Pythonic - 파이썬스러운 코드

프로그래밍에서 관용구(idiom) 은 특정 작업을 수행하기 위해 코드를 작성하는 특별한 방법이다.

매번 동일한 구조를 반복하고 따르는 것이 일반적이다.

언어와 무관한 고차원의 개념인 디자인 패턴과 달리 관용구는 특정 작업을 할 때 사용할수 있는 실제 코드이다.

관용구는 코드이므로 언어에 따라 다르다. 이 관용구를 따른 코드를 관용적이라 부르고, 파이썬에서는 특히 `Pythonic` (파이썬스럽다)이라고 한다.

관용적인 방식으로 코드를 작성했을 때 일반적으로 더 나은 성능을 낸다. 또한 코드도 더 작고 이해하기 쉽다.

목표:

- 인덱스와 슬라이스를 이해하고 인덱싱 가능한 객체를 올바른 방식으로 구현하기
- 시퀀스와 이터러블 구별하기
- 컨텍스트 관리자를 만드는 모범사례 연구 그리고 어떻게 효율적으로 작성할 수 있는지
- 매직 메서드를 사용해 보다 관용적인 코드 구현
- 파이썬에서 부작용을 유발하는 흔한 실수 피하기

인덱스와 슬라이스

일반적인 언어와 같이 첫번째 요소의 인덱스는 0부터 시작한다.

파이썬은 다른 언어와 다른 방법으로 접근하는 것을 지원한다.

```
>>>> my_numbers=(4,5,3,9)
>>>> my_numbers[-1]
9
```

```
>>> my_numbers=(1,2,3,4,5,6,7,8)
>>> my_numers[1:4]
(2,3,4)
```

slice의 시작 인덱스는 포함, 끝 인덱스는 제외하고 선택한 구간의 값을 가져온다는 것에 유의하자

```
>>> my_numbers[:3]
(1,2,3)
>>> my_numbers[1:]
(2,3,4,5,6,7)
>>> my_numbers[::] # my_numbers[:]도 마찬가지로 복사본을 만든다
(1,2,3,4,5,6,7)
>>> my_numbers[1:6:2]
(2,4)
```

위의 모든 예제는 실제로는 slice 함수에 파라미터를 전달하는 것과 같다.

slice함수는 파이썬의 내장 객체이므로 다음과 같이 직접 호출할 수 있다.

```
>>> interval=slice(1,7,2)
>>> my_numbers[interval]
(2,4,6)
```

```
>>> interval=slice(None,3)
>>> my_numbers[interval]==my_numvers[:3]
True
```

slice의 (시작,중지,간격)중 지정하지 않은 파라미터는 None으로 간주한다.

→ 튜플,문자열,리스트의 특정 요소를 가져오려고 한다면 for 루프를 돌며 수작업으로 요소를 선택하지 말고 이와 같은 방법을 사용하는 것이 좋다.

자체 시퀀스 생성

방금 설명한 기능은 `__getitem__` 이라는 매직 메서드(파이썬에서 특수한 동작을 수행하기 위해 예약한 메서드로 이중 언더바로 둘러싸여 있다.) 덕분에 동작한다.

이건은 myobject[key]와 같은 형태를 사용할 때 호출되는 메서드로 key에 해당하는 대괄호 안의 값을 파라미터로 전달한다.

특히 시퀀스는 `__getitem__` 과 `__len__` 을 모두 구현한 객체이므로 반복이 가능하다.

리스트, 튜플과 문자열은 표준 라이브러리에 있는 시퀀스 객체의 예이다.

이 섹션에서는 시퀀스나 이터러버블 객체를 만들지 않고 키로 객체의 특정 요소를 가져오는 방법에 대해 다룬다.

업무 도메인에서 사용하는 사용자 정의 클래스에 `__getitem__` 을 구현하려는 경우 파이썬스러운 접근 방식을 따르기 위해 몇 가지를 고려해야 한다.

클래스가 표준 라이브러리 객체를 감싸는 래퍼(wrapper)인 경우 기본 객체에 가능한 많은 동작을 위임할 수 있다.

예를 들어, 클래스가 리스트의 래퍼인 경우 리스트의 동일한 메서드를 호출하여 호환성을 유지할 수 있다.

다음 클래스는 객체가 어떻게 리스트를 래핑하는지 보여준다.

```
from collections.abc import Sequence

class items:
    def __init__(self,*values):
        self._values=list(values)
```

```
    def __len__(self):
        return len(self,values)

    def __getitem__(self,item):
        return self._values.__getitem__(item)
```

필요한 메서드가 있는 경우 그저 list 객체에 있는 동일한 메서드에 위임하면 된다.

이 예제에서는 컴포지션을 사용한다. 왜냐하면 내부적으로 list 클래스를 상속받지 않고 직접 작성한 구현체를 가지고 있기 때문이다. 다른 방법으로 상속을 사용할 수 있다.

이 경우는 collections.UserList 부모 클래스를 상속해야 한다.

만약 래퍼도 아니고 내장 객체를 사용하지도 않은 경우는 자신만의 시퀀스를 구현할 수 있다. 이때는 다음 사항에 유의해야 한다.

- 범위로 인덱싱한 결과는 해당 클래스와 같은 타입의 인스턴스여야 한다.
- slice에 의해 제공된 범위는 파이썬이 하는 것처럼 마지막 요소는 제외해야 한다.

컨텍스트 관리자

사전 조건과 사후 조건이 있는 일부 코드를 실행해야 하는 상황이 있다.

즉, 어떤 중요한 작업 전후에 실행을 하려는 것이다. 컨텍스트 관리자는 이러한 상황에서 사용할수 있는 가장 훌륭한 도구이다.

일반적으로 리소스 관리와 관련하여 컨텍스트 관리자를 자주 볼 수 있다. 예를 들어 일단 파일을 열면 파일 디스크립터 누수를 막기 위해 작업이 끝나면 적절히 닫히길 기대한다.

이 경우에 일반적으로 할당된 모든 리소스를 해제해야 한다. 모든 것이 잘 처리되었을 경우의 해제는 쉽지만 예외가 발생하거나 오류를 처리해야 한느 경우는 어떻게 될까?

가능한 모든 조합과 실행 경로를 처리하여 디버깅하는 것이 어렵다는 것을 감안할때 가장 일반적인 방법은 finally 블록에 정리 코드를 넣는 것이다.

```
fd=open(filename)
try:
    process_file(fd)
finally:
```

```
    fd.close()
```

똑같은 기능을 매우 우아하고 파이썬스러운 방법으로 구현할 수 있다.

```
with open(filename) as fd:
    process_file(fd)
```

with 문 (PEP-343)은 컨텍스트 관리자로 진입하게 한다. 이 경우 open 함수는 컨텍스트 관리자 프로토콜을 구현한다.

즉 예외가 발생한 경우에도 블록이 완료되면 파일이 자동으로 닫힌다.

컨텍스트 관리자는 `__enter__` 와 `__exit__` 두 개의 매직 메서드로 구성된다.

첫번째 줄에서 with문은 `__enter__` 메서드를 호출하고 이 메서드가 무엇을 반환하든 as 이후에 지정된 변수에 할당된다.

`__enter__` 메서드가 특정한 값을 반환할 필요는 없다. 설사 값을 반환하더라도 필요하지 않으면 변수에 할당하지 않아도 된다.

이 라인이 실행되면 다른 파이썬 코드가 실행될 수 있는 새로운 컨텍스트로 진입한다.

해당 블록에 대한 마지막 문장이 끝나면 컨텍스트가 종료되며 이는 파이썬이 처음 호출한 원래 컨텍스트 객체의 `__exit__` 메서드를 호출함을 의미한다.

컨텍스트 관리자 블록 내에 예외 또는 오류가 있는 경우에도 `__exit__` 메서드가 여전히 호출되므로 정리 조건을 안전하게 실행하는데 편하다.

실제로 `__exit__` 메서드는 블록에서 예외가 발생한 경우 해당 예외를 파라미터로 받기 때문에 임의의 방법으로 처리할 수 있다.

블록의 전후에 필요로 하는 특정 로직을 제공하기 위해 자체 컨텍스트 관리자를 구현할 수도 있다.

컨텍스트 관리자는 관심사를 분리하고 독립적으로 유지되어야 하는 코드를 분리하는 좋은 방법이다. 왜냐하면 이들을 섞으면 로직을 관리하기가 더 어려워지기 때문이다.

예를 들어 스크립트를 사용해 db 백업을 하려는 경우를 생각해보자. 주의 사항은 백업은 오프라인 상태에서 해야 한다는 것이다.

db가 실행되고 있지 않은 동안에만 백업을 할 수 있으며, 이를 위해 서비스를 중지해야 한다. 백업이 끝나면 백업 프로세스가 성공적으로 진행되었는지와 관계없이 프로세스를 다시 시작해야 한다.

첫번째 방법은 서비스를 중지하고 백업을 하고 예외 및 특이 사항을 처리하고 서비스를 다시 시작하는 거대한 단일 함수를 만드는 것이다.

진짜 이렇게 구현하는 경우가 있기 때문에 컨텍스트 관리자를 사용한 문제 해결법을 제시하지 않고 좀 더 자세히 살펴보겠다.

```
def stop_database():
    run("systemctl stop postgresql.service")

def start_database():
    run("systemctl start postgresql.service")

class DBHandler:
    def __enter__(self):
        stop_database()
        return self
    def __exit__(self,exc_type,ex_value, ex_traceback):
        start_database()

    def db_backup():
        run("pg_dump database")

    def main():
        with DBHandler():
            db_backup()
```

이 예제에서는 DBHandler를 사용한 블록 내부에서 컨텍스트 관리자의 결과를 사용하지 않았다. 적어도 이 경우의 `__enter__` 의 반환값은 쓸모가 없다.

컨텍스트 관리자를 디자인할 때 블록이 시작된 후에 무엇이 필요한지 고려해야 한다. 일반적으로 필수는 아니지만 `__enter__`에서 무언가를 반환한느 것이 좋은 습관이다.

main() 함수에서는 유지보수 작업과 상관 없이 백업을 실행한다. 또한 백업에 오류가 있어도 여전히 `__exit__` 를 호출한다.

`__exit__` 메서드의 서명을 주목할 필요가 있다. 블록에서 발생한 예외를 파라미터로 받는다. 블록의 예외가 없으면 모두 None이다.

`__exit__` 의 반환값을 잘 생각해야 한다. 특별한 작업을 할 필요가 없다면 아무것도 반환하지 않아도 된다.

만약 `__exit__` 가 True를 반환한다면 잠재적으로 발생한 예외를 호출자에게 전파하지 않고 멈춘다는 것을 뜻한다. 일반적으로 발생한 예외를 삼키는 것은 좋지않은 습관이다.

컨텍스트 관리자 구현

앞의 예제와 같은 방법으로 컨텍스트 관리자를 구현 할 수 있다. `__enter__` 와 `__exit__` 매직 매서드만 구현하면 해당 객체는 컨텍스트 관리자 프로토콜을 지원할 수 있다.

이렇게 컨텍스트 관리자를 구현하는 것이 일반적인 방법이지만 유일한 방법은 아니다.

contextlib 모듈을 통해 보다 쉽게 구현하는 방법을 살펴볼 것이다.

함수에 contextlib.contextmanager 데코레이터를 적용하면 해당 함수의 코드를 컨텍스트 관리자로 변환한다.

함수는 제너레이터라는 특수한 함수의 형태여야 하는데 이 함수는 코드의 문장을 `__enter__` 와 `__exit__` 매직 메서드로 분리한다.

```
import contextlib

@contextlib.contextmanager
def db_handler():
    try:
        stop_database()
        yield

    finally:
        start_database()

with db_handler():
    db_backup()
```

먼제 제너레이터 함수를 정의하고 @contextlib.contextmanager 데코레이터를 적용했다. 이 함수는 yield 문을 사용했으므로 제너레이터 함수가 된다.

데코레이터를 정용하면 yield 문의 모든 것은 `__enter__` 메서드의 일부처럼 취급된다는 것이다. 여기서 생성된 값은 컨텍스트 관리자의 평가 결과로 사용된다.

`__enter__` 메서드의 변환 값과 같은 역할을 하는 것으로 as x: 와 같은 형태로 변수에 할당할 수 있다. 이번 경우는 yield 문에서 아무것도 반환하지 않았다. 이것은 암묵적으로 None을 반환하는 것과 같다.

그러나 컨텍스트 관리자의 블록 내에서 사용하기를 원하는 경우 반환 값을 지정할 수 있다.

이 지점에서 제너레이터 함수가 중단되고 컨텍스트 관리자로 진입하여 데이터베이스의 백업 코드가 실행된다. 이 작업이 완료되면 다음 작업이 이어서 실행되므로 yield 문 다음에 오는 모든 것들을 `__exit__` 로직으로 볼수 있다.

이렇게 컨텍스트 매니저를 작성하면 기존 함수를 리팩토링하기 쉬운 장점이 있다. 일반적으로 어느 특정 객체에도 속하지 않은 컨텍스트 관리자가 필요한 경우 좋은 방법이다.

그렇지 않으면 객체 지향적인 의미에서 보면 아무런 목적을 가지지 않은 “가짜” 부모 클래스를 생성해야 한다.

많은 상태를 관리할 필요가 없고 다른 클래스와 독립되어 있는 경우라면 컨텍스트 관리자를 만드는 것이 좋은 방법이다.

컨텍스트 관리자를 구현할 수 있는 더 많은 방법이 있으며, 이것 역시 표준 라이브러리인 contextlib 패키지에 있다.

또 다른 도우미 클래스는 `contextlib.Contextdecorator` 이다. 이 클래스는 컨텍스트 관리자 안에서 실행될 함수에 데코레이터를 적용하기 위한 로직을 제공하는 믹스인 클래스이다.

(믹스인 클래스는 다른 클래스에서 필요한 기능만 섞어서 사용될수 있도록 메서드만을 제공하는 유틸리티 형태의 클래스이다.)

이렇게 하려면 ContextDecorator를 상속하여 새로운 클래스를 만들고 매직 메서드에 필요한 로직을 구현하면 된다.

```
class dbhandler_decorator(contextlib.ContextDecorator):
    def __enter__(self):
        stop_database()
        return self

    def __enter__(self,ext_type, ex_value, ex_traceback):
        start_database()
```

```
@dbhandler_decorator()
    def offline_backup():
        run("pg_dump database")
```

데코레이터를 사용하면 로직을 한번만 정의하면 된다는 장점이 있다. 변하지 않는 동일한 로직을 필요한 곳에 원하는 만큼 재사용할 수 있다.

이처럼 컨텍스트 관리자는 파이썬을 차별화하는 상당히 독특한 기능이다,

컴프리헨션(Comprehension)과 할당 표현식

컴프리헨션을 사용하면 코드를 보다 간결하게 작성할 수 있고, 일반적으로 가독성도 높아지기 때문이다.

그렇지만 수집한 데이터에 대해서 어떤 변환을 해야 하는 경우라면 오히려 코드가 더 복잡해질 수 있다.

이런 경우에는 간단하게 for 루프를 사용하는 것이 더 나은 선택이다. 그러나 이런 상황에서 적용할수 있는 할당 표현식이 있다.

여러 연산이 아니라 단일 명령어로 데이터 구조를 생성하려면 컴프리헨션을 사용하는 것이 좋다.

예를 들어, 다음과 같이 어떤 숫자들에 대해서 단순 계산이 포함된 목록을 만들고 싶다면

```
numbers=[]
for i in range(10):
    numbers.append(run_calculation(i))
```

다음과 같이 바로 리스트를 만들수 있다.

```
numbers=[run_caculation(i) for i in range(10)]
```

이러한 형식으로 작성된 코드는 list.append 를 반복적으로 호출하는 대신 단일 파이썬 명령어를 호출하므로 일반적으로 더 나은 성능을 보인다.

클라우딩 컴퓨팅 환경에서 ARN 같은 문자열을 받아서 해당 리소스의 계정 정보를 반환하는 함수를 생각해보자

```
from typing import Iterable,Set

def collect_account_id_from_arns(arns: Iterable[str]) -> Set[
"""
arn:partition:service:region:account-id:resource-id 형태의 arn
고유한 계정ID 를 찾아서 반환
"""
    collected_account_ids=set()
    for arn in arns
        matched=re.match(ARN_REGEX,arn)
        if matched_is not None:
            account_id=matched.groupdict()["account_id"
            collected_account_ids.add(account_id)

    return collected_account_ids
```

```
def collected_account_ids_from_arn(arns):
    matched_arns= filter(None, (re.match(ARN_REGEX,arn) for a
    return {m.groupdict()["account_id"] for m in matched_arns
```

```
def collect_account_ids_from_arns(arns: Iterable[str])->Set[s
    return {
        matched.groupdict()["account_id"]
        for arn in arns
        if (matched:=re.match(ARN_REGEX,arn)) is Not None
        }
```

할당 표현식을 사용하는 또 다른 이유는 성능 때문이다. 어떤 변환 작업을 위해 호출하는 경우 필요 이상으로 호출되기를 원하지 않을 것이다.

뿐만 아니라 할당 표현식에서 함수의 결과를 임시 식별자에 할당하는 것은 코드의 가독성을 높이는 좋은 최적화 기술 중 하나이다.

더 간결한 코드가 항상 더 나은 코드를 의미하는 것은 아니라는 것은 아니라는 것을 명심하자.

프로퍼티, 속성(Attribute)과 객체 메서드의 다른 타입들

public,private,protected 와 같은 접근 제어자를 가지는 다른 언어들과는 다르게 파이썬 객체의 모든 속성과 함수는 퍼블릭이다.

즉, 호출자가 객체의 속성을 호출하지 못하도록 할 방법이 없다.

엄격한 강제상황은 아니지만 파이썬에는 변수명 지정과 관련하여 몇가지 네이밍 컨벤션이 있다.

그 중에 하나로 밑줄로 시작하는 속성은 private 속성을 의미하는 것이 있는데, 외부에 호출되지 않기를 기대한다는 의미이다.

그러나 이것이 외부로부터의 호출을 금지시켜주는 것은 아니라는 것에 주의하자.

파이썬에서의 밑줄

파이썬에서 밑줄을 사용하는 몇가지 규칙과 구현 세부 사항이 있다.

```
class Connector:
    def __init__(self,source):
        self.source=source
        self._timeout=60

conn=Connector("postgresql://localhost")
>>> conn.source
'postgresql://localhost'
>>> conn._timeout
60
>>> conn.__dict__
```

```
{'source':'postgresql://localhost' ,'_timeout':60"}
```

이 코드를 해석해보면 _timeout은 connector 자체에서만 사용되고 호출자는 이 속성에 접근하지 않아야 한다.

즉, timeout 속성은 내부에서만 사용하고 바깥에서는 호출하지 않을 것이므로 외부 인터페이스를 고려하지 않고 언제든 안전하게 리팩토링 할 수 있다.

너무 많은 내부 메서드와 속성을 사용하는 것은 해당 클래스가 너무 많은 일을 하고 있고 단일 책임 원칙을 준수하지 않았다는 신호일 수 있다

단일 밑줄을 접두사로 사용하는 것은 객체의 인터페이스를 명확하게 구분하는 파이썬스러운 방법이다.

그러나 일부 속성과 메서드를 실제로 private로 만든다는 오해가 있다.

timeout 속성을 이중 밑줄로 정의했다고 가정해 보자.

```
class Connector:
    def __init__(self,source):
        self.source=source
        self.__timeout=60
    def connect(self):
        print("connecting with {0}s".format(self.__timeout))

>>> conn= Connector("postgresql://localhost")
>>> conn.connect():
connecting with 60s
>>conn.__timeout
Traceback (most recent cll last):
    File "<stdin>, line 1, in <module>
    AttributeError: 'connector' object hs no attribute '__tim
```

밑줄 두개를 사용하면 파이썬은 다른 이름을 만든다. **이름 맹글링**(name mangling) 이라는 것으로

이중 밑줄을 사용한 변수의 이름을 `“_<class_name>__<attribute_name>”` 형태로 변경하는 것이다.

여기서는 `_connector__timeout` 이라는 속성이 만들어지며 이 속성은 다음과 같이 접근할 수 있다.

```
>>> vars(conn)
{'source': :'postgresql://localhost', '_Connector_timeout':60
>>>conn._Connector__timeout
60
>>> conn._Connector__timeout = 30
>>>conn.connect()
connecting with 30s
```

파이썬에서 이중 밑줄을 사용하는 것은 완전히 다른 경우를 위한 것이다.

여러번 확장되는 클래스의 메서드를 이름 충돌 없이 오버라이드하기 위해 만들어졌다.

지금 예제가 이러한 매커니즘을 사용하기 위한 것이라고 정당화하기에는 거리가 멀다.

이중 밑줄은 파이썬스러운 코드가 아니다. 속성을 private로 정의하려는 경우 하나의 밑줄을 사용하고 파이썬스러운 관습을 지키도록 해야 한다.

반대의 경우, 즉 객체의 일부 속성을 public으로 공개하고 싶은 경우에 대해서 살펴보자. 이런 경우에는 일반적으로 다음 섹션에서 살펴볼 프로퍼티를 사용한다.

프로퍼티

일반적으로 객체 지향 설계에서는 도메인 엔티티를 추상화하는 객체를 만든다. 이러한 객체는 어떤 동작이나 데이터를 캡슐화할 수 있다.

그리고 종종 데이터의 정확성이 객체를 생성할 수 있는지 여부를 동작한다.

다시 말하면 일부 엔터티는 데이터가 특정 값을 가질 경우에만 존재할 수 있고, 잘못된 값을 가진 경우에는 존재할 수 없다.

이것이 유효성 검사 메서드를 만드는 이유이다. 그러나 파이썬에서는 프로퍼티를 사용해 setter와 getter 메서드를 더 간결하게 캡슐화 할 수 있다.

좌표 값을 처리하는 지리 시스템을 생각해보자. 위도와 경도는 특정 범위에서만 의미가 있다.

해당 범위를 벗어나는 좌표는 존재할 수 없다. 좌표를 나타내는 객체를 생성할 수 있지만, 어떤 값을 사용할 때는 항상 허용 가능한 범위 내에 있는지 확인해야 한다.

이런 경우 프로퍼티를 사용할 수 있다.

```
class Coordinate:
    def __init__(self,lat:float, long: float) -> None:
        self._latitude=self_longitude=None
        self.latitude=lat
        self.longitude=long

    @property
    def latitude(self) -> float:
        return self._latitude

    @latitude.setter
    def latitude(self,lat_value:float) ->None:
        if lat_value not in range(-90,90+1):
            raise ValueError(f"유효하지 않은 위도 값: {lat_value}
        self._latitude=lat_value

    @property
    def longitude(self) -> float
        return self._longitude

    @longitude.setter
    def longitude(self,long_value:float) ->None:
        if long_value not in range(-180,180+1):
            raise ValueError(f"유효하지 않은 경도 값: {long_value
        self._longitude=long_value
```

여기에서 프로퍼티는 latitude와 longitude를 정의하기 위해 사용했다. 이렇게 함으로써 private 변수에 저장된 값을 반환하는 별도의 값을 만들었다.

더 중요한 것은 사용자가 이러한 속성 중 하나를 수정하려는 경우다.

```
coordinate.latitude=<-new-latitude-value>
```

지금까지 내부 데이터를 일관성이 있고 투명하게 관리하기 위해 프로퍼티가 어떻게 도움이 되는지 알아봤다.

그러나 때로는 내부 데이터에 따라 어떤 계산을 하고 싶은 경우가 있을 수도 있다. 이런 경우에도 프로퍼티가 좋은 선택이다.

예를들어 특정 포맷이나 데이터 타입으로 값을 반환해야 하는 객체가 있는 경우 프로퍼티를 사용할 수 있다.

소수점 이하 네 자리까지의 좌표 값을 반환하기로 결정했다면 값을 읽을때 호출되는 @property 메소드에서 반올림하는 계산을 만들 수 있다.

프로퍼티는 명령-쿼리 분리 원칙(command and query seperation -CC08) 을 따르기 위한 좋은 방법이다.

명령 쿼리 분리 원칙은 객체의 메서드가 무언가의 상태를 변경하는 커맨드이거나 무언가의 값을 반환하는 커리이거나 둘중 하나만 수행해야 한다는 것이다.

객체의 메서드가 무언가 기능을 수행하면서 동시에 질문에 대답하기 위해 상태를 반환한다면

이는 동시에 두가지 작업을 하고 있는 것이고 명령-쿼리 분리 원칙을 위배하는 것이므로 반드시 하나만 하도록 수정해야 한다.

메서드 이름에 따라 실제 코드가 무엇을 하는지 혼돈스럽고 이해하기가 어려운 경우가 있다.

예를 들어 set_email이라는 메서드를 if self.set_email(”a@j.com”) 처럼 사용했다면 이 코드는 무엇을 의미하는 것일까?

프로퍼티를 사용한다면 이런 종류의 혼동을 피할 수 있다. @property 데코레이터는 무언가에 응답하기 위한 쿼리이고, @<property_name>.setter는 무언가를 하기 위한 커맨드이다.

이 예제를 착안할 수 있는 또 다른 조언 하나는 한 메서드에서 한가지 이상의 일을 하지 말라는 것이다.

무언가를 할당하고 유효성 검사를 하고 싶으면 두개 이상의 문장으로 나누어야 한다.⇒ 하나의 setter와 getter를 가져야 한다.

왜냐하면 객체의 현재 상태를 구할 때마다 부작용 없이 현재 상태를 그대로 반환해야 하기 때문이다.

보다 간결한 구문으로 클래스 만들기

파이썬에서는 객체의 값을 초기화하는 일반적인 보일러플레이트 코드가 있다.

`__init__` 메서드에 객체에서 필요한 모든 속성을 파라미터로 받은 다음 내부 변수에 할당하는 것이다.

```
class MyClass:
    def __init__(self,x,y,...):
        self.x=x
        self.y=y
```

daatclasses 모듈을 사용하여 위 코드를 훨씬 단순화할 수 있다.(PEP-557)

dataclasses 모듈은 @dataclass 데코레이터를 제공한다.

이 데코레이터를 클래스에 적용하면 모든 클래스에 속성에 대해서 마치 `__init__` 메서드에서 정의한 것처럼 인스턴스 속성으로 처리한다.

@dataclass 데코레이터를 사용하면 `__init__` 메서드를 자동으로 생성하므로 또 다시 `__init__` 메서드를 구현할 필요가 없다.

또한 field 라는 객체도 제공한다. 이 field 객체는 해당 속성에 특별한 특징이 있음을 표시한다. 예를 들어, 속성 중 하나가 리스트처럼 변경 가능한 데이터 타입인 경우, `__init__` 에서 비어있는 리스트를 할당할 수 없고, 대신에 None으로 초기화한 다음에 인스턴스마다 적절한 값으로 다시 초기화를 해야 한다.

```
@dataclass
class Myclass:
    list1 :list =field(default_factory=list)
    int1:int
    size=R
```

field 객체를 사용하면 default_factory 파라미터에 list 객체를 전달하여 초기값을 지정할 수 있다.

default_factory에 전달되는 객체는 호출 가능한 객체여야하고, 초기화 시 특별한 값을 지정하지 않는다면 비어있는 인자와 함께 해당 객체를 호출한다.

마지막 속성인 size에 주의하자. 이 멤버는 어노테이션이 없으므로 일반적인 클래스 속성으로 처리된다. 즉 모든 객체가 모든 값을 공유한다.

int1 은 어노테이션이 있다. 정수형이지만 기본 값은 가지고 있지 않으므로 객체 생성시 반드시 값을 정해줘야한다.

복잡하게 유효성 검사를 하거나 특별한 변환을 하지 않는 데이터를 저장하려는 경우 이러한 데이터 클래스가 좋은 대안이 될 수 있다.

다만, 어노테이션이 데이터 변환을 해주지는 않는점을 명심하자. 예를들어 float 타입이거나 integer 타입이어야 한다면 `__init__` 메서드 안에서 이 변환을 해야 한다.

어노테이션만을 사용해서 데이터 클래스로 구현하면 나중에 발견하기 힘든 미묘한 오류를 일으킬 수 있다.

즉 `__init__` 메서드 안에서 별도의 처리를 하거나, 유효성 검사가 엄격하게 필요하지 않은 경우에 데이터 클래스를 사용하는 것이 적합하다.

아마도 데이터 컨테이너나 래퍼 클래스의 용도로 사용되는 모든 경우에 데이터클래스가 유용할 것이다.

이터러블 객체

파이썬에서는 기본적으로 반복 가능한 객체가 있다. 예를 들어 리스트, 튜플, 세트 및 딕셔너리는 특정한 형태의 데이터를 보유할 수 있을 뿐만 아니라 for 루프를 통해 반복적으로 값을 가져오는데 사용할 수 있다.

파이썬의 반복은 이터러블 프로토콜이라른 자체 프로토콜을 사용해 동작한다. **for e in myobject:** 형태로 객체를 반복할수 있는지 확인하기 위해 파이썬은 고수준에서 다음 두 가지를 차례로 검사한다.

- 객체가 `__next__` 나 `__iter__` 이터레이터 메서드 중 하나를 포함하는지 여부
- 객체가 시퀀스이고 `__len__` 과 `__getitem__` 을 모두 가졌는지 여부

즉, fallback 메커니즘으로 시퀀스도 반복을 할수 있으므로 for 루프에서 반복 가능한 객체를 만드는 방법은 두가지가 있다.

이터러블 객체 만들기

객체를 반복하려고 하면 파이썬은 해당 객체의 iter() 함수를 호출한다. 이 함수가 처음으로 하는 것은 해당 객체의 `__iter__` 메서드가 있는지를 확인하는 것이다. 만약 있으면, `__iter__` 메서드를 실행한다.