6 클래스와 객체 완전 정복

본문에서 이미 클래스의 기본 개념들은 모두 살펴보았다. 이 부록에서는 본문에서 다루지 않은 내용들을 중심으로 설명하겠다. 특히 파이썬의 최상위 클래스인 object가 제공하는 특수 메소드(special method)를 상속받거나 재정의(overriding)하는 방법, 데코레이터(decorator)를 사용해서 네임 맹글링하는 방법, 속성값을 검증하거나 속성값으로 사용할 수 있는 범위 등을 제한하는 방법 등을 집중적으로 살펴보겠다. 이런 개념들을 이해하기 위해 이 부록에서는 SimpleTime, Time, Clock 클래스를 만들면서 설명하도록 하겠다.

1. 특수 메소드

파이썬에서 객체(object)는 데이터에 대한 추상화(abstraction)다. 따라서 파이썬이 다루는 모든 데이터는 객체 또는 객체 간의 관계로 표현된다. 그리고 파이썬은 사용자가 클래스를 정의해서 사용할 수 있도록 하는데 이를 '사용자 정의 클래스(custom class)'라고 부른다. 맞춤형 클래스라고도 번역할 수 있겠지만 사용자가 직접 정의해서 만든 클래스라는 것을 강조하기 위해서 이 책에서는 사용자 정의 클래스 또는 사용자 클래스라 하겠다.

object는 파이썬의 최상위 클래스다. object는 사용자 정의 클래스에서 상속받아 그대로 사용하거나 재정의(overriding)해서 사용할 수 있도록 여러가지 메소드를 제공하는데 이런 메소드를 특수 메소드(special method)라 한다. 이 절에서는 파이썬 object가 제공하는 특수 메소드 중 기본적인 메소드 몇 가지를 소개하도록 하겠다.

1.1. 초기화 메소드

객체를 생성할 때는 두 단계를 거친다. 첫 번째 단계는 초기화하지 않은 원상태의 객체를 생성하는 단계다. 이때 호출하는 메소드는 __new__()이다. 이 메소드는 주로 불변 자료형인 정수, 실수, 튜플의 하위 클래스가 인스턴스 생성을 사용자화할 때 사용하며 대부분의 경우 재정의할 필요가 없다.

두 번째 단계는 __new__()가 생성한 원상태의 객체를 호출한 프로그램으로 반환하기 전에 초기화하는 단계다. 이때 호출하는 메소드는 __init__()이다. 따라서 원상태의 객체를 초기화하기 위해서는 이 메소드를 재정의하는 것이 좋다. __init__() 메소드는 새로 생성하는 객체를 초기화할 때 자동으로 호출하는 메소드로, 본문에서 이미 소개했으니 본문 내용(490p~)을 참조하기 바란다.

1.2. 객체 표현 메소드

객체를 화면으로 출력할 때 사용하는 표현 메소드는 두 가지가 있다. __repr__() 메소드는 내장 함수 repr()에 의해 호출되며 해당 객체를 대표 형식(representational form)의 문자열로 반환한다. __str__() 메소드는 내장 함수 str()에 의해 호출되며 해당 객체를 사람이 쉽게 이해할 수 있는 형태인 텍스트 형식(string form)의 문자열로 반환한다. __str__() 메소드는 print() 함수를 통해 객체를 출력할 때 자동으로 호출되는 메소드다.

객체를 화면으로 출력하는 방식으로는 대표 형식과 텍스트 형식이 있다. 이 두 방식에 대해서는 본문에서 자세히 다루었다. 다음 예는 분수를 대표 형식과 텍스트 형식으로 출력한 결과를 보여준다.

```
>>> from fractions import Fraction
>>> Fraction(7, 11)                                    # 대표 형식으로 출력한다.
Fraction(7, 11)
>>> print(Fraction(7, 11))                             # 텍스트 형식으로 출력한다.
7/11
```

뒤에서 다룰 SimpleTime 클래스에서는 이 두 메소드를 재정의할 것이다. 다음 예는 SimpleTime 클래스의 인스턴스를 대표 형식과 텍스트 형식으로 출력한 결과다.

```
>>> from mytime import SimpleTime
>>> t1 = SimpleTime(1, 25, 57)
>>> t1                                                    # 대표 형식으로 출력한다.
SimpleTime<1:25:57>
>>> print(t1)                                             # 텍스트 형식으로 출력한다.
1:25:57
```

1.3. 비교 메소드

object 클래스가 제공하는 특수 메소드 중에는 객체들을 비교할 수 있는 비교 메소드들이 있다. 이들 비교 메소드는 비교 연산자를 사용할 때마다 자동으로 호출된다. 파이썬은 정수, 실수, 문자열, 리스트, 튜플 등 모든 자료형의 값들을 객체로 취급하기 때문에 이들을 비교할 때도 메소드를 사용한다. 이때 호출하는 메소드가 바로 비교 메소드다. 따라서 사용자 정의 클래스를 만들 때도, 같은 클래스가 생성한 객체들을 비교하거나 서로 다른 클래스의 객체를 비교해야 한다면 object 클래스가 제공하는 비교 메소드들을 재정의해서 사용해야 한다.

비교 연산자 부호들과 비교 메소드들과의 관계는 다음과 같다. x == y는 x.__eq__(y)를, x != y는 x.__ne__(y)를, x < y는 x.__lt__(y)를, x <= y는 x.__le__(y)를, x > y는 x.__gt__(y)를, x => y는 x.__ge__(y)를 각각 호출한다.

비교 메소드의 일반 형식은 다음과 같다.

```
객체.__eq__(self, other)
객체.__ne__(self, other)
객체.__lt__(self, other)
객체.__le__(self, other)
객체.__gt__(self, other)
객체.__ge__(self, other)
```

특징은 다음과 같다.

- 객체.__eq__(self, other)
  - self == other 표현식이 호출하는 메소드다. 즉, 객체 자신인 self와 other가 같으면 '참(True)'을 반환하고, 그렇지 않으면 '거짓(False)'을 반환한다.
  - 사용자 정의 클래스의 모든 인스턴스(즉, 객체)는 기본적으로 등호 연산(==)을 지원하는데, 자기 자신과 비교하지 않는 한 '거짓(False)'을 반환한다. 즉, 객체 참조가 같아야만 '참(True)'을 반환한다. 따라서 만약 값이 같은 경우도 '참(True)'을 반환하려면 __eq__() 메소드를 재정의해서 사용해야 한다.
- 객체.__ne__(self, other)
  - self != other 표현식이 호출하는 메소드다. 즉, 객체 자신인 self와 other가 다르면 '참(True)'을 반환하고, 그렇지 않으면 '거짓(False)'을 반환한다.
  - 참고로 파이썬은 등호 연산(==)을 기반으로 부등호 연산(!=)을 자동으로 인식한다. 따라서 굳이 __ne__() 메소드를 재정의하지 않아도 이 연산을 스스로 판단할 수 있다.
- 객체.__lt__(self, other)
  - self < other 표현식이 호출하는 메소드다. 즉, 객체 자신인 self가 other보다 작으면 '참(True)'을 반환하고, 그렇지 않으면 '거짓(False)'을 반환한다.
- 객체.__le__(self, other)
  - self <= other 표현식이 호출하는 메소드다. 즉, 객체 자신인 self가 other보다 작거나 같으면 '참(True)'을 반환하고, 그렇지 않으면 '거짓(False)'을 반환한다.
- 객체.__gt__(self, other)
  - self > other 표현식이 호출하는 메소드다. 즉, 객체 자신인 self가 other보다 크면 '참(True)'을 반환하고, 그렇지 않으면 '거짓(False)'을 반환한다.
- 객체.__ge__(self, other)
  - self >= other 표현식이 호출하는 메소드다. 즉, 객체 자신인 self가 other보다 크거나 같으면 '참(True)'을 반환하고, 그렇지 않으면 '거짓(False)'을 반환한다.

이 메소드들은 잠시 후에 만들 SimpleTime과 Time 클래스에서 재정의하여 다음 예처럼 시간들을 비교하는 데 사용할 것이다.

```
>>> from mytime import SimpleTime
>>> t1 = Time(0, 25, 59)                    # 시간이 00:25:59인 인스턴스를 생성한다.
>>> t2 = Time(1, 52, 5)                     # 시간이 01:52:05인 인스턴스를 생성한다.
>>> t1 > t2                                 # 00:25:59는 01:52:05 보다 큰가?
False
>>> t1 <= t2                                # 00:25:59는 01:52:05 보다 작거나 같은가?
True
```

1.4. 산술 메소드

object 클래스는 비교 메소드뿐만 아니라 숫자 자료형의 산술 연산을 모방할 수 있게 산술 메소드들을 제공하고 있다. 예를 들어 더하기(+) 연산은 __add__()와 __radd__(), 빼기(-) 연산은 __sub__()와 __rsub__()와 관련이 있다(아래 표 참조). 이 외에 논리 연산(not, and, or)도 모두 각 연산에 해당하는 특수 메소드를 제공하고 있다.

산술 메소드의 경우 호출하는 객체가 좌변에 위치하는지 우변에 위치하는지에 따라 메소드의 종류가 달라진다. 즉, myobject가 __add__() 메소드를 제공하는 클래스의 인스턴스면, myobject + x 표현식을 평가할 때 myobject.__add__(x)를 호출한다. 예를 들어, 사용자 클래스의 인스턴스인 myobject가 정수 5를 더하는 연산을 가능하게 하기 위해 __add__() 메소드를 구현했다고 해 보자. myobject + 5 표현식을 사용하면 myobject.__add__(5)를 호출하게 된다.

그런데 5 + myobject 표현식을 사용하면, 정수의 인스턴스인 5가 5.__add__(myobject)를 호출하게 된다. 하지만 정수는 myobject의 존재를 모르기 때문에 NotImplemented라는 특별한 값을 반환한다. 파이썬 인터프리터는 이 값을 인식하고 TypeError를 발생시킨다. 그런데 이 경우 파이썬은 TypeError를 발생시키기 전에 먼저 우변에 있는 객체인 myobject가 __radd__() 메소드를

[표] 산술 메소드

연산자	메소드
+	__add__() __radd__()
-	__sub__() __rsub__()
*	__mul__() __rmul__()
/	__truediv__() __rtruediv__()
//	__floordiv__() __rfloordiv__()
%	__mod__() __rmod__()
+=	__iadd__()
-=	__isub__()
*=	__imul__()
/=	__itruediv__()
//=	__ifloordiv__()
%=	__imod__()

구현했는지 확인한다. myobject가 __radd__()를 구현했다면, myobject.__radd__(5)를 호출한다. myobject가 __radd__() 메소드를 구현하지 않았다면, TypeError를 발생시킨다.

증강 할당 연산의 경우, 예를 들어 myobject가 __iadd__() 메소드를 제공하는 클래스의 인스턴스라면, myobject += x 표현식에 대해서, myobject.__iadd__(x)를 호출한다.

산술 메소드의 일반 형식은 다음과 같다. 여기서는 산술 메소드 중 Time 클래스에서 사용하는 산술 메소드 위주로만 설명한다. 나머지 메소드들도 유사한 방법으로 작동하니, 자세한 내용은 파이썬 문서[4]를 참조하기 바란다.

```
객체.__add__(self, other)
객체.__sub__(self, other)
객체.__radd__(self, other)
객체.__rsub__(self, other)
객체.__iadd__(self, other)
객체.__isub__(self, other)
```

특징은 다음과 같다.

- 객체.__add__(self, other)
  - self + other 표현식이 호출하는 메소드다. 객체 자신인 self에 other를 더한 결괏값을 반환한다.
- 객체.__sub__(self, other)
  - self - other 표현식이 호출하는 메소드다. 객체 자신인 self에서 other를 뺀 결괏값을 반환한다.
- 객체.__radd__(self, other)
  - other + self 표현식을 실행할 때, other가 self를 더하는 연산을 지원하지 않으면 호출하는 메소드이다. 즉, 표현식 좌변의 other가 self를 더할 수 없으면, 우변에 있는 객체 자신인 self로 other를 더한 결괏값을 반환한다.
- 객체.__rsub__(self, other)
  - other - self 표현식을 실행할 때, other가 self를 빼는 연산을 지원하지 않으면 호

4 https://docs.python.org/3.7/reference/datamodel.html

출하는 메소드이다. 즉, 표현식 좌변의 other가 self를 뺄 수 없으면, 우변에 있는 객체 자신인 self로 other를 뺀 결괏값을 반환한다.

- 객체.__iadd__(self, other)
  - self += other 표현식이 호출하는 메소드다. self에 other를 더한 결괏값으로 객체 자신인 self의 값을 갱신한다.
- 객체.__isub__(self, other)
  - self -= other 표현식이 호출하는 메소드다. self에 other를 뺀 결괏값으로 객체 자신인 self의 값을 갱신한다.

이 메소드들은 잠시 후 살펴볼 Time 클래스에서 재정의해서 다음 예처럼 시간들을 더하거나 빼기 연산을 할 때 사용할 것이다.

```
>>> from mytime import Time
>>> t1 = Time(1, 20, 30)                          # 시간이 01:20:30인 인스턴스를 생성한다.
>>> t2 = Time(23, 20, 30)                         # 시간이 23:20:30인 인스턴스를 생성한다.
>>> t1 + t2                          # 01:20:30 + 23:20:30 = 24:41:00, 즉 익일 00:41:00가 된다.
Time<0:41:0>
>>> t1 = Time(hour = 3, minute = 25)              # 시간이 03:25:00인 인스턴스를 생성한다.
>>> t2 = Time(1, 25, 59)                          # 시간이 01:25:59인 인스턴스를 생성한다.
>>> t2 -= t1                                      # 03:25:00 - 01:25:59 = 01:59:01이다.
>>> t2
Time<1:59:1>
```

2. SimpleTime 클래스

그럼 이제부터 시간과 관련된 클래스들을 만들어 보자. 먼저 단순 시간을 표현하는 클래스를 구현하자. 이 클래스의 이름은 SimpleTime이며, hour(시), minute(분), second(초)를 인스턴스 속성으로 가진다. 이 속성들은 공용(public)이기 때문에 객체이름.속성이름 형식으로 직접 접근이 가능하다. 만약 SimpleTime의 인스턴스를 생성할 때 hour, minute, second를 지정하지 않으면, 다음 예처럼 초깃값이 0시, 0분, 0초를 가진 인스턴스를 생성하게된다.

```
>>> t1 = SimpleTime()                          # 시간이 00:00:00인 인스턴스를 생성한다.
>>> t1.hour                                    # hour 속성값을 확인한다.
0
>>> t1.minute                                  # minute 속성값을 확인한다.
0
>>> t1.second                                  # second 속성값을 확인한다.
0
```

SimpleTime 클래스는 시간 단위에 대한 오류 입력을 처리하지 않는 아주 단순한 시간 클래스이다. 예를 들어, 25시, -2시, 63분, -1분, 60초, -15초라고 해도 오류 처리를 하지 않는다. 즉, 0~23시, 0~59분, 0~59초 범위 밖의 시간 단위가 와도 된다. SimpleTime 클래스의 하위 클래스인 Time 클래스에서는 이런 오류들을 처리할 것이다.

SimpleTime 클래스의 인스턴스를 생성할 때 다음 예처럼 다양한 방법으로 시간을 입력해서 생성해보자. 이때, 초기화 메소드에서 hour, minute, second 속성을 키워드 전달인자로 받아 처리하도록 정의한다.

```
>>> t2 = SimpleTime(5)                         # 시간이 05:00:00인 인스턴스를 생성한다.
>>> t3 = SimpleTime(5, 15)                     # 시간이 05:15:00인 인스턴스를 생성한다.
>>> t4 = SimpleTime(5, 15, 38)                 # 시간이 05:15:38인 인스턴스를 생성한다.
>>> t5 = SimpleTime(minute=29)                 # 시간이 00:29:00인 인스턴스를 생성한다.
>>> t6 = SimpleTime(second=52, minute=29)      # 시간이 00:29:52인 인스턴스를 생성한다.
>>> t7 = SimpleTime(hour=12, second=52)        # 시간이 12:00:52인 인스턴스를 생성한다.
```

앞에서 설명한 기능을 하는 SimpleTime 클래스를 다음과 같이 정의해서 'mytime.py'[5]에 저장하자.

```
class SimpleTime:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour                        # 인스턴스 속성 hour를 초기화한다.
        self.minute = minute                    # 인스턴스 속성 minute을 초기화한다.
        self.second = second                    # 인스턴스 속성 second를 초기화한다.
```

5 파일 이름이 모듈 이름으로 되기 때문에 mytime.py가 아닌 다른 이름으로 저장하면 나중에 SimpleTime 클래스를 사용하기 위해 import mytime.SimpleTime 또는 from mytime import SimpleTime으로 불러오기를 할 수 없다. mytime 대신 사용한 파일 이름을 모듈 이름으로 해서 불러와야 한다.

방금 정의한 초기화 메소드 외에 세 개의 특수 메소드를 재정의해야 한다. 먼저, 객체 표현 메소드인 __repr__()과 __str__() 메소드를 재정의해 다음과 같이 시간을 표현해보자.

```
>>> t = SimpleTime(11, 29, 52)                    # 시간이 11:29:52인 인스턴스를 생성한다.
>>> t                                             # 대표 형식으로 출력한다.
SimpleTime<11:29:52>
>>> print(t)                                      # 텍스트 형식으로 출력한다.
11:29:52
```

SimpleTime의 인스턴스를 위와 같이 대표 형식으로 출력하려면 object 클래스로부터 상속받은 __repr__() 메소드를 다음과 같이 재정의하면 된다.

```
def __repr__(self):
    return (f'{self.__class__.__name__}'
            f'<{self.hour!r}:{self.minute!r}:{self.second!r}>')
```

__class__.__name__은 self의 클래스 이름으로 대체되기 때문에 SimpleTime이 출력되며, 이어서 꺽쇠 괄호 안에 있는 대체 필드 {}에서 출력 형식 변환자인 !r을 사용해 출력할 시간을 대표 형식으로 변환한 문자열을 반환한다. 따라서 SimpleTime<11:29:52>와 같은 형식으로 출력하게 된다.

print() 함수를 사용해서 텍스트 형식으로 출력할 때 필요한 특수 메소드는 __str__()이다. 이 메소드를 다음과 같이 재정의하면 SimpleTime 인스턴스를 출력할 때 11:29:52와 같은 형식으로 출력된다.

```
def __str__(self):
    return f'{self.hour!s}:{self.minute!s}:{self.second!s}'
```

여기서 대체 필드 {} 안에 출력 형식 변환자인 !s을 사용해 출력할 내용을 텍스트 형식으로 변환한 문자열을 반환한다. 사실 단순히 숫자만 출력하기 때문에 대표 형식과 텍스트 형식으로 구분하는 출력 형식 변환자를 사용할 필요는 없지만 연습 차원에서 사

용해보기로 한다. 출력 형식 변환자인 {!}에 관해서는 '부록 4 - 문자열 서식 설정'을 참조하기 바란다.

마지막으로 두 개의 시간 객체가 같은 시간인지 판단하는 특수 메소드인 __eq__()를 구현해보자. 시간이 같은지 여부를 확인하는 방법 중 하나로 시, 분, 초를 각각 비교할 수도 있겠지만, 시간 단위를 모두 초로 변환하는 메소드를 만들어서 사용하면 나중에 시간 관련 연산을 할 때도 편리하게 사용할 수가 있다. 예를 들어, 시간끼리 더하거나 뺄 경우에 먼저 시간을 초로 변환한 후 계산하고, 이를 다시 시, 분, 초로 환산해서 반환하면 된다. 따라서 __eq__() 메소드를 재정의하기 전에 먼저 시간을 초 단위로 환산하는 메소드를 구현하자. 다음의 seconds() 메소드는 전체 시간을 초 단위로 계산해서 반환하는 메소드다. 예를 들어, 시간이 21분 25초라면 seconds() 메소드는 1285초를 반환한다.

```
def seconds(self):
    minutes = self.hour * 60 + self.minute          # 시를 분으로 환산한다.
    seconds = minutes * 60 + self.second            # 분을 초로 환산한다.
    return seconds
```

seconds() 메소드를 사용해서 __eq__() 메소드를 재정의해보자.

```
def __eq__(self, other):
    return self.seconds() == other.seconds()
```

__eq__() 메소드를 구현하면 다음과 같이 등호 연산자 ==을 사용해서 시간을 비교할 수 있다. 그리고 앞서 설명했듯이__eq__()만 구현하면 굳이 __ne__() 메소드를 구현하지 않아도 파이썬은 다음 예의 마지막 코드처럼 != 연산을 스스로 판단할 수 있다.

```
>>> # 시간이 17:58:12인 인스턴스를 생성한다.
... t1 = SimpleTime(hour=17, minute=58, second=12)
>>> t2 = SimpleTime(17, 58, 12)                # 시간이 17:58:12인 인스턴스를 생성한다.
>>> t3 = SimpleTime(12)                        # 시간이 12:00:00인 인스턴스를 생성한다.
>>> t1 == t2                                   # 17:58:12는 17:58:12와 같은가?
True
>>> t1 == t3                                   # 17:58:12는 12:00:00과 같은가?
False
>>> t2 != t3                                   # 17:58:12는 12:00:00과 다른가?
True
```

지금까지 설명한 SimpleTime 클래스를 다음과 같이 정의한 후 'mytime.py'에 저장하자. 여기서는 주석을 생략했지만 소스 코드에는 주석으로 코드를 설명했으니 필요한 경우에 내려받아 참고하기 바란다.

```
class SimpleTime:
    # --- 초기화 메소드 ------------------------------------------------------ #
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    # --- 특수 메소드 -------------------------------------------------------- #
    def __str__(self):
        return f'{self.hour!s}:{self.minute!s}:{self.second!s}'

    def __repr__(self):
        return (f'{self.__class__.__name__}'
                f'<{self.hour!r}:{self.minute!r}:{self.second!r}>')

    def __eq__(self, other):
        return self.seconds() == other.seconds()

    # --- 일반 메소드 -------------------------------------------------------- #
    def seconds(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds
```

다음 예는 SimpleTime 클래스를 실행한 결과를 보여준다.

```
>>> from mytime import SimpleTime
>>> t1 = SimpleTime(1, 25, 57)                # 시간이 01:25:57인 인스턴스를 생성한다.
>>> t1                                        # 대표 형식으로 출력한다.
SimpleTime<1:25:57>
>>> print(t1)                                 # 텍스트 형식으로 출력한다.
1:25:57
>>> t1.hour, t1.minute, t1.second             # 시, 분, 초에 각각 접근한다.
(1, 25, 57)
>>> t1.seconds()                              # 시간을 초 단위로 환산해서 반환한다.
5157
>>> t2 = SimpleTime(second=30)                # 시간이 00:00:30인 인스턴스를 생성한다.
>>> t3 = SimpleTime(0, 0, 30)                 # 시간이 00:00:30인 인스턴스를 생성한다.
>>> t2 == t3                                  # 00:00:30는 00:00:30과 같은가?
True
>>> t1 != t2                                  # 01:25:57는 00:00:30과 다른가?
True
```

3. Time 클래스

이번에는 SimpleTime의 하위 클래스인 Time을 구현해보자. Time 클래스의 가장 큰 특징은 SimpleTime의 속성인 hour, minute, second를 사적 속성(private attribute) 형식으로 설계하는 것이다. 이를 위해서 속성을 네임 맹글링(name mangling)하고 데코레이터(decorator)를 사용해서 유효한 입력 값만 처리하도록 한다. 다시 말해, 시, 분, 초 단위 중 하나라도 잘못된 값을 입력하면 오류 처리를 할 수 있게끔 구현한다. 예를 들어, 다음 예처럼 시를 25, 분을 10, 초를 59라고 입력할 경우 AssertionError가 발생한다.

```
>>> t = Time(25, 10, 59)
 Traceback (most recent call last):
 ...
 AssertionError: <hour>는 0-23 사이의 정수여야 합니다.
```

시간 인스턴스들을 서로 비교(==, !=, <, <=, >, >=)하기 위해 특수 메소드 중 비교 메소드 __eq__(), __ne__(), __lt__(), __le__(), __gt__(), __ge__()를 재정의해서 구현한

다. 이 경우에도 등호 연산(==)을 구현하면 부등호 연산(!=)을 자동으로 인식하기 때문에 __ne__() 메소드는 구현하지 않아도 된다.

시간을 더하고 빼는 연산 메소드들도 구현할 것이다. 즉, +, +=, -, -= 연산을 할 수 있도록 하기 위해 특수 메소드인 산술 메소드 중 __add__(), __radd__(), __iadd__(), __sub__(), __rsub__(), __isub__()를 재정의해서 구현한다. 이 외에도 시간 계산을 위한 메소드들도 필요한데, 이들은 Time 클래스 내부에서만 비공개로 사용하기 때문에 메소드 이름 앞에 밑줄 두 개를 접두어로 사용하는 명명 규칙에 따라서 메소드를 정의한다.

3.1. 네임 맹글링

파이썬은 엄밀한 사적 속성을 지원하지 않기 때문에, 다음 예처럼 속성에 바로 접근하는 것이 가능하다.

```
>>> stime = SimpleTime(hour=11, minute=35, second=47)
>>> stime.hour                                          # 시에 접근한다.
11
>>> stime.minute                                        # 분에 접근한다.
35
>>> stime.second                                        # 초에 접근한다.
47
```

하지만 대부분의 객체지향 프로그래밍 언어는 사적 속성을 지원하기 때문에 다른 객체가 자신의 속성 데이터에 직접 접근을 못하도록 해서 자신의 데이터를 보호한다. 따라서 이러한 사적 속성 데이터의 값에 접근해서 속성값을 읽거나 쓰기 위해서는 공용(public) 메소드인 get~ 메소드와 set~ 메소드를 사용한다. 사실 메소드 이름은 아무렇게나 지정해서 사용해도 되지만, 명명 규칙(naming convention)에 따르면 주로 속성값을 가져 오는 메소드의 이름은 get으로 시작하고 속성값을 수정하는 메소드의 이름은 set으로 시작하기 때문에 이 책에서도 이러한 메소드를 각각 get~ 메소드와 set~ 메소드라고 부르도록 한다.

그런데 본문에서도 언급했듯이 파이썬의 속성과 메소드는 공용이기 때문에 get~ 메소드와 set~ 메소드가 불필요하다. 파이썬 기본 철학에 따르면 모든 것을 비공개로 하는

것을 강조하지는 않는다. 하지만 만약 클래스나 인스턴스 속성에 아무나 접근하는 것이 부담스럽다면 네임 맹글링을 통해서 이 문제를 어느 정도 해결할 수 있다.

네임 맹글링(name mangling)이란 말 그대로 이름을 알아보지 못하게 함으로써 외부에서 의도적으로 속성에 직접 접근하는 것을 어렵게 한다. 파이썬에서는 클래스의 속성이나 메소드 이름을 외부에서 볼 수 없도록 하기 위해 이름 앞에 밑줄 두 개(__)를 붙여서 사용하는 명명 규칙을 따른다.

예를 들어 만약 SimpleTime 클래스의 속성을 다음과 같이 네임 맹글링해서 정의했다고 가정해보자.

```
>>> class SimpleTime:
...     def __init__(self, hour=0, minute=0, second=0):
...         self.__hour = hour                  # 인스턴스 속성 __hour를 초기화한다.
...         self.__minute = minute              # 인스턴스 속성 __minute을 초기화한다.
...         self.__second = second              # 인스턴스 속성 __second를 초기화한다.
...
>>>
```

이렇게 정의한 후에 속성에 바로 접근해보자.

```
>>> stime = SimpleTime(hour=11, minute=35, second=47)
>>> stime.__hour                                          # 시에 접근한다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'SimpleTime' object has no attribute '__hour'
>>> stime.__minute                                        # 분에 접근한다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'SimpleTime' object has no attribute '__minute'
>>> stime.__second                                        # 초에 접근한다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'SimpleTime' object has no attribute '__second'
```

모두 AttributeError가 발생하는 것을 알 수 있다. 이렇듯이 속성 이름 앞에 밑줄을 사

용하게 되면 직접적으로 속성 이름에는 접근할 수 없게 된다. 파이썬에서는 모든 것이 기본적으로는 공용(public)이기 때문에 속성 이름을 알고 있으면 접근이 가능하지만, 앞의 경우처럼 접두어로 밑줄을 사용해서 속성 이름을 선언하고 나면 해당 속성 이름을 알고 있어도 접근할 수 없다.

그렇다고 속성에 접근이 전혀 불가능한 것은 아니다. 앞의 경우도 다음 예처럼 객체이름._클래스이름__속성이름으로 접근하면 속성에 접근이 가능하다.

```
>>> stime = SimpleTime(hour=11, minute=35, second=47)
>>> stime._SimpleTime__hour                                    # 시에 접근한다.
11
>>> stime._SimpleTime__minute                                  # 분에 접근한다.
35
>>> stime._SimpleTime__second                                  # 초에 접근한다.
47
```

결론적으로, 속성 이름을 맹글링해도 실질적으로 속성을 완벽하게 비공개로 보호할 수는 없다. 네임 맹글링은 외부에서 의도적으로 속성에 직접 접근하는 것을 어렵게 만드는 최소한의 장치인 것이다.

3.2. 데코레이터

데코레이터(decorator)란 @로 시작하는 구문을 말한다. 함수나 메소드의 원래 기능을 수정하거나 다른 기능을 추가할 때 사용한다. 또한 코드를 더 짧게 만들고 파이썬답게(Pythonic) 하는 역할을 담당한다. 여기서 데코레이터 기능을 모두 설명하는 것은 이 책의 범위를 벗어나기 때문에, 이 가운데 @property를 사용하는 방법만 살펴보기로 한다.

앞서 SimpleTime 클래스를 다음처럼 정의하였다. 즉, 인스턴스 속성 이름 앞에 밑줄을 사용하지 않고 속성들을 선언하였다.

```
class SimpleTime:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour                          # 인스턴스 속성 hour를 초기화한다.
        self.minute = minute                      # 인스턴스 속성 minute을 초기화한다.
        self.second = second                      # 인스턴스 속성 second를 초기화한다.
```

Time은 SimpleTime의 하위 클래스이기 때문에 SimpleTime 클래스의 모든 속성과 메소드를 상속받게 된다. 따라서 Time 클래스의 속성은 재정의하지 않고 초기화 메소드에서 내장 함수인 super() 함수를 사용해서 상위 클래스인 SimpleTime의 초기화 메소드를 다음과 같이 불러온다.

```
class Time(SimpleTime):                           # SimpleTime의 하위 클래스로 정의한다.
    # --- 초기화 메소드 ------------------------------------------------------ #
    def __init__(self, hour=0, minute=0, second=0):
        super().__init__(hour, minute, second)
```

super() 함수는 하위 클래스(subclass)에서 상위 클래스(superclass)의 메소드를 호출할 때 사용한다. 만약 하위 클래스에서 __init__() 메소드를 구현하면, 결과적으로 상위 클래스의 __init__() 메소드를 재정의한 것이 된다. 따라서, 더 이상 상위 클래스의 __init__() 메소드를 자동으로 호출하지 않기 때문에 위의 예처럼 명시적으로 super() 함수를 사용해서 상위 클래스인 SimpleTime의 __init__() 메소드를 호출해야 한다.

이제 데코레이터 @property를 사용해서 Time 클래스의 메소드를 구현해보자. 데코레이터 @property를 사용해서 메소드를 구현하면 실제 클래스나 인스턴스의 속성과 매우 유사해 보이지만 다음과 같은 기능과 장점을 제공하게 된다.

- get~ 메소드와 set~ 메소드를 만들지 않고도 속성에 더 간단히 접근할 수 있다. 즉, 메소드를 속성으로 대체하는 효과를 가져온다.
- 속성에 접근할 때 값을 계산해서 반환하는 기능을 제공한다.
- 속성이 가지고 있는 데이터 값에 접근하거나 데이터 값을 변경할 때 제한(constraint)을 둘 수 있다.

@property를 사용하면 마치 속성에 직접 접근하는 것처럼 하면서, 실제 속성 이름은 맹글링되기 때문에 외부에서 의도적으로 접근하는 것을 어렵게 할 수 있다.

이를 위해 @property를 사용해서 다음과 같이 선언하면 된다.

- get~ 역할을 하는 메소드 앞에 @property 데코레이터를 선언한다.
- set~ 역할을 하는 메소드 앞에 앞서 @property로 선언한 메소드 이름을 사용해 @메소드이름.setter 데코레이터를 선언한다.

다음 예를 보면 앞서 설명한 내용을 이해하기 쉬울 것이다.

```
@property
def hour(self):                         # get~ 메소드 역할을 해서 시를 반환한다.
    return self.__hour                  # 속성 hour를 __hour로 네임 맹글링한다.

@hour.setter
def hour(self, hour):                   # set~ 메소드 역할을 해서 시를 수정할 수 있다.
    self.__hour = hour                  # 속성 hour를 __hour로 네임 맹글링한다.

@property
def minute(self):                       # get~ 메소드 역할을 해서 분을 반환한다.
    return self.__minute                # 속성 minute를 __minute으로 네임 맹글링한다.

@minute.setter
def minute(self, minute):               # set~ 메소드 역할을 해서 분을 수정할 수 있다.
    self.__minute = minute              # 속성 minute를 __minute으로 네임 맹글링한다.

@property
def second(self):                       # get~ 메소드 역할을 해서 초를 반환한다.
    return self.__second                # 속성 second를 __second로 네임 맹글링한다.

@second.setter
def second(self, second):               # set~ 메소드 역할을 해서 시를 수정할 수 있다.
    self.__second = second              # 속성 second를 __second로 네임 맹글링한다.
```

위의 코드를 보면 우선 속성 hour, minute, second의 역할을 하는 메소드를 각각 두 개씩 정의했다. 이 중에서 @property 데코레이터를 사용해서 정의한 메소드

인 hour(self), minute(self), second(self)는 각각 시, 분, 초의 값을 반환하는 역할을 한다. 이때 이 메소드들의 내부는 @property 데코레이터로 hour(시), minute(분), second(초) 속성을 네임 맹글링하여, 내부적으로는 각각 __hour, __minute, __second로 접근해야 한다.

하지만 다음 예처럼 외부적으로는 마치 속성에 직접 접근을 해서 속성값을 추출하거나 갱신하는 것처럼 보여 굳이 따로 get~ 메소드와 set~ 메소드를 만들 필요가 없다.

```
>>> t = Time(hour=11, minute=35, second=47)
>>> # gethour() 역할을 하는 hour() 메소드가 속성에 접근하듯 hour로 접근해서 __hour의 값을 추출한다.
... t.hour                                  # Time 인스턴스의 시(hour)에 접근한다.
11
>>> t                                       # Time 인스턴스를 대표 형식으로 출력한다.
Time<11:35:47>
>>> # sethour(5) 역할을 하는 hour() 메소드가 속성을 갱신하듯 hour로 접근해서 __hour의 값을 바꾼다.
... t.hour = 5                              # 시(hour)를 5로 바꾼다.
>>> t.hour                                  # 시간이 바뀌었다.
5
>>> t                                       # 수정한 Time 인스턴스를 대표 형식으로 출력한다.
Time<5:35:47>
```

즉, Time 클래스에서 @property 데코레이터를 사용하면 메소드를 속성처럼 취급할 수 있기 때문에 여러 장점을 가진다. 그중 하나는 속성에 접근할 때 값을 계산해서 반환하는 기능을 제공하는 것이다. 다음 예를 통해 살펴보자. 앞서 SimpleTime에서 seconds() 메소드를 다음처럼 정의하였다.

```
def seconds(self):                                  # SimpleTime 클래스 메소드
    minutes = self.hour * 60 + self.minute
    seconds = minutes * 60 + self.second
    return seconds
```

이번에는 Time 클래스에서 SimpleTime의 seconds() 메소드를 파이썬다운(Pythonic) 코드로 만들기 위해 @property 데코레이터를 사용해서 구현해보자.

```
@property                                                    # Time 클래스 메소드
def seconds(self):
    minutes = self.hour * 60 + self.minute
    seconds = minutes * 60 + self.second
    return seconds
```

SimpleTime의 seconds() 메소드와 Time 클래스에서 @property 데코레이터를 사용해서 구현한 seconds() 메소드의 차이점은 다음 코드를 보면 알 수 있다.

```
>>> t1 = SimpleTime(1, 25, 57)
>>> t1.seconds()                                    # seconds() 메소드를 호출한다.
5157
>>> t2 = Time(1, 25, 57)
>>> t2.seconds                                      # 속성처럼 seconds로 접근한다.
5157
```

하지만 set~ 메소드 역할을 하는 hour(self, hour), minute(self, minute), second(self, second) 메소드와는 달리 seconds() 메소드는 @seconds.setter 데코레이터로 seconds(self, seconds)를 구현하지 않았기 때문에 사용자가 임의로 이 값을 바꿀 수 없다. 따라서 만약 새로운 값을 seconds에 할당하고자 하면 다음과 같은 오류가 발생한다.

```
>>> t3 = Time(1, 25, 57)
>>> t3.seconds                          # 1시간 25분 57초를 초로 계산한 결과를 확인한다.
5157
>>> t3.seconds = 1000                   # 하지만 이 값을 임의로 바꿀 수는 없다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

이처럼 @property 데코레이터와 @메소드이름.setter 데코레이터를 필요한 곳에 적절히 사용해서 클래스의 기능을 조절할 수가 있다.

마지막으로 조건에 따라 속성에 접근하거나 속성값을 변경하도록 제한(constraint)을 두는 기능을 @property 데코레이터를 사용해서 추가해보자. 이 기능은 특히 값을 저장하

거나 새로운 값으로 바꿀 때 허용 가능한 범위의 값이 아니면 오류를 발생시킬 수 있다.

SimpleTime 클래스의 경우 다음 예처럼 시, 분, 초에 음수나 범위 밖의 숫자가 와도 오류가 발생하지 않는다.

```
>>> t1 = SimpleTime(hour=24, minute=-10, second=123456)
>>> t1                    # 시, 분, 초 모두 범위 밖을 숫자를 입력했지만 오류가 발생하지 않는다.
SimpleTime<24:-10:123456>
```

하지만 Time 클래스의 경우 앞서 사용한 시, 분, 초 값을 각각 입력하면 다음과 같은 오류가 발생하는 코드를 작성할 수 있다.

```
>>> t2 = Time(hour=24, minute=30, second=30)          # 시(hour)가 잘못 입력되었다.
Traceback (most recent call last):
...(중략)
AssertionError: '시'는 0-23 사이의 정수여야 합니다.
>>> t3 = Time(hour=12, minute=-10, second=30)         # 분(minute)이 잘못 입력되었다.
Traceback (most recent call last):
...(중략)
AssertionError: '분'은 0-59 사이의 정수여야 합니다.
>>> t4 = Time(hour=12, minute=30, second=123456)      # 초(second)가 잘못 입력되었다.
Traceback (most recent call last):
...(중략)
AssertionError: '초'는 0-59 사이의 정수여야 합니다.
```

위의 예처럼 시간 값을 지정할 때 그 값이 합당한 범위에 있는지 검증하기 위해서는 @메소드이름.setter 데코레이터로 선언한 메소드(set~메소드의 역할을 하는 메소드)에 다음과 같은 코드를 추가하면 된다.

```
@hour.setter
def hour(self, hour):
    assert hour >= 0 and hour < 24, "'시'는 0-23 사이의 정수여야 합니다."
    self.__hour = hour
```

```
@minute.setter
def minute(self, minute):
    assert minute >= 0 and minute < 60, "'분'은 0-59 사이의 정수여야 합니다."
    self.__minute = minute

@second.setter
def second(self, second):
    assert second >= 0 and second < 60, "'초'는 0-59 사이의 정수여야 합니다."
    self.__second = second
```

3.2. 비교 메소드 재정의

SimpleTime 클래스는 특수 메소드 __eq__()를 재정의해서 구현했기 때문에 다음과 같이 두 개의 시간이 서로 같은지 다른지 알 수 있다.

```
>>> from mytime import SimpleTime
>>> t1 = SimpleTime(hour=17, second=15)        # 시간이 17:00:55인 인스턴스를 생성한다.
>>> t2 = SimpleTime(hour=12)                   # 시간이 12:00:00인 인스턴스를 생성한다.
>>> t1 == t2
False
```

하지만 이 외의 비교 메소드는 구현하지 않았기 때문에 <, <=, >, >=를 사용하면 다음처럼 오류가 발생한다.

```
>>> t1 > t2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'SimpleTime' and 'SimpleTime'
```

반면에 Time 클래스는 오류가 나지 않는다.

```
>>> from mytime import Time
>>> t1 = Time(hour=17, second=55)              # 시간이 17:00:55인 인스턴스를 생성한다.
>>> t2 = Time(hour=12)                         # 시간이 12:00:00인 인스턴스를 생성한다.
>>> t1 >= t2                                   # 17:00:55는 12:00:00보다 크거나 같은가?
True
>>> t1 < t2                                    # 17:00:55는 12:00:00보다 작은가?
False
```

이는 Time 클래스에서 비교 메소드를 다음처럼 모두 구현했기 때문이다. 비교 특수 메소드를 재정의해서 구현하면 시간 인스턴스들을 비교해 어떤 시간이 먼저이고 나중인지 혹은 같은지를 알 수 있다.

```
def __eq__(self, other):                    # TypeError가 발생하면 파이썬이 처리하도록 한다.
    return self.seconds == other.seconds

def __lt__(self, other):                    # TypeError가 발생하면 파이썬이 처리하도록 한다.
    return self.seconds < other.seconds

def __le__(self, other):                    # TypeError가 발생하면 파이썬이 처리하도록 한다.
    return self.seconds <= other.seconds

def __gt__(self, other):                    # TypeError가 발생하면 파이썬이 처리하도록 한다.
    return self.seconds > other.seconds

def __ge__(self, other):                    # TypeError가 발생하면 파이썬이 처리하도록 한다.
    return self.seconds >= other.seconds
```

이번에도 __eq__() 메소드만 있으면 파이썬이 그 반대의 경우를 자동으로 인식하기 때문에 __ne__() 메소드는 구현하지 않았다.

참고로 서로 다른 클래스의 인스턴스를 비교할 경우, 즉 비교가 불가능한 자료형이나 객체를 비교하면(숫자와 문자열 비교 등) TypeError가 발생하는데 이를 파이썬이 알아서 처리하도록 하기 위해 위 코드에서는 따로 오류를 처리하는 코드를 구현하지 않았다.

!=을 포함하여 모든 비교 연산을 실행해보자.

```
>>> t1 = Time(0, 25, 59)                    # 시간이 00:25:59인 인스턴스를 생성한다.
>>> t2 = Time(hour = 1)                     # 시간이 01:00:00인 인스턴스를 생성한다.
>>> t1 == t2                                # 00:25:59와 01:00:00는 같은가?
False
>>> t1 != t2                                # 00:25:59와 01:00:00는 다른가?
True
>>> t1 < t2                                 # 00:25:59는 01:00:00보다 작은가?
True
>>> t1 <= t2                                # 00:25:59는 01:00:00보다 작거나 같은가?
True
>>> t1 > t2                                 # 00:25:59는 01:00:00보다 큰가?
False
>>> t1 >= t2                                # 00:25:59는 01:00:00보다 크거나 같은가?
False
```

3.3. 산술 메소드 재정의

Time 클래스에 시간을 더하고 뺄 수 있는 메소드도 구현해보자. 특수 메소드인 산술 메소드 가운데 __add__(), __radd__(), __iadd__(), __sub__(), __rsub__(), __isub__()를 재정의해 +, +=, -, -= 연산을 할 수 있도록 한다. 먼저 + 연산은 두 개의 시간을 합한 시간을 반환한다.

```
>>> t1 = Time(1, 20, 30)                    # 시간이 01:20:30인 인스턴스를 생성한다.
>>> t2 = Time(1, 25, 30)                    # 시간이 01:25:30인 인스턴스를 생성한다.
>>> t1 + t2                                 # 01:20:30 + 01:25:30 = 02:46:00이 된다.
Time<2:46:0>
```

이때 만약 시간의 합이 24시간을 초과하면 24시간을 초과한 시간만 반환한다. 이는 하루가 지난 시간을 보여주는 것인데 Time 클래스에는 날짜 개념이 없기 때문에 그냥 24시간을 초과한 시간만 반환한다.

```
>>> t1 = Time(1, 20, 30)                    # 시간이 01:20:30인 인스턴스를 생성한다.
>>> t3 = Time(23, 35, 45)                   # 시간이 23:35:45인 인스턴스를 생성한다.
>>> t1 + t3                                 # 01:20:30 + 23:35:45 = 24:56:15, 즉 익일 00:56:15가 된다.
Time<0:56:15>
```

이때 Time 클래스의 인스턴스가 아닌 객체와 + 연산을 수행하면 오류가 난다. 예를 들

어 다음과 같이 Time 클래스의 인스턴스에 정수나 문자열을 더하면 TypeError가 발생한다.

```
>>> t1 + 12                          # Time 인스턴스에 정수를 더하면 Time이 예외 처리를 한다.
Traceback (most recent call last):
...(중략)
TypeError: '12'는 'int'의 인스턴스며 'Time' 클래스의 유효한 객체가 아닙니다.
>>> t1 + '23:35:45'                  # Time 인스턴스에 문자열을 더하면 Time이 예외 처리를 한다.
Traceback (most recent call last):
...(중략)
TypeError: '23:35:45'는 'str'의 인스턴스며 'Time' 클래스의 유효한 객체가 아닙니다.
```

+ 연산을 할 때 앞서 설명한 기능들을 수행하는 __add__()를 재정의해서 구현해보자.

```
def __add__(self, other):
    if isinstance(other, self.__class__):
        return self.__add_time(other)
    else:
        raise TypeError(
            f"'{other}'는(은) '{other.__class__.__name__}'의 인스턴스며 "
            f"'Time' 클래스의 유효한 객체가 아닙니다.")
```

__add__() 메소드를 포함한 모든 산술 메소드는 산술 연산 대상이 같은 클래스 또는 하위 클래스의 인스턴스인지를 먼저 확인한다. 만약 같은 종류의 인스턴스면 __add_time() 메소드를 호출해서 자신과 대상 인스턴스의 시간을 합한 결과를 반환한다. 하지만 연산 처리가 불가능한 다른 종류의 클래스(또는 자료형) 인스턴스면 TypeError를 발생시킨다.

앞서 설명했듯이 속성이나 메소드 이름 앞에 밑줄 두 개(__)를 붙여서 사용하는 것은 해당 속성이나 메소드를 외부에서 사용할지 말라는 일종의 약속이다. __add_time() 메소드가 여기에 해당하는데 이 메소드는 Time 클래스 내부에만 사용하는 일종의 'Helper(도우미)' 기능을 한다. 추후에 매개변수의 개수 또는 역할이나 이름 등이 바뀔 수 있는 메소드는 이처럼 밑줄을 붙여서 외부에서는 사용하지 않도록 하는 것이 좋다.

__add_time() 메소드는 다음과 같이 구현했다.

```
def __add_time(self, other):
    total = self.seconds + other.seconds
    return self.__convert_seconds_to_time(total)
```

이 메소드는 자신(self)의 시간과 대상 객체(other)의 시간을 초로 환산해서 합친 초 단위의 결과를 다시 시간(시, 분, 초)으로 바꿔서 반환하는 메소드다. 이 메소드의 경우 __convert_seconds_to_time()이라는 도우미 메소드를 사용하는데, 다음과 같이 구현했다.

```
def __convert_seconds_to_time(self, seconds):
    converted_seconds = self.__getseconds_for_oneday(seconds)
    minutes, second = divmod(converted_seconds, 60)
    hour, minute = divmod(minutes, 60)
    return self.__class__(hour, minute, second)
```

이 메소드는 전달받은 초(second) 단위의 숫자를 다시 __getseconds_for_oneday() 메소드를 호출해 처리한 결과를 converted_seconds 변수로 반환한다. converted_seconds의 값은 다시 divmod() 함수를 통해 시, 분, 초로 환산해서 반환한다. 참고로 __getseconds_for_oneday() 메소드는 전달받은 값(초)이 하루(24시간)를 초과한 값이라면 하루를 뺀 나머지 시간을 초로 반환한다.

다음 코드는 __getseconds_for_oneday() 메소드를 구현한 것인데, 논리는 매우 간단하다. 전달받은 초가 하루에 해당하는 86400초보다 같거나 크면 86400로 나눈 나머지를 반환한다.

```
def __getseconds_for_oneday(self, seconds):
    if seconds >= 86400:                                    # 하루는 84,400초다.
        seconds %= 86400
    return seconds
```

이번에는 + 연산자의 좌변에 있는 객체가 Time 클래스의 인스턴스를 인식하지 못해 더

하기 연산을 수행할 수 없는 경우에, 다음처럼 Time 클래스의 인스턴스가 대신 더하기 연산을 하도록 __radd__() 메소드를 재정의해서 구현해보자.

```
def __radd__(self, other):
    return self.__add__(other)
```

이때 __radd__() 메소드는 사실상 __add__() 메소드를 호출한다. 즉, 같은 Time 클래스 인스턴스끼리는 시간을 초로 환산해서 더한 후 다시 시, 분, 초로 바꿔 반환한다. 하지만 만약 + 연산자의 좌변에 있는 객체가 Time 클래스의 인스턴스가 아니면 다음과 같이 TypeError를 발생시킨다.

```
>>> 12 + t1                          # 정수에 Time 인스턴스를 더하면 Time이 예외 처리를 한다.
Traceback (most recent call last):
...(중략)
TypeError: '12'는 'int'의 인스턴스며 'Time' 클래스의 유효한 객체가 아닙니다.
>>> '23:35:45' + t1                  # 문자열에 Time 인스턴스를 더하면 Time이 예외 처리를 한다.
Traceback (most recent call last):
...(중략)
TypeError: '23:35:45'는 'str'의 인스턴스며 'Time' 클래스의 유효한 객체가 아닙니다.
```

하지만 이런 경우 굳이 __radd__() 메소드를 구현할 필요 없이 + 연산의 좌변에 오는 객체가 처리하도록 하는 것도 괜찮은 방법이다. 즉, __radd__() 메소드가 없다면 다음처럼 좌변에 있는 객체가 알아서 예외 처리를 하게 된다. 따라서 시간 계산을 할 때 __radd__() 메소드를 반드시 구현할 이유는 없다.

```
>>> 12 + t1                      # 정수에 Time 인스턴스를 더하면 정수가 예외 처리를 한다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'Time'
>>> '23:35:45' + t1              # 문자열에 Time 인스턴스를 더하면 문자열이 예외 처리를 한다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "Time") to str
```

이번에는 증강 할당 연산자인 +=를 사용해서 두 개의 시간을 합한 후 연산자 좌변에

있는 시간에 결과를 다음과 같이 할당해보자.

```
>>> t1 = Time(minute = 25)                    # 시간이 00:25:00인 인스턴스를 생성한다.
>>> t2 = Time(2, 30, 27)                      # 시간이 02:30:27인 인스턴스를 생성한다.
>>> t1 += t2                                  # [증강 할당] 00:25:00 + 02:30:27 = 02:55:27
>>> t1                                        # t1의 시간이 02:55:27로 바뀌었다.
Time<2:55:27>
```

한편, + 연산과 마찬가지로 Time 클래스의 인스턴스가 아닌 객체와 += 연산을 수행할 경우 다음과 같이 TypeError가 발생한다.

```
>>> t1 += 15                          # Time 인스턴스에 정수를 더하면 Time이 예외 처리를 한다.
Traceback (most recent call last):
...(중략)
TypeError: '15'는 'int'의 인스턴스며 'Time' 클래스의 유효한 객체가 아닙니다.
>>> t1 += '02:30:27'                  # Time 인스턴스에 문자열을 더하면 Time이 예외 처리를 한다.
Traceback (most recent call last):
...(중략)
TypeError: '02:30:27'는 'str'의 인스턴스며 'Time' 클래스의 유효한 객체가 아닙니다.
```

이번에는 += 연산을 수행하는 __iadd__()를 재정의해서 구현해보자.

```
def __iadd__(self, other):
    if isinstance(other, self.__class__):
        return self.__add_time(other)
    else:
        raise TypeError(
            f"'{other}'는(은) '{other.__class__.__name__}'의 인스턴스며 "
            f"'Time' 클래스의 유효한 객체가 아닙니다.")
```

그런데 이 코드는 사실상 __add__() 메소드로 구현한 코드와 정확히 같기 때문에 __iadd__() 메소드를 굳이 구현하지 않아도 된다.

이번에는 - 연산에 대해 알아보자. - 연산은 두 시간의 차이를 반환한다.

```
>> Time(23, 55, 59) - Time(12, 30, 30)                    # 23:55:59 - 12:30:30 = 11:25:29
Time<11:25:29>
```

또한 작은 시간에서 큰 시간을 빼도 다음과 같이 음수로 시간을 표시하지는 않는다. 왜냐하면 시간은 음수가 없을 뿐만 아니라 시간 차이를 음수로 표현할 이유도 없기 때문이다.

```
>>> t1 = Time(1, 20, 30)                                  # 시간이 01:20:30인 인스턴스를 생성한다.
>>> t2 = Time(2, 30, 27)                                  # 시간이 02:30:27인 인스턴스를 생성한다.
>>> t1 - t2                          # 01:20:30 - 02:30:27 = -01:09:57, 즉 시차는 1시간 9분 57초이다.
Time<1:9:57>
```

그리고 + 연산과 마찬가지로 만약 Time 클래스의 인스턴스가 아닌 객체와 - 연산을 하면 오류가 난다. 예를 들어 Time 클래스의 인스턴스에 정수나 문자열을 빼면 다음과 같이 TypeError가 발생한다.

```
>>> t1 = Time(1, 20, 30)                  # Time 인스턴스를 정수로 빼면 Time이 예외 처리를 한다.
>>> t1 - 12
Traceback (most recent call last):
...(중략)
TypeError: '12'는 'int'의 인스턴스며 'Time' 클래스의 유효한 객체가 아닙니다.
>>> t1 - '02:30:27'                       # Time 인스턴스를 문자열로 빼면 Time이 예외 처리를 한다.
Traceback (most recent call last):
...(중략)
TypeError: '02:30:27'는 'str'의 인스턴스며 'Time' 클래스의 유효한 객체가 아닙니다.
```

이처럼 - 연산을 수행하는 __sub__()를 메소드를 재정의해서 구현해보자.

```
def __sub__(self, other):
    if isinstance(other, self.__class__):
        return self.__subtract_time(other)
    else:
        raise TypeError(
            f"'{other}'는(은) '{other.__class__.__name__}'의 인스턴스며 "
            f"'Time' 클래스의 유효한 객체가 아닙니다.")
```

__sub__() 메소드도 __add__()와 마찬가지로 우선 연산 처리 대상이 같은 클래스 또는 하위 클래스의 인스턴스인지를 먼저 확인한다. 만약 연산 처리가 가능한 같은 Time 클래스의 인스턴스면 Time 클래스 내부에서만 사용하는 __subtract_time() 메소드를 호출해서 자신으로부터 대상 인스턴스의 시간을 뺀 결과를 반환한다. 만약 연산 처리가 가능하지 않은 다른 종류의 클래스(또는 자료형) 인스턴스면 TypeError를 발생시킨다.

__subtract_time() 메소드는 다음과 같이 구현되었다.

```
def __subtract_time(self, other):
    difference = abs(self.seconds - other.seconds)
    return self.__convert_seconds_to_time(difference)
```

이 메소드는 자신(self)의 시간과 대상 객체(other)의 시간을 초로 환산한 후 이 둘의 차이를 계산한다. 이때 시간 차이를 계산하는 것이 목적이기 때문에 절댓값으로 처리한다. 즉, 작은 시간에서 큰 시간을 빼면 음수가 되지만 절댓값으로 처리하여 두 시간의 차이를 양수로 계산한다. 그러고 나서 앞서 설명한 도우미 메소드인 __convert_seconds_to_time()로 계산 결과를 전달해서 다시 시간(시, 분, 초)으로 바꾼 후 반환한다.

__rsub__() 메소드를 다음과 같이 구현하면 - 연산자의 좌변에 있는 객체가 Time 클래스의 인스턴스가 아니어도 우변에 있는 Time 클래스의 인스턴스가 대신 빼기 연산을 할 수 있다.

```
def __rsub__(self, other):
    return self.__sub__(other)
```

물론 __rsub__() 메소드가 없어도 - 연산자의 좌변에 있는 객체가 Time 인스턴스를 인식하지 못하기 때문에 예외 처리 등 적절한 조치를 취하기 때문에 반드시 구현할 필요는 없다.

__isub__()를 구현하면 다음과 같이 증강 할당 연산자인 -=가 이 메소드를 호출해서 좌변의 객체 값에서 우변의 객체 값을 뺀 결과를 다시 좌변에 있는 Time 인스턴스에

할당할 수 있다

```
>>> t1 = Time(hour = 3, minute = 25)          # 시간이 03:25:00인 인스턴스를 생성한다.
>>> t2 = Time(1, 25, 59)                      # 시간이 01:25:59인 인스턴스를 생성한다.
>>> t2 -= t1        # [증강 할당] 01:25:59 - 03:25:00 = -01:59:01, 즉 시차는 1시간 59분 1초이다.
>>> t2
Time<1:59:1>
```

__isub__()를 구현한 다음 코드를 보면 알겠지만 __sub__() 메소드의 코드와 정확히 같다. 따라서 이 경우도 __iadd__()처럼 __isub__() 메소드를 따로 구현하지 않아도 증강 할당 연산을 할 수 있다.

```
def __isub__(self, other):
    if isinstance(other, self.__class__):
        return self.__subtract_time(other)
    else:
        raise TypeError(
            f"'{other}'는(은) '{other.__class__.__name__}'의 인스턴스며 "
            f"'Time' 클래스의 유효한 객체가 아닙니다.")
```

위의 내용을 포함하는 Time 클래스를 'mytime.py'의 SimpleTime 클래스 아래에 정의해서 저장하도록 한다. 여기서는 주석을 생략했지만 소스 코드에는 주석으로 코드를 설명했으니 필요한 경우에 내려받아 참고하기 바란다.

```
class Time(SimpleTime):
    # --- 초기화 메소드 ------------------------------------------------------ #
    def __init__(self, hour=0, minute=0, second=0):
        super().__init__(hour, minute, second)

    # --- 데코레이터 -------------------------------------------------------- #
    @property
    def hour(self):
        return self.__hour

    @hour.setter
    def hour(self, hour):
        assert hour >= 0 and hour < 24, "'시'는 0-23 사이의 정수여야 합니다."
        self.__hour = hour
```

```
    @property
    def minute(self):
        return self.__minute

    @minute.setter
    def minute(self, minute):
        assert minute >= 0 and minute < 60, "'분'은 0-59 사이의 정수여야 합니다."
        self.__minute = minute

    @property
    def second(self):
        return self.__second

    @second.setter
    def second(self, second):
        assert second >= 0 and second < 60, "'초'는 0-59 사이의 정수여야 합니다."
        self.__second = second

    @property
    def seconds(self):
        minutes = self.__hour * 60 + self.__minute
        seconds = minutes * 60 + self.__second
        return seconds

    # --- 특수 메소드 ---------------------------------------------------------- #
    # - 비교 연산자 ------------------------------------------------------------ #
    def __eq__(self, other):     # TypeError 오류는 파이썬이 처리하도록 한다.
        return self.seconds == other.seconds

    def __lt__(self, other):     # TypeError 오류는 파이썬이 처리하도록 한다.
        return self.seconds < other.seconds

    def __le__(self, other):     # TypeError 오류는 파이썬이 처리하도록 한다.
        return self.seconds <= other.seconds

    def __gt__(self, other):     # TypeError 오류는 파이썬이 처리하도록 한다.
        return self.seconds > other.seconds

    def __ge__(self, other):     # TypeError 오류는 파이썬이 처리하도록 한다.
        return self.seconds >= other.seconds
```

```
    # - 산술 연산자 ------------------------------------------------------------ #
    def __add__(self, other):
        if isinstance(other, self.__class__):
            return self.__add_time(other)
        else:
            raise TypeError(
                f"'{other}'는(은) '{other.__class__.__name__}'의 인스턴스며 "
                f"'Time' 클래스의 유효한 객체가 아닙니다.")

    def __radd__(self, other):
        return self.__add__(other)

    # __iadd__()는 __add__()와 같은 연산 논리이기 때문에 생략해도 된다.
    def __iadd__(self, other):
        if isinstance(other, self.__class__):
            return self.__add_time(other)
        else:
            raise TypeError(
                f"'{other}'는(은) '{other.__class__.__name__}'의 인스턴스며 "
                f"'Time' 클래스의 유효한 객체가 아닙니다.")

    def __sub__(self, other):
        if isinstance(other, self.__class__):
            return self.__subtract_time(other)
        else:
            raise TypeError(
                f"'{other}'는(은) '{other.__class__.__name__}'의 인스턴스며 "
                f"'Time' 클래스의 유효한 객체가 아닙니다.")

    def __rsub__(self, other):
        return self.__sub__(other)

    # __isub__()는 __sub__()와 같은 연산 논리이기 때문에 생략해도 된다.
    def __isub__(self, other):
        if isinstance(other, self.__class__):
            return self.__subtract_time(other)
        else:
            raise TypeError(
                f"'{other}'는(은) '{other.__class__.__name__}'의 인스턴스며 "
                f"'Time' 클래스의 유효한 객체가 아닙니다.")
```

```
    # --- 내부 전용 메소드 ---------------------------------------------------- #
    def __add_time(self, other):
        total = self.seconds + other.seconds
        return self.__convert_seconds_to_time(total)

    def __subtract_time(self, other):
        difference = abs(self.seconds - other.seconds)
        return self.__convert_seconds_to_time(difference)

    def __convert_seconds_to_time(self, seconds):
        converted_seconds = self.__getseconds_for_oneday(seconds)
        minutes, second = divmod(converted_seconds, 60)
        hour, minute = divmod(minutes, 60)
        return self.__class__(hour, minute, second)

    def __getseconds_for_oneday(self, seconds):
        if seconds >= 86400:  # 하루는 86,400초다.
            seconds %= 86400
        return seconds
```

4. Clock 클래스

이번에는 Time 클래스의 하위 클래스인 Clock 클래스를 구현해보자. Clock 클래스의 가장 큰 특징은 시간(시, 분, 초)을 24시간 형식이 아닌 12시간 형식(AM/PM)으로 표현하는 것이다. 이를 위해 SimpleTime 클래스에서 사용한 특수 메소드 중 객체 표현 메소드인 __repr__()와 __str__()를 Clock 클래스에서 다시 재정의해서 구현할 것이다.

아래 예는 Clock 클래스의 인스턴스를 대표 형식과 텍스트 형식으로 출력한 결과를 보여준다.

```
>>> from mytime import Clock
>>> t1 = Clock(11, 59, 59)                    # 시간이 11:59:59인 인스턴스를 생성한다.
>>> t2 = Clock(hour = 13)                     # 시간이 13:00:00인 인스턴스를 생성한다.
>>> t1                                        # 대표 형식으로 출력한다.
Clock<11:59:59>AM
>>> print(t1)                                 # 텍스트 형식으로 출력한다.
11:59:59AM
>>> t2                                        # 대표 형식으로 출력한다.
Clock<1:0:0>PM
>>> print(t2)                                 # 텍스트 형식으로 출력한다.
1:0:0PM
```

Clock의 인스턴스를 앞과 같은 대표 형식(Clock<hh:mm:ss>AM/PM)으로 출력하기 위해 __repr__()를 다음처럼 재구현한다.

```
def __repr__(self):
    if self.hour < 12:
        return (f'{self.__class__.__name__}'
                f'<{self.hour!r}:{self.minute!r}:{self.second!r}>AM')
    else:
        return (
            f'{self.__class__.__name__}'
            f'<{self.__getclockhour()!r}:{self.minute!r}:{self.second!r}>PM')
```

__class__.__name__은 self의 클래스 이름으로 대체가 되기 때문에 'Clock'이 출력되며, 이어서 꺾쇠 괄호 안에 있는 대체 필드 {}에서 출력 형식 변환자인 !r을 사용해 출력할 시간을 대표 형식으로 변환한 문자열을 반환한다. 이때, hour(시)가 0~11시이면 'AM'을 꺾쇠 괄호 다음에 오게 해서 'Clock<11:20:59>AM'과 같은 형식으로 표현한다. 그리고 12~23이면 먼저 __getclockhour() 메소드를 호출해서 오후 시간으로 변환(예를 들어 15시면 3시로)하고 꺾쇠 괄호(<>) 다음에 'PM'을 오게 해서 'Clock<12:52:9>PM' 또는 'Clock<3:15:27>PM'과 같은 형식으로 출력한다.

hour(시)가 12보다 클 경우 오후 시간으로 바꾸는 __getclockhour() 메소드는 Clock 클래스의 내부에서 사용하는 도우미 메소드이며 다음과 같이 구현한다.

```
def __getclockhour(self):
    return self.hour if self.hour <= 12 else self.hour % 12
```

시(hour)가 0~12이면 그대로 반환하고 13~23이면 12로 나눈 나머지를 반환한다.

print() 함수를 사용해서 텍스트 형식으로 출력할 때 필요한 __str__() 메소드를 다음처럼 재구현했다.

```
def __str__(self):
    if self.hour < 12:
        return f'{self.hour!s}:{self.minute!s}:{self.second!s}AM'
    else:
        return f'{self.__getclockhour()!s}:{self.minute!s}:{self.second!s}PM'
```

__str__() 메소드의 구현 논리도 __repr__() 메소드와 유사한데, 차이점은 출력을 할 때 클래스 이름과 꺾쇠 괄호(<>) 없이 오전 시간이면 '11:20:59AM'과 같은 형식으로 표현하고, 오후 시간이면 __getclockhour() 메소드를 호출해서 오후 시간으로 변환한 후 '12:52:19PM' 또는 '3:15:27PM'과 같은 형식으로 출력한다는 점이다.

Clock은 Time의 하위 클래스이기 때문에 Time 클래스가 제공하는 비교 연산과 산술 연산을 그대로 상속받아서 다음과 같이 사용할 수 있다.

```
>>> from mytime import Clock
>>> t1 = Clock(11, 59, 59)        # 시간이 11:59:59인 인스턴스를 생성한다.
>>> t2 = Clock(hour = 13)         # 시간이 13:00:00인 인스턴스를 생성한다.
>>> t1 > t2                       # 11:59:59는 13:00:00보다 큰가?
False
>>> t1 + t2                       # 11:59:59 + 13:00:00 = 24:59:59, 즉 익일 00:59:59가 된다.
0:59:59AM
>>> t1 - t2                       # 11:59:59 - 13:00:00 = -01:00:01, 즉 시차는 1시간 1초이다.
1:0:1am
>>> t1 -= t2                      # [증강 할당] 11:59:59 - 13:00:00 = 01:00:01
>>> t1
Clock<1:0:1>AM
>>> t2 += t1                      # 13:00:00 + 01:00:01 = 14:00:01, 즉 02:00:01PM이다.
>>> print(t2)
2:0:1PM
```

위의 내용을 포함하는 Clock 클래스를 'mytime.py'의 Time 클래스 아래에 정의해서 저장하도록 한다. 다음 코드는 주석을 생략했지만 소스 코드에는 주석으로 코드를 설명했으니 필요한 경우에 내려받아 참고하기 바란다.

```
class Clock(Time):
    # --- 초기화 메소드 ------------------------------------------------------ #
    def __init__(self, hour=0, minute=0, second=0):
        super().__init__(hour, minute, second)

    # --- 특수 메소드 -------------------------------------------------------- #
    def __str__(self):
        if self.hour < 12:
            return f'{self.hour!s}:{self.minute!s}:{self.second!s}AM'
        else:
            return f'{self.__getclockhour()!s}:{self.minute!s}:{self.
second!s}PM'

    def __repr__(self):
        if self.hour < 12:
            return (f'{self.__class__.__name__}'
                    f'<{self.hour!r}:{self.minute!r}:{self.second!r}>AM')
        else:
            return (
                f'{self.__class__.__name__}'
                f'<{self.__getclockhour()!r}:{self.minute!r}:{self.
second!r}>PM')

    # --- 내부 전용 메소드 ---------------------------------------------------- #
    def __getclockhour(self):
        return self.hour if self.hour <= 12 else self.hour % 12
```