4 format( ) 메소드를 활용한 문자열 서식 설정

이 부록에서는 문자열의 format() 메소드에서 제공하는 문자열 서식 설정 중 자주 사용하는 기능들을 소개한다. format() 메소드의 일반적인 내용은 본문에서 다루었다. 여기서는 본문에서 다루지 않은 내용을 위주로 설명하겠다.

본문에서는 문자열 내부에 위치한 대체 필드인 '{}'가 format() 메소드의 전달인자를 대체하여 새로운 문자열을 반환하는 역할을 하는 것을 살펴보았다. 대체 필드를 설정할 때는 '{}' 안에 '필드명'을 입력하는데, 이때 '필드명'에는 전달하려는 위치 전달인자(positional argument)의 인덱스 번호나 키워드 전달인자(keyword argument)의 이름 중 하나를 선택해서 입력하면 된다. 본문에서는 위치 전달인자를 사용한 예만 설명하였다. 여기서는 문자열 서식을 설정하기 위해 대체 필드를 사용하는 방법을 상세히 알아보자.

대체 필드는 다음과 같은 형식으로 작성할 수 있다.

```
{필드명}
{필드명!형식변환}
{필드명:서식명세}
{필드명!형식변환:서식명세}
```

대체 필드 '{}' 안에 들어가는 필드명, 형식변환, 서식명세에 대해 좀 더 자세히 알아보자.

1. 필드명

대체 필드 '{}' 안에 들어가는 필드 이름을 정하는 형식은 다음과 같다.

{필드명}

- 필드명으로는 인덱스나 키워드 식별자가 온다.
- 필드명으로 인덱스를 사용하는 경우
  - format() 메소드의 전달인자로 위치 전달인자가 올 경우에는 대체 필드인 '{}' 안에 위치 전달인자의 위치를 알려주는 인덱스 번호(0, 1, ...)[1]를 필드명으로 사용한다.
  - format() 메소드의 위치 전달인자로 복합자료형이나 클래스가 올 경우에는 다음의 규칙이 적용된다.
    - 리스트, 튜플: {인덱스[인덱스]}

      예) {0[0]}, {1[2]}, {2[5]}, ...
    - 딕셔너리: {인덱스[키]}

      예) {0[name]}, {1[age]}, {2[price]}, ...
    - 클래스: {인덱스[속성이름]}

      예) {0[name]}, {1[grade]}, {2[color]}, ...
- 필드명으로 키워드 식별자를 사용하는 경우
  - format() 메소드의 전달인자로 키워드 전달인자가 올 경우에는 대체 필드인 '{}' 안에 키워드 전달인자의 키워드 식별자를 필드명으로 사용한다.
  - format() 메소드의 키워드 전달인자로 복합자료형이나 클래스가 올 경우에는 다음의 규칙이 적용된다.
    - 리스트, 튜플: {키워드-식별자[인덱스]}

      예) {stock[0]}, {person[2]}, {stationery[2]}, ...
    - 딕셔너리: {키워드-식별자[키]}

      예) {stock[name]}, {person[age]}, {stationery[price]}, ...
    - 클래스: {키워드-식별자[속성이름]}

      예) {student[name]}, {student[grade]}, {cloth[color]}, ...
- format() 메소드 전달인자로는 위치 전달인자가 키워드 전달인자보다 항상 먼저 와야 한다.

1 파이썬 버전 3.1 이후부터는 인덱스 번호를 생략할 수 있다.

다음 예시는 format() 메소드에 위치 전달인자가 왔기 때문에 대체 필드의 필드명으로 인덱스를 사용한 경우이다.

```
>>> # 필드명으로 인덱스 번호를 사용한다.
... '{0} : {1}악기에 속한다.'.format('드럼', '타')
'드럼 : 타악기에 속한다.'
```

이 코드에서는 첫 번째 위치 전달인자인 '드럼'의 값을 대체하는 필드명으로 0이 왔고, '타'의 값을 대체하는 필드명으로 1이 왔다. 파이썬 버전 3.1부터는 위치 전달인자를 사용하면 필드명을 생략할 수 있는데, 이 경우에 파이썬이 문자열 안에 있는 대체 필드의 왼쪽부터 순서대로 위치 전달인자의 값을 대체한다. 다음 예시는 필드명을 생략하고 코드를 실행한 결과를 보여준다.

```
>>> # 위치 전달인자가 오면 필드명을 생략할 수 있다.
... '{} : {}악기에 속한다.'.format('드럼', '타')
'드럼 : 타악기에 속한다.'
```

필드명을 생략하면 코드가 좀 더 간결해 보이는 장점이 있다. 반면에 필드명으로 인덱스를 사용하면 인덱스 번호로 위치 전달인자의 값이 대체된다. 따라서 위치 전달인자의 순서를 바꾸지 않아도 단순히 필드명을 바꿈으로서 결괏값의 순서를 조정할 수 있다. 다음 예시의 두 번째 코드는 첫 번째 코드에서 사용한 전달인자의 순서는 그대로 두고, 필드명만 바꾸어서 문자열을 출력한 결과를 보여준다.

```
>>> '{0}는 {1}를 사랑한다.'.format('나', '너')
'나는 너를 사랑한다.'
>>> '{1}는 {0}를 사랑한다.'.format('나', '너')
'너는 나를 사랑한다.'
```

이번에는 키워드 전달인자를 사용해서 format() 메소드를 호출해보자. 이 경우에는 대체 필드의 필드명으로 키워드 전달인자의 키워드 식별자를 필드명으로 사용해야 한다.

```
>>> '{instrument} : {type}악기에 속한다'.format(type='현', instrument='기타')
'기타 : 현악기에 속한다'
```

키워드 전달인자가 올 경우에는 대체 필드로 대체되는 전달인자의 값이 키워드 식별자로 결정되기 때문에 전달인자의 순서에 영향을 받지 않는다. 다음 예시는 키워드 전달인자의 순서를 바꾸어 코드를 실행했지만 같은 결과가 출력되는 것을 보여준다.

```
>>> '{instrument} : {type}악기에 속한다'.format(instrument='기타', type='현')
'기타 : 현악기에 속한다'
```

필드명으로 인덱스와 키워드 식별자 둘 다 사용할 수 있다. 이 경우에도 인덱스 번호는 생략할 수 있는데, format() 메소드의 전달인자로는 반드시 위치 전달인자가 키워드 전달인자보다 항상 먼저 와야 한다.

```
>>> '{instrument} : {0}악기에 속한다'.format('건반', instrument='키보드')
'키보드 : 건반악기에 속한다'
```

만약 키워드 전달인자가 위치 전달인자보다 먼저 오면 다음처럼 예외가 발생한다.

```
>>> '{instrument} : {0}악기에 속한다'.format(instrument='키보드', '건반')
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

필드명으로 복합자료형이나 클래스를 참조할 수도 있다. 바로 복합자료형이나 클래스가 format() 메소드의 전달인자로 오는 경우이다. 이때 필드명을 어떻게 사용하는지 알아보자. 필드명은 format() 메소드의 전달인자로 위치 전달인자가 오는 경우와 키워드 전달인자가 오는 경우 두 가지로 나눌 수 있다. 먼저 위치 전달인자로 복합자료형이나 클래스가 오는 경우를 살펴본 다음 키워드 전달인자로 복합자료형이나 클래스가 오는 경우를 알아보기로 하자.

1.1. 위치 전달인자로 복합자료형이나 클래스가 오는 경우의 필드명 사용법

1.1.1. 리스트와 튜플

복합자료형 중에서도 리스트나 튜플이 가지고 있는 객체를 참조할 경우 리스트나 튜플의 인덱스 번호를 사용해서 리스트나 튜플의 객체를 추출할 수 있다. 이때 만약 대체 필드와 위치 전달인자의 수가 같으면 다음 두 번째 코드처럼 필드명을 생략할 수 있다.

```
>>> stock = ['펜', '연필', '봉투', '클립', '메모장']                    # 리스트
>>> '현재 재고로 가지고 있는 품목은 {0[1]}입니다.'.format(stock)
'현재 재고로 가지고 있는 품목은 연필입니다.'
>>> # 필드명을 생략할 수 있다.
... '현재 재고로 가지고 있는 품목은 {[1]}입니다.'.format(stock)
'현재 재고로 가지고 있는 품목은 연필입니다.'
```

만약 대체 필드의 수와 위치 전달인자의 수가 일치하지 않는 경우에는, 다음 예시처럼 대체 필드 안의 '필드명'에 위치 전달인자의 인덱스를 반드시 입력해야 한다.

```
>>> stock = ['펜', '연필', '봉투', '클립', '메모장']                    # 리스트
>>> '현재 재고로 가지고 있는 품목은 {0[1]}, {0[4]}입니다.'.format(stock)
'현재 재고로 가지고 있는 품목은 연필, 메모장입니다.'
```

그렇지 않으면 다음과 같이 IndexError가 발생한다.

```
>>> stock = ['펜', '연필', '봉투', '클립', '메모장']                    # 리스트
>>> '현재 재고로 가지고 있는 품목은 {[1]}, {[4]}입니다.'.format(stock)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

대체 필드의 수와 위치 전달인자의 수가 일치하면서 두 개 이상 사용할 경우, 필드명으로 인덱스를 사용하지 않으면 다음 예시처럼 대체 필드의 왼쪽부터 오른쪽 순으로 위치 전달인자의 값이 순서대로 대입이 된다.

```
>>> stock = ['펜', '연필', '볼투', '클립', '메모장']                    # 리스트
>>> instrument = '드럼', '기타', '베이스', '키보드'                     # 튜플
>>> '현재 재고로 가지고 있는 품목은 {[1]}, {[-1]}입니다.'.format(stock, instrument)
'현재 재고로 가지고 있는 품목은 연필, 키보드입니다.'
```

이는 다음과 같이 위치 전달인자의 순서대로 인덱스 번호를 필드명으로 사용한 것과 같은 결과다.

```
>>> '현재 재고로 가지고 있는 품목은 {0[1]}, {1[-1]}입니다.'.format(stock,
instrument)
'현재 재고로 가지고 있는 품목은 연필, 키보드입니다.'
```

필드명으로 인덱스를 생략하지 않고 지정하면, format() 메소드로 전달되는 위치 전달인자의 개수와 상관없이 다음과 같이 사용할 수 있다.

```
>>> '현재 재고로 가지고 있는 품목은 {0[1]}, {0[3]}입니다.'.format(stock, instrument)
'현재 재고로 가지고 있는 품목은 연필, 클립입니다.'
>>> '현재 재고로 가지고 있는 품목은 {1[1]}, {1[-1]}입니다.'.format(stock,
instrument)
'현재 재고로 가지고 있는 품목은 기타, 키보드입니다.'
```

1.1.2. 딕셔너리

딕셔너리의 객체를 참조할 경우에는 딕셔너리의 키(key)를 사용해서 접근이 가능하다. 이때 키의 자료형이 문자열이어도 필드명으로 사용할 때에는 따옴표를 묶어 표현하지 않는다. 리스트, 튜플과 마찬가지로 대체 필드와 위치 전달인자의 수가 같으면 다음 마지막 코드처럼 필드명을 생략할 수 있다.

```
>>> stock = {'타악기': '드럼', '현악기': ['기타', '베이스'], '건반악기': '키보드'}
>>> '현재 재고로 가지고 있는 품목은 {0[현악기]}입니다.'.format(stock)
"현재 재고로 가지고 있는 품목은 ['기타', '베이스']입니다."
>>> # 필드명을 생략할 수 있다.
... '현재 재고로 가지고 있는 품목은 {[현악기]}입니다.'.format(stock)
"현재 재고로 가지고 있는 품목은 ['기타', '베이스']입니다."
```

만약 추출하고자 하는 키의 매핑값이 리스트나 튜플처럼 복합자료형인 경우에는 해당 자료형에서 객체를 추출하는 방식을 사용하면 된다. 다음 예시는 '현악기'의 매핑값이 리스트이기 때문에 리스트의 마지막 객체를 추출하기 위해 리스트의 인덱스를 사용해서 코드를 실행한 결과를 보여준다.

```
>>> stock = {'타악기': '드럼', '현악기': ['기타', '베이스'], '건반악기': '키보드'}
>>> '현재 재고로 가지고 있는 품목은 {[현악기][-1]}입니다.'.format(stock)
'현재 재고로 가지고 있는 품목은 베이스입니다.'
```

딕셔너리의 경우에도 위치 전달인자의 값을 대체할 때, 리스트나 튜플처럼 대체 필드의 수와 위치 전달인자의 수가 일치하지 않으면 대체 필드 안의 '필드명'에 위치 전달인자의 인덱스를 반드시 입력해야 한다.

```
>>> stock = {'타악기': '드럼', '현악기': ['기타', '베이스'], '건반악기': '키보드'}
>>> '현재 재고로 가지고 있는 품목은 {0[건반악기]}, {0[현악기][-1]}입니다.'.format(stock)
'현재 재고로 가지고 있는 품목은 키보드, 베이스입니다.'
```

그렇지 않으면 다음처럼 IndexError가 발생한다.

```
>>> stock = {'타악기': '드럼', '현악기': ['기타', '베이스'], '건반악기': '키보드'}
>>> '현재 재고로 가지고 있는 품목은 {[건반악기]}, {[현악기]}입니다.'.format(stock)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

1.1.3. 클래스

클래스의 속성(attribute)을 참조할 경우에는 클래스의 속성 이름을 사용해서 속성 값을 추출할 수 있다. 먼저 간단한 클래스를 다음과 같이 정의해보자.

```
>>> class myclass:
...     def __init__(self, name, grade):
...         self.name = name
...         self.grade = grade
...
>>>
```

이제 인스턴스를 생성한 후, 대체 필드의 필드명으로 속성 이름을 지정한 후, 인스턴스의 속성 값을 출력해보자. 이때 필드명으로 속성 이름을 지정할 때 점 연산자(.)가 속성 이름 앞에 와야 한다. 그리고 대체 필드와 위치 전달인자의 수가 같으면 다음 마지막 코드처럼 필드명을 생략할 수 있다.

```
>>> ryan = myclass('라이언', 'A+')
>>> '내 성적은 {0.grade}입니다.'.format(ryan)
'내 성적은 A+입니다.'
>>> '내 이름은 {.name}입니다.'.format(ryan)                    # 필드명을 생략할 수 있다.
'내 이름은 라이언입니다.'
```

마찬가지로 대체 필드의 수와 위치 전달인자의 수가 일치하지 않는 경우에는, 대체 필드 안의 '필드명'에 위치 전달인자의 인덱스를 반드시 입력해야 한다.

```
>>> frodo = myclass('프로도', 'P')
>>> '{0.name}의 성적은 {0.grade}입니다.'.format(frodo)
'프로도의 성적은 P입니다.'
```

그렇지 않으면 다음과 같이 IndexError가 발생한다.

```
>>> frodo = myclass('프로도', 'P')
>>> '{.name}의 성적은 {.grade}입니다.'.format(frodo)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

클래스의 속성에 접근할 때 주의할 점이 있다. 만약 클래스를 정의할 때 외부에서 클래스 내부의 속성에 직접 접근하는 것을 어렵게 하기 위해 네임 맹글링(name mangling)을 해서 밑줄 두 개로 시작하는 속성 이름을 정의했다면, 속성에 접근할 수 없다. 다음 예시는 클래스를 정의할 때 네임 맹글링(name mangling)을 하기 위해 속성 이름 앞에 밑줄 두 개를 붙인 후 속성 이름을 선언했다. 네임 맹글링에 대한 설명은 '부록 6 - 클래스와 객체 완전 정복'을 참조하길 바란다.

```
>>> class myclass2:
...     def __init__(self, name, grade):
...         self.__name = name
...         self.__grade = grade
...
>>>
```

네임 맹글링을 한 클래스의 인스턴스에 접근해보자. 다음과 같이 속성을 찾을 수 없다는 예외가 발생한다.

```
>>> ryan = myclass2('라이언', 'A+')
>>> '{0.__name}의 성적은 {0.__grade}입니다.'.format(ryan)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'myclass2' object has no attribute '__name'
```

1.2. 키워드 전달인자로 복합자료형이나 클래스가 오는 경우의 필드명 사용법

이번에는 키워드 전달인자를 사용해서 필드명으로 복합자료형이나 클래스를 참조할 경우에 필드명을 작성하는 방법을 알아보자. 먼저 인덱스로 접근하는 리스트와 튜플을 살펴본 다음, 키로 접근하는 딕셔너리와 클래스의 속성에 접근하는 방법을 알아보도록 하겠다.

1.2.1. 리스트와 튜플

리스트나 튜플이 가지고 있는 객체를 참조할 경우, 앞서 설명한 위치 전달인자를 사용할 때와 큰 차이는 없다. 필드명으로 위치 전달인자의 인덱스 대신에, 다음 예시처럼 키워드 전달인자의 키워드 식별자를 사용할 수 있다.

```
>>> stock = ['펜', '연필', '볼투', '클립', '메모장']                    # 리스트
>>> '재고 품목 : {stationery[1]}'.format(stationery=stock)
'재고 품목 : 연필'
>>> '재고 품목 : {stationery[1]}, {stationery[-1]}'.format(stationery=stock)
'재고 품목 : 연필, 메모장'
```

키워드 전달인자를 사용하면 다음처럼 키워드 전달인자의 순서에 상관없이 사용할 수 있어 편리하다.

```
>>> stock = ['펜', '연필', '볼투', '클립', '메모장']                    # 리스트
>>> instrument = '드럼', '기타', '베이스', '키보드'                     # 튜플
>>> '재고 품목 : {학용품[1]}, {악기[0]}, {학용품[-1]}'.format(
...     학용품=stock, 악기=instrument)
'재고 품목 : 연필, 드럼, 메모장'
```

위의 코드에서는 키워드 식별자로 한글을 사용해보았다. 이처럼 키워드 식별자로 영어와 한글 사용이 모두 가능하지만 일반적으로 한글 사용은 권장하지 않는다.

1.2.2. 딕셔너리

딕셔너리 자료형의 경우는 딕셔너리의 키(key)를 사용해서 접근이 가능한다. 이때 키의 자료형이 문자열이어도 필드명으로 사용할 때에는 키에 따옴표를 묶어 표현하지 않는다.

```
>>> stock = { '타악기': '드럼', '현악기': ['기타', '베이스'], '건반악기': '키보드'}
>>> '재고 품목 : {p[현악기]}'.format(p=stock)
"재고 품목 : ['기타', '베이스']"
>>> '재고 품목 : {p[현악기][-1]}'.format(p=stock)
'재고 품목 : 베이스'
>>> '재고 품목 : {p[건반악기]}, {p[현악기]}'.format(p=stock)
"재고 품목 : 키보드, ['기타', '베이스']"
```

format() 메소드의 전달인자에 매핑형 패킹 및 언패킹 연산자인 **를 적용해서 사용하면 종종 편리할 때가 있다. 예를 들어, 위에서 정의한 stock 앞에 **를 붙여 format()의 전달인자로 사용하면, 다음 예시처럼 문자열 안의 필드명에 따라 딕셔너리의 매핑값을 추출할 수 있다.

```
>>> stock = {'타악기': '드럼', '현악기': ['기타', '베이스'], '건반악기': '키보드'}
>>> '재고 품목 : {타악기}, {건반악기}'.format(**stock)
'재고 품목 : 드럼, 키보드'
```

딕셔너리를 이용하는 대신 locals() 함수를 매핑형 패킹 및 언패킹 연산자인 **와 사용해서 지역 변수의 값에 접근할 수도 있다. locals() 함수는 로컬에서 정의하고 현재 사용 중인 지역 변수들을 딕셔너리 형식으로 관리하고 있기 때문에 이 변수들을 키워드 식별자로 사용할 수도 있다. 다음 예시는 변수를 키워드 식별자로 사용해서 locals()를 통해 해당 값을 언패킹해서 추출한 후 출력한 결과를 보여준다.

```
>>> band = 'Coldplay'
>>> song = 'Yellow'
>>> '{song}는 {band}의 히트곡입니다.'.format(**locals())
'Yellow는 Coldplay의 히트곡입니다.'
```

1.2.3 클래스

키워드 전달인자를 사용할 경우에는 키워드-식별자.속성이름 형식으로 클래스의 속성 값에 접근할 수 있다. 다음 예시는 앞서 사용한 클래스 정의이다.

```
>>> class myclass:
...     def __init__(self, name, grade):
...         self.name = name
...         self.grade = grade
...
>>>
```

이 클래스의 인스턴스를 다음과 같이 생성한 후, 대체 필드의 필드명으로 키워드 식별자를 사용해서 속성 값을 출력해보자.

```
>>> ryan = myclass('라이언', 'A+')
>>> '{student.name}의 성적은 {student.grade}입니다.'.format(student=ryan)
'라이언의 성적은 A+입니다.'
```

2. 출력 형식 변환

파이썬에서 내장된 자료형을 format() 메소드의 전달인자로 사용할 때는 자신을 표현하기 적합한 문자열로 반환한다. 하지만 대체 필드에서 출력 형식 변환자({!})를 사용하면 출력 방식을 원하는 형식으로 변환하는 것이 가능하다. 대체 필드 안에 느낌표(!)를 사용해서 출력 형식 변환자를 지정하는 형식은 다음과 같다.

```
{필드명!형식변환}
{필드명!형식변환:서식명세}
```

- 필드명 뒤에 출력 형식 변환자(conversion specifier)인 !를 추가해주면 된다.
  - 형식 변환자는 필드명과 서식명세 사이에 온다.
- !s : 출력할 내용을 텍스트 형식(string form)으로 변환한다.
- !r : 출력할 내용을 대표 형식(representational form)으로 변환한다.
- !a : 출력할 내용을 아스키 코드만을 사용하는 대표 형식로 변환한다.

모든 파이썬 객체는 두 종류의 출력 형식을 가지고 있다. 첫 번째 출력 형식은 print() 함수를 통해 객체를 출력하는 방식인 텍스트 형식(string form)[2]이다. 텍스트 형식은 사람이 읽기 편한 형식으로 출력하는 방식으로 대화형 모드와 인터프리터 모드 둘 다에서 사용할 수 있다. 만약 해당 객체를 텍스트 형식으로 표현할 수 없는 경우에는 대표 형식으로 출력된다.

두 번째 출력 형식은 print() 함수를 사용하지 않고 변수나 리터럴을 입력한 후 Enter 또는 return 키를 눌러 출력하는 방식인 대표 형식(representational form)이다. 대표 형식이란 해당 객체를 대표하는 형식으로 출력하는 방식을 말하는데 print() 함수를 사용하지 않고 출력할 수 있기 때문에 대화형 모드에서만 사용할 수 있다. 이는 repr() 함수로 객체를 반환한 형식과 같다. 그리고 만약 해당 객체를 대표 형식으로 표현할 수 없으면 꺾쇠 괄호(<>)로 둘러싸인 문자열로 출력된다.

2 자료형인 '문자열'의 의미와 혼동을 피하기 위해 '문자열 형식'으로 번역하지 않고 '텍스트 형식'으로 번역한다. print() 함수를 사용해서 문자열을 출력하면 화이트스페이스 문자 가운데 화면이나 프린터 등과 같은 출력 장치를 제어하는 문자 새줄바꿈이나 탭 문자는 나타내지 않고, 처리된 결과만 텍스트 형식으로 출력하기 때문에 '문자열 형식'보다는 '텍스트 형식'으로 번역하는 것이 좀 더 정확한 번역이라고 간주한다.

예시를 통해 이 둘의 차이를 알아보자. 다음 코드는 분수와 소수를 각각 텍스트 형식과 대표 형식으로 출력한 결과를 보여준다. 각각의 출력 결과를 보면 두 형식의 차이를 쉽게 구분할 수 있다.

```
>>> import fractions, decimal
>>> print(fractions.Fraction(3, 7))                    # 텍스트 형식으로 출력한다.
3/7
>>> fractions.Fraction(3, 7)                           # 대표 형식으로 출력한다.
Fraction(3, 7)
>>> print(decimal.Decimal('123.456789'))               # 텍스트 형식으로 출력한다.
123.456789
>>> decimal.Decimal('123.456789')                      # 대표 형식으로 출력한다.
Decimal('123.456789')
```

앞서 해당 객체를 텍스트 형식으로 표현할 수 없는 경우 대표 형식으로 출력하는데, 대표 형식으로도 표현이 되지 않을 경우에는 꺽쇠 괄호로 둘러싸인 문자열로 출력한다고 설명했다. 다음 예시의 경우 sys 모듈을 출력하면 텍스트 형식과 대표 형식으로 표현되지 않기 때문에 양쪽을 꺽쇠 괄호로 묶어서 sys가 모듈임을 알려주고 있다.

```
>>> import sys
>>> print(sys)      # 텍스트 형식으로 표현할 수 없어 대표 형식으로 출력을 시도한다.
<module 'sys' (built-in)>
>>> sys             # 대표 형식으로도 표현할 수 없어 꺽쇠 괄호로 둘러싸인 문자열로 출력한다.
<module 'sys' (built-in)>
```

그럼 대체 필드에서 출력 형식 변환자를 사용해서 출력 방식을 원하는 형식으로 변환해보자. 다음 예시는 분수를 먼저 텍스트 형식과 대표 형식으로 출력한 후, 대체 필드에서 출력 형식 변환자를 사용해서 순서대로 기본값, 텍스트 형식(!s), 대표 형식(!r), 아스키 코드만 사용하는 대표 형식(!a)으로 출력한 결과를 보여준다.

```
>>> from fractions import Fraction
>>> print(Fraction(3, 7))                              # 텍스트 형식으로 출력한다.
3/7
>>> Fraction(3, 7)                                     # 대표 형식으로 출력한다.
Fraction(3, 7)
>>> '{0} {0!s} {0!r} {0!a}'.format(Fraction(3, 7)) # 형식 변환자를 사용해서 출력한다.
'3/7 3/7 Fraction(3, 7) Fraction(3, 7)'
```

!a를 사용하면 한글 등 아스키 코드가 아닌 경우에 utf-16 형식으로 출력한다. 다음 예시는 영어 문자와 한글 문자를 !a를 사용해 대표 형식으로 출력한 결과를 보여준다.

```
>>> s1 = 'python'
>>> s2 = '파이썬'
>>> '{0!a}, {1!a}'.format(s1, s2)
"'python', '\\ud30c\\uc774\\uc36c'"
```

3. 서식 명세 설정

대체 필드에서 서식 명세(format specification) 기능을 이용하면 문자열, 정수, 실수의 서식 기본값을 변경할 수 있다. 즉, 서식 명세 지정자({:})를 사용하면 문자열이나 숫자 등을 원하는 포맷으로 출력할 수 있다. 서식 명세 지정자는 대체 필드 안에 쌍점(:)을 사용해서 다음과 같은 형식으로 한다.

```
{필드명:서식명세}
{필드명!형식변환:서식명세}
```

- 필드명이나 형식변환이 있는 경우 이 뒤에 서식 명세 지정자(format specifier)인 ':'를 추가하여 서식명세를 지정해주면 된다.

서식명세의 일반 형식은 다음과 같다. 가독성을 위해 각 서식 사이에 공백을 하나씩 두었지만, 실제 코드를 작성할 때는 공백 없이 붙여 써야 한다.

```
{: 채움문자 정렬 음양기호 # 0 폭 , .길이 형}
```

- 채움문자로는 '}'를 제외한 모든 문자가 올 수 있으며 빈 칸을 채우는 문자다.
- 정렬에는 <, >, ^, = 중 하나가 오며 각 부호의 뜻은 다음과 같다.
  - < : 문자열을 왼쪽 정렬한다.
  - > : 문자열을 오른쪽 정렬한다.
  - ^ : 문자열을 중앙 정렬한다.
  - = : 숫자에만 적용되며 음양기호와 숫자(정수나 실수) 사이를 채움문자로 채운다.

- 음양기호는 숫자에만 적용되며 +, -, 공백문자 중 하나가 온다. 각 부호의 뜻은 다음과 같다.
  - + : 양수와 음수 모두 각각 +와 - 기호를 반드시 표시한다.
  - - : 숫자가 음수인 경우에는 – 기호를 표시하지만, 양수의 경우는 표시하지 않는다.
  - 공백문자 : 숫자가 양수인 경우에는 공백으로 표시하고, 음수인 경우에는 - 기호를 표시한다.
- #은 숫자에만 적용되며 # 뒤에는 b, o, x(또는 X) 중 하나가 온다. 각 문자의 뜻은 다음과 같다.
  - b : 숫자를 2진법 형식으로 표현한다.
  - o : 숫자를 8진법 형식으로 표현한다.
  - x (또는 X) : 숫자를 16진법 형식으로 표현한다.
- 0은 숫자에만 적용되며 빈 칸을 0으로 채운다(padding).
- 폭은 문자열의 최소 길이를 설정한다.
- ,는 숫자에만 적용되며 숫자를 세 자리씩 그룹으로 묶어 쉼표(,)로 분리시켜 표시한다.
- .길이는 format() 메소드의 전달인자가 숫자인 경우와 문자열인 경우 다르게 적용된다.
  - 숫자의 경우 소수점 자릿수를 길이만큼 설정한다.
  - 문자열의 경우 문자열의 길이를 설정한다. 길이가 문자열의 길이보다 크면 문자열 길이만큼만, 길이가 문자열의 길이보다 작으면 길이만큼만 문자열을 표현한다.
- 형은 숫자에만 적용되며 다음과 같은 의미를 가진다.
  - 숫자가 정수인 경우 각 문자의 뜻은 다음과 같다.
    - b : 정수를 2진수로 표현한다.
    - o : 정수를 8진수로 표현한다.
    - x : 정수를 16진수 소문자로 표현한다.
    - X : 정수를 16진수 대문자로 표현한다.
    - d : 정수를 10진수로 표현한다.
    - c : 정수에 해당하는 유니코드 문자로 표현한다.
    - n : 정수를 컴퓨터에 설정된 현지(locale) 기본 숫자 표기 방식으로 표현한다. 이 경우에는 서식명세 중 ,를 사용하는 것은 의미가 없다.
  - 숫자가 실수인 경우 각 문자의 뜻은 다음과 같다.
    - e : 실수를 소문자 e를 사용해서 지수 형식으로 표현한다.

- E : 실수를 대문자 E를 사용해서 지수 형식으로 표현한다.
- f : 실수를 표준 부동소수점 형식으로 표현한다.
- g : 실수를 일반적인 실수 형식으로 표현한다. 이는 f와 같지만 큰 숫자의 경우에는 e를 사용하는 것과 같다.
- G : g와 마찬가지로, 큰 숫자의 경우에는 E를 사용하는 것과 같다.
- % : 실수에 100을 곱한 후 끝에 %를 추가해서 f 형식으로 표현한다.
- n : 실수를 컴퓨터에 설정된 현지(locale) 기본 숫자 표기 방식으로 표현한다. 이 경우에는 서식명세 중 ,를 사용하는 것은 의미가 없다.

예시를 통해 서식 명세를 사용하는 방법을 알아보기로 하자.

3.1 문자열 서식 명세

먼저 문자열을 다양한 방법으로 출력해보자. 다음 코드는 서식 명세를 지정하지 않고 기본값을 사용해서 문자열을 출력한 결과이다.

```
>>> s = '바로 쓰는 파이썬'
>>> '{}'.format(s)                                    # 기본값으로 출력한다.
'바로 쓰는 파이썬'
```

문자열의 폭을 설정할 경우, 만약 폭의 길이가 문자열의 길이보다 크면 남는 폭 길이는 공백으로 채운다. 반면에 폭이 문자열 길이보다 작으면 원래 문자열 길이를 사용한다.

```
>>> s = '바로 쓰는 파이썬'                              # 문자열의 길이는 9이다.
>>> '{:15}'.format(s)                                 # 문자열 폭을 15로 설정한다.
'바로 쓰는 파이썬      '
>>> '{:5}'.format(s)                                  # 문자열 폭을 5로 설정한다.
'바로 쓰는 파이썬'
```

이번에는 문자열에 폭을 설정하고 오른쪽 정렬한 후 채움문자로 남는 폭 길이를 채워보자. 이 경우 만약 폭 길이가 문자열 길이보다 작으면 채움문자 없이 원래 문자열만 사용한다.

```
>>> s = '바로 쓰는 파이썬'                    # 문자열의 길이는 9이다.
>>> '{:*>15}'.format(s)                      # 채움문자는 *, 우측 정렬, 폭은 15로 설정한다.
'******바로 쓰는 파이썬'
>>> '{:*>5}'.format(s)                       # 채움문자는 *, 우측 정렬, 폭은 5로 설정한다.
'바로 쓰는 파이썬'
```

이번에는 몇 가지 다른 채움문자와 정렬 방식으로 문자열의 서식을 설정해보자.

```
>>> s = '바로 쓰는 파이썬'                    # 문자열의 길이는 9이다.
>>> '{:!<15}'.format(s)                      # 채움문자는 !, 좌측 정렬, 폭은 15로 설정한다.
'바로 쓰는 파이썬!!!!!!'
>>> '{:#^15}'.format(s)                      # 채움문자는 #, 중앙 정렬, 폭은 15로 설정한다.
'###바로 쓰는 파이썬###'
>>> '{:$>15}'.format(s)                      # 채움문자는 $, 우측 정렬, 폭은 15로 설정한다.
'$$$$$$바로 쓰는 파이썬'
```

문자열의 길이를 지정하면 폭을 지정할 때와는 다른 결과를 가져온다. 문자열의 실제 길이보다 서식 명세에서 지정한 길이가 크면 문자열 길이까지만 문자열을 설정하고, 정렬과 채움문자는 다음 예시처럼 무시된다.

```
>>> s = '바로 쓰는 파이썬'                    # 문자열의 길이는 9이다.
>>> '{:.15}'.format(s)                       # 길이를 15로 설정한다.
'바로 쓰는 파이썬'
>>> '{:!<.15}'.format(s)                     # 채움문자는 !, 좌측 정렬, 길이는 15로 설정한다.
'바로 쓰는 파이썬'
>>> '{:#^.15}'.format(s)                     # 채움문자는 #, 중앙 정렬, 길이는 15로 설정한다.
'바로 쓰는 파이썬'
>>> '{:$>.15}'.format(s)                     # 채움문자는 $, 우측 정렬, 길이는 15로 설정한다.
'바로 쓰는 파이썬'
```

반대로 문자열의 실제 길이보다 서식 명세에서 지정한 길이가 작으면 서식 명세에서 지정한 길이까지만 문자열을 표현한다. 이 경우에도 정렬과 채움문자는 다음 예시처럼 무시된다.

```
>>> s = '바로 쓰는 파이썬'                    # 문자열의 길이는 9이다.
>>> '{:.5}'.format(s)                        # 길이를 5로 설정한다.
'바로 쓰는'
>>> '{:!<.5}'.format(s)                      # 채움문자는 !, 좌측 정렬, 길이는 5로 설정한다.
'바로 쓰는'
>>> '{:#^.5}'.format(s)                      # 채움문자는 #, 중앙 정렬, 길이는 5로 설정한다.
'바로 쓰는'
>>> '{:$>.5}'.format(s)                      # 채움문자는 $, 우측 정렬, 길이는 5로 설정한다.
'바로 쓰는'
```

3.2. 정수 서식 명세

이번에는 정수를 다양한 방법으로 출력해보자. 다음 코드는 서식 명세를 지정하지 않고 기본값을 사용해서 양의 정수와 음의 정수를 출력한 결과이다.

```
>>> pos_int, neg_int = 123456789, -123456789
>>> '{}'.format(pos_int)                     # 양의 정수를 기본값으로 출력한다.
'123456789'
>>> '{}'.format(neg_int)                     # 음의 정수를 기본값으로 출력한다.
'-123456789'
```

서식 명세에서 폭을 설정할 경우, 폭의 길이가 정수의 길이보다 크다면 숫자를 오른쪽 정렬한 후에 왼쪽에 남는 부분은 공백으로 채운다. 반면에 폭이 정수 길이보다 작다면 원래 정수의 길이를 사용한다.

```
>>> pos_int, neg_int = 123456789, -123456789
>>> '{:15}'.format(pos_int)                  # 폭을 15로 설정하고 양의 정수를 출력한다.
'      123456789'
>>> '{:15}'.format(neg_int)                  # 폭을 15로 설정하고 음의 정수를 출력한다.
'     -123456789'
>>> '{:5}'.format(pos_int)                   # 폭을 5로 설정하고 양의 정수를 출력한다.
'123456789'
>>> '{:5}'.format(neg_int)                   # 폭을 5로 설정하고 음의 정수를 출력한다.
'-123456789'
```

서식 명세에서 채움문자와 정렬 방식을 다양하게 이용해서 양의 정수와 음의 정수의 서식을 설정해보자.

```
>>> pos_int, neg_int = 123456789, -123456789
>>> '{:*<15}'.format(pos_int)          # 채움문자는 *, 좌측 정렬, 폭은 15로 양수를 설정한다.
'123456789******'
>>> '{:$>15}'.format(neg_int)          # 채움문자는 $, 우측 정렬, 폭은 15로 음수를 설정한다.
'$$$$$-123456789'
>>> '{:=^15}'.format(pos_int)          # 채움문자는 =, 중앙 정렬, 폭은 15로 양수를 설정한다.
'===123456789==='
>>> '{:#^15}'.format(neg_int)          # 채움문자는 #, 중앙 정렬, 폭은 15로 음수를 설정한다.
'##-123456789###'
```

만약 서식 명세에서 폭을 설정했지만 폭이 정수의 길이보다 작다면 채움문자와 정렬 방식을 무시하고 원래 정수만 표시한다.

```
>>> pos_int, neg_int = 123456789, -123456789
>>> '{:$>5}'.format(pos_int)                # 채움문자는 $, 우측 정렬, 폭은 5로 설정한다.
'123456789'
>>> '{:*<5}'.format(neg_int)                # 채움문자는 *, 좌측 정렬, 폭은 5로 설정한다.
'-123456789'
```

서식 명세에서 음양기호를 사용하면 양수와 음수에 기호를 표시할지 여부를 설정할 있다. 다음 예시처럼 + 기호를 사용하면 양수와 음수 모두 음양 기호를 표현하게 된다.

```
>>> pos_int, neg_int = 123456789, -123456789
>>> '{:+}'.format(pos_int)                  # 음양 기호를 반드시 표기한다.
'+123456789'
>>> '{:+}'.format(neg_int)
'-123456789'
```

만약 음수인 경우에만 - 기호를 표시하고, 양수의 경우는 표시하지 않으려면 - 기호를 사용하면 된다. 이는 기본값으로 출력하는 것과 같다.

```
>>> pos_int, neg_int = 123456789, -123456789
>>> '{:-}'.format(pos_int)                  # 음 기호만 표기한다.
'123456789'
>>> '{:-}'.format(neg_int)
'-123456789'
```

공백문자를 지정하면, 다음과 같이 양수인 경우에는 공백으로 표시하고, 음수인 경우에는 - 기호를 표시한다.

```
>>> pos_int, neg_int = 123456789, -123456789
>>> '{: }'.format(pos_int)              # 음 기호는 표기하고 양 기호는 공란으로 표기한다.
' 123456789'
>>> '{: }'.format(neg_int)
'-123456789'
```

이번에는 서식 설정할 때 사용하는 정렬 방식 중 = 부호를 사용해서 정수(실수에도 적용된다)와 기호 사이를 채움문자로 채우는 방식을 알아보자.

```
>>> pos_int, neg_int = 123456789, -123456789
>>> '{:*=+15}'.format(pos_int)            # 양수와 음양 기호 사이를 채움문자 *로 채운다.
'+******123456789'
>>> '{:$=15}'.format(neg_int)             # 음수와 음양 기호 사이를 채움문자 $로 채운다.
'-$$$$$123456789'
```

숫자를 10진수가 아닌 다른 기수법으로 표현하려면 서식 명세에 #을 사용하면 된다.

```
>>> i = 123456789
>>> '{:#b}'.format(i)                     # 숫자를 2진법 형식으로 표현한다.
'0b111010110111100110100010101'
>>> '{:#o}'.format(i)                     # 숫자를 8진법 형식으로 표현한다.
'0o726746425'
>>> '{:#x}'.format(i)                     # 숫자를 소문자 16진법 형식으로 표현한다.
'0x75bcd15'
>>> '{:#X}'.format(i)                     # 숫자를 대문자 16진법 형식으로 표현한다.
'0X75BCD15'
```

서식 명세에서 0을 사용하면 빈 칸을 모두 0으로 채운다. 이를 덧대기 또는 패딩(padding)이라 하는데 다음과 같이 사용할 수 있다.

```
>>> pos_int, neg_int = 123456789, -123456789
>>> '{:015}'.format(pos_int)                          # 빈 칸을 0으로 채운다.
'000000123456789'
>>> '{:015}'.format(neg_int)
'-00000123456789'
```

참고로 서식 설정의 정렬 방식 중 = 부호를 사용해서 채움문자로 0을 지정하면 다음과 같은 결과를 가져올 수 있다.

```
>>> pos_int, neg_int = 123456789, -123456789
>>> '{:0=15}'.format(pos_int)          # 숫자와 음양 기호 사이를 채움문자 0으로 채운다.
'000000123456789'
>>> '{:0=15}'.format(neg_int)
'-00000123456789'
```

서식 설정에서 콤마(,)를 지정하면 숫자를 세 자리마다 쉼표로 구분하여 표시한다.

```
>>> pos_int, neg_int = 123456789, -123456789
>>> '{:,}'.format(pos_int)              # 세 자리마다 쉼표로 숫자를 구분하여 표시한다.
'123,456,789'
>>> '{:,}'.format(neg_int)
'-123,456,789'
```

마지막으로 숫자 서식 명세에서 정수를 표현하는 형을 지정해서 출력해보자.

```
>>> i = 8361
>>> '{:b}'.format(i)                   # 정수를 2진수로 표현한다.
'10000010101001'
>>> '{:o}'.format(i)                   # 정수를 8진수로 표현한다.
'20251'
>>> '{:x}'.format(i)                   # 정수를 16진수 소문자로 표현한다.
'20a9'
>>> '{:X}'.format(i)                   # 정수를 16진수 대문자로 표현한다.
'20A9'
>>> '{:d}'.format(i)                   # 정수를 10진수로 표현한다.
'8361'
>>> '{:c}'.format(i)                   # 정수를 해당 유니코드 문자로 표현한다.
'₩'
>>> '{:n}'.format(i)                   # 정수를 현지의 기본 숫자 표기 방식으로 표현한다.
'8361'
```

3.3. 실수 서식 명세

실수는 바로 앞에서 설명한 정수 서식 설정과 동일한 방식으로 서식 명세를 설정할 수 있다. 따라서 여기서는 실수에만 적용되는 부분만 설명한다.

실수와 정수의 서식 명세에 있어 가장 큰 차이는 형 설정에 있다. 즉, 형을 설정할 때 사용하는 문자가 다르다. 다음 예시는 숫자 서식 명세에서 실수를 표현하는 형을 지정해서 출력한 결과를 보여준다.

```
>>> real = 1234.5678912345678987654321
>>> '{:e}'.format(real)          # 실수를 소문자 e를 사용해서 지수 형식으로 표현한다.
'1.234568e+03'
>>> '{:E}'.format(real)          # 실수를 대문자 E를 사용해서 지수 형식으로 표현한다.
'1.234568E+03'
>>> '{:f}'.format(real)          # 실수를 표준 부동소수점 형식으로 표현한다.
'1234.567891'
>>> '{:g}'.format(real)          # 실수를 일반적인 실수 형식으로 표현한다.
'1234.57'
>>> '{:G}'.format(real)          # 실수를 일반적인 실수 형식으로 표현한다.
'1234.57'
>>> '{:%}'.format(real) # 100을 곱한 후 끝에 %를 추가해서 표준 부동소수점 형식으로 표현한다.
'123456.789123%'
>>> '{:n}'.format(real)          # 실수를 현지의 기본 숫자 표기 방식으로 표현한다.
'1234.57'
```