는 특별한 순서 요소값을 갖는 리스트 객체로 참조한다. 만약 또 다른 리스트가 동일한 요소를 가진다면, 그 리스트는 같은 값을 가진다고 말한다.

제 12 절 에일리어싱(Aliasing)

`a`가 객체를 참조하고, `b = a` 대입하다면, 두 변수는 동일한 객체를 참조한다.

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

객체와 변수의 연관짖는 것을 참조(reference)라고 한다. 상기의 경우 동일한 객체에 두 개의 참조가 있다.

하나 이상의 참조를 가진 객체는 한개 이상의 이름을 갖게 되어서, 객체가 에일리어스(aliased) 되었다고 한다.

만약 에일리어스된 객체가 변경 가능하면, 변화의 여파는 다른 객체에도 파급된다.

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

이와 같은 행동이 유용하기도 하지만, 오류를 발생시키기도 쉽다. 일반적으로, 변경가능한 객체(mutable object)로 작업할 때 에일리어싱을 피하는 것이 안전하다.

문자열 같이 변경 불가능한 객체에 에일리어싱은 그렇게 문제가 되지 않는다.

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

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

제 13 절 리스트 인수

리스트를 함수에 인자로 전달할 때, 함수는 리스트에 참조를 얻는다. 만약 함수가 리스트 매개 변수를 변경한다면, 호출자는 변경된 것을 보게된다. 예를 들어, `delete_head`**는 리스트에서 첫 요소를 제거한다.**

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

다음에 `delete_head` **함수가 사용된 예제가 있다.**

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

매개 변수 `t`**와 변수** `letters`**는 동일한 객체에 대한 에일리어스(aliases)다.**

리스트를 변경하는 연산자와 신규 리스트를 생성하는 연산자를 구별하는 것이 중요하다. 예를 들어, 덧붙이기 `(append)` **메쏘드는 리스트를 변경하지만,** `+` **연산자는 신규 리스트를 생성한다.**

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

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

리스트를 변경하는 함수를 작성할 때, 이 차이는 매우 중요하다. 예를 들어, 다음 함수는 리스트의 머리 부문(head)을 삭제하지 않는다.

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

슬라이스 연산자는 새로운 리스트를 생성하고 대입문을 통해서 `t`**가 참조하게 하지만, 어떤 것도 인자로 전달된 리스트에는 영향도 주지 못한다.**

대안은 신규 리스트를 생성하고 반환하는 함수를 작성하는 것이다. 예를 들어, `tail`**은 리스트의 첫 요소를 제외하고 모든 요소를 반환한다.**

```
def tail(t):
    return t[1:]
```

상기 함수는 원 리시트를 변경하지는 않는다. 다음에 사용 예시가 있다.

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

Exercise 8.1 리스트를 인자로 받아 리스트를 변경하여, 첫 번째 요소와 마지막 요소를 제거하고 None을 반환하는 chop 함수를 작성하게요.

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

제 14 절 디버깅

부주의한 리스트 사용이나 변경가능한 객체를 사용하는 경우 디버깅을 오래 할 수 있다. 다음에 일반적인 함정 유형과 회피하는 방법을 소개한다.

1. **대부분의 리스트 메쏘드는 인자를 변경하고,** None**을 반환한다. 이는 새로운 문자열을 반환하고 원 문자열은 그대로 두는 문자열의 경우와 정반대다.**

   다음과 같이 문자열 코드를 쓰는데 익숙해져 있다면,

   ```
   word = word.strip()
   ```

   다음과 같이 리스트 코드를 작성하고 싶은 유혹이 있을 것이다.

   ```
   t = t.sort()            # 틀림(WRONG)!
   ```

   정렬 (sort) **메쏘드는** None**을 반환하기 때문에, 리스트** t**로 수행한 다음 연산은 실패한다.**

   리스트 메쏘드와 연산자를 사용하기 전에, 문서를 주의깊게 읽고, 인터랙티브 모드에서 시험하는 것을 권한다. 리스트가 문자열과 같은 다른 순서(sequence)와 공유하는 메쏘드와 연산자는 `docs.python.org/lib/typesseq.html` **에 문서화되어 있다. 변경가능한 순서(sequence)에만 적용되는 메쏘드와 연산자는** `docs.python.org/lib/typesseq-mutable.html`**에 문서화되어 있다.**

2. **관용구를 선택하고 고수하라.**

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

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

   하지만, 다음은 잘못됐다.

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

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

3. **에일리어싱을 회피하기 위해서 사본 만들기.**

   인자를 변경하는 정렬 (sort) **같은 메쏘드를 사용하지만, 원 리스트도 보관되길 원한다면, 사본을 만든다.**

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

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

4. **리스트,** 분할 (split)**, 파일**

   파일을 읽고 파싱할 때, 프로그램이 중단될 수 있는 입력값을 마주할 수많은 기회가 있다. 그래서 파일을 훑어 "건초더미에서 바늘"을 찾는 프로그램을 작성할 때 사용한 가디언 패턴(guardian pattern)을 다시 살펴보는 것은 좋은 생각이다.

   파일 라인에서 요일을 찾는 프로그램을 다시 살펴보자.

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

   각 라인을 단어로 나누었기 때문에, startswith**를 사용하지 않고, 라인에 관심있는 단어가 있는지 살펴보기 위해서 단순하게 각 라인의 첫 단어를 살펴본다. 다음과 같이** continue **문을 사용해서 "From"이 없는 라인을 건너 뛴다.**

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

   프로그램이 훨씬 간단하고, 파일 끝에 있는 새줄(newline)을 제거하기 위해 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 Sat 를 출력하고 나서 역추적 오류(traceback error)로 프로그램이 정상 동작에 실패한다. 무엇이 잘못되었을까? 어딘가 엉망이 된 데이터가 있어 우아하고, 총명하며, 매우 파이썬스러운 프로그램을 망가뜨린건가?

오랜 동안 프로그램을 응시하고 머리를 짜내거나, 다른 사람에게 도움을 요청할 수 있지만, 빠르고 현명한 접근법은 print문을 추가하는 것이다. print문을 넣는 가장 좋은 장소는 프로그램이 동작하지 않는 라인 앞이 적절하고, 프로그램 실패를 야기할 것 같은 데이터를 출력한다.

이 접근법이 많은 라인을 출력하지만, 즉석에서 문제에 대해서 손에 잡히는 단서는 최소한 준다. 그래서 words를 출력하는 출력문을 5번째 라인 앞에 추가한다. "Debug:"를 접두어로 라인에 추가하여, 정상적인 출력과 디버그 출력을 구분한다.

```
for line in fhand:
    words = line.split()
    print 'Debug:', words
    if words[0] != 'From' : continue
    print words[2]
```

프로그램을 실행할 때, 많은 출력결과가 스크롤되어 화면 위로 지나간다. 마지막에 디버그 결과물과 역추적(traceback)을 보고 역추적(traceback) 바로 앞에서 무슨 일이 생겼는지 알 수 있다.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

각 디버그 라인은 리스트 단어를 출력하는데, 라인을 분할 (split)해서 단어로 만들 때 얻어진다. 프로그램이 실패할 때 리스트 단어는 비었다 '[]'. 텍스트 편집기로 파일을 열어 살펴보면 그 지점은 다음과 같다.

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

프로그램이 빈 라인을 만났을 때, 오류가 발생한다. 물론, 빈 라인은 '0' 단어 ("zero words")다. 프로그램을 작성할 때, 왜 이것을 생각하지 못했을까? 첫 단어(word[0])가 "From"과 일치하는지 코드가 점검할 때, "인덱스 범위 오류(index out of range)"가 발생한다.

물론, 첫 단어가 없다면 첫 단어 점검을 회피하는 가디언 코드(guardian code)를 삽입하기 최적 장소이기는 하다. 코드를 보호하는 방법은 많다. 첫 단어를 살펴보기 전에 단어의 갯수를 확인하는 방법을 택한다.

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
```

```
# print 'Debug:', words
if len(words) == 0 : continue
if words[0] != 'From' : continue
print words[2]
```

변경한 코드가 실패해서 다시 디버그할 경우를 대비해서, `print`문을 제거하는 대신에 `print`문을 주석 처리한다. 그리고 나서, 단어가 '0' 인지를 살펴보고 만약 '0' 이면, 파일 다음 라인으로 건너뛰도록 `continue`문을 사용하는 가디언 문장(guardian statement)을 추가한다.

두 개 `continue`문이 "흥미롭고" 좀더 처리가 필요한 라인 집합을 정제하도록 돕는 것으로 생각할 수 있다. 단어가 없는 라인은 "흥미 없어서" 다음 라인으로 건너뛴다. 첫 단어에 "From"이 없는 라인도 "흥미 없어서" 건너뛴다.

변경된 프로그램이 성공적으로 실행되어서, 아마도 올바르게 작성된 것으로 보인다. 가디언 문장(guardian statement)이 `words[0]`가 정상작동할 것이라는 것을 확인해 주지만, 충분하지 않을 수도 있다. 프로그램을 작성할 때, "무엇이 잘못 될 수 있을까?"를 항상 생각해야만 한다.

Exercise 8.2 상기 프로그램의 어느 라인이 여전히 적절하게 보호되지 않은지를 생각해 보세요. 텍스트 파일을 구성해서 프로그램이 실패하도록 만들 수 있는지 살펴보세요. 그리고 나서, 프로그램을 변경해서 라인이 적절하게 보호되게 하고, 새로운 텍스트 파일을 잘 다룰 수 있도록 시험하세요.

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

제 15 절 용어정의

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

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

요소(element): 리스트 혹은 다른 순서(sequence) 값의 하나로 항목(item)이라고도 한다.

동등한(equivalent): 같은 값을 가짐.

인덱스(index): 리스트의 요소를 지칭하는 정수 값.

동일한(identical): 동등을 함축하는 같은 객체임.

리스트(list): 순서(sequence) 값.

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

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

객체(object): 변수가 참조할 수 있는 무엇. 객체는 자료형(type)과 값(value)을 가진다.

참조(reference): 변수와 값의 연관.

제 16 절 연습문제

Exercise 8.4 `www.py4inf.com/code/romeo.txt`에서 파일 사본을 다운로드 받으세요.

`romeo.txt` 파일을 열어, 한 줄씩 읽어들이는 프로그램을 작성하세요. 각 라인마다 분할 `(split)` 함수를 사용하여 라인을 단어 리스트로 쪼개세요.

각 단어마다, 단어가 이미 리스트에 존재하는지를 확인하세요. 만약 단어가 리스트에 없다면, 리스트에 새 단어로 추가하세요.

프로그램이 완료되면, 알파벳 순으로 결과 단어를 정렬하고 출력하세요.

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

Exercise 8.5 전자우편 데이터를 읽어 들이는 프로그램을 작성하세요. ”From” 으로 시작하는 라인을 발견했을 때, 분할 `(split)` 함수를 사용하여 라인을 단어로 쪼개세요. ”From” 라인의 두번째 단어, 누가 메시지를 보냈는지에 관심이 있다.

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

”From” 라인을 파싱하여 각 ”From”라인의 두번째 단어를 출력한다. 그리고 나서, ”From:”이 아닌 ”From”라인 갯수를 세고, 끝에 갯수를 출력한다.

여기 몇 줄을 삭제한 출력 예시가 있다.

```
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

Exercise 8.6 사용자가 숫자 리스트를 입력하고, 입력한 숫자 중에 최대값과 최소값을 출력하고 사용자가 ”done”을 입력할 때 종료하는 프로그램을 다시 작성하세요. 사용자가 입력한 숫자를 리스트에 저장하고, `max()` 과 `min()` 함수를 사용하여 루프가 끝나면, 최대값과 최소값을 출력하는 프로그램을 작성하세요.

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```

제 9 장

딕셔너리(Dictionaries)

딕셔너리(dictionary)는 리스트 같지만 좀더 일반적이다. 리스트에서 위치(인덱스)는 정수이지만, 딕셔너리에서는 인덱스는 임의 자료형(type)이 될 수 있다.

딕셔너리를 인덱스 집합(키(keys)라고 부름)에서 값(value) 집합으로 사상(mapping)하는 것으로 생각할 수 있다. 각각의 키는 값에 대응한다. 키와 값을 연관시키는 것을 키-값 페어(key-value pair)라고 부르고, 종종 항목(item)으로도 부른다.

한 예제로, 영어 단어에서 스페인 단어에 매핑되는 사전을 만들 것이다. 키와 값은 모두 문자열이다.

`dict` **함수는 항목이 전혀 없는 사전을 신규로 생성한다.** `dict`**는 내장함수명이어서, 변수명으로 사용하는 것을 피해야 한다.**

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

구불구불한 괄호 `{}`**는 빈 딕셔너리를 나타낸다. 딕셔너리에 항목을 추가하기 위해서 꺾쇠 괄호를 사용한다.**

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

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

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

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

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

`eng2sp`**을 출력하면, 놀랄 것이다.**

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

키-값 페어(key-value pair) 순서가 같지 않다. 사실 동일한 사례를 여러분의 컴퓨터에서 입력하면, 다른 결과를 얻게 된다. 일반적으로, 딕셔너리 항목 순서는 예측 가능하지 않다.

딕셔너리 요소가 정수 인덱스로 색인되지 않아서 문제되지는 않는다. 대신에, 키를 사용해서 상응하는 값을 찾을 수 있다.

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

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

만약 키가 딕셔너리에 존재하지 않으면, 예외 오류가 발생한다.

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

len 함수를 딕셔너리에 사용해서, 키-값 페어(key-value pair) 항목 개수를 반환한다.

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

in 연산자도 딕셔너리에 작동되는데, 어떤 것이 딕셔너리 *키(key)*에 있는지 알려준다. (값(value)으로 나타내는 것은 충분히 좋지는 않다.)

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

딕셔너리에 무엇이 값으로 있는지 확인하기 위해서, values 메쏘드를 해서 리스트로 값을 반환받고 나서 in 연산자를 사용하여 확인한다.

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

in 연산자는 리스트와 딕셔너리에 각기 다른 알고리즘을 사용한다. 리스트에 대해서 선형 검색 알고리즘을 사용한다. 리스트가 길어짐에 따라 검색 시간은 리스트 길이에 비례하여 길어진다. 딕셔너리에 대해서 파이썬은 해쉬 테이블(hash table)로 불리는 놀라운 특성을 가진 알고리즘을 사용한다. 얼마나 많은 항목이 딕셔너리에 있는지에 관계없이 in 연산자는 대략 동일한 시간이 소요된다. 왜 해쉬 함수가 마술 같은지에 대해서는 설명하지 않지만, wikipedia.org/wiki/Hash_table 에서 좀더 많은 정보를 얻을 수 있다.

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

제 1 절 계수기(counter) 집합으로서 딕셔너리

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

1. 26개 변수를 알파벳 문자 각각에 대해 생성한다. 그리고 나서 아마도 연쇄 조건문을 사용하여 문자열을 훑고 해당 계수기(counter)를 하나씩 증가한다.
2. 26개 요소를 가진 리스트를 생성한다. 내장함수 `ord`를 사용해서 각 문자를 숫자로 변환한다. 리스트 안에 인덱스로 숫자를 사용해서 적당한 계수기(counter)를 증가한다.
3. 키(key)로 문자, 계수기(counter)로 해당 값(value)을 갖는 딕셔너리를 생성한다. 처음 문자를 만나면, 딕셔너리에 항목으로 추가한다. 추가한 후에는 기존 항목 값을 증가한다.

상기 3개 옵션은 동일한 연산을 수행하지만, 각각은 다른 방식으로 연산을 구현한다.

구현(implementation)은 연산(computation)을 수행하는 방법이다. 어떤 구현 방법이 다른 것 보다 낫다. 예를 들어, 딕셔너리 구현의 장점은 사전에 문자열에서 어떤 문자가 나타날지 몰라도 된다. 다만 나타날 문자에 대한 공간만 준비하면 된다는 것이다.

다음에 딕셔너리로 구현한 코드가 있다.

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print d
```

계수기(counter) 혹은 빈도에 대한 통계 용어인 히스토그램(histogram)을 효과적으로 계산해보자.

`for` 루프는 문자열을 훑는다. 매번 루프를 반복할 때마다 딕셔너리에 문자 `c`가 없다면, 키 `c`와 초기값 1을 가진 새로운 항목을 생성한다. 문자 `c`가 이미 딕셔너리에 존재한다면, `d[c]`을 증가한다.

다음 프로그램 실행 결과가 있다.

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

히스토그램은 문자 `'a'`, `'b'`는 1회, `'o'`는 2회 등등 나타남을 보여준다.

딕셔너리에는 키와 디폴트(default) 값을 갖는 `get` 메쏘드가 있다. 딕셔너리에 키가 나타나면, `get` 메쏘드는 해당 값을 반환하고, 해당 값이 없으면 디폴트 값을 반환한다. 예를 들어,

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print counts.get('jan', 0)
100
>>> print counts.get('tim', 0)
0
```

get 메쏘드를 사용해서 상기 히스토그램 루프를 좀더 간결하게 작성할 수 있다. get 메쏘드는 딕셔너리에 키가 존재하지 않는 경우를 자동적으로 다루기 때문에, if문을 없애 4줄을 1줄로 줄일 수 있다.

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c,0) + 1
print d
```

계수기(counter) 루프를 단순화하려고 get메쏘드를 사용하는 것은 파이썬에서 흔히 사용되는 "숙어(idiom)"가 되고, 책 후반부까지 많이 사용할 것이다. 시간을 가지고서 잠시 if 문과 in 연산자를 사용한 루프와 get메쏘드를 사용한 루프를 비교해 보세요. 동일한 연산을 수행하지만, 하나가 더 간결한다.

제 2 절 딕셔너리와 파일

딕셔너리의 흔한 사용법 중의 하나는 텍스트로 작성된 파일에 단어 빈도를 세는 것이다. **http://shakespeare.mit.edu/Tragedy/romeoandjuliet/romeo_juliet.2.2.html** 사이트 덕분에 *로미오와 쥴리엣(Romeo and Juliet)* 텍스트 파일에서 시작합시다.

처음 연습으로 구두점이 없는 짧고 간략한 텍스트 버젼을 사용한다. 나중에 구두점이 포함된 전체 텍스트로 작업할 것이다.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

파일 라인을 읽고, 각 라인을 단어 리스트로 쪼개고, 루프를 돌려 사전을 이용하여 각 단어의 빈도수를 세는 파이썬 프로그램을 작성한다.

두 개의 for 루프를 사용한다. 외곽 루프는 파일 라인을 읽고, 내부 루프는 특정 라인의 단어 각각에 대해 반복한다. 하나의 루프는 *외곽* 루프가 되고, 또 다른 루프는 *내부* 루프가 되어서 중첩 루프(nested loops)라고 불리는 패턴 사례다.

외곽 루프가 한번 반복을 할 때마다 내부 루프는 모든 반복을 수행하기 때문에 내부 루프는 "좀더 빨리" 반복을 수행하고 외곽 루프는 좀더 천천히 반복을 수행하는 것으로 생각할 수 있다.

두 중첩 루프의 조합이 입력 파일의 모든 라인에 있는 모든 단어의 빈도를 계수(count)하도록 보증합니다.

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

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print counts
```

프로그램을 실행하면, 정렬되지 않은 해쉬 순서로 모든 단어의 빈도수를 출력합니다. romeo.txt **파일은** **www.py4inf.com/code/romeo.txt**에서 다운로드 가능하다.

```
python count1.py
Enter the file name: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

가장 높은 빈도 단어와 빈도수를 찾기 위해서 딕셔너리를 훑는 것이 불편하다. 좀더 도움이 되는 출력결과를 만들려고 파이썬 코드를 추가하자.

제 3 절 반복과 딕셔너리

for**문에 순서(sequence)로서 딕셔너리를 사용한다면, 딕셔너리 키를 훑는다. 루프는 각 키와 해당 값을 출력한다.**

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    print key, counts[key]
```

출력은 다음과 같다.

```
jan 100
chuck 1
annie 42
```

다시 한번, 키는 특별한 순서가 없다.

이 패턴을 사용해서 앞서 기술한 다양한 루프 숙어를 구현한다. 예를 들어, 딕셔너리에서 10 보다 큰 값을 가진 항목을 모두 찾고자 한다면, 다음과 같이 코드를 작성한다.

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    if counts[key] > 10 :
        print key, counts[key]
```

for 루프는 딕셔너리 *키(keys)*를 반복한다. 그래서, 인덱스 연산자를 사용해서 각 키에 상응하는 *값(value)*을 가져와야 한다. 여기 출력값이 있다.

```
jan 100
annie 42
```

10 이상 값만 가진 항목만 볼 수 있다.

알파벳 순으로 키를 출력하고자 한다면, 딕셔너리 객체의 keys 메쏘드를 사용해서 딕셔너리 키 리스트를 생성한다. 그리고 나서 리스트를 정렬하고, 정렬된 리스트를 루프 돌리고, 아래와 같이 정렬된 순서로 키/값 페어(key/value pair)를 출력한다.

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

다음에 출력결과가 있다.

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

처음에 keys 메쏘드로부터 얻은 정렬되지 않은 키 리스트가 있고, 그리고 나서 for 루프로 정렬된 키/값 페어(key/value pair)가 있다.

제 4 절 고급 텍스트 파싱

romeo.txt 파일을 사용한 상기 예제에서, 수작업으로 모든 구두점을 제거해서 가능한 단순하게 만들었다. 실제 텍스트는 아래 보여지는 것처럼 많은 구두점이 있다.

```
But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,
```

파이썬 split 함수는 공백을 찾고 공백으로 구분되는 토큰으로 단어를 처리하기 때문에, “soft!” 와 “soft”는 다른 단어로 처리되고 각 단어에 대해서 별도 딕셔너리 항목을 생성한다.

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

lower, punctuation, translate 문자열 메쏘드를 사용해서 상기 문제를 해결할 수 있다. translate 메쏘드가 가장 적합하다. translate 메쏘드에 대한 문서가 다음에 있다.

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

(만약 존재한다면) deletechars 에 있는 모든 문자를 삭제한다. 그리고 나서 순서수(ordinal)로 색인된 각 문자를 번역하는 256-문자열 테이블(table)을 사용해서 문자를 번역한다. 만약 테이블이 None 이면, 문자 삭제 단계만 수행된다.

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

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

프로그램을 다음과 같은 수정을 한다.

```
import string                                          # 신규 코드

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

counts = dict()
for line in fhand:
    line = line.translate(None, string.punctuation)    # 신규 코드
    line = line.lower()                                # 신규 코드
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print counts
```

translate 메쏘드를 사용해서 모든 구두점을 제거했고, lower 메쏘드를 사용해서 라인을 소문자로 수정했다. 나머지 프로그램은 변경된게 없다. 파이썬 2.5 이전 버젼에는 translate 메쏘드가 첫 매개변수로 None을 받지 않아서 translate 메쏘드를 호출하기 위해서 다음 코드를 사용하세요.

```
print a.translate(string.maketrans(' ',' '), string.punctuation
```

"파이썬 예술(Art of Python)" 혹은 "파이썬스럽게 생각하기(Thinking Pythonically)"를 배우는 일부분은 파이썬은 흔한 자료 분석 문제에 대해서 내장 기능을 가지고 있는 것을 깨닫는 것이다. 시간이 지남에 따라, 충분한 예제 코드를 보고 충분한 문서를 읽는다. 작업을 편하게 할 수 있게 이미 다른 사람이

작성한 코드가 존재하는지를 살펴보기 위해서 어디를 찾아봐야 하는지도 알게 될 것이다.

다음은 출력결과의 축약 버젼이다.

```
Enter the file name: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

출력결과는 여전히 다루기 힘들어 보입니다. 파이썬을 사용해서 정확히 찾고자는 하는 것을 찾았으나 파이썬 튜플(tuples)에 대해서 학습할 필요성을 느껴진다. 튜플을 학습할 때, 다시 이 예제를 살펴볼 것이다.

제 5 절 디버깅

점점 더 큰 데이터로 작업함에 따라, 수작업으로 데이터를 확인하거나 출력을 통해서 디버깅을 하는 것이 어려울 수 있다. 큰 데이터를 디버깅하는 몇가지 방법이 있다.

입력값을 줄여라(Scale down the input):] 가능하면, 데이터 크기를 줄여라. 예를 들어, 프로그램이 텍스트 파일을 읽는다면, 첫 10줄로 시작하거나, 찾을 수 있는 작은 예제로 시작하라. 데이터 파일을 편집하거나, 프로그램을 수정해서 첫 `n` 라인만 읽도록 프로그램을 변경하라.

오류가 있다면, `n`을 줄여서 오류를 재현하는 가장 작은 값으로 만들어라. 그리고 나서, 오류를 찾고 수정해 나감에 따라 점진적으로 늘려나가라.

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

실행 오류(runtime errors)의 일반적인 원인은 올바른 자료형(right type)이 아니기 때문이다. 이런 종류의 오류를 디버깅하기 위해서, 종종 값의 자료형을 출력하는 것으로 충분하다.

자가 진단 작성(Write self-checks): 종종 오류를 자동적으로 검출하는 코드를 작성한다. 예를 들어, 리스트 숫자의 평균을 계산한다면, 결과값은 리스트의 가장 큰 값보다 클 수 없고, 가장 작은 값보다 작을 수 없다는 것을 확인할 수 있다. "완전히 비상식적인" 결과를 탐지하기 때문에 "건전성 검사(sanity check)"라고 부른다.

또 다른 검사법은 두가지 다른 연산의 결과를 비교해서 일치하는지 살펴보는 것이다. "일치성 검사(consistency check)"라고 부른다.

고급 출력(Pretty print the output): 디버깅 출력결과를 서식화하는 것은 오류 발견을 용이하게 한다.

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

제 6 절 용어정의

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

해쉬테이블(hashtable): 파이썬 딕셔너리를 구현하기 위해 사용된 알고리즘

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

히스토그램(histogram): 계수기(counter) 집합.

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

항목(item): 키-값 페어(key-value pair)에 대한 또 다른 이름.

키(key): 키-값 페어(key-value pair)의 첫번째 부분으로 딕셔너리에 나타나는 객체.

키-값 페어(key-value pair): 키에서 값으로 매핑 표현.

룩업(lookup): 키를 가지고 해당 값을 찾는 딕셔너리 연산.

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

값(value): 키-값 페어(key-value pair)의 두번째 부분으로 딕셔너리에 나타나는 객체. 앞에서 사용한 단어 "값(value)" 보다 더 구체적이다.

제 7 절 연습문제

Exercise 9.2 커밋(commit)이 무슨 요일에 수행되었는지에 따라 전자우편 메세지를 구분하는 프로그램을 작성하세요. "From"으로 시작하는 라인을 찾고, 3번째 단어를 찾아서 요일별 횟수를 계수(count)하여 저장하세요. 프로그램 끝에 딕셔너리 내용을 출력하세요. (순서는 문제가 되지 않습니다.)

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

Sample Execution:
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

Exercise 9.3 전자우편 로그(log)를 읽고, 히스토그램을 생성하는 프로그램을 작성하세요. 딕셔너리를 사용해서 전자우편 주소별로 얼마나 많은 전자우편이 왔는지를 계수(count)하고 딕셔너리를 출력합니다.

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
```

```
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

Exercise 9.4 상기 프로그램에 누가 가장 많은 전자우편 메시지를 가졌는지 알아내는 코드를 추가하세요.

결국, 모든 데이터를 읽고, 딕셔너리를 생성한다. 최대 루프(장 7.2 참조)를 사용해서 딕셔너리를 훑어서 누가 가장 많은 전자우편 메시지를 갖는지, 그리고 그 사람이 얼마나 많은 메시지를 갖는지 출력한다.

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

Exercise 9.5 다음 프로그램은 주소 대신에 도메인 이름을 기록한다. 누가 메일을 보냈는지 대신(즉, 전체 전자우편 주소) 메시지가 어디에서부터 왔는지 출처를 기록한다. 프로그램 마지막에 딕셔너리 내용을 출력한다.

```
python schoolcount.py
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

제 10 장

튜플(Tuples)

제 1 절 튜플은 불변이다.

튜플(tuple)[1]은 리스트와 마찬가지로 순서(sequence) 값이다. 튜플에 저장된 값은 임의 자료형(type)이 될 수 있고, 정수로 색인 된다. 중요한 차이점은 튜플은 불변(immutable)하다는 것이다. 튜플은 또한 비교 가능(comparable)하고 해쉬형(hashable)이다. 따라서, 리스트 값을 정렬할 수 있고, 파이썬 딕셔너리 키 값으로 튜플을 사용할 수 있다.

구문론적으로, 튜플은 콤마로 구분되는 리스트 값이다.

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

꼭 필요하지는 않지만, 파이썬 코드를 봤을 때, 가독성을 높여 튜플을 빠르게 알아볼 수 있도록 괄호로 튜플을 감싸는 것이 일반적이다.

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

단일 요소를 가진 튜플을 생성하기 위해서 마지막 콤마를 포함해야 한다.

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

콤마가 없는 경우 파이썬에서는 `('a')`을 괄호를 가진 문자열 표현으로 간주하여 문자열로 평가한다.

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

튜플을 구축하는 다른 방법은 내장함수 `tuple`을 사용하는 것이다. 인자가 없는 경우, 빈 튜플을 생성한다.

---

[1] 재미난 사실: 단어 "튜플(tuple)"은 가변 길이 (한배, 두배, 세배, 네배, 다섯배, 여섯배, 일곱배 등) 숫자열에 붙여진 이름에서 유래한다.

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

만약 인자가 문자열, 리스트 혹은 튜플 같은 순서(sequence)인 경우, `tuple`**에 호출한 결과는 요소 순서(sequence)를 가진 튜플이 된다.**

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

튜플(`tuple`)**이 생성자 이름이기 때문에 변수명으로 튜플 사용을 피해야 한다.**

대부분의 리스트 연산자는 튜플에서도 사용 가능하다. 꺾쇠 연산자가 요소를 색인한다.

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

그리고, 슬라이스 연산자(slice operator)는 요소 범위를 선택한다.

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

하지만, 튜플 요소 중 하나를 변경하고 하면, 오류가 발생한다.

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

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

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

제 2 절 튜플 비교하기

비교 연산자는 튜플과 다른 순서(sequence)와 함께 쓸 수 있다. 파이썬은 각 순서(sequence)에서 비교를 첫 요소부터 시작한다. 만약 두 요소가 같다면, 다음 요소 비교를 진행하며 서로 다른 요소를 찾을 때까지 계속한다. 후속 요소가 아무리 큰 값이라고 하더라도 비교 고려대상에서 제외된다.

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

정렬 (`sort`) **함수도 동일한 방식으로 동작한다. 첫 요소를 먼저 정렬하지만, 동일한 경우 두 번째 요소를 정렬하고, 그 후속 요소를 동일한 방식으로 정렬한다.**

이 기능이 다음 DSU라고 불리는 패턴이 된다.

데코레이트 (Decorate) : 순서(sequence)에서 요소 앞에 하나 혹은 그 이상의 키를 가진 튜플 리스트를 구축해서 순서(sequence)를 장식한다.

정렬 (Sort) : 파이썬 내장 함수 `sort`를 사용한 튜플 리스트를 정렬한다.

언데코레이트 (Undecorate) : 순서(sequence)의 정렬된 요소만 추출하여 장식을 지웁니다.

예를 들어, 단어 리스트가 있고 가장 긴 단어부터 가장 짧은 단어 순으로 정렬한다고 가정하자.

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
   t.append((len(word), word))

t.sort(reverse=True)

res = list()
for length, word in t:
    res.append(word)

print res
```

첫번째 루프는 튜플 리스트를 생성하고, 각 튜플은 단어 앞에 길이 정보를 가진다.

정렬(`sort`)함수는 첫번째 요소, 길이를 우선 비교하고, 동률일 경우에만 두 번째 요소를 고려한다. 정렬(`sort`) 함수의 인자 `reverse=True`는 내림차순으로 정렬한다는 의미다.

두 번째 루프는 튜플 리스트를 운행하여 훑고, 길이에 따라 내림차순으로 단어 리스트를 생성한다. 그래서, 5 문자 단어는 역 알파벳 순으로 정렬되어 있다. 다음 리스트에서 "what"이 "soft" 보다 앞에 나타난다.

프로그램의 출력은 다음과 같다.

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

물론, 파이썬 리스트로 변환하여 내림차순으로 정렬된 문장은 시적인 의미를 많이 잃어버렸다.

제 3 절 튜플 대입(Tuple Assignment)

파이썬 언어의 독특한 구문론적인 기능중의 하나는 대입문의 왼편에 튜플을 놓을 수 있다는 것이다. 왼편이 순서(sequence)인 경우 한번에 하나 이상의 변수에 대입할 수 있다.

다음 예제에서, 순서(sequence)인 두개 요소를 갖는 리스트가 있다. 하나의 문장으로 순서(sequence)의 첫번째와 두번째 요소를 변수 `x`와 `y`에 대입한다.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

마술이 아니다. 파이썬은 *대략* 튜플 대입 구문을 다음과 같이 해석한다.[2]

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

스타일적으로 대입문 왼편에 튜플을 사용할 때, 괄호를 생략한다. 하지만 다음은 동일하게 적합한 구문이다.

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

튜플 대입문을 사용하는 특히 똑똑한 응용사례는 단일 문장으로 두 변수 값을 교환(swap)하는 것이다.

```
>>> a, b = b, a
```

양쪽 문장이 모두 튜플이지만, 왼편은 튜플 변수이고 오른편은 튜플 표현식이다. 오른편 각각의 값이 왼편 해당 변수에 대입된다. 대입이 이루어지기 전에 오른편의 모든 표현식이 평가된다.

왼편의 변수 갯수와 오른편의 값의 갯수는 동일해야 한다.

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

좀더 일반적으로, 오른편은 임의 순서(문자열, 리스트 혹은 튜플)가 될 수 있다. 예를 들어, 전자우편 주소를 사용자 이름과 도메인으로 분할하기 위해서 다음과 같이 프로그램을 작성할 수 있다.

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

[2]파이썬은 구문을 문자 그대로 해석하지는 않는다. 예를 들어, 동일한 것을 딕셔너리로 작성한다면, 기대한 것처럼 동작하지는 않는다.

분할 (split) 함수로부터 반환되는 값은 두개 요소를 가진 리스트다. 첫번째 요소는 uname에 두번째 요소는 domain에 대입된다.

```
>>> print uname
monty
>>> print domain
python.org
```

제 4 절 딕셔너리와 튜플

딕셔너리에는 튜플 리스트를 반환하는 items 메쏘드가 있다. 각 튜플은 키-값 페어(key-value pair)다.[3]

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = d.items()
>>> print t
[('a', 10), ('c', 22), ('b', 1)]
```

딕셔너리로부터 기대했듯이, 항목은 특별한 순서가 없다.

하지만 튜플 리스트는 리스트여서 비교가 가능하기 때문에, 튜플 리스트를 정렬할 수 있다. 딕셔너리를 튜플 리스트로 변환하는 것이 키로 정렬된 딕셔너리 내용을 출력할 수 있게 한다.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = d.items()
>>> t
[('a', 10), ('c', 22), ('b', 1)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

새로운 리스트는 키 값으로 오름차순 알파벳 순으로 정렬된다.

제 5 절 딕셔너리로 다중 대입

items 함수, 튜플 대입, for문을 조합해서, 단일 루프로 딕셔너리의 키와 값을 운행하여 훑는 멋진 코드 패턴을 만들 수 있다.

```
for key, val in d.items():
    print val, key
```

상기 루프에는 두개의 반복 변수(iteration variables)가 있다. items 함수가 튜플 리스트를 반환하고, key, val는 튜플 대입하여 딕셔너리에 있는 각각의 키-값 페어(key-value pair)를 성공적으로 반복한다.

매번 루프를 반복할 때마다, key와 value는 딕셔너리(여전히 해쉬 순으로 되어 있음)의 다음 키-값 페어(key-value pair)로 진행한다.

[3] 파이썬 3.0으로 가면서 살짝 달라졌다.

루프의 출력결과는 다음과 같다.

```
10 a
22 c
1 b
```

다시 한번 해쉬 키 순서다. (즉, 특별한 순서가 없다.)

두 기술을 조합하면, 딕셔너리 내용을 키-값 페어(key-value pair)에 저장된 *값*의 순서로 정렬하여 출력할 수 있다.

이것을 수행하기 위해서, 각 튜플이 `(value, key)` **형태인 튜플 리스트를 작성한다.** `items` **메쏘드를 사용하여 리스트** `(key, value)` **튜플을 만든다. 하지만 이번에는 키가 아닌 값으로 정렬한다. 키-값(key-value) 튜플 리스트를 생성하면, 역순으로 리스트를 정렬하고 새로운 정렬 리스트를 출력하는 것은 쉽다.**

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
...     l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

조심스럽게 각 튜플 첫번째 요소로 값(value)을 갖는 튜플 리스트를 생성했다. 튜플 리스트를 정렬하여 값으로 정렬된 딕셔너리가 생성되었다.

제 6 절 가장 빈도수가 높은 단어

로미오와 쥴리엣 2장 2막 **텍스트 파일로 다시 돌아와서, 텍스트에서 가장 빈도수가 높은 단어를 10개를 출력하기 위해서 상기 학습한 기법을 사용하여 프로그램을 보강해보자.**

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
    line = line.translate(None, string.punctuation)
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

# Sort the dictionary by value
lst = list()
```

```
for key, val in counts.items():
    lst.append( (val, key) )

lst.sort(reverse=True)

for key, val in lst[:10] :
    print key, val
```

파일을 읽고 각 단어를 문서의 단어 빈도수에 매핑(사상)하여 딕셔너리를 계산하는 프로그램 첫 부분은 바뀌지 않는다. 하지만, `counts` 를 단순히 출력하는 대신에 `(val, key)` 튜플 리스트를 생성하고 역순으로 리스트를 정렬한다.

값이 첫 위치에 있기 때문에, 비교 목적으로 값을 사용한다. 만약 동일한 값을 가진 튜플이 하나이상 존재한다면, 두번째 요소(키, key)를 살펴본다. 그래서 값이 동일한 경우 키의 알파벳 순으로 추가 정렬된다.

마지막에 다중 대입 반복을 수행하는 멋진 `for` 루프를 작성한다. 그리고, 리스트 슬라이스(`lst[:10]`)를 통해 가장 빈도수가 높은 상위 10개 단어를 출력한다.

이제 단어 빈도 분석을 위해서 작성한 프로그램의 마지막 출력결과는 원하는 바를 완수한 것처럼 보인다.

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

복잡한 데이터 파싱과 분석 작업이 이해하기 쉬운 19줄 파이썬 프로그램으로 수행된 사실이 왜 파이썬이 정보 탐색 언어로서 좋은 선택인지 보여준다.

제 7 절 딕셔너리 키로 튜플 사용하기

튜플은 해쉬형(hashable)이고, 리스트는 그렇지 못하기 때문에, 만약 딕셔너리에 사용할 복합(composite)키를 생성하려면, 키로 튜플을 사용해야 한다.

만약 성(last-name)과 이름(first-name)을 가지고 전화번호에 사상(매핑, mapping)하는 전화번호부를 생성하려고 하면, 복합키와 마주친다. 변수 `last`, `first`, `number`을 정의했다고 가정하면, 다음과 같이 딕셔너리 대입문을 작성할 수 있다.

```
directory[last,first] = number
```

꺾쇠 괄호 표현은 튜플이다. 딕셔너리를 훑기 위해서 `for` 루프에 튜플 대입을 사용한다.

```
for last, first in directory:
    print first, last, directory[last,first]
```

상기 루프가 튜플인 directory에 키를 훑는다. 각 튜플 요소를 last, first에 대입하고 나서, 이름과 해당 전화번호를 출력한다.

제 8 절 순서(sequence) : 문자열, 리스트, 튜플

여기서 리스트 튜플에 초점을 맞추었지만, 이장의 거의 모든 예제가 또한 리스트의 리스트, 튜플의 튜플, 리스트 튜플에도 동작한다. 가능한 모든 조합을 열거하는 것을 피하기 위해서, 순서의 순서(sequences of sequences)에 대해서 논의하는 것이 때로는 쉽다.

대부분의 문맥에서 다른 종류의 순서(문자열, 리스트, 튜플)는 상호 호환해서 사용될 수 있다. 그런데 왜 그리고 어떻게 다른 것보다 이것을 선택해야 될까?

명확한 것부터 시작하자. 문자열은 요소가 문자여야 하기 때문에 다른 순서(sequence)보다 제약이 따른다. 또한 문자열은 불변(immutable)이다. 새로운 문자열을 생성하는 것과 반대로, 문자열에 있는 문자를 변경하고자 한다면, 대신에 문자 리스트를 사용하는 것을 생각할 수 있다.

리스트는 튜플보다 좀더 일반적으로 사용된다. 이유는 대체로 변경가능(mutable)하기 때문이다. 하지만, 다음 몇가지 경우에 튜플이 좀더 선호된다.

1. return 문처럼 어떤 맥락에서, 리스트보다 튜플을 생성하는 것이 구문론적으로 더 간략하다. 다른 맥락에서는 리스트가 더 선호될 수 있다.
2. 딕셔너리 키로서 순서(sequence)를 사용하려면, 튜플이나 문자열같은 불변 자료형(immutable type)을 사용해야 한다.
3. 함수에 인자로 순서(sequence)를 전달하려면, 튜플을 사용하는 것이 에일리어싱(aliasing)으로 생기는 예기치 못한 행동에 대한 가능성을 줄여준다.

튜플은 불변(immutable)이어서, 기존 리스트를 변경하는 sort, reverse 같은 메쏘드를 제공하지는 않는다. 하지만, 파이썬이 제공하는 내장함수 sorted, reversed 를 통해서, 매개 변수로 임의 순서(sequence)를 전달 받아서, 같은 요소를 다른 순서로 정렬된 새로운 리스트를 반환한다.

제 9 절 디버깅

리스트, 딕셔너리, 튜플은 자료 구조(data structures)로 일반적으로 알려져 있다. 이번장에서 리스트 튜플, 키로 튜플, 값으로 리스트를 담고 있는 딕셔너리 같은 복합 자료 구조를 보기 시작했다. 복합 자료 구조는 유용하지만, 저자가 작명한 모양 오류(shape errors)라고 불리는 오류에 노출되어 있다. 즉, 자료

구조가 잘못된 자료형(type), 크기, 구성일 경우 오류가 발생한다. 혹은 코드를 작성하고, 자료의 모양이 생각나지 않는 경우도 오류의 원인이 된다.

예를 들어, 정수 하나를 가진 리스트를 기대하고, (리스트가 아닌) 일반 정수를 넘긴다면, 작동하지 않는다.

프로그램을 디버깅할 때, 정말 어려운 버그를 잡으려고 작업을 한다면, 다음 네 가지를 시도할 수 있다.

코드 읽기(reading): 코드를 면밀히 조사하고, 스스로에게 다시 읽어 주고, 코드가 자신이 작성한 의도를 담고 있는지 점검하라.

실행(running): 변경해서 다른 버젼을 실행해서 실험하라. 종종, 프로그램이 적절한 곳에 적절한 것을 보여준다면, 문제가 명확하다. 발판(scaffolding)을 만들기 위해서 때때로 시간을 들일 필요도 있다.

반추(ruminating): 생각의 시간을 갖자. 어떤 종류의 오류인가: 구문, 실행, 의미론(semantic). 오류 메시지로부터 혹은 프로그램 출력결과로부터 무슨 정보를 얻을 수 있는가? 어떤 종류 오류가 지금 보고 있는 문제를 만들었을까? 문제가 나타나기 전에, 마지막으로 변경한 것은 무엇인가?

퇴각(retreating): 어느 시점에선가, 최선은 물러서서, 최근의 변경을 다시 원복하는 것이다. 잘 동작하고 이해하는 프로그램으로 다시 돌아가서, 다시 프로그램을 작성한다.

초보 프로그래머는 종종 이들 활동 중 하나에 사로잡혀 다른 것을 잊곤 한다. 활동 각각은 고유한 실패 방식과 함께 온다.

예를 들어, 프로그램을 정독하는 것은 문제가 인쇄상의 오류에 있다면 도움이 되지만, 문제가 개념상 오해에 뿌리를 두고 있다면 그다지 도움이 되지 못한다. 만약 작성한 프로그램을 이해하지 못한다면, 100번 읽을 수는 있지만, 오류를 발견할 수는 없다. 왜냐하면, 오류는 여러분 머리에 있기 때문입니다.

만약 작고 간단한 테스트를 진행한다면, 실험을 수행하는 것이 도움이 될 수 있다. 하지만, 코드를 읽지 않거나, 생각없이 실험을 수행한다면, 프로그램이 작동될 때까지 무작위 변경하여 개발하는 "랜덤 워크 프로그램(random walk programming)" 패턴에 빠질 수 있다. 말할 필요없이 랜덤 워크 프로그래밍은 시간이 오래 걸린다.

생각할 시간을 가져야 한다. 디버깅은 실험 과학 같은 것이다. 문제가 무엇인지에 대한 최소한 한 가지 가설을 가져야 한다. 만약 두개 혹은 그 이상의 가능성이 있다면, 이러한 가능성 중에서 하나라도 줄일 수 있는 테스트를 생각해야 한다.

휴식 시간을 가지는 것은 생각하는데 도움이 된다. 대화를 하는 것도 도움이 된다. 문제를 다른 사람 혹은 자신에게도 설명할 수 있다면, 질문을 마치기도 전에 답을 종종 발견할 수 있다.

하지만, 오류가 너무 많고 수정하려는 코드가 매우 크고, 복잡하다면 최고의 디버깅 기술도 무용지물이다. 가끔, 최선의 선택은 퇴각하는 것이다. 작동하고 이해하는 곳까지 후퇴해서 프로그램을 간략화하라.

초보 프로그래머는 종종 퇴각하기를 꺼려한다. 왜냐하면, 설사 잘못되었지만, 한줄 코드를 지울 수 없기 때문이다. 삭제하지 않는 것이 기분을 좋게 한다면, 다시 작성하기 전에 프로그램을 다른 파일에 복사하라. 그리고 나서, 한번에 조금씩 붙여넣어라.

정말 어려운 버그(hard bug)를 발견하고 고치는 것은 코드 읽기, 실행, 반추, 때때로 퇴각을 요구한다. 만약 이들 활동 중 하나도 먹히지 않는다면, 다른 것들을 시도해 보세요.

제 10 절 용어정의

비교가능한(comparable): 동일한 자료형의 다른 값과 비교하여 큰지, 작은지, 혹은 같은지를 확인하기 위해서 확인할 수 있는 자료형(type). 비교가능한(comparable) 자료형은 리스트에 넣어서 정렬할 수 있다.

자료 구조(data structure): 연관된 값의 집합, 종종 리스트, 딕셔너리, 튜플 등으로 조직화된다.

DSU: "decorate-sort-undecorate,"의 약어로 리스트 튜플을 생성, 정렬, 결과 일부 추출을 포함하는 패턴.

모음(gather): 가변-길이 인자 튜플을 조합하는 연산.

해쉬형(hashable): 해쉬 함수를 가진 자료형(type). 정수, 소수점, 문자열 같은 불변형은 해쉬형이다. 리스트나 딕셔너리 처럼 변경가능한 형은 해쉬형이 아니다.

스캐터(scatter): 순서(sequence)를 리스트 인자로 다루는 연산.

(자료 구조의) 모양 (shape (of a data structure)): 자료 구조의 자료형(type), 크기, 구성을 요약.

싱글톤(singleton): 단일 요소를 가진 리스트 (혹은 다른 순서(sequence)).

튜플(tuple): 불변 요소들의 순서 (sequence).

튜플 대입(tuple assignment): 오른편 순서(sequence)와 왼편 튜플 변수를 대입. 오른편이 평가되고나서 각 요소들은 왼편의 변수에 대입된다.

제 11 절 연습문제

Exercise 10.1 앞서 작성한 프로그램을 다음과 같이 수정하세요. "From"라인을 읽고 파싱하여 라인에서 주소를 뽑아내세요. 딕셔너리를 사용하여 각 사람으로부터 메시지 숫자를 계수(count)한다.

모든 데이터를 읽은 후에 가장 많은 커밋(commit)을 한 사람을 출력하세요. 딕셔너리로부터 리스트 (count, email) 튜플을 생성하고 역순으로 리스트를 정렬한 후에 가장 많은 커밋을 한 사람을 출력하세요.

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

Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

Exercise 10.2 이번 프로그램은 각 메시지에 대한 하루 중 시간의 분포를 계수(count)한다. "From" 라인으로부터 시간 문자열을 찾고 콜론(:) 문자를 사용하여 문자열을 쪼개서 시간을 추출합니다. 각 시간별로 계수(count)를 누적하고 아래에 보여지듯이 시간 단위로 정렬하여 한 라인에 한시간씩 계수(count)를 출력합니다.

```
Sample Execution:
python timeofday.py
Enter a file name: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

Exercise 10.3 파일을 읽고, 빈도(frequencey)에 따라 내림차순으로 *문자(letters)*를 출력하는 프로그램을 작성하세요. 작성한 프로그램은 모든 입력을 소문자로 변환하고 a-z 문자만 계수(count)한다. 공백, 숫자, 문장기호 a-z를 제외한 다른 어떤 것도 계수하지 않습니다. 다른 언어로 구성된 텍스트 샘플을 구해서 언어마다 문자 빈도가 어떻게 변하는지 살펴보세요. 결과를 `wikipedia.org/wiki/Letter_frequencies` 표와 비교하세요.