17장. 고급 문법

파이썬 정복

Contents

❖ 목차

1. 반복자

❖ 열거 가능 객체

▪ for 반복문

```
for num in [11, 22, 33]:
    print(n)
```

- 객체 요소를 순서대로 읽는 제어문
- 컨테이너 순회
  - __iter__ 메서드 호출하여 반복자 구하고,
  - 이것이 __next__ 메서드 호출하여 컨테이너 요소 읽으며 이동

▪ 반복가능(Iterable) 객체

- 반복자로 요소를 순서대로 읽을 수 있는 것
- for문과 자주 함께 사용

1. 반복자

foriter

```
nums = [11, 22, 33]
it = iter(nums)
while True:
    try:
        num = next(it)
    except StopIteration:
        break
    print(num)
```

실행결과

```
11
22
33
```

- for문 내부동작 이해하고 필요한 조건 맞추어 반복가능 객체 만들 수 있음
  - 요소 순회할 반복자 제공

- Seq 클래스
  - 일련의 이름 목록 저장해두고 순서대로 하나씩 읽어 줌

1. 반복자

❖ 제너레이터 (Generator)

- 매번 반복자 관련 메서드 작성하는 수고 덜 수 있음
- yield 명령
  - 값 반환
  - return 문과 유사하되 변수의 마지막 값과 상태 저장

generator

```
def seqgen(data):
    for index in range(0, len(data), 2):
        yield data[index:index+2]

solarterm = seqgen("입춘우수경칩춘분청명곡우입하소만망종하지소서대서")
for k in solarterm:
    print(k, end = ',')
```

  - seqgen 제너레이터
    - __iter__, __next__ 메서드 내부 자동생성
    - 인수로 전달받은 문자열 데이터 분리하여 yield 명령으로 리턴

1. 반복자

genexpr

```
data = "인촌우수경칩춘분청명곡우입하소만망종하지소서대서"
for k in (data[index:index+2] for index in range(0, len(data), 2)):
    print(k, end = ',')
```

2. 데코레이터

❖ 일급 시민 (First Class Citizen)

- 파이썬 등 함수형 언어에서의 함수
  - 이름 가짐
  - 다른 변수에 대입할 수 있음
  - 인수로 전달할 수 있음
  - 리턴할 수 있음
  - 컬렉션에 저장할 수 있음

funcvalue

```
def add(a, b):
    print(a + b)

plus = add
plus(1, 2)
```

실행결과

```
3
```

2. 데코레이터

❖ 지역 함수

- 다른 함수 안에 정의되는 도우미 함수
- 함수 내부의 반복되는 코드 통합하여 관리 용이하게 함

localfunc

```
def calcsum(n):
    def add(a, b):
        return a+b

    sum = 0
    for i in range(n+1):
        sum = add(sum, i)
    return sum

print("~10 =", calcsum(10))
```

실행결과

```
~10 = 55
```

2. 데코레이터

- 상호 평등한 관계로 작성할 경우
  - 동작에는 문제 없음
  - calcsum이 add에 종속되며 독립성 떨어지고 재사용 번거로움

```
def add(a, b):
    return a + b

def calcsum(n):
    sum = 0
    for i in range(n + 1):
        sum = add(sum, i)
    return sum
```

2. 데코레이터

❖ 함수 데코레이터 (Decorator)

- 함수에 원하는 코드 추가하는 기법

- 함수 래핑 (Wrapping)
  - 원하는 코드 추가 및 원래 함수 대리호출하여 기능 확장

wrapper

```
def inner():
    print("결과를 출력합니다.")

def outer(func):
    print("-" * 20)
    func()
    print("-" * 20)

outer(inner)
```

실행결과

```
--------------------
결과를 출력합니다.
--------------------
```

2. 데코레이터

- **호출 구문의 비직관성 해결하려면?**

wrapper2

```
def inner():
    print("결과를 출력합니다.")

def outer(func):
    def wrapper():
        print("-" * 20)
        func()
        print("-" * 20)
    return wrapper

inner = outer(inner)
inner()
```

2. 데코레이터

- 내부함수 정의 시 데코레이터 붙임
- @outer 데코레이터
  - inner = outer(inner) 구문으로 함수 포장하여 재정의

decorator

```
def outer(func):
    def wrapper():
        print("-" * 20)
        func()
        print("-" * 20)
    return wrapper

@outer
def inner():
    print("결과를 출력합니다.")

inner()
```

2. 데코레이터

- 예시

tagdeco

```
def para(func):
    def wrapper():
        return "<p>" + str(func()) + "</p>"
    return wrapper

@para
def outname():
    return "김상형"

@para
def outage():
    return "29"

print(outname())
print(outage())
```

실행결과

```
<p>김상형</p>
<p>29</p>
```

2. 데코레이터

- 래핑되는 함수가 인수 가질 경우 대리호출 시에도 인수 그대로 전달

decoarg

```
def para(func):
    def wrapper():
        return "<p>" + str(func()) + "</p>"
    return wrapper

@para
def outname(name):
    return "이름:" + name + "님"

@para
def outage(age):
    return "나이:" + str(age)

print(outname("김상형"))
print(outage(29))
```

실행결과

```
Traceback (most recent call last):
  File "C:/PyStudy/CharmTest/CharmTest.py", line 14, in <module>
    print(outname("김상형"))
TypeError: wrapper() takes 0 positional arguments but 1 was given
```

2. 데코레이터

- wrapper 는 func() 형태로만 대리호출하여 인수 적용되지 않음
  - wrapper가 가변 인수 받아야

decoarg2

```
def para(func):
    def wrapper(*args, **kwargs):
        return "<p>" + str(func(*args, **kwargs)) + "</p>"
    return wrapper

@para
def outname(name):
    return "이름:" + name + "님"

@para
def outage(age):
    return "나이:" + str(age)

print(outname("김상형"))
print(outage(29))
print(outname.__name__)
```

실행결과

```
<p>이름:김상형님</p>
<p>나이:29</p>
wrapper
```

2. 데코레이터

- 마지막 줄 outname 함수의 __name__ 속성이 wrapper로 출력
  - wraps 데코레이터
    - 데코레이터 간 중첩 시 문제를 해결

decoarg3

```
from functools import wraps

def para(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        return "<p>" + str(func(*args, **kwargs)) + "</p>"
    return wrapper

@para
def outname(name):
    return "이름:" + name + "님"

@para
def outage(age):
    return "나이:" + str(age)

print(outname("김상형"))
print(outage(29))
print(outname.__name__)
```

실행결과

```
<p>이름:김상형님</p>
<p>나이:29</p>
outname
```

2. 데코레이터

❖ 클래스 데코레이터

- 객체를 괄호 붙여 호출할 경우 __call__ 특수 메서드가 자동 호출
- __call__메서드에서 원래 함수 호출 전후 추가 작업 활용

classwrapper

```
class Outer:
    def __init__(self, func):
        self.func = func

    def __call__(self):
        print("-" * 20)
        self.func()
        print("-" * 20)

def inner():
    print("결과를 출력합니다.")

inner = Outer(inner)
inner()
```

실행결과

```
--------------------
결과를 출력합니다.
--------------------
```

2. 데코레이터

- inner()
  - 객체를 함수 형식으로 호출하면 래핑한 함수 호출됨
  - 위 구문을 간단하게 줄여 클래스 데코레이터 만듦

classdeco

```
class Outer:
    def __init__(self, func):
        self.func = func

    def __call__(self):
        print("-" * 20)
        self.func()
        print("-" * 20)

@Outer
def inner():
    print("결과를 출력합니다.")

inner()
```

3. 동적 코드 실행

❖ eval

- 문자열 형태로 된 파이썬 표현식 평가하여 결과 반환
- 실시간으로 코드 만들어 실행할 수 있음

eval

```
result = eval("2 + 3 * 4")
print(result)

a = 2
print(eval("a + 3"))

city = eval("['seoul', 'osan', 'suwon']")
for c in city:
    print(c, end = ', ')
```

실행결과

```
14
5
seoul, osan, suwon,
```

3. 동적 코드 실행

❖ repr

- 객체로부터 문자열 표현식을 생성
  - 해석기를 위한 표현식이라는 점에서 str 함수와 차이
  - 이 표현식으로 다시 객체 만들 수 있어야
  - 따옴표까지 문자열에 함께 담아야 유효한 표현식

strrepr

```
print(str(1234), end = ', ')
print(str(3.14), end = ', ')
print(str(['seoul', 'osan', 'suwon']), end = ', ')
print(str('korea'))

print(repr(1234), end = ', ')
print(repr(3.14), end = ', ')
print(repr(['seoul', 'osan', 'suwon']), end = ', ')
print(repr('korea'))
```

실행결과

```
1234, 3.14, ['seoul', 'osan', 'suwon'], korea
1234, 3.14, ['seoul', 'osan', 'suwon'], 'korea'
```

3. 동적 코드 실행

- **문자열 표현식을 문자열로 출력**

```
>>> str('korea')
'korea'
>>> repr('korea')
"'korea'"
```

repreval

```
intexp = repr(1234)
intvalue = eval(intexp)
print(intvalue)

strexp = repr('korea')
strvalue = eval(strexp)
print(strvalue)
```

실행결과

```
1234
korea
```

3. 동적 코드 실행

❖ exec

- 파이썬 코드를 직접 실행하는 함수
- eval 함수는 표현식 평가하나 문장을 실행하는 것은 아님

exec

```
exec("value = 3")
print(value)
exec("for i in range(5):print(i, end = ', ')")
```

실행결과

```
3
0, 1, 2, 3, 4,
```

3. 동적 코드 실행

- 여러 줄 코드의 반복 처리도 가능
  - 계속 실행할 코드 미리 해석해 놓으면 동작 빨라짐
    - compile(source, filename, mode)

compile

```
code = compile("""
for i in range(5):
    print(i, end = ', ')
print()
    """, '<string>', 'exec')

for n in range(10):
    exec(code)
```

Thank You !

파이썬 정복