3. 내장 자료구조, 함수, 파일

이 장에서는 이 책 전반에 걸쳐 사용하게 될 파이썬 언어에 내장되어 있는 기능을 알아보자.

pandas나 Numpy 같은 애드온 라이브러리는 대규모 데이터 계산을 위한 진보된 기능을 제공 하지만, 내장되어 있는 기능은 파이썬 내장 자료 처리 도구와 함께 사용해야 한다.

먼저 파이썬의 기본 자료구조인 튜플, 리스트, 사전 그리고 집합부터 알아보고 재사용 가능한 파이썬 함수를 작성하는 방법을 살펴본다. 마지막으로 파이썬 file 객체의 원리를 살펴보고 하드디스크에 직접 파일을 읽고 쓰는 방식을 알아보자.

3.1 자료구조와 순차 자료형

파이썬의 자료구조는 단순하지만 강력하다. 자료구조의 사용법을 숙지하는 것이 파이썬의 고수가 되는 지름길이다.

3.1.1 튜플

튜플은 1차원의 고정된 크기를 가지는 변경 불가능한 순차 자료형이다. 튜플을 생성하는 가장 쉬운 방법은 쉼표로 구분된 값을 대입하는 것이다.

```
In [7]: tup = 4,5,6

In [8]: tup
Out[8]: (4, 5, 6)
```

괄호를 사용해서 값을 묶어줌으로써 중첩된 튜플을 정의할 수 있다. 아래 예제는 튜플의 튜플을 생성한다.

```
In [9]: nested_tup = (4,5,6), (7,8)

In [10]: nested_tup
Out[10]: ((4, 5, 6), (7, 8))
```

모든 순차 자료형이나 이터레이터는 tuple 매서드를 호출해 튜플로 변환할 수 있다.

```
In [11]: tuple ([4,0,2])
Out[11]: (4, 0, 2)

In [12]: tup = tuple('string')

In [13]: tup
Out[13]: ('s', 't', 'r', 'i', 'n', 'g')

In [14]: tuple('string')
Out[14]: ('s', 't', 'r', 'i', 'n', 'g')
```

튜플의 각 원소는 대괄호 []을 이용해서 다른 순차 자료형처럼 접근할 수 있다. C, C++, 자바 그리고 다른 많은 언어처럼 순차 자료형의 색인은 0 부터 시작한다.

```
In [15]: tup[0]
Out[15]: 's'
```

튜플에 저장된 객체 자체는 변경이 가능하지만 한 번 생성되면 각 슬롯에 저장된 객체를 변경하는 것은 불가하다.

```
In [16]: tup = tuple(['foo',[1,2],True])

In [17]: tup[2] = False
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-b89d0c4ae599> in <module>
----> 1 tup[2] = False

TypeError: 'tuple' object does not support item assignment
```

튜플 내에 저장된 객체는 그 위치에서 바로 변경이 가능하다.

```
In [18]: tup[1].append(3)

In [19]: tup
Out[19]: ('foo', [1, 2, 3], True)
```

+연산자를 이용해서 튜플을 이어붙일 수 있다.

```
In [20]: (4,None,'foo')+(6,0)+('bar',)
Out[20]: (4, None, 'foo', 6, 0, 'bar')
```

튜플에 정수를 곱하면 리스트와 마찬가지로 튜플의 복사본이 반복되어 늘어난다.

```
In [21]: ('foo', 'bar')*4
Out[21]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

튜플 안에 있는 객체는 복사되지 않고 그 객체에 대한 참조만 복사된다는 점을 기억하자.

튜플에서 값 분리하기

만일 튜플과 같은 표현의 변수에 튜플을 **대입**하면 파이썬은 등호(=) 오른쪽에 있는 변수에서 값을 **분리**한다.

```
In [22]: tup = (4,5,6)

In [23]: a,b,c = tup

In [24]: b
Out[24]: 5
```

중첩된 튜플을 포함하는 순차 자료형에서도 값을 분리해낼 수 있다.

```
In [25]: tup = 4,5,(6,7)

In [26]: a,b,(c,d)=tup

In [27]: d
Out[27]: 7
```

이 기능을 사용하면 변수의 이름을 바꿀 때 다른 언어에서는 아래처럼 처리하는 것을 쉽게 해결할 수 있다.

```
tmp = a
a = b
b = tmp
```

즉, 파이썬에서는 다음과 같이 하여 두 변수의 값을 쉽게 바꿀 수 있다.

```
In [28]: a,b =1,2

In [29]: a
Out[29]: 1

In [30]: b
Out[30]: 2

In [31]: b, a=a,b

In [32]: a
Out[32]: 2

In [33]: b
Out[33]: 1
```

튜플이나 리스트를 순회할 때도 흔히 이 기능을 활용한다.

```
In [34]: seq = [(1,2,3),(4,5,6),(7,8,9)]

In [35]: for a,b,c in seq:
    ...:     print('a={0},b={1}, c={2}'.format(a,b,c))
    ...:
a=1,b=2, c=3
a=4,b=5, c=6
a=7,b=8, c=9
```

이 기능은 함수에게 여러 개의 값을 반환할 때도 자주 사용하는데 이는 나중에 다시 살펴보겠다. 파이썬은 최근에 튜플의 처음 몇몇 값을 '끄집어내야' 하는 상황을 위해 튜플에서 값을 분리하는 보다 진보된 방법을 수용했다. 이 경우 특수한 문법인 *rest를 사용하는데 함수의 시그니처에서 길이를 알 수 없는 건 인자를 담기 위한 방법으로도 사용된다.

```
In [36]: values = 1,2,3,4,5

In [37]: a,b, *rest = values

In [38]: a,b
Out[38]: (1, 2)

In [39]: rest
Out[39]: [3, 4, 5]
```

rest는 필요 없는 값을 무시하기 위해 사용하기도 한다. rest라는 이름 자체에는 특별한 의미가 없다. 불필요한 변수라는 것을 나타내기 위해 _를 사용하는 관습도 있다.

```
In [40]: a,b, *_=values

In [41]: _
Out[41]: [3, 4, 5]
```

튜플 메서드

튜플의 크기와 내용은 변경 불가능하므로 인스턴스 메서드가 많지 않다. 유용하게 사용되는 메서드 중 하나는 주어진 값과 같은 값이 몇 개 있는지 반환하는 count 메서드다. (리스트에서도 사용가능 하다).

```
In [42]: a = (1,2,2,2,3,4,2)

In [43]: a.count(2)
Out[43]: 4
```

3.1.2 리스트

튜플과는 대조적으로 리스트는 크기나 내용의 변경이 가능하다. 리스트는 대괄호 []나 list 함수를 사용해서 생성할 수 있다.

```
In [44]: a_list = [2,3,7, None]

In [45]: tup = ('foo','bar','baz')

In [46]: b_list = list(tup)

In [47]: b_list
Out[47]: ['foo', 'bar', 'baz']

In [48]: b_list[1]='peekaboo'

In [49]: b_list
Out[49]: ['foo', 'peekaboo', 'baz']
```

리스트와 튜플은 의미적으로 비슷한 (바로 튜플을 수정할 수 없지만) 객체의 1차원 순차 자료형이며 많은 함수에서 교차적으로 사용할 수 있다.

list 함수는 이터레이터나 제너레이터 표현에서 실제 값을 모두 담기 위한 용도로도 자주 사용된다.

```
In [50]: gen = range(10)

In [51]: gen
Out[51]: range(0, 10)

In [52]: list(gen)
Out[52]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

원소 추가하고 삭제하기

append 메서드를 사용해서 리스트의 끝에 새로운 값을 추가할 수 있다.

```
In [53]: b_list.append('dwarf')

In [54]: b_list
Out[54]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

insert 메서드를 사용해서 리스트의 특정 위치에 값을 추가할 수 있다.

```
In [55]: b_list.insert(1,'red')

In [56]: b_list
Out[56]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

CAUTION insert는 append에 비해 연산비용이 많이 든다. insert로 값을 추가하면 추가된 위치 이후의 원소들은 새로 추가될 원소를 내부적으로 모두 자리를 옮겨야 하기 때문이다. 순차 자료형의 시작과 끝 지점에 원소를 추가하고 싶다면 이런 용도로 사용할 수 있는 양방향 큐인 collections.deque를 사용하자.

insert 메서드와 반대 개념으로 pop 메서드가 있다. pop 메서드는 특정 위치의 값을 반환하고 해당 값을 리스트에서 삭제한다.

```
In [57]: b_list.pop(2)
Out[57]: 'peekaboo'

In [58]: b_list
Out[58]: ['foo', 'red', 'baz', 'dwarf']
```

remove 메서드를 이용해서 원소를 삭제할 수 있는데, 삭제는 리스트에서 제일 앞에 위치한 값부터 이뤄진다.

```
In [59]: b_list.append('foo')

In [60]: b_list
Out[60]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [61]: b_list.remove('foo')

In [62]: b_list
Out[62]: ['red', 'baz', 'dwarf', 'foo']
```

성능이 큰 문제가 되지 않는다면 append와 remove 메서드를 사용해서 파이썬의 리스트를 여러 종류의 데이터를 담을 수 있는 자료구조로 사용할 수 있다.

리스트 이어붙이기

튜플과 마찬가지로 +연산자를 이용하면 두 개의 리스트를 합칠 수 있다.

```
In [63]: [4,None, 'foo'] + [7,8,(2,3)]
Out[63]: [4, None, 'foo', 7, 8, (2, 3)]
```

만일 리스트를 미리 정의해두었다면 extend 메서드를 사용해서 여러개의 값을 추가할 수 있다. 리스트를 이어붙이면 새로운 리스트를 생성하고 값을 복사하게 되므로 상대적으로 연산비용이 높다는 점을 기억하자. 큰 리스트일수록 extend 메서드를 사용해서 기존의 리스트에 값을 추가하는 것이 일반적으로 더 나은 선택이다.

```
everything = []
for chunk in list_of_lists:
    everything.extend(chunk)
```

위 코드가 아래 코드보다 좀 더 빠르다.

```
everything = []
for chunk in list_of_lists:
    everything = everything + chunk
```

정렬

sort 함수를 이용해서 새로운 리스트를 생성하지 않고 있는 그대로 리스트를 정렬할 수 있다.

```
In [64]: a = [7,2,5,1,3]

In [65]: a.sort()

In [66]: a
Out[66]: [1, 2, 3, 5, 7]
```

sort는 편의를 위해 몇 가지 옵션을 제공한다. 그중 하나는 정렬 기준으로 사용할 값을 반환하는 함수 이다. 예를 들어 다음과 같이 문자열이 들어 있는 리스트를 문자열의 길이 순으로 정렬할 수 있다.

```
In [67]: b= ['saw','small','He','foxes','six']

In [68]: b.sort(key=len)

In [69]: b
Out[69]: ['He', 'saw', 'six', 'small', 'foxes']
```

일반적인 순차 자료형의 정렬된 복사본을 생성하는 sorted 함수도 살펴보자.

이전 탐색과 정렬된 리스트 유지하기

내장 bisect 모듈은 이진 탐색과 정렬된 리스트에 값을 추가하는 기능을 제공한다. bisect.bisect 메서드는 값이 추가될 때 리스트가 정렬된 상태를 유지할 수 있는 위치를 반환하며 bisect.insort는 실제로 정렬된 상태를 유지한 채 값을 추가한다.

```
In [70]: import bisect

In [71]: c = [1,2,2,2,3,4,7]
```

```
In [72]: bisect.bisect(c,2)
Out[72]: 4

In [73]: bisect.bisect(c,1)
Out[73]: 1

In [74]: bisect.bisect(c,3)
Out[74]: 5

In [75]: bisect.bisect(c,5)
Out[75]: 6

In [76]: bisect.bisect(c,4)
Out[76]: 6

In [77]: bisect.bisect(c,7)
Out[77]: 7

In [78]: bisect.bisect(c,9)
Out[78]: 7

In [79]: bisect.insort(c,6)

In [80]: c
Out[80]: [1, 2, 2, 2, 3, 4, 6, 7]
```

CAUTION bisect 모듈 함수는 리스트가 정렬된 상태인지 검사하지 않으므로 연산비용이 높을 수 있다. 그리고 정렬되지 않은 리스트에 대해 모듈 함수를 수행하면 오류 없이 수행되지만 정확하지 않은 값을 반환한다.

슬라이싱

리스트와 같은 자료형 (배열, 튜플, ndarray)은 색인 연산자 []안에 start:stop을 지정해서 원하는 크기만큼 잘라낼 수 있다.

```
In [81]: seq = [7,2,3,7,5,6,0,1]

In [82]: seq[1:5]
Out[82]: [2, 3, 7, 5]
```

슬라이스에 다른 순차 자료형을 대입하는 것도 가능하다.

```
In [83]: seq[3:4] = [6,3]

In [84]: seq
Out[84]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

색인의 시작 (start) 위치에 있는 값은 포함되지만 끝(stop) 위치에 있는 값은 포함되지 않는다.

따라서 슬라이싱 결과의 개수는 stop - start 이다. 색인의 시작(start) 값이나 끝 (stop) 값은 생략할 수 있는데, 이 경우 생략된 값은 각각 순차 자료형의 처음 혹은 마지막 값이 된다.

```
In [85]: seq[:5]
Out[85]: [7, 2, 3, 6, 3]

In [86]: seq[3:]
Out[86]: [6, 3, 5, 6, 0, 1]
```

음수 색인은 순차 자료형의 끝에서부터의 위치를 나타낸다.

```
In [87]: seq[-4:]
Out[87]: [5, 6, 0, 1]

In [88]: seq[-6:-2]
Out[88]: [6, 3, 5, 6]
```

R이나 매트랩 사용자라면 슬라이싱 문법에 익숙해지는 데 시간이 걸릴 수 있다. 그림에서는 슬라이스의 가장자리에 색인을 표시하여 어디서 슬라이싱이 시작되고 끝나는지 양수와 음수 색인으로 확인할 수 있도록 하였다.

두 번째 콜론 다음에 간격(step)을 지정할 수 있는데, 하나 걸러 다음 원소를 선택하려면 다음과 같이 표현한다.

```
In [89]: seq[::2]
Out[89]: [7, 3, 3, 6, 1]

In [90]: seq[::-1]
Out[90]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

3.1.3 내장 순차 자료형 함수

파이썬에는 순차 자료형에 사용할 수 있는 매우 유용한 함수가 있는데, 이 함수들은 꼭 익혀서 기회가 될 때마다 사용할 수 있어야 한다.

enumerate

이 함수는 순차 자료형에서 현재 아이템의 색인을 함께 처리하고자 할 때 흔히 사용한다. 다음 코드를 보자.

```
i=0
for value in collection:
    #value를 사용하는 코드 작성
    i+=1
```

이는 매우 흔한 코드인데 파이썬에는 (i,value) 튜플을 반환하는 enumerate 라는 함수가 있다.

위 코드를 enumerate를 사용해서 다시 작성하면 다음과 같다.

```
for i,value in enumerate(collection):
    #value를 사용하는 코드 작성
```

색인을 통해 데이터에 접근할 때 enumerate를 사용하는 유용한 패턴은 순차 자료형에서의 값과 그 위치를 dict에 넘겨주는 것이다.

```
In [91]: some_list = ['foo', 'bar', 'baz']

In [92]: mapping = {}

In [93]: for i,v in enumerate(some_list):
    ...:     mapping[v] = i
    ...:

In [94]: mapping
Out[94]: {'foo': 0, 'bar': 1, 'baz': 2}
```

sorted

sorted 함수는 정렬된 새로운 순차 자료형을 반환한다.

```
In [95]: sorted([7,1,2,6,0,3,2])
Out[95]: [0, 1, 2, 2, 3, 6, 7]

In [96]: sorted(['horse race'])
Out[96]: ['horse race']

In [97]: sorted('horse race')
Out[97]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

sorted 함수는 리스트의 sort 메서드와 같은 인자를 취한다.

zip

zip함수는 여러 개의 리스트나 튜플 또는 다른 순차 자료형을 서로 짝지어서 튜플의 리스트를 생성한다.

```
In [98]: seq1 = ['foo','bar', 'baz']

In [99]: seq2 = ['one','two', 'three']

In [100]: zipped = zip(seq1, seq2)

In [102]: list(zipped)
Out[102]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

zip함수는 여러 개의 순차 자료형을 받을 수 있으며 반환되는 리스트의 크기는 넘겨받은 순차 자료형 중 **가장 짧은**크기로 정해진다.

```
In [103]: seq3 = [False, True]

In [104]: list(zip(seq1,seq2,seq3))
Out[104]: [('foo', 'one', False), ('bar', 'two', True)]
```

```
In [107]: seq3.append('fff')

In [108]: seq3
Out[108]: [False, True, 'fff']
```

zip함수의 아주 흔한 사용 예는 여러 개의 순차 자료형을 동시에 순회하는 경우인데, enumerate와 함께 사용되기도 한다.

```
In [116]: for i,(a,b) in enumerate (zip(seq1,seq2)):
     ...:     print('{0}:{1},{2}'.format(i,a,b))
     ...:
0:foo,one
1:bar,two
2:baz,three
```

zip함수를 사용해서 이렇게 짝지어진 순차 자료형을 다시 풀어낼 수 있다. 이를 이용해서 리스트의 **로우**를 리스트의 **칼럼**으로 변환하는 것도 가능하다. 문법은 다음과 같이 약간 복잡하다.

```
In [122]: pitchers = [('Nolan','Ryan'),('Roger','Clemens'),('Schilling','Curt')]

In [123]: pitchers
Out[123]: [('Nolan', 'Ryan'), ('Roger', 'Clemens'), ('Schilling', 'Curt')]

In [124]: first_names, last_names = zip(*pitchers)

In [126]: first_names
Out[126]: ('Nolan', 'Roger', 'Schilling')

In [127]: last_names
Out[127]: ('Ryan', 'Clemens', 'Curt')
```

reversed

reversed는 순차 자료형을 역순으로 순회한다.

```
In [128]: list(reversed(range(10)))
Out[128]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

3.1.4 사전

dict(사전)는 파이썬 내장 자료구조 중에서 가장 중요하다. 일반적으로 **해시맵**또는 **연관 배열**이라고 널리 알려져 있다. 사전은 유연한 크기를 가지는 **키-값** 쌍으로, **키**와 **값**은 모두 파이썬 객체이다 .

사전을 생성하는 방법은 중괄호 {}를 사용하여 콜론으로 구분된 키와 값을 둘러싸는 것이다.

```
In [129]: empty_dict = {}

In [130]: d1 = {'a':'some value', 'b':[1,2,3,4]}

In [131]: d1
Out[131]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

리스트나 튜플을 사용하는 것처럼 사전의 값에 접근하거나 값을 입력할 수 있다.

```
In [132]: d1[7] = 'an interger'

In [133]: d1
Out[133]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an interger'}

In [134]: d1['b']
Out[134]: [1, 2, 3, 4]
```

사전에 어떤 키가 있는지 확인하는 것도 리스트나 튜플과 같은 문법으로 확인할 수 있다.

```
In [135]: 'b' in d1
Out[135]: True

In [136]: 'c' in d1
Out[136]: False

In [138]: '7' in d1
Out[138]: False

In [139]: 7 in d1
Out[139]: True

In [140]: 'a' in d1
Out[140]: True
```

del1 예약어나 pop메서드(값을 반환함과 동시에 해당 키를 삭제한다)를 사용해서 사전의 값을 삭제할 수 있다.

```
In [141]: d1[5] = 'some value'

In [142]: d1
Out[142]: {'a': 'some value', 'b': [1, 2, 3, 4]
            , 7: 'an interger', 5: 'some value'}
```

```
In [143]: d1['dummy']='another value'

In [144]: d1
Out[144]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an interger',
 5: 'some value',
 'dummy': 'another value'}

In [145]: del d1[5]

In [146]: d1
Out[146]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an interger',
 'dummy': 'another value'}

In [147]: ret = d1.pop('dummy')
```

```
In [148]: ret
Out[148]: 'another value'


In [149]: d1
Out[149]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an interger'}
```

keys와 values 메서드는 각각 키와 값이 담긴 이터레이터를 반환한다. 키-값 쌍은 일정한 기준으로 정렬되어 있진 않지만 keys 메서드와 values 메서드에서 반환하는 리스트는 같은 순서를 가진다.

```
In [150]: list(d1.keys())
Out[150]: ['a', 'b', 7]

In [151]: list(d1.values())
Out[151]: ['some value', [1, 2, 3, 4], 'an interger']
```

update 메서드를 사용해서 하나의 사전을 다른 사전과 합칠 수 있다.

```
In [152]: d1.update({'b':'foo','c':12})

In [153]: d1
Out[153]: {'a': 'some value', 'b': 'foo', 7: 'an interger', 'c': 12}
```

순차 자료형에서 사전 생성하기

두 개 자료형의 각 원소를 짝지어서 사전으로 만드는 일은 흔히 접한다. 그렇게 하기 위해 다음과 같은 코드를 작성해보자.

```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```

본질적으로 사전은 2개짜리 튜플로 구성되어 있으므로 dict 함수가 2개 짜리 튜플의 리스트를 인자로 받아서 사전을 생성하는 일은 그다지 놀라운 일은 아니다.

```
In [154]: mapping = dict(zip(range(5), reversed(range(5))))

In [155]: mapping
Out[155]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

사전을 생성하는 세련된 방법인 **사전 표기법**을 알아본다.

기본 값

아래는 매우 일반적인 로직이다.

```
if key in some_dict:
    value = some_dict[key]
else:
    value = defalut_value
```

사전 메서드인 get과 pop은 반환할 기본값을 받아서 위 코드에서 if-else 블록을 아래처럼 간단하게 작성할 수 있다.

```
value = some_dict.get(key, defalut_value)
```

get메서드는 기본적으로 해당 키가 존재하지 않을 경우 None을 반환하며, pop 매세드는 예외를 발생시킨다. 보통 사전에 값을 **대입**할 때는 리스트 같은 다른 컬렉션에 있는 값을 이용하는데, 예를 들어 여러 단어를 시작 글자에 따라 사전에 리스트로 저장하고 싶다면 다음처럼 할 수 있다.

```
In [156]: words = ['apple', 'bat', 'atom']

In [157]: by_letter = {}

In [158]: for word in words:
     ...:     letter = word[0]
     ...:     if letter not in by_letter:
     ...:         by_letter[letter] = [word]
     ...:     else:
     ...:         by_letter[letter].append(word)
     ...:

In [159]: by_letter
Out[159]: {'a': ['apple', 'atom'], 'b': ['bat']}
```

사전의 setdefalut 메서드를 바로 이 목적으로 사용한다. 위 코드에서 if-else 블록은 다음처럼 작성할 수 있다.

```
for word in words:
    letter = word[0]
    by_letter.setdefalut(letter, []).append(word)
```

내장 collerctions 모듈은 defaultfict 라는 유용한 클래스를 담고 있는데. 이 클래스를 사용하면 위 과정을 좀 더 쉽게 할 수 있다.

자료형 혹은 사전의 각 슬롯에 담길 기본값을 생성하는 함수를 넘겨서 사전을 생성하는 것이다.

```
from collections import defaultdict
by_letter = defalutdict(list)
for word in words:
    by_letter[word[0]].append(word)
```

유효한 사전 키

사전의 값으로 어떤 파이썬 객체라도 가능하지만 키는 스칼라형(정수, 실수, 문자열)이나 튜플(튜플에 저장된 값 역시 값이 바뀌지 않는 객체여야 한다)처럼 값이 바뀌지 않는 객체만 가능하다.

기술적으로는 **해시 가능**해야 한다는 뜻이다. 어떤 객체가 해시 가능한지 (즉, 사전의 키로 사용할 수 있는지)는 hash함수를 사용해서 검사할 수 있다.

```
In [162]: hash('string')
Out[162]: -4458886828098416009

In [164]: hash((1,2,(2,3)))
Out[164]: 1097636502276347782
```

리스트를 키로 사용하기 위한 한 가지 방법은 리스트를 튜플로 변경하ㅡㄴ 것이다.

```
In [165]: d = {}

In [166]: d[tuple([1,2,3])]=5

In [167]: d
Out[167]: {(1, 2, 3): 5}
```

3.1.5 집합

집한은 유일한 원소만 담는 정렬되지 않은 자료형이다. 사전과 유사하지만 값은 없고 키만 가지고 있다. 집합은 두 가지 방법으로 생성할 수 있는데 set함수를 이용하거나 중괄호를 이용해서 생성할 수 있다.

```
In [168]: set([2,2,2,1,3,3])
Out[168]: {1, 2, 3}

In [169]: {2,2,2,1,3,3}
Out[169]: {1, 2, 3}
```

집합은 합집합, 교집합, 차집합, 대칭차집합 같은 산술 **집합 연산**을 제공한다. 다음과 같은 두 개의 집합이 있다고 하자.

두 집합의 합집합은 두 집합의 모든 원소를 모은 집합이다. union 메서드를 사용하거나 | 이항 연산자로 구할 수 있다.

교집합은 두 집합에 공통으로 존재하는 원소만 모은 집합이다. intersectioin 메서드를 사용하거나 & 이항 연산자로 구할 수 있다.

```
In [170]: a = {1,2,3,4,5}

In [171]: b = {3,4,5,6,7,8}

In [172]: a.union(b)
Out[172]: {1, 2, 3, 4, 5, 6, 7, 8}

In [173]: a|b
Out[173]: {1, 2, 3, 4, 5, 6, 7, 8}

In [174]: a.intersection(b)
Out[174]: {3, 4, 5}

In [175]: a & b
Out[175]: {3, 4, 5}
```

함수	대체 문법	설명
a.add(x)	N/A	a에 x를 추가한다.
a.clear()	N/A	모든 원소를 제거하고 빈 상태로 되돌린다.
a.remove(x)	N/A	a에서 x를 제거한다.
a.pop	N/A	a에서 임의의 원소를 제거한다. 비어 있는 경우 KeyError를 발생시킨다.
a.union(b)	a | b	a와 b의 합집합
a.update(b)	a |= b	a에 a와 b의 합집합을 대입한다.
a.intersection(b)	a & b	a와 b의 교집합
a.intersection_update(b)	a &=b	a에 a와 b의 교집합을 대입한다.
a.difference(b)	a -b	a와 b의 차집합
a.difference_update(b)	a -=b	a에 a와 b의 차집합을 대입한다.
a.symmetric_difference(b)	a^b	a와 b의 대칭차집합
a.symmetric_difference_update(b)	a^=b	a에 a와 b의 대칭차집합을 대입한다.
a.issubset(b)	N/A	a의 모든 원소가 b에 속할 경우 true
a.issuperset(b)	N/A	a가 b의 모든 원소를 포함할 경우 true
a.isdisjoint(b)	N/A	a와 b 모두에 속하는 원소가 없을 경우 true

```
In [2]: In [170]: a = {1,2,3,4,5}^M
   ...: ^M
   ...: In [171]: b = {3,4,5,6,7,8}

In [4]: a.symmetric_difference(b)
Out[4]: {1, 2, 6, 7, 8}
```

모든 논리 집합 연산은 연산 결과를 좌항에 대입하는 함수도 따로 제공한다. 큰 집합을 다룰 때 유용하게 사용할 수 있다.

```
In [7]: a = {1,2,3,4,5,6,7,8}

In [8]: c = a.copy()

In [9]: b
Out[9]: {3, 4, 5, 6, 7, 8}

In [10]: c |=b

In [11]: c
Out[11]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [13]: d = b.copy()

In [14]: d = a.copy()

In [15]: d &=b

In [16]: d
Out[16]: {3, 4, 5, 6, 7, 8}
```

사전처럼 집합 원소들도 일반적으로 변경이 불가능해야 한다. 리스트 같은 원소를 담으려면 튜플로 변경해야 한다.

```
In [17]: my_data = {1,2,3,4}

In [18]: my_set = {tuple(my_data)}

In [19]: my_set
Out[19]: {(1, 2, 3, 4)}
```

어떤 집합이 다른 집합의 부분집합인지 확대집합인지 검사할 수도 있다.

```
In [21]: a_set = {1,2,3,4,5}

In [22]: {1,2,3}.issubset(a_set)
Out[22]: True

In [23]: a_set.issuperset({1,2,3})
Out[23]: True
```

만일 집합의 내용이 같다면 두 집합은 동일하다.

```
In [24]: {1,2,3}=={3,2,1}
Out[24]: True
```

3.1.6 리스트, 집합,사전 표기법

리스트 표기법은 파이썬 언어에서 가장 사랑받는 기능 중 하나이다. 이를 이용하면 간결한 표현으로 새로운 리스트를 만들 수 있다. 기본 형식은 다음과 같다.

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

필터 조건은 생략 가능하다. 예를 들어 문자열 리스트가 있다면 아래처럼 문자열의 길이가 2 이하인 문자열은 제외하고 나머지를 대문자로 바꾸는게 가능하다.

```
In [25]: strings = ['a','as','bat','car','dove','python']

In [27]: [x.upper() for x in strings if len(x) > 2]
Out[27]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

집합과 사전에 대해서도 리스트 표기법과 같은 방식으로 적용할 수 있다. 사전 표기법은 다음과 같다.

```
dict_comp = {key-expr : value-expr for value in collection
             if condition}
```

집합 표기법은 대괄호 대신 중괄호를 쓴다는 점만 빼면 리스트 표기법과 동일하다.

```
set_comp = {expr for value in collection if condition}
```

리스트 표기법과 마찬가지로 집합과 사전 표기법 역시 문법적 관용으로, 간결한 코드 작성을 통해 코드의 가독성을 높여준다. 위에서 살펴본 문자열 리스트를 생각해보자. 리스트 내의 문자열들의 길이를 담고 있는 집합을 생성하려면 집합 표기법을 이용하여 다음과 같이 처리할 수 있다.

```
In [28]: unique_lengths = {len(x) for x in strings}

In [29]: unique_lengths
Out[29]: {1, 2, 3, 4, 6}
```

map 함수를 이용해서 함수적으로 표현할 수도 있다.

```
In [30]: set(map(len, strings))
Out[30]: {1, 2, 3, 4, 6}
```

사전 표기법의 예제로, 리스트에서 문자열의 위치를 담고 있는 사전을 생성해보자.

```
In [31]: loc_mapping = {val : index for index, val in enumerate(strings)}

In [32]: loc_mapping
Out[32]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

중첩된 리스트 표기법

다음과 같이 영어 이름과 스페인어 이름을 담고 있는 리스트의 리스트가 있다고 하자.

```
In [40]: all_data = [['john','emily','michael','mary','steven'],
                     ['maria','juan','javier','matalia']]
```

몇몇 파일에서 이들 이름을 읽어 와서 영어와 스페인어 이름을 따로 저장했으며 각 이름에서 알파벳 e가 2개 이상 포함된 이름의 목록을 구한다고 가정하자. 리스트는 for문을 사용해서 다음처럼 구할 수 있다.

```
In [43]: result = [name for names in all_data for name in names
    ...:           if name.count('e') >= 2 ]

In [44]: result
Out[44]: ['steven']
```

위 코드 전체를 **중첩된 리스트 표기법**을 이용해서 위와 같은 코드를 작성할 수 있다.

중첩된 리스트 표기법을 처음 접하면 머릿속에서 그려내기가 어려울 수 있다. 리스트 표기법에서 for 부분은 중첩의 순서에 따라 나열되며 필터 조건은 끝에 위치한다. 숫자 튜플이 담긴 리스트를 그냥 단순한 리스트로 변환하는, 다음 예제를 살펴보자.

```
In [45]: some_tuples = [(1,2,3),(4,5,6),(7,8,9)]

In [46]: flattened = [x for tup in some_tuples for x in tup]

In [47]: flattened
Out[47]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

리스트 표기법 대신 for문을 사용할 경우 for 표현식의 순서도 리스트 표기법의 순서와 동일함을 기억하자.

```
flattened = []
for tup in some_tuples:
    for x in tup:
        flattened.append(x)
```

몇 단계의 중첩이라도 가능하지만 만약 2단계 이상의 중첩이 필요하다면 자료구조 설계에 대해 다시한번 생각해봐야 한다. 위 문법과 리스트 표기법 안에서 리스트 표기법을 사용하는 것의 차이를 구별하는 것은 중요하다.

```
In [48]: [[x for x in tup]for tup in some_tuples]
Out[48]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

위 코드는 내부 리스트의 원소를 모두 끄집어낸 리스트를 생성하는 것이 아닌 리스트의 리스트를 생성한다.

3.2 함수

함수는 파이썬에서 코드를 재사용하고 조직화하기 위한 가장 중요한 수단이다. 경험적으로 같은 일을 반복하거나 비슷한 코드를 한 번 이상 실행햐애 할 것이 예상되면 재사용 가능한 함수를 작성하는 것이 더 나을 것이다. 함수는 파이썬 명령들의 집합에 이름을 지어 보다 가독성이 좋은 코드로 짜야한다.

함수는 def 예약어로 정의하고 return 예약어를 사용해서 값을 반환한다.

```
In [49]: def my_function(x,y,z=1.5):
    ...:     if z>1:
    ...:         return z*(x+y)
    ...:     else:
    ...:         return x /(x+y)
    ...:

In [50]: my_function(5,6,0.7)
Out[50]: 0.45454545454545453

In [51]: my_function(5,6)
Out[51]: 16.5
```

return 문은 몇 개가 되든 상관없다. 함수 블록이 끝날 때까지 return문이 없다면 None이 자동으로 반환된다.

함수는 여러 개의 일반 인자와 키워드 인자를 받을 수 있다. 키워드 인자는 흔히 기본값 또는 부수적인 인자를 지정하기 위해 사용한다. 위 함수에서 x와 y는 일반인자이며, z는 키워드인자다.

함수의 키워드 인자는 항상 일반 인자 다음에 와야 한다는 규칙이 있다. 키워드 인자의 순서에는 제약이 없으므로 키워드 인자의 이름만 기억하고 있으면 된다. 키워드 인자의 순서는 기억할 필요없다.

3.2.1 네임스페이스, 스코프, 지역 함수

함수는 전역과 지역 두 가지 스코프 영역에서 변수를 참조한다. 변수의 스코프를 설명하는 다른 용어로 **네임스페이스**가 있다. 함수 내에서 선언된 변수는 기본적으로 모두 지역 네임스페이스에 속한다. 지역 네임스페이스는 함수가 호출될 때 생성되며 함수의 인자를 통해 즉시 생성된다. 함수의 실행이 끝나면 지역 네임스페이스는 사라진다. (예외가 있지만 이 장에서는 다루는 내용을 벗어난다.) 다음 함수를 보자.

```
def func():
    a = []
    for i in range(5):
        a.append(i)
```

func()를 호출하면 비어있는 리스트 a가 생성되고 다섯 개의 원소가 리스트에 추가된다. 그리고 함수가 끝나면 이 리스트 a는 사라진다. 하지만 리스트 a를 다음과 같이 선언했다고 하자.

함수의 스코프 밖에서 변수에 값을 대입하려면 그 변수를 global 예약어로 전역변수 선언해야 한다.

```
In [52]: a = None

In [53]: def bind_a_variable():
    ...:     global a
    ...:     a = []
    ...: bind_a_variable()
    ...:
    ...:

In [54]: print(a)
[]
```

CAUTION global 예약어는 자주 사용하지 않도록 한다. 일반적으로 전역 변수는 시스템 전체의 상태를 저장하기 위한 용도로 사용한다. 만약 전역 변수를 많이 사용하면 클래스를 사용한 객체지향 프로그래밍이 적절한 상황이라는 것이다.

3.2.2 여러값 반환하기

자바와 C++로 프로그래밍을 하다가 처음으로 파이썬을 접했을 때 내가 가장 좋아했던 기능은 하나의 함수에서 여러 개의 값을 반환할 수 있는 기능이었다.

```
def f():
    a =5
    b = 6
    c = 7
    return a,b,c

a,b,c = f()
```

데이터 분석과 과학 계산 어플에서는 많은 함수가 여러 개의 값을 반환하는 일이 잦다. 앞서 살펴본 튜플을 생각해보면 이 함수는 하나의 객체, 말하자면 튜플을 반환한다고 생각할 수 있다. 즉, 위 예제에서는 튜플을 반환하며 아래 코드처럼 변수에 대입할 수 있다.

```
return_value = f()
```

여기서 return_value는 짐작한 대로 반환된 세 개의 값을 가지고 있는 튜플이 된다. 다른 매력적인 대안으로는 여러 값을 반환하는 대신 사전 형태로 반환하는 것이다.

```
def f():
    a =5
    b = 6
    c = 7
    return {'a':a,'b':b,'c':c}
```

경우에 따라서는 따라서는 사전을 반환하는 이 대안이 보다 유용할 수 있다.

3.2.3 함수도 객체다

파이썬에서는 함수도 객체이므로 다른 언어에서는 힘든 객체 생성 표현을 쉽게 할 수 있다. 데이터를 정제하기 위해 다음과 같은 문자열 리스트를 변형해야 한다고 가정하자.

```
In [55]: states = ['alabama', 'georgia', 'georgia!', 'FLOrida',
                   'west virginia?']
```

사용자가 입력하는 설문 조사 데이터를 다뤄본적이 있다면 이와 같이 엉망인 데이터를 많이 보았을 것이다. 분석을 위해 이런 문자열 리스트를 정형화할 필요가 있다. 공백 문자를 제거 하고 필요없는 문장 부호를 제거하거나 대소문자를 맞추는 등의 작업이 필요하다. 이는 내장 문자열 메서드나 정규 표현식을 위한 re 표준 라이브러리를 이용해서 쉽게 해결할 수 있다.

```
import re

def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub('[!#?]','',value)
        value = value.title()
        result.append(value)
    return result
```

실행 결과는 다음과 같다.

```
In [58]: clean_strings(states)
Out[58]: ['Alabama', 'Georgia', 'Georgia', 'Florida', 'West Virginia']
```

다른 유용한 접근법으로는 적용할 함수를 리스트에 담아두고 각각의 문자열에 적용하는 것이다.

```
def remove_punctuation(value):
    return re.sub('[!#?]','',value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result
```

```
In [60]: clean_strings(states, clean_ops)
Out[60]: ['Alabama', 'Georgia', 'Georgia', 'Florida', 'West Virginia']
```

이와 같이 좀 더 **실용적인** 패턴은 문자열 변형을 상위 레벨에서 쉽게 처리할 수 있다. 이렇게 해서 clean_strings 함수는 재사용이 용이해졌다.

순차적자료형에 대해 함수를 적용하는 내장 함수인 map 함수를 이용해서 함수를 인자로 사용할 수도 있다.

```
In [61]: for x in map(remove_punctuation, states ):
    ...:     print(x)
    ...:
alabama
georgia
georgia
FLOrida
west virginia
```

3.2.4 익명 함수

파이썬은 **익명**함수 혹은 **람다**함수라고 하는 값을 반환하는 단순한 한 문장으로 이뤄진 함수를 지원한다. lambda 예약어로 정의하며, 이는 '익명 함수를 선언한다'라는 의미이다.

```
def short_function(x):
    return x *2

equiv_anon = lambda x:x*2
```

이 책에서는 이를 람다 함수라고 하겠다. 람다 함수는 데이터 분석에서 특히 편리한데, 이는 앞으로 알게 되겠지만 데이터를 변형하는 함수에서 인자로 함수를 받아야 하는 경우가 매우 많기 때문이다. 즉, 람다 함수를 사용하면 실제 함수를 선언하거나 람다 함수를 지역 변수에 대입하는 것보다 코드를 적게 쓰고 더 간결해지기 때문이다. 약간 억지스럽지만 다음 예제를 보자.

```
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4,0,1,5,6]
apply_to_list(ints, lambda x:x*2)
```

물론 [x * 2 for x in ints]라고 해도 되지만 이렇게 하면 apply_to_list 함수에 사용자 연산을 간결하게 전달할 수 있다.

다른 예제로, 다음 문자열 리스트를 각 문자열에서 다양한 문자가 포함된 순서로 정렬한다고 가정하자.

```
In [62]: def apply_to_list(some_list, f):^M
    ...:     return [f(x) for x in some_list]^M
    ...: ^M
    ...: ints = [4,0,1,5,6]^M
    ...: apply_to_list(ints, lambda x:x*2)
Out[62]: [8, 0, 2, 10, 12]
```

다른 예제로, 다음 문자열 리스트를 각 문자열에서 다양한 문자가 포함된 순서로 정렬한다고 가정하자.

```
In [63]: strings = ['foo','card','bar','aaaa','abab']
```

리스트의 sort 메서드에 람다 함수를 넘겨 다음과 같이 정렬할 수 있다.

```
In [64]: strings.sort(key = lambda x: len(set(list(x))))

In [65]: strings
Out[65]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

Note_ 람다 함수가 익명 함수라고 불리는 이유 중 하나는 이 함수 객체에는 명시적인 _name_ 속성이 없기 때문이다.

3.2.5 커링 : 일부 인자만 취하기

커링은 수학자인 하스켈 커리의 이름에서 따온 컴퓨터 과학 용어로, 함수에서 **일부 인자만 취하는**새로운 함수를 만드는 기법이다. 예를 들어 2개의 숫자를 더하는 함수가 있다고 가정하자.

```
def add_numbers(x,y):
    return x+y
```

이 함수를 이용해서 하나의 변수만 인자로 받아 5를 더해주는 새로운 함수 add_five 를 생성하자.

```
add_five = lambda y: add_numbers(5,y)
```

add_numbers의 두 번째 인자를 커링했다. 여기서는 기존 함수를 호출하는 새로운 함수를 하나 정의했을 뿐이므로 그리 복잡하지 않다. 내장 functools 모듈의 partial 함수를 이용하면 이 과정을 단순화 할 수 있다.

```
from functools import partial
add_five = partial(add_numbers, 5)
```

3.2.6 제너레이터

파이썬은 리스트 내의 객체나 파일의 각 로우 같은 순차적인 자료를 순회하는 일관적인 방법을 제공한다. **이터레이터 프로토콜**을 이용해 순회 가능한 객체를 만들 수 있다. 예를 들어 사전을 순회하면 사전의 키가 반환된다.

```
In [66]: some_dict = {'a':1, 'b':2, 'c':3}

In [69]: for key in some_dict:
    ...:     print(key )
    ...:
a
b
c
```

for key in some_dict 라고 작성하면 파이썬 인터프리터는 some_dict에서 이터레이터를 생성한다.

```
In [70]: dict_iterator = iter(some_dict)

In [71]: dict_iterator
Out[71]: <dict_keyiterator at 0x1b1ce99bd68>
```

이터레이터는 for문 같은 컨텍스트에서 사용될 경우 객체를 반환한다. 리스트나 리스트와 유사한 객체를 취하는 대부분의 메서드는 순회 가능한 객체도 허용한다. 여기에는 min, max, sum 같은 내장 메서드와 list.tuple 같은 자료구조를 생성하는 메서드도 포함된다.

```
In [72]: list(dict_iterator)
Out[72]: ['a', 'b', 'c']
```

제네레이터는 순회 가능한 객체를 생성하는 간단한 방법이다. 일반 함수는 실행되면 단일 값을 반환하는 반면 제너레이터는 순차적인 값을 매 요청 시마다 하나씩 반환한다. 제너레이터를 생성하려면 함수에서 return을 하는 대신 yield 예약어를 사용한다.

```
def squares(n=10):
    print('Generating squares from 1 to {0}'.format(n**2))
    for i in range(1, n+1):
        yield i **2
```

제너레이터를 호출하더라도 코드가 즉각적으로 실행되지 않는다.

```
In [87]: def squares(n=10):^M
    ...:     print('Generating squares from 1 to {0}'.format(n**2))^M
    ...:     for i in range(1, n+1):^M
    ...:         yield i **2

In [88]: gen1 = squares()

In [89]: gen1
Out[89]: <generator object squares at 0x000001B1CE6DC948>

In [90]: for x in gen1:
    ...:     print(x, end=' ')
    ...:

Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

제너레이터로부터 값을 요청하면 그때서야 제너레이터의 코드가 실행된다.

제너레이터 표현식

제너레이터를 생성하는 더 간단한 방법은 **제네레이터 표현식**을 사용하는 것이다. 다음은 리스트, 사전, 집합 표현식과 유사한 방식으로 제네레이터를 생성한다. 리스트 표현식에서 대괄호를 사용하듯이 괄호를 사용해서 제너레이터를 생성할 수 있다.

```
In [91]: gen = (x**2 for x in range(100))

In [92]: gen
Out[92]: <generator object <genexpr> at 0x000001B1CDC9C0C8>

In [93]: def _make_gen():
    ...:     for x in range(100):
    ...:         yield x ** 2
```

제너레이터 표현식은 리스트 표현식을 인자로 받는 어떤 파이썬 함수에서도 사용할 수 있다.

```
In [97]: sum(x ** 2 for x in range(100))
Out[97]: 328350

In [98]: dict((i, i**2) for i in range(5))
Out[98]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

itertools 모듈

표준 라이브러리인 itertools 모듈은 일반 데이터 알고리즘을 위한 많은 제너레이터를 포함 하고 있다. 예를 들어 groupby는 순차 자료구조와 함수를 받아 인자로 받은 함수에서 반환하는 값에 따라 그룹을 지어준다. 다음예제를 보자.

```
In [99]: import itertools

In [100]: first_letter = lambda x: x[0]

In [103]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']

In [104]: for letter, names in itertools.groupby(names, first_letter):
     ...:     print(letter, list(names))
     ...:
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

아래 유용하다고 생각하는 itertools 함수를 정리해두었다. 이 유용한 내장 유틸리티 모듈을 더 자세히 알아보고 싶다면 공식 파이썬 문서를 참고하라.

함수	설명
combination(iterable,k)	iterable에서 순서를 고려하지 않고 길이가 k인 모든 가능한 조합을 생성 한다.
permutations(iterable,k)	iterable에서 순서를 고려하여 길이가 k인 모든 가능한 조합을 생성한다.
groupby(iterable[,keyfunc])	iterable에서 각각의 고유한 키에 따라 그룹을 생성한다.
product(*iterables, repeat = 1)	iterable에서 카테시안 곱을 구한다. 중첩된 for 문 사용과 유사하다.

3.2.7 에러와 예외 처리

견고한 프로그램을 작성하려면 파이썬의 오류와 **예외**를 잘 처리해야 한다. 데이터 분석 어플에서는 많은 함수가 특정한 종류의 입력만 처리하도록 되어 있다. 예를 들어 파이썬의 float 함수는 문자열을 부동소수점수로 변환할 수 있지만 적절하지 않은 입력에 대해서는 ValueError와 함께 실패한다.

```
In [107]: float('1.2345')
Out[107]: 1.2345

In [108]: float('something')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-108-2649e4ade0e6> in <module>
----> 1 float('something')

ValueError: could not convert string to float: 'something'
```

적절하지 않은 입력에 대해서 입력을 그대로 반환하는 개선된 float 함수를 작성한다고 가정하자. 이렇게 하려면 try/except 블록을 사용해서 float 함수를 호출하면 된다.

```
def attempt_float(x):
    try:
        return float(x)
    except:
        return x
```

except 블록에 있는 코드는 float(x)가 예외를 발생했을 때 실행된다.

```
In [109]: def attempt_float(x):^M
     ...:     try:^M
     ...:         return float(x)^M
     ...:     except:^M
     ...:         return x
     ...:

In [110]: attempt_float('1.2345')
Out[110]: 1.2345

In [111]: attempt_float('something')
Out[111]: 'something'
```

float 함수가 ValueError가 아닌 예외를 발생시키는 경우도 있다.

```
In [112]: float((1,2))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-112-a101b3a3d6cd> in <module>
----> 1 float((1,2))

TypeError: float() argument must be a string or a number, not 'tuple'
```

입력이 문자열이나 숫자가 아니라는 뜻의 TypeError는 정당한 오류이므로 그대로 두고 ValueError 만 무시하고 싶다면 except 뒤에 처리할 예외의 종류를 적어준다.

```
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        return x
```

이 함수는 다음처럼 동작한다.

```
In [115]: attempt_float((1,2))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-115-102527222085> in <module>
----> 1 attempt_float((1,2))

<ipython-input-113-12f95d1462b6> in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x

TypeError: float() argument must be a string or a number, not 'tuple'
```

튜플을 사용해서 여러 개의 예외를 한 번에 처리할 수도 있다.(괄호로 묶어준다)

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x
```

```
In [117]: attempt_float((1,2))
Out[117]: (1, 2)
```

예외를 무시하지 않고, try 블록의 코드가 성공적으로 수행되었는지 여부와 관계없이 실행시키고 싶은 코드는 finally 블록을 이용해서 적어준다.

```
f = open(path, 'w')
try:
    write_to_file(f)
finally:
    f.close()
```

여기서 파일 핸들 f는 항상 닫히게 된다. 이와 유사하게 try 블록이 성공적으로 수행되었을 때만 else 블록을 사용해서 수행할 코드를 적어준다.

```
f = open(path, 'w')
try:
    write_to_file(f)
except:
    print('failed')
else:
    print('Succeeded')
finally:
    f.close()
```

IPython에서 예외 처리

%run을 이용해서 코드를 실행시키던 중에 예외가 발생하면 IPython은 기본적으로 전체 트레이스백을 출력하고 해당 위치 주변의 코드를 함께 보여준다.

추가 내용을 보여주지 않는 표준 파이썬 인터프리터에 비해 IPython은 추가 내용을 함께 포함해서 보여주므로 매우 편리하다. %xmode 매직 명령어를 이용하면 표준 파이썬 인터프리터와 같은 수준인 Plain에서부터 함수 인잣값 등을 포함해서 보여주는 Verbose 단계까지 직접 제어할 수 있다. 나중에 다른 장에서 살펴보겠지만 에러가 발생했을 때 대화형 디버깅을 통해 스택의 내용을 직접 살펴볼 수 있다.

3.3 파일과 운영체제

이 책에서는 대부분 pandas.read_csv 같은 고수준의 도구를 사용해서 디스크로부터 파일을 읽어와 파이썬 자료구조에 저장한다. 하지만 파이썬에서 파일을 어떻게 다루는지 이해하는 것도 중요하다. 다행히도 파이썬에서 파일을 다루는 방법은 전혀 어렵지 않다. 이는 파이썬이 텍스트와 파일 처리에 인기 있는 중 하나이다.

파일을 읽고 쓰기 위해 열 때는 내장함수인 open을 이용해서 파일의 상대 경로나 절대 경로를 넘겨주어야 한다.

```
In [123]: path = 'C:\wd\st1.txt'

In [124]: f = open(path)

In [125]: f
Out[125]: <_io.TextIOWrapper name='C:\\wd\\st1.txt' mode='r' encoding='cp949'>

In [126]: for line in f:
     ...:     pass

In [127]: lines = [x.rstrip() for x in open(path)]

In [128]: lines
Out[128]:
```

```
['"번호" "이름" "키" "몸무게"',
 '101 "hong" 175 65',
 '201 "lee" 185 85',
 '301 "kim" 173 60',
 '401 "park" 180 70']
```

기본적으로 파일은 읽기 전용 모드인 'r'로 열린다. 파일 핸들 f를 리스트로 생각할 수 있으며 파일의 매 줄을 순회할 수 있다.

파일에서 읽은 줄은 줄끝문자가 그대로 남아 있으므로 파일에서 읽은 줄에서 이를 제거하는 다음과 같은 코드를 종종 보게 될 것이다.

파일 객체를 생성하기 위해 open을 사용했다면 작업이 끝났을 때 명시적으로 닫아주어야 한다.

파일을 닫으면 해당 자원을 OS로 되돌려준다.

```
In [129]: f.close()
```

with문을 사용하면 파일 작업이 끝났을 때 필요한 작업을 쉽게 처리할 수 있다.

```
In [131]: with open(path) as f:
     ...:     lines = [x.rstrip() for x in f]
```

만일 파일을 f = open(path, 'w')로 연다면 txt파일이 **새롭게 생성** 되고 파일의 내용을 새로운 내용으로 덮어쓴다. 'x'모드는 쓰기 목적으로 파일을 새로 만들지만 이미 해당 파일이 존재하면 실패하게 된다.

파일을 읽을 때는 read, seek, tell 메서드를 주로 사용하는데, read 메서드는 해당 파일에서 특정 개수만큼의 문자를 반환한다. 여기서 '문자'는 인코딩 (UTF-8)으로 결정되거나 이진 모드인 경우에는 단순히 바이트로 결정된다.

```
In [142]: f = open(path)

In [144]: f.read(18)
Out[144]: '"번호" "이름" "키" "몸무게'

In [145]: f2 = open(path, 'rb')

In [146]: f2.read(10)
Out[146]: b'"\xb9\xf8\xc8\xa3" "\xc0\xcc'
```

read 메서드는 읽은 바이트 만큼 파일 핸들의 위치를 옮긴다. tell 매서드는 현제 위치를 알려준다.

```
In [147]: f.tell()
Out[147]: 26

In [148]: f2.tell()
Out[148]: 10
```

파일에서 10개의 문자를 읽었어도 위치가 11인 이유는 기본 인코딩에서 10개의 문자를 인코딩하기 위해 그 만큼의 바이트가 필요했기 때문이다. 기본 인코딩은 sys 모듈에서 확인할 수 있다.

```
In [149]: import sys

In [150]: sys.getdefaultencoding()
Out[150]: 'utf-8'
```

seek 매서드는 파일 핸들의 위치를 해당 파일에서 지정한 바이트 위치로 옮긴다.

```
In [151]: f.seek(3)
Out[151]: 3

In [152]: f.read(1)
Out[152]: '호'
```

마지막으로 파일을 닫는 것을 잊지마라!!

```
In [153]: f.close()

In [154]: f2.close()
```

파이썬 파일 모드

모드	설명
r	읽기 전용 모드
w	쓰기 전용 모드, 새로운 파일을 생성한다(같은 이름의 파일이 존재하면 덮어쓴다)
x	쓰기 전용 모드, 새로운 파일을 생성한다, 이미 존재하는 경우에는 쓰기가 안된다.
a	기존 파일에 추가한다.(파일이 존재하지 않을 경우 새로 생성한다.)
r+	읽기/쓰기 모드
b	이진 파일 모드. 읽기/쓰기 모드에 추가해서 'rb' 또는 'wb' 처럼 사용한다.
t	텍스트 모드(자동으로 바이트를 유니코드로 디코딩한다.) 모드를 지정하지 않으면 t가 기본 모드로 지정된다. 다른 모드에 추가해서 'rt' 또는 'xt' 처럼 사용한다.

파일에 텍스트를 기록하려면 write나 writelines 메서드를 사용하면 된다. 예를 들어 빈 줄이 포함되지 않도록 prof_mod.py를 작성하려면 다음과 같이 하면 된다.

```
In [156]: with open('st1.txt','w') as handle:
     ...:     handle.writelines(x for x in open(path) if len(x)>1)
     ...:

In [158]: with open('st1.txt') as f:
     ...:     lines = f.readlines()
     ...:
     ...:

In [159]: lines
Out[159]: []
```

중요 file 메서드와 속성

메서드	설명
read([size])	파일에서 데이터를 읽어서 문자열로 반환한다. size인자를 사용해서 몇 바이트를 읽을 것인지 지정할 수 있다.
readlines([size])	파일의 매 줄을 모두 읽어 리스트로 반환한다. size 인자를 사용해서 얼마나 읽을 것인지 지정할 수 있다.
write(str)	전달받은 문자열을 파일에 기록한다.
writelines(strings)	전달 받은 일련의 문자열을 파일에 기록한다.
close()	파일 핸들을 닫는다.
flush()	내부 I/O 버퍼를 디스크로 비운다.
seek(pos)	파일 내에서 지정한 위치(정수)로 이동한다.
tell()	현재 파일의 위치를 정수 형태로 반환한다.
closed	파일 핸들이 닫힌 경우 True를 반환한다.

3.3.1 바이트와 유니코드

읽기든 쓰기든 파이썬 파일은 파이썬 문자열(즉, 유니코드)을 다루기 위한 텍스트 모드를 기본으로 한다. 이는 파일 모드에 b를 추가해서 열 수 있는 **이진 모드**와는 다르다. UTF-8 인코딩을 사용하는 비-아스키 문자가 포함된 파일을 살펴보자.

```
¬In [160]: f = open(path)

In [161]: with open(path) as f:
     ...:     data = f.read(10)
     ...:

In [163]: data
Out[163]: '"번호" "이름" '
```

UTF-8은 가변길이 유니코드 인코딩으므로 파일에서 일부 문자를 읽어오도록 한다면 파이썬은 파일에서 필요한 만큼의 바이트 (최소 10바이트에서 40바이트까지 될 수 있다)를 읽은 다음 10문자로 디코딩 한다. 만일 파일을 'rb' 모드로 열었다면 read는 딱 10바이트만 읽어올 것이다.

```
In [164]: with open(path, 'rb') as f:
     ...:     char = f.read(10)

In [165]: char
Out[165]: b'"\xb9\xf8\xc8\xa3" "\xc0\xcc'
```

텍스트 인코딩에 따라 읽어온 바이트를 str 객체로 직접 디코딩할 수도 있다. 다만 온전한 유니코드 문자로 인코딩되어 있을 경우에만 가능하다.

```
In [166]: char.decode('utf8')
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-166-5a4c7a92098b> in <module>
----> 1 char.decode('utf8')

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb9 in position 1: invalid
start byte
```