04. 문자열

4.1 시퀀스 자료형

- 시퀀스 자료형이란

이 장에서 다루는 문자열은 시퀀스 자료형이다. 시퀀스 자료형은 여러 객체를 저장하는 자료형이며, 각 객체는 순서를 가진다. 각 요소는 인덱스를 사용하여 참조할수 있다. 시퀀스 자료형에 속하는 객체에는 앞에서 살펴본 문자열과 리스트, 튜플이 있다. 이들 자료형은 파이썬에서 가장 많이 사용된다. 따라서 이들의 일반적인 특성을 이해해야 한다. 이들의 특성은 내장 자료형에만 적용되는 것이 아니라, 여러분이 나중에 설계할 시퀀스 클래스 객체들이 가져야 할 특성이기도 하다. 우선 게 가지 자료형에 대한 예를 간단히 들어 보자.

```
>>> s = '파이썬만세'              # 문자열
>>> L = [100, 200, 300]          # 리스트
>>> t = ('튜플', '객체', 1, 2)  # 튜플
```

문자열은 작은 따옴표나 '큰따옴표' 로 묶인 문자들의 모임이다. 리스트는 대괄호 []안에 둘러싸인 객체들의 모임이고, 튜플은 소괄호 () 안에 둘러싸인 객체들의 모임이다. 이들은 조금씩 상이한 특성이 있지만, 모두 시퀀스 자료형이라는 공통적인 특성이 있다. 이 절에서는 이들의 공통적인 특성에 대해서만 설명한다. 시퀀스 자료형이 가지는 공통적인 연산에는 다음과 같은 것들이 있다.

구분	연산	설명
인덱싱	[k]형식	k번 위치의 값 하나를 취한다.
슬라이싱	[s:t:p]형식	s부터 t사이 구간의 값을 p 간격으로 취한다.
연결하기	+ 연산자	두 시퀀스형 데이터를 붙여서 새로운 데이터를 만든다.
반복하기	● 연산자	시퀀스형 데이터를 여러번 반복해서 새로운 데이터를 만든다.
멤버검사	in 연산자	어떤 값이 시퀀스 자료형에 속하는지를 검사한다.
길이 정보	len() 내장 함수	시퀀스형 데이터의 크기를 나타낸다.

- 인덱싱

인덱싱은 순서가 있는 데이터에서 오프셋으로 하나의 객체를 참조하는 것이다. 오프셋능 정수이며, 0부터 시작한다. 다음과 같은 형식으로 사용한다.

[오프셋]

다음은 인덱싱을 사용한 예이다.

```
>>> s = 'abcdef'          # 문자열
>>> l = [100, 200, 300] # 리스트
>>> s[0]                  # 참조
'a'
>>> s[1]
'b'
>>> s[-1]                 # 맨오른쪽 값
```

```
'f'
>>> l[1]
200
>>> l[1] = 900          # 치환
```

- 슬라이싱

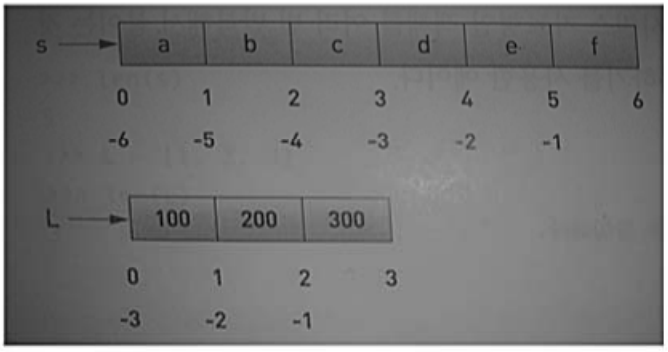
슬라이싱은 시퀀스 자료형의 일정 영역에서 새로운 객체를 반환하며, 결과의 자료형은 원래의 자료형과 같다. 다음과 같은 형식으로 사용한다.

[시작오프셋:끝오프셋]

오프셋은 생략할수 있다. 시작 오프셋을 생략하면 0, 끝 오프셋을 생략하면 마지막 값으로 처리한다. 다음은 슬라이싱을 사용한 예이다.

```
>>> s = 'abcdef'
>>> L= [100, 200, 300]
>>> s[1:3]      # ①
'bc'
>>> s[1:]       # 1부터 끝까지이다.
'bcdef'
>>> s[:]        # 처음(0)부터 끝까지이다.
'abcdef'
>>> s[-100:100] # 범위를 넘어서면 범위 내의 값으로 자동으로 처리한다.
'abcdef'
>>> L[:-1]      # 맨 오른쪽 값을 제외하고 모두이다.
[100, 200]
```

① 첫 번째 문자부터 세 번째 문자까지가 아니라, 1번위치와 3번 위치 사이를 나나낸다. 참고로 문자열 s와 리스트 L의 구조를 그리면 그림 4-1과 같다.



글라이싱은 실제로 세가지 값을 가진다. 세 번째 step은 데이터를 취하는 간격이다 이것을 확장슬라이싱이라고 한다.

```
>>> s = 'abcd'
>>> s[::2]      # 2칸 단위로 데이터를 취한다.
'ac'
>>> s[::-1]     # 거꾸로 데이터를 취한다.
'dcba'
```

- 연결하기

연결하기는 +연산자로 같은 시퀀스 자료형인 두객체를 연결하는 것이다. 일반적으로 두 객체의 자료형이 동일해야 하며, 새로 만들어지는 객체도 같은 자료형이다. 다음은 연결하기를 사용한 예이다.

```
>>> s = 'abc' + 'def'
>>> s
'abcdef'
>>> L = [1, 2, 3] + [4, 5, 6]
>>> L
[1, 2, 3, 4, 5, 6]
```

- 반복하기

반복하기는 * 연산자를 사용하여 시퀀스 자료형인 객체를 여러번 반복해서 붙이는 것이다. 새로운 객체를 반환한다. 다음은 반복하기를 사용한 예이다.

```
>>> s = 'ABC'
>>> s * 4       # s + s + s + s와 동일하다.
'ABCABCABCABC'
>>> L = [1, 2, 3]
>>> L * 2
[1, 2, 3, 1, 2, 3]
```

- 멤버검사

어떤 객체가 시퀀스 자료형인 객체에 포함된 것인지 검사하는 것을 멤버 검사라고 한다. 반환값 1은 참을, 0은 거짓을 의미한다. 다음은 멤버 검사를 사용한 예이다.

```
>>> '파' in '파이썬'
True
>>> t = (1, 2, 3, 4, 5)
>>> 2 in t
True
>>> 10 not in t
True
```

문자열인 경우는 in 연산자로 부분 문자열을 확인할수 있다.

```
>>> '파이' in '파이썬'
True
>>> '이선' in '파이썬'
False
```

- 길이 정보

시퀀스형 객체 안에 있는 전체 요소의 개수를 얻기 위해서 len() 내장함수를 사용할수 있다.

```
>>> s = '파이썬만세'
>>> len(s)
5
>>> L = [1, 2, 3]
>>> len(L)
3
```

4.2 문자열 정의하기

- 한줄 문자열

문자열은 앞서 설명한 시퀀스 자료형이다. 따라서 이 자료형의 특성(인덱싱, 슬라이싱, 연결하기, 반복하기 등)을 모두 가진다. 우선 문자열을 정의하는 예부터 보자.

```
>>> s = ''                                          # 빈 문자열이다.
>>> str1 = 'Python is great'                        # 작은따옴표 '를 사용하여 정의한다.
>>> str2 = "Yes, it is."                            # 큰따옴표 "를 사용하여 정의한다.
>>> str3 = "It' s not like any other languages"     # ①
>>> type(s)                                         # 자료형을 확인한다.
<class 'str'>
>>> isinstance(s, str)
True
```

① 세 번째 예에서 It' s의 '는 "로 둘러싸여 있는 전체 문자열 안에 포함되어 있으므로 문자열을 구분하는 문자가 아닌 출력 문자로 처리된다. 인용부호의 의미를 없애기 위하여 \를 사용할수 있다.

```
>>> str4 = 'Don₩'t walk, "Run"'
>>> print(str4)
Don't walk, "Run "
```

\n은 줄바꾸기 문자 그리고 \t는 탭문자를 나타낸다. 또한, 줄의 맨 마지막에 \를 사용하면 다음 줄이 하나로 이어진다는 것을 의미한다.

```
>>> long_str = "This is a rather long string  ₩
... containing back slash and new line. ₩n Good!"
>>> print(long_str)
This is a rather long string  containing back slash and new line.
 Good!
```

- 여러줄 문자열

파이썬에서는 작은따옴표 '나 큰따옴표 "를 세 번 연속해서 사용하여 여러줄 문자열을 손쉽게 표현한다. 이렇게 하면 표현식 내의 모든 텍스트를 적힌 그대로 표현할수 있다.

```
>>> multiline = """ 만일 우리에게 겨울이 없다면, 봄은 그토록 즐겁지 않을 것이다.
... 우리들이 이따금 역경을 맛보지 않는다면, 성공은 그토록 환영받지 못할 것이다.
... """
>>> print(multiline)
 만일 우리에게 겨울이 없다면, 봄은 그토록 즐겁지 않을 것이다.
우리들이 이따금 역경을 맛보지 않는다면, 성공은 그토록 환영받지 못할 것이다.
```

- 특수 문자

파이썬에서도 문자열을 표현할 때 백슬래스 \를 사용하여 키보드로 표현하기 힘든 문자들을 표현한다.

백슬래스 \를 사용하는 특수 문자

문자	설명	문자	설명
\ enter	다음줄과 연속임을 표현	\n 또는 \012	줄바꾸기
\\	\문자 자체	\t	tab 키
\‘	‘문자	\0xx	8진수 코드 xx
\“	“문자	\xXX	16진수 코드 XX
\b	백스페이스	\e	Esc 키

다음은 특수 문자를 사용하는 예이다.

```
>>> a = ₩                # 줄연속
... 2
>>> a
2
>>> print('abc\tdef\tghi')
abc\tdef\tghi
>>> print('a₩nb₩nc')
a
b
c
```

- 문자열 연산

문자열은 앞서 설명한 것처럼 시퀀스형의 모든 연산(인덱싱, 슬라이싱, 연결하기, 반복하기, 멤버 검사 등)을 제공한다. 다음은 이들 연산을 사용한 예이다.

```
>>> s1 = '첫 문자열'
>>> s2 = '두번째 문자열'
>>> s3 = s1 +  ''  + s2          # 문자열 붙이기
>>> s3
'첫 문자열두번째 문자열'
>>> s1 * 3
'첫 문자열첫 문자열첫 문자열'
>>> s1 * 3                      # 문자열 3회 반복하기
'첫 문자열첫 문자열첫 문자열'
>>> s1[1:-1]                    # 슬라이싱
' 문자'
>>> len(s1)                     # 문자열 길이 정보
5
>>> '문자' in s111
```

```
True
```

문자열은 그 자체 값을 변경할수 없는, 변경 불가능 자료형이다.

```
>>> print(str1)
Python is great
>>> print(str1[0])
P
>>> str1[0] = 'f'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

문자열이 시퀀스 자료형으로 for 문에 적용하면 각 문자에 대해서 반복한다.

```
>>> for c in '파이선만세':
...     print(c, end = '')
...
파이선만세
```

문자열을 어떻게 변경할수 있을까? 간단하다. 슬라이싱을 사용하면 원하는 문자열을 얻을 수 있다. 이것은 실제로 문자열의 변경이라기보다는 새로 문자열을 정의하는 것이다.

```
>>> s = 'spam and egg'
>>> s = s[:5] + 'cheese' + s[5:]
>>> s
'spam cheeseand egg'
```

4.3 문자열의 서식 지정하기

print() 함수를 사용하면 출력 문자열의 서식을 자요롭게 지정할수 있다. 여기서 서식이 란 문서의 틀이나 양식을 말한다. 문서는 양식과 내용으로 구성된다. 양식은 고정된 틀을 의미하고 빈칸을 갖는다. 어떤 문서 양식을 만들어 놓고 빈칸에 필요한 내용을 채워서 문서를 자유롭게 만들어 내는데 이때 서식지정을 사용한다.

- 이전 방식의 서식 지정

파이썬1과 2에서 주로 사용하던 방식으로 문자열 안에 서식 문자를 지정하고 튜플로 데이터를 공급하는 방식이다. 파이썬3에서는 이 방식을 권장하기 않는다.]

```
>>> int_val = 23
>>> float_val = 2.34567
>>> print("%3d %s %0.2f" % (int_val, 'any value',float_val)) )
 23 any value 2.35
```

여기서 %d와 %f 등을 서식 문자라고 부른다. 각각 정수형과 문자열, 실수형 데이터를 지정한다. %3d와 %0.2f와 같이 포함되어 있는 숫자는 출력할 자릿수와 실수의 정밀도 등을 나타낸다. %3d에서 3은 세자리로 정수를 출력하라는 의미이고, %0.2f에서 0은 자릿수를 특별히 제한하지 않으며, 2는 소수점 이하 두자리만 출력한다는 의미이다.

- format() 함수를 사용한 서식 지정

format() 함수는 하나의 값을 주어진 형식의 문자열로 변환하는데 사용한다.

```
>>> int_val = 23
>>> float_val = 2.34567
>>> print(format(int_val, '3d'), format(float_val, '.2f'))
 23 2.35
```

3d에서 3은 세자리로 정수를 출력하는 의미이고, 0.2f에서 2는 소수점 이하 두자리로 실수를 출력하라는 의미이다. 천 단위마다 쉼표, ,를 추가하려면 다음과 같이 변환 기호 앞에 쉽표,를 사용한다.

```
>>> format(123456789, ',d')
'123,456,789'
>>> format(1234567.89, ',.2f')
'1,234,567.89'
```

- format() 메서드를 사용한 서식 지정

세 번째로 문자열의 format() 메서드는 여러 값을 출력할 때 가장 많이 사용하는 방식이다. 문자열 내 중괄호 {}은 데이터로 채워질 자리를 의미한다. 공급되는 순서대로 데이터를 출력한다.

```
>>> '{} {}'.format(23, 2.12345) )
'23 2.12345'
```

{0}은 format() 메서드의 첫 번째 인수, {1}은 두 번째 인수로 치환되는 것을 나타낸다. 참조 순서가 바뀔수도 있고, 여러번 나타날수도 있다.

```
>>> L = {1, 5, 3, 7, 4, 5}
>>> '최댓값:{0}, 최솟값:{1}'.format(max(L), min(L))
'최댓값:7, 최솟값:1'
>>> '최솟값:{1},최댓값{0},최대값{0}'.format(max(L), min(L))
'최솟값:1,최댓값7,최대값7'
```

변환 기호를 함께 사용해서 다음과 같이 사용할수도 있다. 실수 형식으로 출력하려면 치환 영역내에 변환 기호를 사용할수 있다. {0:5}나 {0:5d}는 format() 메서드의 첫 번째 인수를 5자리로 출력하라는 의미이고, {1:0.5}는 5자리의 유효 자리로, {1:0,5f}는 두 번째 인수를 소수점 이하 5자리로 실수로 출력하라는 의미이다. {1:10.5f}는 출력 자리 10자리를 확보하고 소수점 이하 5자리를 출력한다.

```
>>> 'sqrt({:3}) = {:0.5}'.format(2, 2 ** 0.5)   #  자릿수만 지정
'sqrt(  2) = 1.4142'
>>> 'sqrt({0:3}) = {1:0.5}'.format(2, 2 ** 0.5) # 앞의 결과와 동일
'sqrt(  2) = 1.4142'
>>> 'sqrt({0:3d}) = {1:0.5f}'.format(2, 2 ** 0.5)       # d : decimal, f : float
'sqrt(  2) = 1.41421'
>>> 'sqrt({:3d}) = {:0.5f}'.format(2, 2 ** 0.5)
'sqrt(  2) = 1.41421'
>>> '{:,d}'.format(123456789)
'123,456,789'
>>> '{:,.2f}'.format(123456789.0123)
'123,456,789.01'
```

리스트가 인수로 전달될때는 인덱싱으로 참조할수 있다.

```
>>> L = [0, 1, 1, 2, 3, 5, 8, 13, 21]
>>> 'next value of {0[4]} is {0[5]}',format(L)
('next value of {0[4]} is {0[5]}', '[0, 1, 1, 2, 3, 5, 8, 13, 21]')
```

키워드 인수로 전달되면 이름으로 접근할수 있다.

```
>>> '나이:{age} 키:{height}'.format(age = 49, height = 173)
'나이:49 키:173'
```

사전이 인수로 전달될때는 format_map() 메서드를 사용하여 키로 참조한다.

```
>>> info = {'size':32, 'height':173, 'age':49}
>>> "나이:{age}, 키:{height}".format_map(info)
```

'나이:49, 키:173'

모듈인 경우에는 이름으로 접근할수 있다.

```
>>> import sys          # 모듈의 값을 우선 확인한다.
>>> sys.float_info.max
1.7976931348623157e+308
>>> '실수 최댓값 : {0.float_info.max}'.format(sys))
'실수 최댓값 : 1.7976931348623157e+308'
```

{0.float_info.max}는 sys.float_info.max을 의미한다. 기타 다른 객체인 경우에도 이름으로 접근할수 있다.

```
>>> print('{0.__doc__}'.format(list))                    # list._doc_
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

다음은 0도부터 360도까지 10도 간격으로 사인 값을 표로 만드는 예이다.

```
>>> for deg in range(0, 361, 10):
...     rad = radians(deg)
...     print('sin({0:3d}) = {1:10.5f}'.format(deg, sin(rad)))
...
sin(  0) =    0.00000
sin( 10) =    0.17365
sin( 20) =    0.34202
sin( 30) =    0.50000
sin( 40) =    0.64279
sin( 50) =    0.76604
sin( 60) =    0.86603
...<중략>
```

- 수칭 변환 기호

수치 자료형에 맞는 변환 기호는별도로 준비되어 있다. 예를들어, b는 2진수이고 d는 진수등을 출력하는데 사용한다. 다음예에서 n은 d와 같이 10진수를 출력하지만 로캘이 적절하게 설정되어 있다면 큰 숫자를 세자리마다 쉼표, 로 분리하는 형식으로 출력할수 있다.

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, '')
'Korean_Korea.949'
>>> # 2진수, 10진수, 8진수, 16진수, 16진수, 10진수(d), 10진수(n)
```

...
>>> '{0:b} {0:d} {0:o} {0:x} {0:X} {1:d} {1:n}'.format(13, 123456789)
'1101 13 15 d D 123456789 123,456,789'

실수형 값에 사용하는 변환 기호로는 다음과 같은 것들이 있다.

>>> '{0:e} {0:E} {0:f} {0:F}'.format(0.6789)
'6.789000e-01 6.789000E-01 0.678900 0.678900'
>>> '{1:g} {1:G} {1:n} {0:%}'.format(0.6789, 123E30)
'1.23e+32 1.23E+32 1.23e+32 67.890000%'

e와 E는 지수형식으로 출력하고 f와 F는 고정 소수점 형식으로 출력한다. 이들 그룹은 e와 f는 소문자를 그리고 E와 F는 대문자를 출력하는 것 외에는 동일하다. g와 G는 고정소수점 형식이나 지수형식 중 적절한 것을 알아서 선택하여 출력한다. n은 g와 같으나 현재 로캘을 적용하여 출력하며 %는 백분율을 적용하여 출력한다. 표 4-2에 있는 변환 기호에 보조적인 서식문자를 함께 사용해서 출력 양식을 조절할수 있다.

보조 서식 문자

기호	설명	예
m	m개의 최소 자리를 확보한다. '|egg| egg|;	>>> '{0:5s}'.format('egg') 'egg'
m.n	m개의 최소 자리를 확보하고, n개의 소수점 이하 자리를 출력한다.	>>> '{0:10.3f}'.format(123.456789) ' 123.457'

<
>
	왼쪽이나 오른쪽으로 맞추어서 출력한다.	>>> '{0:<5d}'.format(123)
		'123 '
		>>> '(0:>5d)'.format(123)
		' 123'

+	+/- 부호를 출력한다. '+123, -123'	>>> '{0:+d}'.format(123) '+123'
공백	양수일 때 공백을 삽입한다. '123, -123'	>>> '{0: d}'.format(123) ' 123'
#	8진수 출력에는 0을 16진수 출력에는 0x이나 0X를 앞에 붙인다.	>>> '{0:#o} {0:#x}'.format(123) '0o173 0x7b'
0	왼쪽 빈 공간을 0으로 채운다.	>>> '{0:05d}'.format(123) '00123'

4.4 문자열 메서드

다음은 대소문자 변환에 관련된 메서드를 사용한 예이다.

```
>>> s = 'i like porgramming.'
>>> s.upper()   # 대문자로 변환한다.
'I LIKE PORGRAMMING.'
>>> s.lower()   # 소문자로 변환한다.
'i like porgramming.'
>>> 'I Like Programming'.swapcase() )
'i lIKE pROGRAMMING'
>>> s.capitalize()      # 첫 문자를 대문자로 변환한다.
'I like porgramming.'
>>> s.title()
'I Like Porgramming. '
```

다음은 검색에 관련된 메서드를 사용한 예이다.

```
>>> s = 'i like programming, i like swimming.'
>>> s.count('like')            # 문자열 s에서  'like '라는 부분 문자열이 발생한 횟수를 반환한다.
2
>>> s.find('like')             # 문자열 s에서  'like '의 오프셋을 반환한다. 검색에 해당한다.
2
>>> s.find('like',3)           # 문자열 3번째 위치부터 검색한다.
22
>>> s.find('my')               # 찾는 문자열이 없을 경우 -1을 반환한다.
-1
>>> s.rfind('like')            # fine() 메서드와 같지만 문자열 s의 뒤쪽부터 검색한다.
22
>>> s.index('like')            # find() 메서드와 같지만,
2
>>> s.index('my')              # 찾는 문자열이 없을 경우 예외가 발생한다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> s.rindex('like')           # index() 메서드와 같지만 문자열 s의 뒤쪽부터 검색한다.
22
>>> s.startswith('i like')     # i like로 시작하는 문자열인가?
True
>>> s.endswith('swimming.')    # swimming.으로 끝나는 문자열인가?
True
>>> s.startswith('progr', 7)   # 7번째 문자열이 progr로 시작하는가?
True
```

```
>>> s.endswith('like', 0, 26)   # 0부터 26번째 사이의 문자열이 like로 끝나는가?
True
```

다음은 편집과 치환에 관련된 메서드를 사용한 예이다.

```
>>> u = 'spam and ham'
>>> u.strip()           # 좌우 공백을 없앤다.
'spam and ham'
>>> u.rstrip()          # 오른쪽 공백을 없앤다.
'spam and ham'
>>> u.lstrip()          # 왼쪽 공백을 없앤다.
'spam and ham'
>>> 'abc'.strip()
'abc'
>>> '><><abc><><><>'.strip('<>')
'abc'
>>> '><><abc><><><>\n'.strip('<>')
'abc<><><>\n'
>>> 'ㅎㅎ 파이썬 만세 ㅎㅎ'.strip('g')
'ㅎㅎ 파이썬 만세 ㅎㅎ'
>>> u.replace('spam','spam, egg')       #  'spam '을 ' spam, egg '로 치환한다.
'spam, egg and ham'
```

문자열의 분리와 결합에 간련된 메서드는 가장 자주 사용한다.

```
>>> u = 'spam and ham'
>>> u.split()   # 공백으로 분리한다.
['spam', 'and', 'ham']
>>> u.split('and')      # 'and'로 분리한다...
['spam ', ' ham']
>>> t = u.split()       # ':'문자로 결합한다. 틀리기 쉬우므로 주의가 필요하다...
>>> print('\n'.join(t)) # 줄 바꾸기로 결합한다.합한다.
spam\nand\nham
>>> print('₩n'.join(t)) # 줄 바꾸기로 결합한다.
spam
and
ham
>>> lines = '''first line       # 3줄 문자열이다.
... second line
... third line'''
>>> lines.splitlines()          # 줄 단위로 분리한다.
['first line', 'second line', 'third line']
```

여기서 ':' .join(t)을 사용하는 방법을 종종 혼동하는 경우가 있다. t는 리스트이고 ':' 는 리스트의 문자열들을 연결할 문자열이다.

split() 메서드는 두 번째 인수로 최대 분리 개수를 지정할수 있다. rsplit() 메서드는 오른쪽부터 분리하는 것을 제외하며 split() 메서드와 같다.

```
>>> s = 'one:two:three:four'
>>> s.split(':', 2)     # 두 번만 분리한다.
['one', 'two', 'three:four']
>>> s.rsplit(':',1)     # 오른쪽부터 처리한다...
['one:two:three', 'four']
```

다음은 정렬에 관련된 메서드를 사용한 예이다.

```
>>> s = 'one:two:three:four'
>>> s.split(':', 2)     # 두 번만 분리한다.
['one', 'two', 'three:four']
>>> s.rsplit(':',1)# 오른쪽부터 처리한다...
['one:two:three', 'four']
```

다음은 정렬에 관련된 메서드를 사용한 예이다.

```
>>> u = 'spam and egg'
>>> u.center(60)        # 전체 60 문자의 가운데에 맞춘다.
'                        spam and egg                         '
>>> u.ljust(60)# 왼쪽에 맞춘다.
'spam and egg                                                 '
>>> u.rjust(60)         # 오른쪽에 맞춘다.
'                                             spam and egg'
```

center()와 ljust(), rjust() 메서드는 모두 채워질 문자를 다음과 같이 선택할수 있다.

```
>>> u.center(60, '-')   # 공백대신 '-' 문자로 채운다...
'------------------------spam and egg------------------------ '
```

expandtabs() 메서드는 탭 문자를 공백 문자로 변경한다.

```
>>> '1₩tand₩t2'.expandtabs()           # 탭(₩t)을 8자 공백으로 변경한다.
'1       and     2'
>>> '1₩tand₩t2'.expandtabs(4)          # 탭을 4자 공백으로 변경한다.
'1   and 2'
```

숫자로만 구성된 문자열인지 영문자로만 구성된 문자열인지를 판별할수 있다.

```
>>> '1234'.isdigit()            # 문자열이 숫자인가?
True
>>> '123₩u2155₩u2156'
'123⅕⅖'
>>> '123₩u2155₩u2156'.isnumeric()       # 문자열이 유니코드 수치 혹은 일반 수치 문자인가?
True
>>> print('123₩u0661')                  # ₩u0661은 아라빅 숫자 1을 나타내는 유니코드이다.
123١
>>> '123\u0661'.isdecimal()     # 일반 수치 혹은 유니코드 수치(Nd category) 문자인가?
False
>>> 'abcd한글'.isalpha()        # 문자열이 영문자 혹은 유니코드 Letter 문자인가?
True
>>> '1abc234'.isalnum() # 문자열이 숫자나 영문자 혹은 유니코드 Letter문자인가?자인가?
True
>>> 'abc'.islower()             # 문자열이 소문자인가?
True
>>> 'ABC'.isupper()
True
>>> '₩t₩r₩n'.isspace()          # 공백 문자인가?
True
>>> 'This Is A Title'.istitle() # 제목 문자열인가??
True
>>> '₩n₩t'.isspace()            # 공백 문자열인가?
True
>>> 'def'.isidentifier()        # 문자열이 예약어인가?
True
>>> '₩n₩t'.isprintable()        # 문자열이 인쇄 가능한 문자들의 모임인가?
False
```

maketrans()와 translate() 메서드를 사용하면 문자를 매핑하여 변환한 결과를 얻을수 있다.

```
>>> instr = 'abcdef'
>>> outstr = '123456'
>>> trantab = ''.maketrans(instr, outstr)       # 'abcdef'를 '123456'으로
>>> trantab
{97: 49, 98: 50, 99: 51, 100: 52, 101: 53, 102: 54}
>>> 'as soon as possible'.translate(trantab)
'1s soon 1s possi2l5'
```

4.5 유니코드 문자열과 바이트

- 파이썬 3과 유니코드

파이썬 3의 문자열은 유니코드이다. 유니코드는 전 세계적으로 사용하는 문자 집합을 하나로 모은 것이다. 유니코드는 31비트로 표현되는 문자 세트이며, 온전히 한 문자를 표현하려면 4바이트를 필요로 한다. 파이썬의 문자열은 다수의 바이트를 하나의 문자로 해석하는 유니코드 문자의 모임이다.

```
>>> '₩uac00'
'가'
>>> len('가')
1
>>> '₩u8a9e'
'語'
```

예와 같이 파이썬 3에서 '₩u' 를 사용하여 유니코드 문자를 지정할수 있다. 유니코드에서는 16비트로 표현할 수 있는 65,536개(2^{16})의 문자 세트를 하나의 코드 평면이라 부르며 현재 총 17개의 코드 평면이 있다.
코드평면 0을 기본 다국어 평면이라고 부르며 코드로는 U+0000에서 U+FFFF에 해당한다. 코드 평면 0인 BMP에는 한글과 한자를 포함하여 우리가 필요로 하는대부분의 다국어 문자가 정의되어 있다. 참고로 한글은 U+AC00~U+D7AF에 할당되어 있다. 파이썬 3으로는 이 문자들을 '₩uxxxx' 와같이 표현할수 있다. 코드평면 1은 추가적인 심볼들의 정의에 사용하며, 코드 평면 2는 추가적인 표의 문자르 정의에 사용한다. 코드 평면 3에서 13은 현재 정의되어 있지 않다. 코드평면 14는 특별한 목적으로 할당되어 있고, 코드 평면 15와 16은 개인적인 용도로 할당되어 있다.

```
>>> import sys
>>> sys.maxunicode
1114111
```

그렇다면 코드 평면 0 이외의 문자들을 어떻게 표현할수 있을까? 다른 방법이 있긴 하지만, 파이썬에서는 '₩U(대문자에 주의)' 를 사용하여 32비트를 표현하도록 하고 있다. 예를들어, U+1F193 문자는 파이썬에서 다음과 같이 표현한다.

```
>>> len('₩U0010FFFF')          # 문자 두 개가 아닌 문자 하나이다.
1
```

파이썬은 코드 값에 따라서 1, 2, 4 바이트로 내부에서문자들을 표현한다. 다음은 유니코드 범위와 바이트 수의 관계이다.

- ASCII 문자와 Latin1 문자 U+0000 ~ U+00FF는 1바이트를 사용한다.
- BMP 문자 U+0000 ~ U+FFFF는 2바이트를 사용한다.
- BMP 문자가 아닌 U+10000 ~ U+10FFFF는 4바이트를 사용한다.

- 인코딩과 바이트

유니코드 문자열을 바이트의 열로 변환하는 것을 인코딩이라고 한다. 가장 간단한 인코딩 방법으로 유니코드

는 4바이트로 표현하므로 각각의 문자를 32비트정수로 표현하는 것이다. 하지만, 1바이트나 2바이트로 표현할 수 있는 문자를 4바이트로 표현하는 방법은 공간 낭비와 호환성 부족, 바이트 저장 순서 등 여러 가지 문제를 낳는다. 따라서 호환성이 있으며, 메모의 낭비도 줄일수 있는 인코딩을 사용해야 하는데 가장 많이 사용하는 유니코드 인코딩 기법을 UTF-8이다. UTF-8은 가변길이 인코딩 기법으로 코드값에 따라서 1~4바이트로 표현한다. 아스키 코드와 호환되어 U+0000 ~ U+007F 범위의 코드는 1바이트로 표현한다. 자세한 내용은 표 4-3을 참고하자.

```
>>> '파이썬만세'.encode()       # 기본값이 UTF-8이다.
b'₩xed₩x8c₩x8c₩xec₩x9d₩xb4₩xec₩x8d₩xac₩xeb₩xa7₩x8c₩xec₩x84₩xb8'
>>> '파이썬 만세'.encode('utf-16')      # UFT-16으로 인코딩을 한다.
b'₩xff₩xfe₩x0c₩xd3t₩xc7l₩xc3 ₩x00₩xcc₩xb98₩xc1'
>>> '파이썬만세'.encode('utf-32')       # UTF-32로 인코딩을 한다.
b'₩xff₩xfe₩x00₩x00₩x0c₩xd3₩x00₩x00t₩xc7₩x00₩x00l₩xc3₩x00₩x00₩xcc₩xb9₩x00₩x008₩xc1₩x00₩x00'
>>> '파이썬 만세'.encode('cp949')       # 유니코드가 아닌 코드 세트로도 인코딩이 가능하다
b'₩xc6₩xc4₩xc0₩xcc₩xbd₩xe3 ₩xb8₩xb8₩xbc₩xbc'
```

인코딩을 한출력 결과는 문자열이 아닌 바이트의 열임에 주의해야 한다.

```
>>> type('한글'.encode())
<class 'bytes'>
```

문자열은 멀티바이트를 하나의 문자로 해석하는 유니코드로만 표현하기 때문에 인코딩이 된 개개의 바이트는 bytes 자료형으로 전환된다. 즉, 바이트는 0 ~ 255 사이의 코드 값의 열이며, 문자열은 여러 바이트를 하나의 무자로 해석하는 추상적인 문자의 열이다.
바이트는 문자열과 같이 변경 불가능 자료형이며 문자열이 지원하는 대부분의 메서드를 지원한다.

```
>>> b = b'bytes'         # 바이트를 정의한다.
>>> type(b)              # 자료형을 확인한다.
<class 'bytes'>
>>> len(b)               # 바이트 길이 정보
5
>>> b[0]                 # 인덱싱
98
>>> b.upper()            # 문자열이 지원하는 대부분의 메서드를 지원한다.
b'BYTES'
```

하지만, 바이트와 문자열 간의 직접적인 연산은 허용되지 않는다. 바이트와 바이트 간에 또는 문자열과 문자열 간에 연산이 이루어지도록 변환해야 한다.

```
>>> b + 'string'         # 바이트와 문자열 간에 연산이 허용되지 않는다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: can't concat str to bytes
>>> b + 'string'.encode()       # 바이트와 바이트 간의 연산이다.
b'bytesstring'
>>> b.decode() + 'string'       # 문자열과 문자열 간의 연산이다..
'bytesstring'
```

바이트를 문자열로 변환하려면 decode() 메서드를 사용해야 한다.

```
>>> b = '파이썬만세'.encode('cp949')
>>> b.decode('cp949')
'파이썬만세'
```

UTF-8 인코딩

유니코드 범위	UTF-8 인코딩
U+00000000 ~ U+0000007F	0xxxxxxx
U+00000080 ~ U+0000007F	110xxxxx 10xxxxxx
U+00000800 ~ U+0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U+00010000 ~ U+001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
U+00200000 ~ U+03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
U+04000000 ~ U+7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

U+0000부터 U+007F까지의 문자들은 그대로 사용하여 ASCII 문자와의 호환성이 유지된다. U+007F 보다 큰 모든 문자는 각각 독자적인 바이트의 연속으로 인코딩하며, 이것들은 각각 고유한 비트 세트를 가진다.

```
>>> '\u0080'.encode('utf-8')    # U+0080을 UTF-8로 인코딩한다.
b'₩xe2₩x88₩x96u0080'
>>> bin(0xc2), bin(0x80)        # 2진수로 출력해 보면 1100 0010 1000 0000이다.
('0b11000010', '0b10000000')
>>> '가나',encode( 'utf-16' ))
  File "<stdin>", line 1
    '가나',encode( 'utf-16' )
                   ^
SyntaxError: invalid character in identifier
>>> '가나',encode('utf-16') )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'encode' is not defined
>>>
>>> '가나'.encode('utf-16')
b'₩xff₩xfe₩x00₩xac₩x98₩xb0'
>>> bytes.fromhex('FF BE 00 AC 98 B0').decode('utf-16')
```

'₩ueeff가나'
>>> bytes.fromhex('FE FF AC 00 B0 98').decode('utf-16')))))
'가나'

- 형변환
파이썬 3의 문자의 코드 값을 알려면 ord() 함수를 사용하고 반대로 코드 값에서 문자로 변환하려면 chr() 함수를 사용한다.

```
>>> ord('가')            # 문자를 코드 값으로 변환한다.
44032
>>> hex(ord('가'))))
'0xac00'
>>> chr(0xac00)          # 코드 값을 문자로 변환한다.
'가'
```

첫 번째 예는 16진수 문자열을 바이트 열로 변환한다. 두 번째 예는 16진수 문자열을 유니코드 문자열로 변환한다.

- 예제 : 유니코드에서 한글 자소 추출하기
유니코의 한글은 초성과 중성, 종성의 순서만 알면 조합이 가능하도록 되어있다. 결론적으로 유니코드 한글 코드 값은 다음식에 의해서 결정된다.

한글 유니코드 = 0xAC00 + ((초성순서 * 21) + 중성순서) * 28 + 종성순서

여기서 각 자소의 순서는 다음 표와 같다.

초성 순서(19개)

ㄱ/0	ㄲ/1	ㄴ/2	ㄷ/3	ㄸ/4	ㄹ/5	ㅁ/6	ㅂ/7
ㅃ/8	ㅅ/9	ㅆ/10	ㅇ/11	ㅈ/12	ㅉ/13	ㅊ/14	ㅋ/15
ㅌ/16	ㅍ/17	ㅎ/18					

중성 순서(21개)

ㅏ/0	ㅐ/1	ㅑ/2	ㅒ/3	ㅓ/4	ㅔ/5	ㅕ/6	ㅖ/7
ㅗ/8	ㅘ/9	ㅙ/10	ㅚ/11	ㅛ/12	ㅜ/13	ㅝ/14	ㅞ/15
ㅟ/16	ㅠ/17	ㅡ/18	ㅢ/19	ㅣ/20			

종성 순서(28개)

없음/0	ㄱ/1	ㄲ/2	ㄱㅅ/3	ㄴ/4	ㄴㅈ/5	ㄴㅎ/6	ㄷ/7
ㄹ/8	ㄹㄱ/9	ㄹㅁ/10	ㄹㅂ/11	ㄹㅅ/12	ㄹㅌ/13	ㄹㅍ/14	ㄹㅎ/15
ㅁ/16	ㅂ/17	ㅂㅅ/18	ㅅ/19	ㅆ/20	ㅇ/21	ㅈ/22	ㅊ/23
ㅋ/24	ㅌ/25	ㅍ/26	ㅎ/27				

예를들어, '가' 는 초성 0, 중성 0, 종성 0이므로 0xAC00에 해당한다. '한' 은 초성 18, 중성 0, 종성 4이므로 0xd55c가 된다.

```
>>> 0xac00 + ((18 * 21) + 0) * 28 + 4
54620
>>> hex(54620)          # 16진수로 표시하면 0xd55c이다.
'0xd55c'
>>> chr(0xd55c)
'한'
```

자소의 상대적인 인덱스 값을 사용해서 한글을 완성하는 함수 compose_hangul()를 만들어 보자.

```
>>> def compose_hangul(cho, jung, jong):
...     code = 0xac00 + ((cho * 21) + jung) * 28 + jong
...     return chr(code)
...
>>> print(compose_hangul(18, 0, 4))
한
```

독립된 자소로부터 한글을 조합하려면 다음과 같이 compose_hamgul2() 함수로 가능하다.

```
>>> cho_list = [ "ㄱ", "ㄲ", "ㄴ", "ㄷ", "ㄸ", "ㄹ", "ㅁ", "ㅂ", "ㅃ", "ㅅ", "ㅆ", "ㅇ", "ㅈ", "ㅉ", "ㅊ", "ㅋ", "ㅌ", "ㅍ", "ㅎ"]
>>> jung_list = ["ㅏ", "ㅐ", "ㅑ", "ㅒ", "ㅓ", "ㅔ", "ㅕ", "ㅖ", "ㅗ", "ㅘ", "ㅙ", "ㅚ", "ㅛ", "ㅜ", "ㅝ", "ㅞ", "ㅟ", "ㅠ", "ㅡ", "ㅢ", "ㅣ"]
>>> jong_list = ["", "ㄱ", "ㄲ", "ㄳ", "ㄴ", "ㄵ", "ㄶ", "ㄷ", "ㄹ", "ㄺ", "ㄻ", "ㄼ", "ㄽ", "ㄾ", "ㄿ", "ㅀ", "ㅁ", "ㅂ", "ㅄ", "ㅅ", "ㅆ", "ㅇ", "ㅈ", "ㅊ", "ㅋ", "ㅌ", "ㅍ", "ㅎ"]

>>> def compose_hangul2(cho_c, jung_c, jong_c):
...     cho = cho_list.index(cho_c)            # 초성의 순서 위치를 얻는다
...     jung = jung_list.index(jung_c)         # 중성의 순서 위치를 얻는다
...     jong = jong_list.index(jong_c)         # 종성의 순서 위치를 얻는다
...     code = 0xac00 + ((cho*21) + jung)*28 + jong          # 한글 조합 코드 계산
...     return chr(code)
...
>>> print (compose_hangul2('ㅎ', 'ㅏ', 'ㄴ'))
한
```

반대로 한글이 주어져 있을 때 자소로 분리해 보자. 우선 0xAC00을 빼고 난후에 각 자소를 분리하면 된다.

```
>>> c = '한'
>>> code = ord(c) - 0xac00      # 수치로 형변환
```

```
>>> chosung = code // (21*28)   # 초성
>>> jungsung = (code - chosung*21*28) // 28     # 중성
>>> jongsung = (code - chosung*21*28 - jungsung*28)     # 종성
>>>
>>> print(chosung, jungsung, jongsung)
18 0 4
```

만일 분리된 독립 자소를 얻으려면 다음과 같이 리스트에서 값을 추출하면 된다.

```
>>> print(cho_list[chosung], jung_list[jungsung], jong_list[jongsung])
ㅎ ㅏ ㄴ
```

다음예는 영타로 '한글' 을 입력할 경우 gksrmf이 된다. 이것을 한글로 변환하는 코드의 예이다. 방법은 자소 목록이 한글로 되어있지 않고 알파벳으로 되어 있는 것을 제외하오는 compose_hangul2() 함수와 동일하다.

```
>>> cho_list_eng = [ "r", "R", "s", "e", "E", "f", "a", "q", "Q", "t", "T", "d", "w", "W", "c", "z", "x", "v", "g"]
>>> jung_list_eng = ["k", "o", "I", "O", "j", "p", "u", "P", "h", "hk", "ho", "hl", "y", "n", "nj", "np", "nl", "b", "m", "ml", "l"]
>>> jong_list_eng = ["", "r", "R", "rt", "s", "sw", "sg", "e", "f", "fr", "fa", "fq", "ft", "fx", "fv", "fg", "a", "q", "qt", "t", "T", "d", "w", "C", "z", "x", "v", "g"]

>>> def compose_hangul3(cho_e, jung_e, jong_e=''):
...     cho = cho_list_eng.index(cho_e)
...     jung = jung_list_eng.index(jung_e)
...     jong = jong_list_eng.index(jong_e)
...     code = 0xac00 + ((cho*21) + jung)*28 + jong
...     return chr(code)

>>> print (compose_hangul3('g', 'k', 's'), compose_hangul3('r', 'm', 'f'))
한 글
```