

3

유용한 문자열 메소드



여기에서는 5장의 ‘자주 사용하는 문자열 메소드’(152-164p)에서 다루지 않은 유용한 문자열 몇 가지를 추가로 소개한다.

문자열 분할 메소드 `split()`

본문에서 다룬 `split()` 메소드에 대해 좀 더 자세히 알아보자. 여기서 설명하는 모든 내용은 `rsplit()`에도 적용된다. 본문에서도 설명했지만 구분자 `sep`를 지정하지 않으면 연속으로 붙어 있는 화이트스페이스 문자는 하나로 취급된다. 따라서 화이트스페이스 문자로만 이루어져 있는 문자열에 대해 구분자 `sep`를 지정하지 않고 `split()` 메소드를 사용하면 빈 리스트를 반환한다.

```
>>> ' \t \n \n\t \t '.split()
[]
>>> '          '.split()
[]
>>> ''.split()
[]
```

이때 주의할 점은 구분자 `sep`를 지정하면 문자열 안의 연속된 구분자는 하나의 그룹으로 취급되지 않고 반환되는 리스트에서 빈 문자열로 처리된다는 점이다.

```
>>> '가,,나'.split(',') # 구분자','가 연속으로 두 개 있다.
['가', '', '나']
>>> '1/2//3/4//5//'.split('/') # 구분자 '/'가 연속으로 두 개, 세 개 있다.
['1', '2', '', '3', '4', '', '5', '', '', '']
>>> '빅데이터\n'.split('\n') # 구분자 '\n'이 끝에 있다.
['빅데이터', '']
>>> '빅데이터\n\n'.split('\n') # 구분자 '\n'이 끝에 두 개 있다.
['빅데이터', '', '']
>>> '분석 자료\n\n파이썬 언어\r초급\r\n'.split('\n') # 구분자 '\n'이 연속으로 두 개 있다.
['분석 자료', '', '파이썬 언어\r초급\r', '']
>>> '분석 자료\n\n\n파이썬 언어\r초급\r\n'.split('\n') # 구분자 '\n'이 연속으로 세 개 있다.
['분석 자료', '', '', '파이썬 언어\r초급\r', '']
```

만약 빈 문자열을 분할하려고 시도하면 구분자 sep를 지정하더라도 구분자를 적용할 수 없기 때문에 빈 문자열 그대로를 담고 있는 리스트를 반환한다.

```
>>> ''.split('\n')
['']
>>> ''.split(',')
['']
>>> ''.split(' ')
['']
```

문자열 분할 메소드 splitlines()

문자열.splitlines([keepends])

- 이 메소드는 줄 경계 문자를 기준으로 문자열을 분리해서 리스트 형태로 반환한다.
 - 줄 경계 문자에는 새줄바꿈(\n), 캐리지 리턴(\r), 캐리지 리턴과 새줄바꿈(\r\n) 수직 탭(\v 또는 \x0b), 페이지 넘기기(\f 또는 \x0c), 파일 분리자(\x1c), 그룹 분리자(\x1d), 레코드 분리자(\x1e), C1 제어 코드인 다음 줄(\x85), 줄 분리자(\u2028), 문단 분리자(\u2029)가 포함된다.
- keepends가 True로 지정되지 않는 한, 줄 경계 문자들은 분리된 문자열에 포함되지 않는다.

다음 예시는 `splitlines()` 메소드를 호출할 때 `keepends`를 생략하고 기본값을 적용했기 때문에 줄 경계 문자인 새줄바꿈을 기준으로 모두 분할한 후, 새줄바꿈 문자는 포함하지 않은 문자열을 리스트로 반환하는 것을 보여준다.

```
>>> '하나\n둘\n'.splitlines()
['하나', '둘']
```

반면에 `split()` 메소드를 사용해 구분자로 새줄바꿈 문자를 지정해서 실행하면 예시의 문자열 마지막에 구분자인 새줄바꿈 문자가 있기 때문에 빈 문자열 하나가 더 추가된 리스트를 반환하게 된다.

```
>>> '하나\n둘\n'.split('\n')
['하나', '둘', '']
```

`splitlines()` 메소드 역시 문자열의 끝에 새줄바꿈 부호가 연속으로 있으면 하나의 그룹으로 취급하지 않고 빈 문자열을 반환한다.

```
>>> '하나\n\n둘\n\n'.splitlines()
['하나', '', '둘', '']
```

만약 `keepends`를 `True`로 지정해서 이 메소드를 호출하면 새줄바꿈 문자를 포함한 문자열을 리스트로 반환한다.

```
>>> '하나\n둘\n'.splitlines(keepends=True)
['하나\n', '둘\n']
```

`splitlines(True)`와 같다.

또한 `keepends`를 `True`로 지정한 경우, 새줄바꿈 부호가 연속으로 있으면 빈 문자열 대신 새줄바꿈만 있는 문자열을 반환한다.

```
>>> '하나\n\n둘\n\n'.splitlines(True)
['하나\n', '\n', '둘\n', '\n']
```

`splitlines(keepends=True)`와 같다.

이는 split() 메소드의 구분자로 새줄바꿈 문자를 지정해서 실행할 경우 빈 문자열을 반환하는 것과는 차이가 있다.

```
>>> '하나\n\n둘\n\n'.split('\n')
['하나', '', '둘', '', '']
```

그런데, 화이트스페이스 문자 기준으로 분할하기 위해 split() 메소드의 구분자를 지정하지 않고 호출할 경우, 앞서 설명했듯이 연속으로 붙어 있는 화이트스페이스 문자는 하나로 취급해서 분할하기 때문에 다음과 같은 결과를 가져온다.

```
>>> '하나\n\n둘\n\n'.split()
['하나', '둘']
```

문자열 분할 메소드 partition()과 rpartition()

문자열.partition(구분자)

문자열.rpartition(구분자)

- 이 두 메소드는 문자열을 삼등분해서 튜플 형태(3-tuple)로 반환한다.
 - partition()은 문자열의 왼쪽부터 시작해서 구분자가 처음 나온 위치에서 문자열을 분할하며, 구분자 이전, 구분자, 구분자 이후 부분으로 삼등분해서 튜플 형태로 반환한다.
 - rpartition()은 문자열의 오른쪽부터 시작해서 구분자가 처음 나온 위치에서 문자열을 분할하며, 구분자 이전, 구분자, 구분자 이후 부분으로 삼등분해서 튜플 형태로 반환한다.
- 만약 찾는 구분자가 문자열에 없다면, 문자열 전체와 빈 문자열 2개를 튜플 형태(3-tuple)로 반환한다.
- 구분자를 지정하지 않으면 TypeError가 발생한다.

다음 예시는 구분자로 공백문자(' ')를 지정한 후, 문자열의 왼쪽부터 시작해서 구분자가 처음 나온 위치에서 삼등분하고, 그 다음에는 오른쪽부터 처음 나오는 구분자를 기준으로 삼등분하는 코드를 실행한 결과다.

```
>>> eng = 'Introduction to Python'
>>> eng.partition(' ')           # 왼쪽부터 처음 나온 공백을 기준으로 삼등분한다.
('Introduction', ' ', 'to Python')
>>> eng.rpartition(' ')         # 오른쪽부터 처음 나온 공백을 기준으로 삼등분한다.
('Introduction to', ' ', 'Python')
```

이번에는 느낌표(!)를 구분자로 하여 문자열을 왼쪽과 오른쪽에서 처음 나온 위치에서 삼등분하는 코드를 실행해보자.

```
>>> kor = '파이썬을 배우면서 파이썬을 즐기자!!!'
>>> kor.partition('!')          # 왼쪽부터 처음 나온 !를 기준으로 삼등분한다.
('파이썬을 배우면서 파이썬을 즐기자', '!', '!!!')
>>> kor.rpartition('!')        # 오른쪽부터 처음 나온 !를 기준으로 삼등분한다.
('파이썬을 배우면서 파이썬을 즐기자!!', '!', '')
```

partition()과 rpartition()은 문자열을 구분자를 기준으로 양쪽으로 분리한다. 위 예시의 마지막 코드는 구분자가 오른쪽 끝에 있기 때문에 튜플의 마지막 객체가 빈 문자열을 반환한 것을 볼 수 있다. 만약 찾는 구분자가 문자열에 없다면 다음과 같이 문자열 전체와 빈 문자열 두 개를 튜플로 반환한다.

```
>>> s = '/usr/local/bin/python'
>>> s.partition('x')
('/usr/local/bin/python', '', '')
```

문자열 질의 메소드 count()

문자열.count(부분문자열[, 시작[, 끝]])

- 이 메소드는 문자열 가운데 부분문자열과 일치하는 내용의 개수를 반환한다.
- 시작과 끝은 분할 연산자에서 사용하는 표기법과 같은 의미로 사용된다. 즉, 문자열의 시작부터 끝 바로 앞까지의 대상 가운데 부분문자열과 일치하는 부분문자열의 개수를 반환한다.

다음 예시는 문자열 'Introduction to Python'에서 소문자 'o'가 몇 번 나오는지 질의하는 코드다. 처음 코드는 전체 문자열에서 소문자 'o'가 몇 번 나오는지 확인하고, 두 번째와 세 번째 코드는 문자열에서 검색 구간을 지정해서 소문자 'o'가 몇 번 나오는지 확인하는 결과를 보여준다.

```
>>> eng = 'Introduction to Python'
>>> eng.count('o')           # 문자열에서 'o'가 몇 번 나오는지 질의한다.
4
>>> eng.count('o', -2 )     # 끝에서 두 번째부터 마지막까지 'o'가 몇 번 나오는지 질의한다.
1
>>> eng.count('o', 0, -2 )  # 처음부터 끝에서 세 번째까지 'o'가 몇 번 나오는지 질의한다.
3
```

이번에는 한글 문자열에서 원하는 대상이 몇 번 나오는지 개수를 세어보자.

```
>>> kor = '파이썬을 배우면서 파이썬을 즐기자!!!'
>>> kor.count('파이썬')      # 문자열에서 '파이썬'이 몇 번 나오는지 질의한다.
2
```

문자열 질의 메소드 find()와 rfind()

```
문자열.find(부분문자열[, 시작[, 끝]])  
문자열.rfind(부분문자열[, 시작[, 끝]])
```

- 이 두 메소드는 **문자열** 가운데 **부분문자열**이 처음 나온 위치의 가장 낮은 인덱스 번호를 반환한다.
 - find()는 **문자열**의 왼쪽부터 시작해서 **부분문자열**이 처음 나온 위치의 가장 낮은 인덱스 번호를 반환한다.
 - rfind()는 **문자열**의 오른쪽부터 시작해서 **부분문자열**이 처음 나온 위치의 가장 낮은 인덱스 번호를 반환한다.
- **시작**과 **끝**은 분할 연산자에서 사용하는 표기법과 같은 의미로 사용된다. 즉, **문자열**의 **시작**부터 **끝** 바로 앞까지의 대상 가운데 **부분문자열**이 왼쪽/오른쪽부터 처음 나온 위치를 반환한다.
- 만약 찾는 **부분문자열**이 존재하지 않는다면, -1을 반환한다.

다음 예시는 문자열 '파이썬을 배우면서 파이썬을 즐기자!!!'에서 '파이썬'이 왼쪽에 처음 나타나는 위치와 오른쪽에 처음 나타나는 위치를 찾는다. 이때 부분 문자열의 문자가 한 개가 아니라 두 개 이상일 경우 부분 문자열을 찾은 위치에서 가장 낮은 인덱스 번호를 반환한다.

```
>>> kor = '파이썬을 배우면서 파이썬을 즐기자!!!'  
>>> kor.find('파이썬') # 문자열의 처음부터 시작해서 '파이썬'이 처음 나온 위치를 알아본다.  
0  
>>> kor.rfind('파이썬') # 문자열의 끝부터 시작해서 '파이썬'이 처음 나온 위치를 알아본다.  
10
```

이번에는 문자열에서 검색 구간을 정해서 찾아보도록 하자. 이때 반환하는 위치 값은 전체 문자열의 절댓값이지 구간 내의 상대 위치는 아니다.

```
>>> kor.find('파이썬', 5) # 여섯 번째부터 시작해서 '파이썬'이 처음 나온 위치를 알아본다.
10
>>> # 여섯 번째부터 끝에서 네 번째 사이에서 '자'가 처음 나온 위치를 알아본다.
... kor.find('자', 5, -3)
17
```

만약 찾고자 하는 부분 문자열이 없다면 다음 예시처럼 -1을 반환한다.

```
>>> kor.find('자바') # 문자열의 처음부터 시작해서 '자바'가 처음 나온 위치를 알아본다.
-1
>>> # 여섯 번째부터 끝에서 네 번째 사이에서 '!'가 처음 나온 위치를 알아본다.
... kor.rfind('!', 5, -3)
-1
```

문자열 질의 메소드 index()와 rindex()

```
문자열.index(부분문자열[, 시작[, 끝]])
문자열.rindex(부분문자열[, 시작[, 끝]])
```

- 이 두 메소드는 find()와 rfind()처럼 문자열 중 부분문자열이 처음 나온 위치의 가장 낮은 인덱스 번호를 반환한다.
 - index()는 문자열의 왼쪽부터 시작해서 부분문자열이 처음 나온 위치의 가장 낮은 인덱스 번호를 반환한다.
 - rindex()는 문자열의 오른쪽부터 시작해서 부분문자열이 처음 나온 위치의 가장 낮은 인덱스 번호를 반환한다.
- 시작과 끝은 분할 연산자에서 사용하는 표기법과 같은 의미로 사용된다. 즉, 문자열의 시작부터 끝 바로 앞까지의 대상 가운데 부분문자열이 왼쪽/오른쪽부터 처음 나온 위치를 반환한다.
- 만약 찾는 부분문자열이 존재하지 않는다면, ValueError가 발생한다.

앞서 find()와 rfind()에서 사용한 예시를 index()와 rindex()에서 그대로 사용해 보자. 먼저 문자열 '파이썬을 배우면서 파이썬을 즐기자!!!'에서 '파이썬'이 왼쪽에 처음 나타나는 위치를 찾고 그 다음에 오른쪽에 처음 나타나는 위치를 찾는다. find(), rfind()와 마찬가지로 부분 문자열의 문자가 한 개가 아니라 두 개 이상일 경우 부분 문자열을 찾은 위치에서 가장 낮은 인덱스 번호를 반환한다.

```
>>> kor = '파이썬을 배우면서 파이썬을 즐기자!!!'
>>> kor.index('파이썬')    # 문자열의 처음부터 시작해서 '파이썬'이 처음 나온 위치를 알아본다.
0
>>> kor.rindex('파이썬')   # 문자열의 끝부터 시작해서 '파이썬'이 처음 나온 위치를 알아본다.
10
```

문자열에서 구간을 정해서 검색할 경우, 마찬가지로 반환하는 위치 값은 전체 문자열의 절댓값이지 구간 내의 상대 위치는 아니다.

```
>>> kor.index('파이썬', 5)    # 여섯 번째부터 시작해서 '파이썬'이 처음 나온 위치를 알아본다.
10
>>> # 여섯 번째부터 끝에서 네 번째 사이에서 '자'가 처음 나온 위치를 알아본다.
... kor.index('자', 5, -3)
17
```

find()와 rfind() 메소드와는 달리, index()와 rindex()의 경우에는 만약 찾고자 하는 부분 문자열이 없다면 다음처럼 ValueError가 발생한다.

```
>>> # 여섯 번째부터 끝에서 네 번째 사이에서 '!'가 처음 나온 위치를 알아본다.
... kor.rindex('!', 5, -3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

문자열 질의 메소드 startswith()와 endswith()

```
문자열.startswith(접두문자열[, 시작[, 끝]])  
문자열.endswith(접미문자열[, 시작[, 끝]])
```

- startswith() 메소드는 **문자열**이 **접두문자열**로 시작하면 '참(True)'을 반환하고, 아니면 '거짓(False)'을 반환한다.
- endswith() 메소드는 **문자열**이 **접미문자열**로 끝나면 '참(True)'을 반환하고, 아니면 '거짓(False)'을 반환한다.
- **시작**과 **끝**은 분할 연산자에서 사용하는 표기법과 같은 의미로 사용된다. 즉, **문자열**의 **시작**부터 **끝** 바로 앞까지의 대상 가운데 **접두문자열**로 시작하는지 또는 **접미문자열**로 끝나는지를 확인한다.

다음 예시는 문자열 '파이썬을 배우면서 파이썬을 즐기자!!!'를 사용해서 접두 문자열과 접미 문자열을 확인하는 코드를 실행한 결과를 보여준다.

```
>>> kor.startswith('파이썬')           # 문자열이 '파이썬'으로 시작하는지 확인한다.  
True  
>>> kor.endswith('!!!')                 # 문자열이 '!'로 끝나는지 확인한다.  
True
```

이번에는 구간을 정해놓고 접두 문자열과 접미 문자열의 존재 여부를 확인해보자.

```
>>> kor.startswith('파이썬', 5) # 문자열의 여섯 번째가 '파이썬'으로 시작하는지 확인한다.  
False  
>>> # 문자열의 처음부터 끝에서 네 번째까지가 '!'로 끝나는지 확인한다.  
... kor.endswith('!', 0, -3)  
False
```

문자열 질의 메소드 is~()

```
문자열.isspace()  
문자열.isalpha()  
문자열.isdecimal()  
문자열.isdigit()  
문자열.isnumeric()  
문자열.isalnum()  
문자열.isascii()  
문자열.isprintable()
```

- `isspace()` 메소드는 문자열이 모두 화이트스페이스 문자이면 '참(True)'을 반환하고, 하나라도 화이트스페이스 문자가 아니면 '거짓(False)'을 반환한다.
- `isalpha()` 메소드는 문자열이 모두 알파벳 글자이면 '참(True)'을 반환하고, 하나라도 알파벳이 아니면 '거짓(False)'을 반환한다.
- `isdecimal()` 메소드는 문자열이 모두 10진수 문자(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)이면 '참(True)'을 반환하고, 하나라도 10진수 문자가 아니면 '거짓(False)'을 반환한다.
- `isdigit()` 메소드는 문자열이 모두 숫자이면 '참(True)'을 반환하고, 하나라도 숫자가 아니면 '거짓(False)'을 반환한다.
 - 숫자는 10진수 문자(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)와 윗첨자 등 특별한 처리가 필요한 10진수 문자를 포함한다.
- `isnumeric()` 메소드는 문자열이 모두 수치로 표현되면 '참(True)'을 반환하고, 하나라도 수치로 표현되지 않으면 '거짓(False)'을 반환한다.
 - 수치란 숫자와 유니코드 중 숫자 값 속성이 있는 문자를 포함한다.
- `isalnum()` 메소드는 문자열이 모두 알파벳 글자나 수치이면 '참(True)'을 반환하고, 하나라도 알파벳 글자나 수치가 아니면 '거짓(False)'을 반환한다.
 - `isalpha()`, `isdecimal()`, `isdigit()`, `isnumeric()` 메소드 중 하나라도 '참(True)'을 반환하는 문자들로 이루어진 문자열인 경우 '참(True)'을 반환한다.
- `isascii()` 메소드는 문자열의 모든 문자가 비어 있거나 아스키 문자이면 '참(True)'을 반환하고, 하나라도 그렇지 않으면 '거짓(False)'을 반환한다.

- `isprintable()` 메소드는 문자열의 모든 문자가 비어 있거나 인쇄 가능한 문자이면 '참(True)'을 반환하고, 인쇄 가능하지 않은 문자가 하나라도 있으면 '거짓(False)'을 반환한다.

다음 예시는 `isspace()` 메소드를 사용해서 문자열의 문자가 모두 화이트스페이스 문자인지를 확인하는 코드다.

```
>>> '\n\r\t '.isspace()      # 모든 문자가 화이트스페이스 문자이면 '참'을 반환한다.
True
>>> '\nr\t '.isspace()      # 모든 문자가 화이트스페이스 문자가 아니면 '거짓'을 반환한다.
False
```

문자열의 모든 문자가 알파벳 글자로만 구성되어 있는지 확인하려면 `isalpha()` 메소드를 사용하면 된다.

```
>>> 'a가'.isalpha()          # 모든 문자가 알파벳 글자이면 '참'을 반환한다.
True
>>> 'a3가'.isalpha()        # 하나라도 알파벳 글자가 아니면 '거짓'을 반환한다.
False
>>> 'a.가'.isalpha()        # 하나라도 알파벳 글자가 아니면 '거짓'을 반환한다.
False
```

`isdecimal()` 메소드를 사용하면 문자열의 모든 문자가 0, 1, 2, 3, 4, 5, 6, 7, 8, 9로만 이루어진 문자인지, 즉 10진수 숫자로만 구성된 문자열인지를 확인할 수 있다.

```
>>> '0123456789'.isdecimal() # 모든 문자가 10진수로 표현되면 '참'을 반환한다.
True
>>> '-5'.isdecimal()         # 하나라도 10진수 숫자가 아니면 '거짓'을 반환한다.
False
>>> '1.23'.isdecimal()       # 하나라도 10진수 숫자가 아니면 '거짓'을 반환한다.
False
>>> '007'.isdecimal()        # 모든 문자가 10진수로 표현되면 '참'을 반환한다.
True
```

만약 문자열에 포함된 문자 가운데 특별한 문자인 첨자들을 포함해서 모든 문자가 10진수(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)로만 이루어져 있는지 확인하려면 `isdigit()` 메소드를 사용하면 된다. 이 메소드는 `isdecimal()` 메소드가 ‘참(True)’을 반환하는 문자를 포함하여, 그렇지 않은 첨자 등도 10진수로 표현할 수 있는 문자인지 확인할 수 있다. 예를 들어, 다음 예시처럼 윗첨자 ‘3’의 경우 `isdecimal()` 메소드는 ‘거짓(False)’을 반환하지만, `isdigit()` 메소드는 ‘참(True)’을 반환한다.

```
>>> print('\u00b3')           # 윗첨자(superscript) 3
3
>>> '\u00b3'.isdecimal()
False
>>> '\u00b3'.isdigit()
True
```

하지만 `isdigit()` 메소드도 `isdecimal()` 메소드와 마찬가지로 숫자 이외의 문자가 하나라도 포함되면 ‘거짓(False)’을 반환한다.

```
>>> '-5'.isdigit()           # 하나라도 숫자가 아니면 '거짓'을 반환한다.
False
>>> '1.23'.isdigit()         # 하나라도 숫자가 아니면 '거짓'을 반환한다.
False
```

만약 첨자 문자를 포함하여 유니코드 문자 가운데 숫자로 표현할 수 있는 문자들도 확인하려면 `isnumeric()` 메소드를 사용하면 된다. 이 메소드는 문자열의 모든 문자가 수치로 표현되면 ‘참(True)’을 반환한다. 따라서 이 메소드는 `isdecimal()`과 `isdigit()` 메소드가 ‘참(True)’을 반환하는 문자를 포함하여, 숫자를 의미하는 유니코드 문자들을 포괄한 수치들로 문자열이 이루어져 있는지 확인할 때 사용할 수 있다. 다음 예시는 이 세 가지 메소드가 분수 1/5을 의미하는 유니코드를 어떻게 처리하는지를 보여준다.

```
>>> print('\u2155')           # chr(8533)와 같다. 분수 1/5을 표현하는 유니코드 문자다.
½
>>> '\u2155'.isdecimal()
False
>>> '\u2155'.isdigit()
False
>>> '\u2155'.isnumeric()
True
```

isnumeric() 메소드 역시 수치 이외의 문자가 하나라도 포함되면 다음 예시처럼 ‘거짓(False)’을 반환한다.

```
>>> '-5'.isnumeric()          # 문자 중 하나라도 수치가 아니면 '거짓'을 반환한다.
False
>>> '1.23'.isnumeric()        # 문자 중 하나라도 수치가 아니면 '거짓'을 반환한다.
False
```

isalnum() 메소드는 문자열의 모든 문자가 알파벳 글자이거나 수치로 표현되면 ‘참(True)’을 반환한다. 즉, 앞에서 다룬 isdecimal(), isdigit(), isnumeric(), isalpha() 메소드 중 어느 하나라도 ‘참(True)’을 반환하는 문자열이라면, isalnum() 메소드도 ‘참(True)’을 반환한다. 다음은 isalnum() 메소드로 몇 가지 예를 테스트한 결과다.

```
>>> '3동a호'.isalnum()        # 모든 문자가 알파벳 글자나 수치이면 '참'을 반환한다.
True
>>> '07호C동'.isalnum()       # 모든 문자가 알파벳 글자나 수치이면 '참'을 반환한다.
True
>>> '-3동a호'.isalnum()       # 하나라도 알파벳 글자나 수치가 아니면 '거짓'을 반환한다.
False
>>> '-3'.isalnum()            # 하나라도 알파벳 글자나 수치가 아니면 '거짓'을 반환한다.
False
>>> '1.23'.isalnum()          # 하나라도 알파벳 글자나 수치가 아니면 '거짓'을 반환한다.
False
```

isascii() 메소드는 문자열의 모든 문자가 비어 있거나 아스키 문자이면 '참(True)'을 반환하고, 그렇지 않으면 '거짓(False)'을 반환한다.

```
>>> ''.isascii()           # 모든 문자가 비어 있거나 아스키 문자이면 '참'을 반환한다.
True
>>> 'abCD'.isascii()       # 모든 문자가 비어 있거나 아스키 문자이면 '참'을 반환한다.
True
>>> ' abCD'.isascii()      # 모든 문자가 비어 있거나 아스키 문자이면 '참'을 반환한다.
True
>>> 'abCD123'.isascii()    # 모든 문자가 비어 있거나 아스키 문자이면 '참'을 반환한다.
True
>>> '-3.14abCD'.isascii()  # 모든 문자가 비어 있거나 아스키 문자이면 '참'을 반환한다.
True
```

isascii() 메소드는 문자열 가운데 하나라도 아스키 문자가 아니면 '거짓(False)'을 반환한다. 다음 예시는 문자열에 한글이 포함되어 있기 때문에 '거짓(False)'을 반환했다.

```
>>> '3아가'.isascii()      # 문자 중 하나라도 아스키 문자가 아니면 '거짓'을 반환한다.
False
```

마지막으로 isprintable() 메소드는 문자열이 포함한 모든 문자가 공백이거나 인쇄 가능한 문자이면 '참(True)'을 반환하고, 화이트스페이스 문자 등 인쇄가 가능하지 않은 문자가 하나라도 있으면 '거짓(False)'을 반환한다.

```
>>> 'a.가'.isprintable()    # 모든 문자가 인쇄 가능하면 '참'을 반환한다.
True
>>> '-3.14'.isprintable()    # 모든 문자가 인쇄 가능하면 '참'을 반환한다.
True
>>> ' a가\n'.isprintable()   # 하나라도 인쇄가 가능하지 않으면 '거짓'을 반환한다.
False
>>> '\n\t'.isprintable()     # 하나라도 인쇄가 가능하지 않으면 '거짓'을 반환한다.
False
```

이때 예외가 있는데, 빈 문자열('')과 공백 문자(' ')는 인쇄가 가능한 문자로 취급한다.

```
>>> ''.isprintable() # 모든 문자가 인쇄 가능하면 '참'을 반환한다.
True
>>> ' '.isprintable() # 모든 문자가 인쇄 가능하면 '참'을 반환한다.
True
>>> '5'.isprintable() # 모든 문자가 인쇄 가능하면 '참'을 반환한다.
True
>>> 'n'.isprintable() # 모든 문자가 인쇄 가능하면 '참'을 반환한다.
True
```

영어 관련 문자열 질의 메소드

```
문자열.isitle()
문자열.islower()
문자열.isupper()
```

- `istitle()`은 문자열에 있는 모든 영어 단어들의 첫 문자만 대문자고 나머지는 모두 소문자로 구성되어 있으면 '참(True)'을 반환하고, 그렇지 않으면 '거짓(False)'을 반환한다.
- `islower()`는 문자열에 있는 모든 영문자가 소문자이면 '참(True)'을 반환하고, 하나라도 대문자가 있으면 '거짓(False)'을 반환한다.
- `isupper()`는 문자열에 있는 모든 영문자가 대문자이면 '참(True)'을 반환하고, 하나라도 소문자이면 '거짓(False)'을 반환한다.

다음 예시는 `istitle()` 메소드를 실행한 결과를 보여준다. 영문자에 대해서만 적용이 된다는 것을 볼 수 있다.

```
>>> 'Right Way To 파이썬'.istitle() # 모든 영어 단어의 첫 문자만 대문자면 '참'을 반환한다.
True
>>> 'Right way to 파이썬'.istitle() # 그렇지 않으면 '거짓'을 반환한다.
False
```


islower() 메소드는 문자열에 있는 영문자가 모두 소문자인 경우 '참(True)'을 반환하고, 그렇지 않으면 '거짓(False)'을 반환한다.

```
>>> 'right way to 파이썬'.islower() # 모든 영문자가 소문자이면 '참'을 반환한다.
True
>>> 'Right way to 파이썬'.islower() # 하나라도 대문자가 있으면 '거짓'을 반환한다.
False
```

반면에 isupper() 메소드는 문자열에 있는 영문자가 모두 대문자인 경우에만 '참(True)'을 반환하고, 그렇지 않으면 '거짓(False)'을 반환한다.

```
>>> 'RIGHT WAY TO 파이썬'.isupper() # 모든 영문자가 대문자이면 '참(True)'을 반환한다.
True
>>> 'RIGHT WAY To 파이썬'.isupper() # 하나라도 소문자가 있으면 '거짓(False)'을 반환한다.
False
```

영어 관련 문자열 교체 메소드

```
문자열.title()
문자열.capitalize()
문자열.swapcase()
문자열.lower()
문자열.upper()
```

- 이 다섯 개의 메소드는 영어 문자열의 대소문자를 변경해서 새로운 문자열을 반환하는 메소드다.
 - title()은 문자열에 있는 각 영단어의 첫 문자는 대문자로, 나머지는 모두 소문자로 바꾼 문자열을 반환한다.
 - capitalize()는 문자열의 첫 문자가 영문자이면 이를 대문자로 변경하고, 나머지 모든 영문자는 소문자로 바꾼 문자열을 반환한다. 문자열의 첫 문자가 영문자가 아니면, 모든 영문자를 소문자로 바꾼 문자열을 반환한다.

- `swapcase()`는 문자열에 있는 영문자를 대문자는 소문자로, 소문자는 대문자로 바꾼 문자열을 반환한다.
- `lower()`는 문자열에 있는 모든 영문자를 소문자로 바꾼 문자열을 반환한다.
- `upper()`는 문자열에 있는 모든 영문자를 대문자로 바꾼 문자열을 반환한다.

다음 예시는 위의 다섯 개 메소드를 실행한 결과를 보여준다.

```
>>> s = 'Right Way to 파이썬 programming'
>>> s.title()                # 문자열의 각 영단어의 첫 문자를 대문자로 반환한다.
'Right Way To 파이썬 Programming'
>>> s.capitalize()          # 문자열의 첫 문자는 대문자로, 나머지는 소문자로 반환한다.
'Right way to 파이썬 programming'
>>> s.swapcase()             # 대문자는 소문자로, 소문자는 대문자로 반환한다.
'rIGHT wAY TO 파이썬 PROGRAMMING'
>>> s.lower()                # 모든 영문자를 소문자로 반환한다.
'right way to 파이썬 programming'
>>> s.upper()                # 모든 영문자를 대문자로 반환한다.
'RIGHT WAY TO 파이썬 PROGRAMMING'
```