파이썬 - 형식 지정 출력

목 차

장식적인 출력 포매팅

https://docs.python.org/ko/3/tutorial/inputoutput.html?highlight=input%20output

지금까지 우리는 값을 쓰는 두 가지 방법을 만났습니다: *표현식 문장* 과 `print()` 함수입니다. (세 번째 방법은 파일 객체의 `write()` 메서드를 사용하는 것입니다; 표준 출력 파일은 `sys.stdout` 로 참조할 수 있습니다. 이것에 대한 자세한 정보는 라이브러리 레퍼런스를 보세요.)

종종 단순히 스페이스로 구분된 값을 인쇄하는 것보다 출력 형식을 더 많이 제어해야 하는 경우가 있습니다. 출력을 포맷하는 데는 여러 가지 방법이 있습니다.

- 포맷 문자열 리터럴을 사용하려면, 시작 인용 부호 또는 삼중 인용 부호 앞에 `f` 또는 `F` 를 붙여 문자열을 시작하십시오. 이 문자열 안에서, `{` 및 `}` 문자 사이에, 변수 또는 리터럴 값을 참조할 수 있는 파이썬 표현식을 작성할 수 있습니다.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- 문자열의 `str.format()` 메서드는 더 많은 수작업을 요구합니다. 변수가 대체 될 위치를 표시하기 위해 `{` 및 `}`를 여전히 사용하고, 자세한 포매팅 디렉티브를 제공할 수 있지만, 포맷할 정보도 제공해야 합니다.

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

- 마지막으로, 문자열 슬라이싱 및 이어붙이기 연산을 사용하여 상상할 수 있는 모든 배치를 만듦으로써, 모든 문자열 처리를 스스로 수행할 수 있습니다. 문자열형에는 주어진 열 너비로 문자열을 채우는 데 유용한 연산을 수행하는 몇 가지 메서드가 있습니다.

장식적인 출력이 필요하지 않고 단지 디버깅을 위해 일부 변수를 빠르게 표시하려면, `repr()` 또는 `str()` 함수를 사용하여 모든 값을 문자열로 변환할 수 있습니다.

`str()` 함수는 어느 정도 사람이 읽기에 적합한 형태로 값의 표현을 돌려주게 되어있습니다. 반면에 `repr()` 은 인터프리터에 의해 읽힐 수 있는 형태를 만들게 되어있습니다 (또는 그렇게 표현할 수 있는 문법이 없으면 `SyntaxError` 를 일으키도록 구성됩니다). 사람이 소비하기 위한 특별한 표현이 없는 객체의 경우, `str()` 는 `repr()` 과 같은 값을 돌려줍니다. 많은 값, 숫자들이나 리스트와 딕셔너리와 같은 구조들, 은 두 함수를 쓸 때 같은 표현을 합니다. 특별히, 문자열은 두 가지 표현을 합니다.

몇 가지 예를 듭니다:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

`string` 모듈에는 문자열에 값을 치환하는 또 다른 방법을 제공하는 `Template` 클래스가 포함되어 있습니다. `$x` 와 같은 자리 표시자를 사용하고 이것들을 딕셔너리에서 오는 값으로 치환하지만, 포매팅에 대한 제어를 훨씬 덜 제공합니다.

1. 포맷 문자열 리터럴

포맷 문자열 리터럴(간단히 f-문자열이라고도 합니다)은 문자열에 `f` 또는 `F` 접두어를 붙이고 표현식을 `{expression}`로 작성하여 문자열에 파이썬 표현식의 값을 삽입할 수 있게 합니다.

선택적인 포맷 지정자가 표현식 뒤에 올 수 있습니다. 이것으로 값이 포맷되는 방식을 더 정교하게 제어할 수 있습니다. 다음 예는 원주율을 소수점 이하 세 자리로 반올림합니다.

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

`':'` 뒤에 정수를 전달하면 해당 필드의 최소 문자 폭이 됩니다. 열을 줄 맞춤할 때 편리합니다.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

다른 수정자를 사용하면 포맷되기 전에 값을 변환할 수 있습니다. `'!a'`는 `ascii()`를, `'!s'`는 `str()`을, `'!r'`는 `repr()`을 적용합니다.:

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

이러한 포맷 사양에 대한 레퍼런스는 포맷 명세 미니 언어에 대한 레퍼런스 지침서를 참조하십시오.

2. 문자열 format() 메서드

`str.format()` 메서드의 기본적인 사용법은 이런 식입니다:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

중괄호와 그 안에 있는 문자들 (포맷 필드라고 부른다) 은 `str.format()` 메서드로 전달된 객체들로 치환됩니다. 중괄호 안의 숫자는 `str.format()` 메서드로 전달된 객체들의 위치를 가리키는데 사용될 수 있습니다.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
```

```
eggs and spam
```

str.format() 메서드에 키워드 인자가 사용되면, 그 값들은 인자의 이름을 사용해서 지정할 수 있습니다.

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

위치와 키워드 인자를 자유롭게 조합할 수 있습니다:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
                                                       other='Georg'))
The story of Bill, Manfred, and Georg.
```

나누고 싶지 않은 정말 긴 포맷 문자열이 있을 때, 포맷할 변수들을 위치 대신에 이름으로 지정할 수 있다면 좋을 것입니다. 간단히 딕셔너리를 넘기고 키를 액세스하는데 대괄호 '[]' 를 사용하면 됩니다.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

〈**〉 표기법을 사용해서 table 을 키워드 인자로 전달해도 같은 결과를 얻을 수 있습니다.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

이 방법은 모든 지역 변수들을 담은 딕셔너리를 돌려주는 내장 함수 vars() 와 함께 사용할 때 특히 쓸모가 있습니다.

예를 들어, 다음 줄은 정수와 그 제곱과 세제곱을 제공하는 빽빽하게 정렬된 열 집합을 생성합니다:

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

str.format() 를 사용한 문자열 포매팅의 완전한 개요는 포맷 문자열 문법 을 보세요.

3. 예전의 문자열 포매팅

% 연산자(모듈로)는 문자열 포매팅에도 사용할 수 있습니다. `'string' % values` 가 주어지면, `string` 에 있는 `%` 인스턴스는 0 개 이상의 `values` 요소로 대체됩니다. 이 연산을 흔히 문자열 보간(interpolation)이라고 합니다. 예를 들면:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

더 자세한 내용은 printf 스타일 문자열 포매팅 섹션에 나옵니다.

부록-1. 포맷 문자열 리터럴

https://docs.python.org/ko/3/reference/lexical_analysis.html#f-strings

버전 3.6 에 추가.

포맷 문자열 리터럴(*formatted string literal*) 또는 *f-문자열* *(f-string)* 은 `'f'` 나 `'F'` 를 앞에 붙인 문자열 리터럴입니다. 이 문자열은 치환 필드를 포함할 수 있는데, 중괄호 `{}` 로 구분되는 표현식입니다. 다른 문자열 리터럴이 항상 상숫값을 갖지만, 포맷 문자열 리터럴은 실행시간에 계산되는 표현식입니다.

이스케이프 시퀀스는 일반 문자열 리터럴처럼 디코딩됩니다 (동시에 날 문자열인 경우는 예외입니다). 디코딩 후에 문자열의 내용은 다음과 같은 문법을 따릅니다:

```
f_string          ::=  (literal_char | "{{" | "}}" | replacement_field)*
replacement_field ::=  "{" f_expression ["="] ["!" conversion] [":" format_spec] "}"
f_expression      ::=  (conditional_expression | "*" or_expr)
                       ("," conditional_expression | "," "*" or_expr)* [","]
                     | yield_expression
conversion        ::=  "s" | "r" | "a"
format_spec       ::=  (literal_char | NULL | replacement_field)*
literal_char      ::=  <any code point except "{", "}" or NULL>
```

중괄호 바깥 부분은 일반 리터럴처럼 취급되는데, 이중 중괄호 `'{{'` 나 `'}}'` 가 대응하는 단일 중괄호로 치환된다는 점만 예외입니다. 하나의 여는 중괄호 `'{'` 는 치환 필드를 시작시키는데, 파이썬 표현식이 뒤따릅니다. 평가 후 표현식 텍스트와 해당 값을 모두 표시하려면 (디버깅에 유용합니다), 표현식 뒤에 등호 `` 〈=〉 \를 추가할 수 있습니다. 느낌표 ``'!'` 로 시작하는, 변환(conversion) 필드가 뒤따를 수 있습니다. 포맷 지정자(format specifier)도 덧붙일 수 있는데, 콜론 `':'` 으로 시작합니다. 치환 필드는 닫는 중괄호 `'}'` 로 끝납니다.

포맷 문자열 리터럴의 표현식은 괄호로 둘러싸인 일반적인 파이썬 표현식으로 취급되는데, 몇 가지 예외가 있습니다. 빈 표현식은 허락되지 않고, `lambda` 와 대입 표현식 `:=` 은 모두 명시적인 괄호로 둘러싸야 합니다. 치환 표현식은 개행문자를 포함할 수 있으나 (예를 들어, 삼중 따옴표 된 문자열) 주석은 포함할 수 없습니다. 각 표현식은 포맷 문자열 리터럴이 등장한 지점의 문맥에서 왼쪽에서 오른쪽으로 계산됩니다.

버전 3.7 에서 변경: 파이썬 3.7 이전에는, 구현 문제로 인해 포맷 문자열 리터럴의 표현식에서 `await` 표현식과 `async for` 절을 포함하는 컴프리헨션은 유효하지 않았습니다.

등호 기호 `'='`이 제공되면, 출력에는 표현식 텍스트, `'='` 및 평가된 값이 포함됩니다. 여는 중괄호 `'{'` 뒤, 표현식 내, `'='` 뒤의 스페이스는 모두 출력에 유지됩니다. 기본적으로, `'='`은 포맷 지정자가 없는 한 표현식의 `repr()`을 제공합니다. 포맷이 지정되면 변환 `'!r'`이 선언되지 않는 한 기본적으로 표현식의 `str()`이 사용됩니다.

버전 3.8 에 추가: 등호 기호 `'='`.

변환(conversion)이 지정되면, 표현식의 결과가 포매팅 전에 변환됩니다. 변환 `'!s'` 는 결과에 `str()` 을 호출하고, `'!r'` 은 `repr()` 을 호출하고, `'!a'` 은 `ascii()` 를 호출합니다.

결과는 `format()` 프로토콜로 포매팅합니다. 포맷 지정자는 표현식이나 변환 결과의 `__format__()` 메서드 로 전달됩니다. 포맷 지정자가 생략되면 빈 문자열이 전달됩니다. 이제 포맷된 결과가 최종 문자열에 삽입됩니다.

최상위 포맷 지정자는 중첩된 치환 필드들을 포함할 수 있습니다. 이 중첩된 필드들은 그들 자신의 변환 필드와 포맷 지정자를 포함할 수 있지만, 깊이 중첩된 치환 필드들을 포함할 수는 없습니다. 포맷 지정자 간이 언어는 `str.format()` 메서드에서 사용되는 것과 같습니다.

포맷 문자열 리터럴을 이어붙일 수는 있지만, 치환 필드가 여러 리터럴로 쪼개질 수는 없습니다.

포맷 문자열 리터럴의 예를 들면:

```
>>> name = "Fred"
>>> f"He said his name is {name!r}."
"He said his name is 'Fred'."
>>> f"He said his name is {repr(name)}."  # repr() is equivalent to !r
"He said his name is 'Fred'."
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
>>> today = datetime(year=2017, month=1, day=27)
>>> f"{today:%B %d, %Y}"  # using date format specifier
'January 27, 2017'
>>> f"{today=:%B %d, %Y}" # using date format specifier and debugging
'today=January 27, 2017'
>>> number = 1024
>>> f"{number:#0x}"  # using integer format specifier
'0x400'
>>> foo = "bar"
>>> f"{ foo = }" # preserves whitespace
" foo = 'bar'"
>>> line = "The mill's closed"
>>> f"{line = }"
'line = "The mill\'s closed"'
>>> f"{line = :20}"
"line = The mill's closed   "
>>> f"{line = !r:20}"
'line = "The mill\'s closed" '
```

일반적인 문자열 리터럴과 같은 문법을 공유하는 것으로 인한 결과는 치환 필드에 사용되는 문자들이 포맷 문자열 리터럴을 감싸는 따옴표와 충돌하지 않아야 한다는 것입니다:

```
f"abc {a["x"]} def"    # error: outer string literal ended prematurely
f"abc {a['x']} def"    # workaround: use different quoting
```

포맷 표현식에는 역 슬래시를 사용할 수 없고, 사용하면 에러가 발생합니다:

```
f"newline: {ord('\n')}"  # raises SyntaxError
```

역 슬래시 이스케이프가 필요한 값을 포함하려면, 임시 변수를 만들면 됩니다.

```
>>> newline = ord('\n')
>>> f"newline: {newline}"
'newline: 10'
```

포맷 문자열 리터럴은 독스트링(docstring)으로 사용될 수 없습니다. 표현식이 전혀 없더라도 마찬가집니다.

```
>>> def foo():
...     f"Not a docstring"
...
>>> foo.__doc__ is None
True
```

포맷 문자열 리터럴 추가에 대한 제안은 **PEP 498** 을 참조하고, 관련된 포맷 문자열 메커니즘을 사용하는 `str.format()` 도 살펴보는 것이 좋습니다.

부록-2. 문자열 format 메서드

https://docs.python.org/ko/3/library/stdtypes.html#str.format

`str.format(*args, **kwargs)`

문자열 포맷 연산을 수행합니다. 이 메서드가 호출되는 문자열은 리터럴 텍스트나 중괄호 `{}` 로 구분된 치환 필드를 포함할 수 있습니다. 각 치환 필드는 위치 인자의 숫자 인덱스나 키워드 인자의 이름을 가질 수 있습니다. 각 치환 필드를 해당 인자의 문자열 값으로 치환한 문자열의 사본을 돌려줍니다.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

포맷 문자열에 지정할 수 있는 다양한 포맷 옵션에 대한 설명은 포맷 문자열 문법 을 참조하십시오.

참고

숫자(int, float, complex, decimal.Decimal 와 서브 클래스)를 n 형식으로 포매팅할 때 (예: '{:n}'.format(1234)), 이 함수는 일시적으로 LC_CTYPE 로케일을 LC_NUMERIC 로케일로 설정하여 localeconv() 의 decimal_point 와 thousands_sep 필드를 디코드하는데, 이 필드들이 ASCII 가 아니거나 1 바이트보다 길고, LC_NUMERIC 로케일이 LC_CTYPE 로케일과 다를 때만 그렇게 합니다. 이 임시 변경은 다른 스레드에 영향을 줍니다.

버전 3.7 에서 변경: 숫자를 n 형식으로 포매팅할 때, 이 함수는 어떤 경우에 일시적으로 LC_CTYPE 로케일을 LC_NUMERIC 로케일로 설정합니다.

str.**format_map**(*mapping*)

str.format(**mapping) 과 비슷하지만, dict 로 복사되지 않고 mapping 을 직접 사용합니다. 예를 들어 mapping 이 dict 서브 클래스면 유용합니다:

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

버전 3.2 에 추가.

부록-3. 포맷 명세 미니 언어

https://docs.python.org/ko/3/library/string.html#formatspec

《포맷 명세》 는 포맷 문자열에 포함된 치환 필드 내에서 개별 값의 표시 방법을 정의하는 데 사용됩니다 (포맷 문자열 문법 과 포맷 문자열 리터럴을 보세요). 이것들은 내장 format() 함수에 직접 전달될 수도 있습니다. 각 포맷 가능한 형은 포맷 명세를 해석하는 방법을 정의 할 수 있습니다.

대부분의 내장형은 포맷 명세에 대해 다음 옵션을 구현하지만, 일부 포맷 옵션은 숫자 형에서만 지원됩니다.

일반적인 관례는 빈 포맷 명세가 값에 str() 을 호출한 것과 같은 결과를 만드는 것입니다. 비어 있지 않은 포맷 명세는 보통 결과를 수정합니다.

표준 포맷 지정자 의 일반적인 형식은 다음과 같습니다:

```
format_spec     ::=
[[fill]align][sign][#][0][width][grouping_option][.precision][type]
fill            ::=  <any character>
align           ::=  "<" | ">" | "=" | "^"
sign            ::=  "+" | "-" | " "
width           ::=  digit+
grouping_option ::=  "_" | ","
precision       ::=  digit+
```

```
type          ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" |
"o" | "s" | "x" | "X" | "%"
```

유효한 *align* 값이 지정되면, *fill* 문자가 앞에 나올 수 있는데 임의의 문자가 될 수 있고, 생략된 경우에는 스페이스가 기본값으로 사용됩니다. 포맷 문자열 리터럴 에서나 `str.format()` 메서드를 사용할 때는, 리터럴 중괄호(《`{`》 또는 《`}`》)를 *fill* 문자로 사용할 수 없습니다. 그러나, 중첩된 치환 필드로 중괄호를 삽입 할 수 있습니다. 이 제한은 `format()` 함수에는 영향을 미치지 않습니다.

다양한 정렬 옵션의 의미는 다음과 같습니다:

옵션	의미
`'<'`	사용 가능한 공간 내에서 필드가 왼쪽 정렬되도록 합니다 (대부분 객체에서 이것이 기본값입니다).
`'>'`	사용 가능한 공간 내에서 필드가 오른쪽 정렬되도록 합니다 (숫자에서 이것이 기본값입니다).
`'='`	채움이 부호 (있다면) 뒤에, 숫자 앞에 오도록 강제합니다. 이것은 〈+000000120〉 형식으로 필드를 인쇄하는 데 사용됩니다. 이 정렬 옵션은 숫자 형에게만 유효합니다. 이것은 필드 너비 바로 앞에 〈0〉 이 있으면 기본값이 됩니다.
`'^'`	사용 가능한 공간 내에서 필드를 가운데에 배치합니다.

최소 필드 너비가 정의되지 않으면, 필드 너비는 항상 필드를 채울 데이터와 같은 크기이므로, 정렬 옵션은 이 경우 의미가 없습니다.

sign 옵션은 숫자 형에게만 유효하며, 다음 중 하나일 수 있습니다:

옵션	의미
`'+'`	음수뿐만 아니라 양수에도 부호를 사용해야 함을 나타냅니다.
`'-'`	음수에 대해서만 부호를 사용해야 함을 나타냅니다 (이것이 기본 동작입니다).
스페이스	양수에는 선행 스페이스를 사용하고, 음수에는 마이너스 부호를 사용해야 함을 나타냅니다.

`'#'` 옵션은 변환에 《대안 형식》 이 사용되도록 만듭니다. 대안 형식은 형별로 다르게 정의됩니다. 이 옵션은 정수, 실수, 복소수와 Decimal 형에게만 유효합니다. 정수의 경우, 이진수, 8 진수 또는 16 진수 출력이 사용될 때, 이 옵션은 출력값에 각각 접두사 `'0b'`, `'0o'` 또는 `'0x'` 를 추가합니다. 실수, 복소수 및 Decimal 의 경우, 대안 형식은 변환 결과의 소수점 아래 숫자가 없어도 항상 소수점 문자가 포함되게 합니다. 보통은, 소수점 문자는 그 뒤에 숫자가 있는 경우에만 변환 결과에 나타납니다. 이에 더해, `'g'` 및 `'G'` 변환의 경우 끝에 붙는 0 이 결과에서 제거되지 않습니다.

`','` 옵션은 천 단위 구분 기호에 쉼표를 사용하도록 알립니다. 로케일을 고려하는 구분자의 경우, 대신 `'n'` 정수 표시 유형을 사용하십시오.

버전 3.1 에서 변경: `','` 옵션을 추가했습니다 (**PEP 378** 도 보세요).

`'_'` 옵션은 부동 소수점 표시 유형 및 정수 표시 유형 `'d'` 에 대해 천 단위 구분 기호에 밑줄을 사용하도록 알립니다. 정수 표시 유형 `'b'`, `'o'`, `'x'` 및 `'X'` 의 경우 밑줄이 4 자리마다 삽입됩니다. 다른 표시 유형의 경우, 이 옵션을 지정하면 에러가 발생합니다.

버전 3.6 에서 변경: `'_'` 옵션을 추가했습니다 (**PEP 515** 도 보세요).

width 는 최소 총 필드 너비를 정의하는 십진 정수인데, 접두사, 구분자 및 다른 포매팅 문자들을 포함합니다. 지정하지 않으면, 필드 너비는 내용에 의해 결정됩니다.

명시적 정렬이 주어지지 않을 때, *width* 필드 앞에 `'0'` 문자를 붙이면 숫자 형에 대해 부호를 고려하는 0 채움을 사용할 수 있습니다. 이것은 `'0'` 의 *fill* 문자와 `'='` 의 *alignment* 유형을 갖는 것과 동등합니다.

precision 는 `'f'` 및 `'F'` 로 포맷된 부동 소수점 값의 소수점 이하 또는 `'g'` 또는 `'G'` 로 포맷된 부동 소수점 값의 소수점 앞, 뒤로 표시할 숫자의 개수를 나타내는 십진수입니다. 숫자가 아닌 유형의 경우 필드는 최대 필드 크기를 나타냅니다 - 즉, 필드 내용에서 몇 개의 문자가 사용되는지 나타냅니다. 정숫값에는 *precision* 이 허용되지 않습니다.

마지막으로 *type* 은 데이터를 표시하는 방법을 결정합니다.

사용 가능한 문자열 표시 유형은 다음과 같습니다:

유형	의미
`'s'`	문자열 포맷. 이것은 문자열의 기본 유형이고 생략될 수 있습니다.
없음	`'s'` 와 같습니다.

사용 가능한 정수 표시 유형은 다음과 같습니다:

유형	의미
`'b'`	이진 형식. 이진법으로 숫자를 출력합니다.
`'c'`	문자. 인쇄하기 전에 정수를 해당 유니코드 문자로 변환합니다.
`'d'`	십진 정수. 십진법으로 숫자를 출력합니다.
`'o'`	8 진 형식. 8 진법으로 숫자를 출력합니다.
`'x'`	16 진 형식. 9 보다 큰 숫자의 경우 소문자를 사용하여 16 진법으로 숫자를 출력합니다.
`'X'`	16 진 형식. 9 보다 큰 숫자의 경우 대문자를 사용하여 16 진법으로 숫자를 출력합니다.
`'n'`	숫자. 이는 현재 로케일 설정을 사용하여 적절한 숫자 구분 문자를 삽입한다는 점을 제외하고는 `'d'` 와 같습니다.
없음	`'d'` 와 같습니다.

위의 표시 유형에 더해, 정수는 아래에 나열된 부동 소수점 표시 유형으로 포맷될 수 있습니다 (`'n'` 및 없음 제외). 그렇게 할 때, 포매팅 전에 정수를 부동 소수점 숫자로 변환하기 위해 `float()` 가 사용됩니다.

부동 소수점 및 Decimal 값에 사용할 수 있는 표시 유형은 다음과 같습니다:

유형	의미
`'e'`	지수 표기법. 지수를 나타내는 문자 〈e'를 사용하여 과학 표기법으로 숫자를 인쇄합니다. 기본 정밀도는 `6` 입니다.
`'E'`	지수 표기법. 구분 문자로 대문자 〈E'를 사용한다는 것을 제외하고 `'e'` 와 같습니다.

유형	의미
`'f'`	고정 소수점 표기법. 숫자를 고정 소수점 숫자로 표시합니다. 기본 정밀도는 `6` 입니다.
`'F'`	고정 소수점 표기법. `'f'` 와 같지만, `nan` 을 `NAN` 으로, `inf` 를 `INF` 로 변환합니다.
`'g'`	범용 형식. 주어진 정밀도 `p >= 1` 에 대해, 숫자를 유효 숫자 `p` 로 자리 올림 한 다음, 결과를 크기에 따라 고정 소수점 형식이나 과학 표기법으로 포맷합니다. 정확한 규칙은 다음과 같습니다: 표시 유형 `'e'` 와 정밀도 `p-1` 로 포맷된 결과의 지수가 `exp` 라고 가정하십시오. 이때 `-m <= exp < p` 이면 (여기서 `m` 은 float 에서 -4 이고 `Decimal` 이면 -6 입니다), 숫자는 표시 형식 `'f'` 와 정밀도 `p-1-exp` 로 포맷됩니다. 그렇지 않으면, 숫자는 표시 유형 `'e'` 와 정밀도 `p-1` 로 포맷됩니다. 두 경우 유효하지 않은 후행 0 은 모두 유효숫자부에서 제거되고, 뒤에 남아있는 숫자가 없다면 `'#'` 옵션이 사용되지 않는 한 소수점도 제거됩니다. 양과 음의 무한대, 양과 음의 0, nans 는 정밀도와 무관하게 각각 `inf`, `-inf`, `0`, `-0`, `nan` 으로 포맷됩니다. `0` 의 정밀도는 `1` 의 정밀도로 처리됩니다. 기본 정밀도는 `6` 입니다.
`'G'`	범용 형식. 숫자가 너무 커지면 `'E'` 로 전환하는 것을 제외하고 `'g'` 와 같습니다. 무한과 NaN 의 표현도 대문자로 바뀝니다.
`'n'`	숫자. 현재 로케일 설정을 사용하여 적절한 숫자 구분 문자를 삽입한다는 점을 제외하면 `'g'` 와 같습니다.
`'%'`	백분율. 숫자에 100 을 곱해서 고정 (`'f'`) 형식으로 표시한 다음 백분율 기호를 붙입니다.
없음	고정 소수점 표기법이 선택될 때, 소수점 이하로 적어도 하나의 자리가 있다는 점을 제외하면 `'g'` 와 비슷합니다. 기본 정밀도는 특정 값을 나타내는 데 필요한 만큼 높습니다. 전체적인 효과는 `str()` 의 출력을 다른 포맷 수정자에 의해 변경된 것처럼 만드는 것입니다.

포맷 예제

이 절은 `str.format()` 문법의 예와 예전 `%`-포매팅과의 비교를 포함합니다.

대부분은 문법이 예전의 `%`-포매팅과 유사하며, `{}` 가 추가되고 `%` 대신 및 `:` 이 사용됩니다. 예를 들어, `'%03.2f'` 는 `'{:03.2f}'` 로 번역될 수 있습니다.

새 포맷 문법은 다음 예제에 보이는 것과 같이 새롭고 다양한 옵션도 지원합니다.

위치로 인자 액세스:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'abracadabra'
```

이름으로 인자 액세스:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-
115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

인자의 어트리뷰트 액세스:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary
part -5.0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

인자의 항목 액세스:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

`%s` 과 `%r` 대체:

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

텍스트 정렬과 너비 지정:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

`%+f`, `%-f`, `% f` 대체와 부호 지정:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
```

```
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

`%x`, `%o` 대체와 다른 진법으로 값 변환:

```
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

쉼표를 천 단위 구분자로 사용:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

백분율 표현:

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

형별 포매팅 사용:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

인자 중첩과 보다 복잡한 예제:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end='
')
...     print()
...
```

```
 5    5    5   101
 6    6    6   110
 7    7    7   111
 8    8   10  1000
 9    9   11  1001
10    A   12  1010
11    B   13  1011
```

부록-4. printf 스타일 문자열 포매팅

https://docs.python.org/ko/3/library/stdtypes.html#old-string-formatting

참고

여기에 설명된 포맷 연산은 여러 가지 일반적인 오류를 (예를 들어 튜플과 딕셔너리를 올바르게 표시하지 못하는 것) 유발하는 다양한 문제점들이 있습니다. 새 포맷 문자열 리터럴 나 `str.format()` 인터페이스 혹은 템플릿 문자열 을 사용하면 이러한 오류를 피할 수 있습니다. 이 대안들은 또한 텍스트 포매팅에 더욱 강력하고 유연하며 확장 가능한 접근법을 제공합니다.

문자열 객체는 한가지 고유한 내장 연산을 갖고 있습니다: `%` 연산자 (모듈로). 이것은 문자열 *포매팅* 또는 *치환* 연산자라고도 합니다. `format % values` 가 주어질 때 (*format* 은 문자열입니다), *format* 내부의 `%` 변환 명세는 0 개 이상의 *values* 의 요소로 대체됩니다. 이 효과는 C 언어에서 `sprintf()` 를 사용하는 것과 비슷합니다.

format 이 하나의 인자를 요구하면, *values* 는 하나의 비 튜플 객체 일 수 있습니다. 5 그렇지 않으면, *values* 는 format 문자열이 지정하는 항목의 수와 같은 튜플이거나 단일 매핑 객체 (예를 들어, 딕셔너리) 이어야 합니다.

변환 명세는 두 개 이상의 문자를 포함하며 다음과 같은 구성 요소들을 포함하는데, 반드시 이 순서대로 나와야 합니다:

1. `'%'` 문자: 명세의 시작을 나타냅니다.
2. 매핑 키 (선택 사항): 괄호로 둘러싸인 문자들의 시퀀스로 구성됩니다 (예를 들어, `(somename)`).
3. 변환 플래그 (선택 사항): 일부 변환 유형의 결과에 영향을 줍니다.
4. 최소 필드 폭 (선택 사항): `'*'` (애스터리스크) 로 지정하면, 실제 폭은 *values* 튜플의 다음 요소에서 읽히고, 변환할 객체는 최소 필드 폭과 선택적 정밀도 뒤에 옵니다.
5. 정밀도 (선택 사항): `'.'` (점) 다음에 정밀도가 옵니다. `'*'` (애스터리스크) 로 지정하면, 실제 정밀도는 *values* 튜플의 다음 요소에서 읽히고, 변환할 값은 정밀도 뒤에 옵니다.
6. 길이 수정자 (선택 사항).
7. 변환 유형.

오른쪽 인자가 딕셔너리 (또는 다른 매핑 형) 인 경우, 문자열에 있는 변환 명세는 *반드시* `'%'` 문자 바로 뒤에 그 딕셔너리의 매핑 키를 괄호로 둘러싼 형태로 포함해야 합니다. 매핑 키는 포맷할 값을 매핑으로부터 선택합니다. 예를 들어:

```
>>> print('%(language)s has %(number)03d quote types.' %
...       {'language': "Python", "number": 2})
Python has 002 quote types.
```

이 경우 `*` 지정자를 사용할 수 없습니다 (순차적인 매개변수 목록이 필요하기 때문입니다).

변환 플래그 문자는 다음과 같습니다:

플래그	뜻
`'#'`	값 변환에 《대체 형식》 (아래에 정의되어있습니다) 을 사용합니다.
`'0'`	변환은 숫자 값의 경우 0 으로 채웁니다.
`'-'`	변환된 값은 왼쪽으로 정렬됩니다 (둘 다 주어지면 `'0'` 변환보다 우선 합니다).
`' '`	(스페이스) 부호 있는 변환 때문에 만들어진 양수 앞에 빈칸을 남겨둡니다 (음수면 빈 문자열입니다).
`'+'`	부호 문자 (`'+'` or `'-'`) 가 변환 앞에 놓입니다 (`' '` 플래그에 우선합니다).

길이 수정자 (`h`, `l`, `L`) 를 제공할 수는 있지만, 파이썬에서 필요하지 않기 때문에 무시됩니다 – 예를 들어 `%ld` 는 `%d` 와 같습니다.

변환 유형은 다음과 같습니다:

변환	뜻	노트
`'d'`	부호 있는 정수 십진 표기.	
`'i'`	부호 있는 정수 십진 표기.	
`'o'`	부호 있는 8 진수 값.	(1)
`'u'`	쓸데없는 유형 – `'d'` 와 같습니다.	(6)
`'x'`	부호 있는 16 진수 (소문자).	(2)
`'X'`	부호 있는 16 진수 (대문자).	(2)
`'e'`	부동 소수점 지수 형식 (소문자).	(3)
`'E'`	부동 소수점 지수 형식 (대문자).	(3)
`'f'`	부동 소수점 십진수 형식.	(3)
`'F'`	부동 소수점 십진수 형식.	(3)
`'g'`	부동 소수점 형식. 지수가 -4 보다 작거나 정밀도 보다 작지 않으면 소문자 지수형식을 사용하고, 그렇지 않으면 십진수 형식을 사용합니다.	(4)
`'G'`	부동 소수점 형식. 지수가 -4 보다 작거나 정밀도 보다 작지 않으면 대문자 지수형식을 사용하고, 그렇지 않으면 십진수 형식을 사용합니다.	(4)
`'c'`	단일 문자 (정수 또는 길이 1 인 문자열을 허용합니다).	
`'r'`	문자열 (`repr()`을 사용하여 파이썬 객체를 변환합니다).	(5)
`'s'`	문자열 (`str()` 을 사용하여 파이썬 객체를 변환합니다).	(5)
`'a'`	문자열 (`ascii()` 를 사용하여 파이썬 객체를 변환합니다).	(5)
`'%'`	인자는 변환되지 않고, 결과에 `'%'` 문자가 표시됩니다.	

노트:

1. 대체 형식은 첫 번째 숫자 앞에 선행 8 진수 지정자 (`'0o'`)를 삽입합니다.
2. 대체 형식은 첫 번째 숫자 앞에 선행 `'0x'` 또는 `'0X'` (`'x'` 나 `'X'` 유형 중 어느 것을 사용하느냐에 따라 달라집니다) 를 삽입합니다.
3. 대체 형식은 그 뒤에 숫자가 나오지 않더라도 항상 소수점을 포함합니다.

   정밀도는 소수점 이하 자릿수를 결정하며 기본값은 6 입니다.
4. 대체 형식은 결과에 항상 소수점을 포함하고 뒤에 오는 0 은 제거되지 않습니다.

   정밀도는 소수점 앞뒤의 유효 자릿수를 결정하며 기본값은 6 입니다.
5. 정밀도가 `N` 이라면, 출력은 `N` 문자로 잘립니다.
6. **PEP 237** 을 참조하세요.

파이썬 문자열은 명시적인 길이를 가지고 있으므로, `%s` 변환은 문자열의 끝이 `'\0'` 이라고 가정하지 않습니다.

버전 3.1 에서 변경: 절댓값이 1e50 을 넘는 숫자에 대한 `%f` 변환은 더는 `%g` 변환으로 대체되지 않습니다.