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

딕셔너리(Dictionaries)

딕셔너리(dictionary)는 리스트 같지만 좀더 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. `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'`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, 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-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'
```

len함수를 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 리스트이 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 에 좀더 많은 것을 읽을 수 있다.

Exercise 1.1 words.txt의 단어를 읽어서 딕셔너리에 키로 저장하는 프로그램을 작성하세요. 값이 무엇이든지 상관없습니다. 딕셔너리에 문자열을 확인하는 가장 빠른 방법으로 in 연산자를 사용할 수 있습니다.

1.1 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, 틱셔너리에 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
```

카우터 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 딕겨너리에 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 "숙어(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.

1.2 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 쥴리엣(Romeo and Juliet)* 텍스트 파일에서 시작합시다.

처음 연습으로 구두점이 없는 짧고 간략한 텍스트 버젼을 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.

1.3 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.

1.4 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 메또드에 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 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 많이 흔한 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 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 필요가 있다. 튜플을 학습하기 위해서 다시 이 예제를 살펴볼 것이다.

1.5 디버깅

점점 더 큰 데이터로 작업함에 따라, 수작업으로 데이터를 확인하거나 출력을 통해서 디버깅을 하는 것이 어려울 수 있다. 큰 데이터를 디버깅하는 몇가지 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, `n`을 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 항못의 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. 또다른 검사법은 두가지 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.

1.6 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): 카운터 집합.

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

항목(item): 키-밸류 페어에 대한 또다른 이름.

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

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

1.7 연습문제

Exercise 1.2 커밋(commit)이 무슨 요일에 수행되었는지에 따라 전자우편 메세지를 구분하는 프로그램을 작성하세요. "From"으로 시작하는 라인을 찾고, 3번째 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 1.3 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 1.4 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 1.5 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}
```