UNIT 39

이터레이터 사용하기

39 이터레이터 사용하기

이터레이터 사용하기

- 이터레이터(iterator)는 값을 차례대로 꺼낼 수 있는 객체(object)임
- 파이썬에서는 이터레이터만 생성하고 값이 필요한 시점이 되었을 때 값을 만드는 방식을 사용함
- 데이터 생성을 뒤로 미루는 것인데 이런 방식을 지연 평가(lazy evaluation)라고 함
- 이터레이터는 반복자라고 부르기도 함

39.1 반복 가능한 객체 알아보기

반복 가능한 객체 알아보기

- 반복 가능한 객체는 말 그대로 반복할 수 있는 객체인데 우리가 흔히 사용하는 문자열, 리스트, 딕셔너리, 세트가 반복 가능한 객체임
- 요소가 여러 개 들어있고, 한 번에 하나씩 꺼낼 수 있는 객체임
- 객체가 반복 가능한 객체인지 알아보는 방법은 객체에 __iter__ 메서드가 들어있는지 확인해보면 됨

• **dir(객체)**

```
>>> dir([1, 2, 3])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__forma
t__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '_
_init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__
reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '_
_subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

```
>>> [1, 2, 3].__iter__()
<list_iterator object at 0x03616630>
```

39.1 반복 가능한 객체 알아보기

반복 가능한 객체 알아보기

- 리스트의 이터레이터를 변수에 저장한 뒤 __next__ 메서드를 호출해보면 요소를 차례대로 꺼낼 수 있음

```
>>> it = [1, 2, 3].__iter__()
>>> it.__next__()
1
>>> it.__next__()
2
>>> it.__next__()
3
>>> it.__next__()
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    it.__next__()
StopIteration
```

- 이터레이터는 __next__로 요소를 계속 꺼내다가 꺼낼 요소가 없으면 StopIteration 예외를 발생시켜서 반복을 끝냄

39.1 반복 가능한 객체 알아보기

» 반복 가능한 객체 알아보기

- 리스트뿐만 아니라 문자열, 딕셔너리, 세트도 __iter__를 호출하면 이터레이터가 나옴
- 이터레이터에서 __next__를 호출하면 차례대로 값을 꺼냄

```
>>> 'Hello, world!'.__iter__()
<str_iterator object at 0x03616770>
>>> {'a': 1, 'b': 2}.__iter__()
<dict_keyiterator object at 0x03870B10>
>>> {1, 2, 3}.__iter__()
<set_iterator object at 0x03878418>
```

- 리스트, 문자열, 딕셔너리, 세트는 요소가 눈에 보이는 반복 가능한 객체임

39.1 반복 가능한 객체 알아보기

반복 가능한 객체 알아보기

- 다음과 같이 range(3)에서 __iter__로 이터레이터를 얻어낸 뒤 __next__ 메서드를 호출해보자

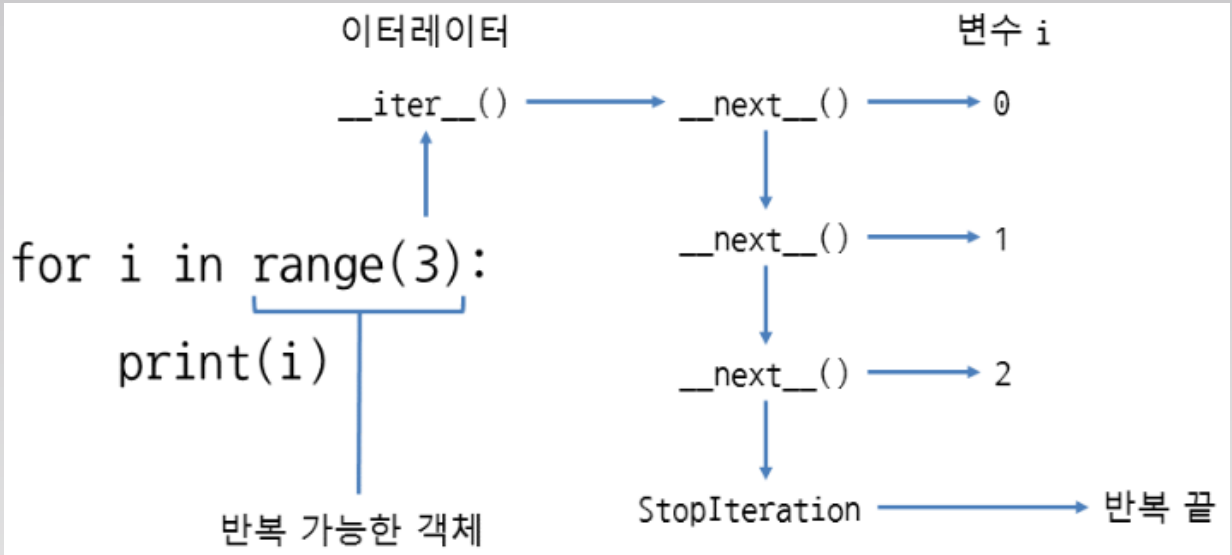
```
>>> it = range(3).__iter__()
>>> it.__next__()
0
>>> it.__next__()
1
>>> it.__next__()
2
>>> it.__next__()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    it.__next__()
StopIteration
```

39.1 반복 가능한 객체 알아보기

for와 반복 가능한 객체

- 다음과 같이 for에 range(3)을 사용했다면 먼저 range에서 __iter__로 이터레이터를 얻음
- 한 번 반복할 때마다 이터레이터에서 __next__로 숫자를 꺼내서 i에 저장하고, 지정된 숫자 3이 되면 StopIteration을 발생시켜서 반복을 끝냄

▼ 그림 for에서 range의 동작 과정



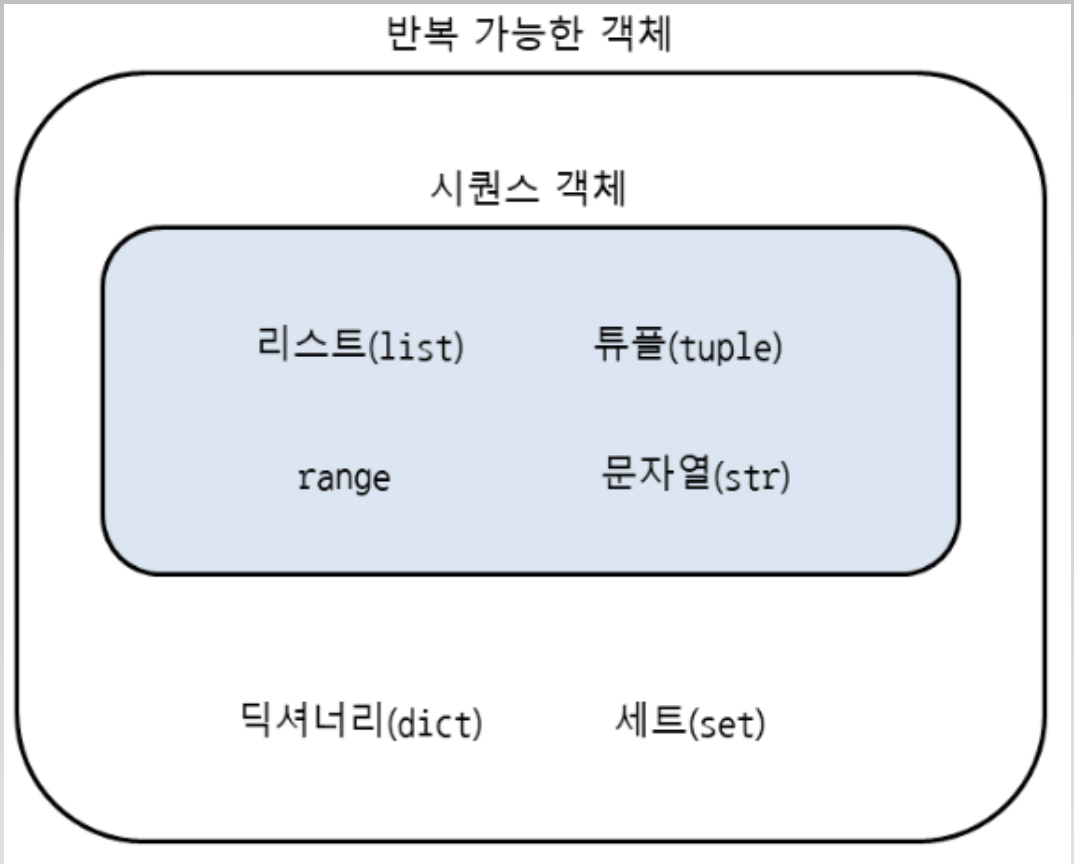
39.1 반복 가능한 객체 알아보기

for와 반복 가능한 객체

- 반복 가능한 객체는 요소를 한 번에 하나씩 가져올 수 있는 객체이고, 이터레이터는 __next__ 메서드를 사용해서 차례대로 값을 꺼낼 수 있는 객체임
- 반복 가능한 객체(iterable)와 이터레이터(iterator)는 별개의 객체이므로 둘은 구분해야 함
- 반복 가능한 객체에서 __iter__ 메서드로 이터레이터를 얻음

39.1 반복 가능한 객체 알아보기

▼ 그림 반복 가능한 객체는 시퀀스 객체를 포함



39.2 이터레이터 만들기

이터레이터 만들기

- 간단하게 range(횟수)처럼 동작하는 이터레이터임

```
class 이터레이터이름:
    def __iter__(self):
        코드

    def __next__(self):
        코드
```

39.2 이터레이터 만들기

이터레이터 만들기

iterator.py

```
class Counter:
    def __init__(self, stop):
        self.current = 0    # 현재 숫자 유지, 0부터 지정된 숫자 직전까지 반복
        self.stop = stop    # 반복을 끝낼 숫자

    def __iter__(self):
        return self         # 현재 인스턴스를 반환

    def __next__(self):
        if self.current < self.stop:    # 현재 숫자가 반복을 끝낼 숫자보다 작을 때
            r = self.current            # 반환할 숫자를 변수에 저장
            self.current += 1           # 현재 숫자를 1 증가시킴
            return r                    # 숫자를 반환
        else:                           # 현재 숫자가 반복을 끝낼 숫자보다 크거나 같을 때
            raise StopIteration         # 예외 발생

for i in Counter(3):
    print(i, end=' ')
```

실행 결과

```
0 1 2
```

39.2 이터레이터 만들기

이터레이터 만들기

- 클래스로 이터레이터를 작성하려면 __init__ 메서드를 만듦

```
def __init__(self, stop):
    self.current = 0    # 현재 숫자 유지, 0부터 지정된 숫자 직전까지 반복
    self.stop = stop    # 반복을 끝낼 숫자
```

- 이 객체는 리스트, 문자열, 딕셔너리, 세트, range처럼 __iter__를 호출해줄 반복 가능한 객체(iterable)가 없으므로 현재 인스턴스를 반환하면 됨
- 이 객체는 반복 가능한 객체이면서 이터레이터임

```
def __iter__(self):
    return self         # 현재 인스턴스를 반환
```

39.2 이터레이터 만들기

이터레이터 만들기

```
    def __next__(self):
        if self.current < self.stop:    # 현재 숫자가 반복을 끝낼 숫자보다 작을 때
            r = self.current            # 반환할 숫자를 변수에 저장
            self.current += 1           # 현재 숫자를 1 증가시킴
            return r                    # 숫자를 반환
        else:                           # 현재 숫자가 반복을 끝낼 숫자보다 크거나 같을 때
            raise StopIteration         # 예외 발생
```

```
for i in Counter(3):
    print(i)
```

- 이터레이터를 만들 때는 __init__ 메서드에서 초깃값, __next__ 메서드에서 조건식과 현재값 부분을 주의해야 함
- 이 부분이 잘못되면 미묘한 버그가 생길 수 있음

39.2 이터레이터 만들기

이터레이터 언패킹

- 다음과 같이 Counter()의 결과를 변수 여러 개에 할당할 수 있음
- 이터레이터가 반복하는 횟수와 변수의 개수는 같아야 함

```
>>> a, b, c = Counter(3)
>>> print(a, b, c)
0 1 2
>>> a, b, c, d, e = Counter(5)
>>> print(a, b, c, d, e)
0 1 2 3 4
```

- 사실 우리가 자주 사용하는 map도 이터레이터임
- a, b, c = map(int, input().split())처럼 언패킹으로 변수 여러 개에 값을 할당할 수 있음

39.3 인덱스로 접근할 수 있는 이터레이터 만들기

» 인덱스로 접근할 수 있는 이터레이터 만들기

```
class 이터레이터이름:
    def __getitem__(self, 인덱스):
        코드
```

iterator_getitem.py

```
class Counter:
    def __init__(self, stop):
        self.stop = stop

    def __getitem__(self, index):
        if index < self.stop:
            return index
        else:
            raise IndexError

print(Counter(3)[0], Counter(3)[1], Counter(3)[2])

for i in Counter(3):
    print(i, end=' ')
```

실행 결과

```
0 1 2
0 1 2
```

39.3 인덱스로 접근할 수 있는 이터레이터 만들기

인덱스로 접근할 수 있는 이터레이터 만들기

- __init__ 메서드부터는 Counter(3)처럼 반복을 끝낼 숫자를 받았으므로 self.stop에 stop을 넣어줌

```
class Counter:
    def __init__(self, stop):
        self.stop = stop            # 반복을 끝낼 숫자
```

- 클래스에서 __getitem__ 메서드를 구현하면 인덱스로 접근할 수 있는 이터레이터가 됨
- Counter(3)과 같이 반복을 끝낼 숫자가 3이면 인덱스는 2까지 지정할 수 있음

```
    def __getitem__(self, index):    # 인덱스를 받음
        if index < self.stop:        # 인덱스가 반복을 끝낼 숫자보다 작을 때
            return index             # 인덱스를 반환
        else:                        # 인덱스가 반복을 끝낼 숫자보다 크거나 같을 때
            raise IndexError         # 예외 발생
```

- 이렇게 하면 Counter(3)[0]처럼 이터레이터를 인덱스로 접근할 수 있음
- 반복할 숫자와 인덱스가 같아서 index를 그대로 반환했지만, index와 식을 조합해서 다른 숫자를 만드는 방식으로 활용할 수 있음

39.4 iter, next 함수 활용하기

iter, next 함수 활용하기

- iter는 객체의 __iter__ 메서드를 호출해주고, next는 객체의 __next__ 메서드를 호출해줌

```
>>> it = iter(range(3))
>>> next(it)
0
>>> next(it)
1
>>> next(it)
2
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    next(it)
StopIteration
```

- iter는 반복 가능한 객체에서 이터레이터를 반환하고, next는 이터레이터에서 값을 차례대로 꺼냄

39.4 iter, next 함수 활용하기

» iter

- iter는 반복을 끝낼 값을 지정하면 특정 값이 나올 때 반복을 끝냄
- 이 경우에는 반복 가능한 객체 대신 호출 가능한 객체(callable)를 넣어줌
- 참고로 반복을 끝낼 값은 sentinel이라고 부르는데 감시병이라는 뜻임
- 반복을 감시하다가 특정 값이 나오면 반복을 끝낸다고 해서 sentinel임

  - **iter(호출가능한객체, 반복을끝낼값)**

- 호출 가능한 객체를 넣어야 하므로 매개변수가 없는 함수 또는 람다 표현식으로 만들어줌

```
>>> import random
>>> it = iter(lambda : random.randint(0, 5), 2)
>>> next(it)
0
>>> next(it)
3
>>> next(it)
1
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    next(it)
StopIteration
```

39.4 iter, next 함수 활용하기

iter

- 다음과 같이 for 반복문에 넣어서 사용할 수도 있음

```
>>> import random
>>> for i in iter(lambda : random.randint(0, 5), 2):
...     print(i, end=' ')
...
3 1 4 0 5 3 3 5 0 4 1
```

- iter 함수를 활용하면 if 조건문으로 매번 숫자가 2인지 검사하지 않아도 되므로 코드가 좀 더 간단해짐

```
import random

while True:
    i = random.randint(0, 5)
    if i == 2:
        break
    print(i, end=' ')
```

39.4 iter, next 함수 활용하기

next

- next는 기본값을 지정할 수 있음
- 기본값을 지정하면 반복이 끝나더라도 StopIteration이 발생하지 않고 기본값을 출력함
- 반복할 수 있을 때는 해당 값을 출력하고, 반복이 끝났을 때는 기본값을 출력함
- 다음은 range(3)으로 0, 1, 2 세 번 반복하는데 next에 기본값으로 10을 지정함

• next(반복가능한객체, 기본값)

```
>>> it = iter(range(3))
>>> next(it, 10)
0
>>> next(it, 10)
1
>>> next(it, 10)
2
>>> next(it, 10)
10
>>> next(it, 10)
10
```