appendix B
제네레이터

부록B에서는

파이썬에는 '제네레이터'라는 특수한 구조가 있습니다. 사실 이것은 따로 신경 쓰지 않아도 프로그래밍을 할 때 문제는 없지만, 가끔 '이건 되는데, 이건 왜 안되지?'라는 위화감을 느낄 수도 있으므로 간단하게 살펴보도록 하겠습니다.

이터레이터

본문에서 반복문을 살펴보면서 구문을 다음과 같이 설명했습니다. 이때 반복할 수 있는 것을 프로그래밍 용어로 '이터러블Iterable'이라고 합니다. 즉, 이터러블은 내부에 있는 요소들을 차례차례 꺼낼 수 있는 객체를 의미합니다.

구문 반복할 수 있는 것과 함께 사용하는 for 반복문

```
for <반복자> in <반복할 수 있는 것>:
```

리스트, 문자열, 튜플, 딕셔너리 등은 모두 내부에서 요소를 차례차례 꺼낼 수 있으므로 이터러블입니다.

그리고 이터러블 중에서 next() 함수를 적용해서 하나하나 꺼낼 수 있는 요소를 '이터레이터Iterator'라고 부르는데, 이터레이터의 예를 간단하게 살펴보고 이야기를 계속 진행하겠습니다.

코드 B-1 reversed( ) 함수와 이터레이터

```
# 변수를 선언합니다.
numbers = [1, 2, 3, 4, 5, 6]
reversed_numbers = reversed(numbers)

# reversed_numbers를 출력합니다.
print("reversed_numbers:", reversed_numbers)
print("next(reversed_numbers):", next(reversed_numbers))
print("next(reversed_numbers):", next(reversed_numbers))
print("next(reversed_numbers):", next(reversed_numbers))
print("next(reversed_numbers):", next(reversed_numbers))
print("next(reversed_numbers):", next(reversed_numbers))
```

리스트에 reversed() 함수를 사용했을 때, <list_reverseditерator object at (주소)>처럼 출력했던 것을 기억하나요? reversed() 함수의 리턴값이 바로 'reverseditерator'로 '이터레이터'인 것입니다.

이와 같은 이터레이터는 반복문의 매개변수로 전달할 수 있으며, 현재 코드처럼 next() 함수로 사용해 내부의 요소를 하나하나 꺼낼 수 있습니다.

실행 결과

```
reversed_numbers: <list_reverseiterator object at 0x034D21D0>
next(reversed_numbers): 6
next(reversed_numbers): 5
next(reversed_numbers): 4
next(reversed_numbers): 3
next(reversed_numbers): 2
```

for 반복문의 매개변수에 넣으면, 반복을 돌 때마다 next() 함수를 사용해서 요소를 하나하나 꺼내주는 것입니다. 왜 reversed() 함수는 리스트를 바로 리턴해주는 것이 아니라, 이터레이터를 리턴해주는 것일까요?

이는 메모리의 효율성을 위해서입니다. 1만 개의 요소가 들어 있는 리스트를 복제한 뒤 뒤집어서 리턴하는 것보다, 기존에 있던 리스트를 활용해서 작업하는 것이 훨씬 효율적이라고 판단하기 때문입니다.

02 제너레이터 기본

이전 절에서 언급했던 이터레이터를 직접 만들 때는 제너레이터Generator를 사용합니다. 처음 보면 굉장히 당황할 수 있는 코드인데요, 당황하지 말고 제너레이터가 무엇인지 살펴보도록 합시다.

제너레이터와 next( ) 함수

함수 내부에서 yield 키워드를 사용하면 해당 함수는 제너레이터 함수가 되며, 일반 함수와 다른 특성을 가지게 됩니다. 예를 들어 다음 코드를 살펴봅시다.

코드 B-2 제너레이터 함수

```
# 함수를 선언합니다.
def test():
    print("함수가 호출되었습니다.")
    yield "test"

# 함수를 호출합니다
print("A 지점 통과")
test()

print("B 지점 통과")
test()

print(test())
```

yield 키워드를 사용한 함수는 함수를 호출해도 함수 내부의 코드가 실행되지 않습니다. 원래 test() 함수를 호출하면 "**함수가 호출되었습니다**"라는 문자열이 출력되어야 하지만, 출력되지 않습니다.

추가로 함수의 리턴값으로 <generator object test at 0x02F20C90> 등이 출력됩니다.

실행 결과

```
A 지점 통과
B 지점 통과
<generator object test at 0x02F20C90>
```

제너레이터 함수는 제너레이터를 리턴합니다. 출력된 `<generator object test at 0x02F20C90>`이 바로 제너레이터 객체를 의미합니다.

제너레이터 객체는 `next()`라는 함수를 사용해 함수 내부의 코드를 실행하게 됩니다. 이때 `yield` 키워드 부분까지만 실행하며, `next()` 함수의 리턴값으로 `yield` 키워드 뒤에 입력한 값이 출력됩니다.

이론 설명만으로는 이해하기 힘들 수 있는데, 다음 코드의 실행 결과를 간단하게 예측해보고 실제로 출력된 실행 결과를 살펴봅시다.

코드 B-3 제너레이터 객체와 next( ) 함수

```
# 함수를 선언합니다.
def test():
    print("A 지점 통과")
    yield 1
    print("B 지점 통과")
    yield 2
    print("C 지점 통과")

# 함수를 호출합니다.
output = test()

# next() 함수를 호출합니다.
print("D 지점 통과")
a = next(output)
print(a)

print("E 지점 통과")
b = next(output)
print(b)
```

```
print("F 지점 통과")
c = next(output)
print(c)
```

코드를 실행하면, 다음과 같이 출력합니다. next() 함수를 호출할 때마다 "A 지점 통과", "B 지점 통과", "C 지점 통과"처럼 함수 내부의 내용이 진행되는 모습을 확인할 수 있습니다.

추가로 next() 함수를 호출한 이후 yield 키워드를 만나지 못하고 함수가 끝나면, 'StopIteration'이라는 예외가 발생합니다.

실행 결과

```
D 지점 통과
A 지점 통과
1
E 지점 통과
B 지점 통과
2
F 지점 통과
C 지점 통과
Traceback (most recent call last):
  File "test.py", line 22, in <module>
    c = next(output)
StopIteration
```

StopIteration 예외가 발생했습니다.

제너레이터 객체는 이처럼 함수의 코드를 조금씩 실행할 때 사용합니다.

제너레이터와 반복문

본문에서 보았던 reversed() 등의 함수가 모두 제너레이터 객체를 생성할 때 사용하는 함수입니다. 이러한 함수를 반복문에 넣어 호출해서 사용했는데, 우리가 만든 제너레이터 함수도 반복문에 넣어 활용할 수 있습니다.

코드 B-4 제너레이터와 반복문

```
# 함수를 선언합니다.
def test():
    print("A 지점 통과")
    yield 1
    print("B 지점 통과")
    yield 2
    print("C 지점 통과")

# 반복문을 실행합니다.
for i in test():
    print("{}번째 반복".format(i))
    print()
```

이렇게 입력하면 함수가 끝날 때까지 계속 반복문을 돌게 됩니다.

실행 결과

```
A 지점 통과
1번째 반복

B 지점 통과
2번째 반복

C 지점 통과
```

TIP 제너레이터 객체 두 번 사용하기

한 번 사용이 완료된 제너레이터 객체는 다시 사용할 수 없습니다. 예를 들어 reversed( ) 함수로 리스트를 뒤집고, 반복문을 두 번 적용하는 경우를 생각해봅시다.

코드 B-5 reversed( ) 함수의 리턴값으로 두 번 반복 돌리기

```
# reversed() 함수로 리스트를 뒤집습니다.
test = reversed([1, 2, 3])

print("# 첫 번째 반복")
```

```
for i in test:
    print(i)

print("# 두 번째 반복")
for i in test:
    print(i)
```

첫 번째 반복문을 돌 때 제너레이터 객체가 모두 사용되므로 두 번째 반복문은 제대로 작동하지 않습니다. 이미 제너레이터 함수의 모든 부분을 종료했기 때문입니다.

실행 결과

```
# 첫 번째 반복
3
2
1
# 두 번째 반복
```

제너레이터 함수를 활용하면, 굉장히 다양한 것들을 할 수 있습니다. 일단 제너레이터가 무엇인지 기억해주세요. 이후에 개발을 하다 보면 알게 모르게 꽤 많이 접할 수 있을 것입니다. 간단한 연습으로 `reversed()`나, `map()`과 같은 함수를 직접 구현해보는 것도 재미있는 공부가 될 것입니다.

appendix C
정규 표현식

부록 C에서는

정규 표현식은 문자열의 패턴을 검사할 때 사용합니다. 원래는 펄Perl 프로그래밍 언어에서 사용한 객체인데, 유용하다고 인정받아 현재는 많은 프로그래밍 언어에서 사용되고 있습니다.

정규 표현식은 분량이 굉장히 많고, 매일 사용하는 내용이 아니라 가끔씩 사용하는 부분이므로 필자도 완벽하게 외우지는 않습니다. 필요할 때 스스로 정리한 내용을 찾아서 사용하는 편입니다. 따라서 부담 갖지 말고 가볍게 살펴보도록 합시다.

Raw 문자열

정규 표현식을 만들 때는 Raw 문자열이라는 특수한 형태의 문자열을 사용하는 경우가 많습니다. Raw 문자열이 무엇이고, 왜 사용하는지 간단하게 살펴보겠습니다.

일단 Raw 문자열은 다음과 같이 만듭니다.

구문 Raw 문자열

```
r"<글자>"
```

Raw 문자열은 일반 문자열과 거의 비슷합니다. 다음 코드를 살펴봅시다.

코드 C-1 Raw 문자열

```
print("# 일반 문자열")
print("안녕하세요")
print()
print("# Raw 문자열")
print(r"안녕하세요")
```

그냥 문자열 앞에 r을 붙이면 됩니다.

실행하면 두 값이 모두 동일하게 나옵니다.

실행 결과

```
# 일반 문자열
안녕하세요

# Raw 문자열
안녕하세요
```

그렇다면 어디에서 차이가 발생할까요? 바로 이스케이프 문자를 사용할 때입니다. Raw 문자열은 이스케이프 문자를 취급하지 않습니다.

코드 C-2 Raw 문자열과 이스케이프 문자

```
print("# 일반 문자열")
print("\note")
print()
print("# Raw 문자열")
print(r"\note")
```

이스케이프 문자 \n(줄바꿈)으로 처리됩니다.

따라서 실행했을 때 "\note"라고 입력했으면, "\note"라고 그대로[raw] 나옵니다.

실행 결과

```
# 일반 문자열

ote

# Raw 문자열
\note
```

정규 표현식을 작성하다 보면 이스케이프 문자 때문에 원하는 결과가 나오지 않는 경우가 있습니다. 예를 들어서 다음 코드를 살펴봅시다. 다음 코드는 역슬래시(\)를 콜론(:)으로 변경하는 예제입니다.

조금 보기 힘들 수 있는데, 정규 표현식에서 "\"를 표현하려면 "\\"를 입력해야 합니다. 그런데 일반 문자열에 "\\"를 입력하려면, 이스케이프 문자를 활용해 "\\\\"를 입력해야 합니다. 이렇게 입력하는 것은 굉장히 귀찮습니다. 이때 Raw 문자열을 활용하면 간단하게 "\\"로 입력할 수 있습니다.

코드 C-3 역슬래시와 정규 표현식

```
import re

print(re.sub("\\\\", ":", r"Hello\World"))
print(re.sub(r"\\", ":", r"Hello\World"))
```

이처럼 역슬래시를 정규 표현식 내부에서 사용하고자 할 때 일반 문자열을 사용하면 코드 입력이 복잡해지거나, 예상하지 못한 결과가 나오는 단점이 있습니다. 그래서 정규 표현식을 사용할 때는 문자열을 Raw 문자열로 사용하는 경우가 많습니다.

02 정규 표현식 모듈

정규 표현식을 활용할 때는 파이썬에 내장되어 있는 re 모듈을 사용합니다. 참고로 이때 re는 'Regular Expression(정규 표현식)'의 앞 글자를 딴 것입니다. 일단 정규 표현식을 본격적으로 활용하기 전에, 어떠한 기능들과 정규 표현식을 함께 사용할 수 있는지 알아봅시다. 아직 정규 표현식을 배우지 않은 상태이므로, 정규 표현식 이외의 것들에 주목해주세요.

findall( ) 함수와 finditer( ) 함수

일단 가장 기본적이며 이해하기 쉬운 함수로는 findall() 함수와 finditer() 함수가 있습니다.

두 함수는 글자 내부에서 정규 표현식에 일치하는 표현을 모두 찾아 줍니다. 이때 findall() 함수는 리스트로 리턴하며, finditer() 함수는 이터레이터로 리턴합니다. 간단한 예제를 살펴봅시다.

코드 C-4 findall( ) 함수와 finditer( ) 함수

```
# 모듈을 읽어 들입니다.
import re

# 정규 표현식 객체를 생성합니다.
regexp = re.compile(r"\w*e\w*")

# findall() 함수
input_a = "coffee and tea"
print("# findall() 함수는 모두 찾아 리스트로 리턴합니다.")
print("findall():", regexp.findall(input_a))
print()

# finditer() 함수
input_a = "coffee and tea"
output = regexp.finditer(input_a)
```

e가 들어간 단어를 검색할 때 사용하는 정규 표현식입니다. 자세한 내용은 이후에 다루겠습니다. 일단 findall( ) 함수와 finditer( ) 함수에 집중해주세요!

```
print("# finditer() 함수는 이터레이터를 리턴합니다.")
for item in output:
    print("findall():", item)
```

이터레이터와 관련된 내용은 부록B를 참고해주세요!

coffee and tea에서 e가 들어가는 단어를 찾아보았습니다. coffee와 tea에 모두 e가 들어가므로, 이 단어들을 추출해줍니다.

이때 findall() 함수는 문자열의 리스트, finditer() 함수는 Match 객체의 이터레이터를 리턴합니다. 따라서 다음과 같이 출력합니다.

실행 결과

```
# findall() 함수는 모두 찾아 리스트로 리턴합니다.
findall(): ['coffee', 'tea']

# finditer() 함수는 이터레이터를 리턴합니다.
finditer(): <_sre.SRE_Match object; span=(0, 6), match='coffee'>
finditer(): <_sre.SRE_Match object; span=(11, 14), match='tea'>
```

Match 객체와 관련된 내용은 이후에 알아보겠습니다.

TIP findall( ) 함수와 finditer( ) 함수의 단순 사용

findall( ) 함수와 finditer( ) 함수는 다음과 같이 간단하게 사용할 수 있습니다.

코드 C-5 findall( ) 함수와 finditer( ) 함수 단순 사용

```
# 모듈을 읽어 들입니다.
import re

# 정규 표현식을 사용합니다.
output_a = re.findall(r"\w*e\w*", "coffee and tea")
output_b = re.finditer(r"\w*e\w*", "coffee and tea")
```

search( ) 함수

이어서 search() 함수에 대해 살펴보겠습니다. search() 함수는 이름 그대로 글자 내부에서 정규 표현식에 일치하는 표현을 찾아줍니다. 이전의 findall(), finditer() 함수와 다르게 하나만 찾아 Match 객체로 리턴합니다. 이때 만약 아무 것도 찾지 못했다면 None을 리턴합니다.

그럼 간단한 예제를 살펴봅시다.

코드 C-6 search( ) 함수

```
# 모듈을 읽어 들입니다.
import re

# 정규 표현식 객체를 생성합니다.
regexp = re.compile(r"\w*e\w*")

# search() 함수
input_a = "coffee and tea"
print("search():", regexp.search(input_a))
```

코드를 실행하면 다음과 같이 출력합니다.

실행 결과

```
search(): <_sre.SRE_Match object; span=(0, 6), match='coffee'>
```

search() 함수는 결과를 앞쪽부터 찾았을 때 가장 먼저 나오는 하나만 찾아주기 때문에 짧은 문자열 확인에 많이 사용됩니다. 예를 들면, 파일 확장자 확인 등에 활용됩니다. search() 함수의 리턴값을 곧바로 if 조건문에 넣으면, 문자열 내부에 정규 표현식에 일치하는 표현이 있는지 없는지 쉽게 구분할 수 있습니다.

코드 C-7 search( ) 함수 활용 예

```
# 모듈을 읽어 들입니다.
import re

# 정규 표현식 객체를 생성합니다.
```

```
regexp = re.compile(r"\w*e\w*")

# search() 함수
input_a = "coffee and tea"
if regexp.search(input_a):
    print("{}에는 e가 들어간 단어가 있어요!".format(input_a))
else:
    print("{}에는 e가 들어간 단어가 없어요!".format(input_a))
```

if 조건문 뒤에 객체를 넣으면 True로 변환되고, None을 넣으면 False로 변환됩니다.

TIP search( ) 함수의 단순 사용

search( ) 함수도 다음과 같이 간단하게 사용할 수 있습니다.

코드 C-8 search( ) 함수 단순 사용

```
# 모듈을 읽어 들입니다.
import re

# 정규 표현식을 사용합니다.
input_a = "coffee and tea"
regexp = re.search(r"\w*e\w*$", input_a)
```

여담이지만 파이썬의 정규 표현식에는 match() 함수도 있습니다. search() 함수와 거의 비슷하지만, 이후에 언급하는 여러 줄 처리를 할 때 차이가 발생합니다. 하지만 대부분의 경우 search() 함수만 사용하므로, 이 책에서는 search() 함수에 대해서만 설명하겠습니다.

sub( ) 함수와 subn( ) 함수

sub() 함수와 subn() 함수는 문자열 내부에서 정규 표현식과 일치하는 표현을 다른 문자열로 변경할 때 사용합니다. 예를 들어, 다음 코드는 'e가 들어가는 단어'를 'banana'로 변경합니다.

이때 sub() 함수는 변환된 문자열을 단순하게 리턴하며, subn() 함수는 변환된 문자열과 함께 변환 횟수를 리턴합니다.

코드 C-9 sub( ) 함수와 subn( ) 함수

```
# 모듈을 읽어 들입니다.
import re

# 정규 표현식 객체를 생성합니다.
regexp = re.compile(r"\w*e\w*")

# sub() 함수
input_a = "coffee and tea"
output_a = regexp.sub("banana", input_a)
output_b = regexp.subn("banana", input_a)

# 출력합니다.
print("# sub() 함수")
print("sub():", output_a)
print("subn():", output_b)
```

간단하게 실행해보면 다음과 같이 출력합니다. e가 들어가는 단어였던 coffee와 tea가 banana로 변환된 것을 확인할 수 있습니다. 2개가 변환되었으므로 subn() 함수의 결과에 2가 출력되는 것도 확인할 수 있습니다.

실행 결과

```
# sub() 함수
sub(): banana and banana
subn(): ('banana and banana', 2)
```

TIP sub( ) 함수와 subn( ) 함수의 단순 사용

sub( ) 함수와 subn( ) 함수도 다음과 같이 간단하게 사용할 수 있습니다.

코드 C-10 sub( ) 함수와 subn( ) 함수 단순 사용

```
# 모듈을 읽어 들입니다.
import re
```

```
# 정규 표현식을 사용합니다.
output_a = re.sub(r"\w*e\w*", "banana", input_a)
output_b = re.subn(r"\w*e\w*", "banana", input_a)
```

re.split( ) 함수

마지막으로 정규 표현식을 활용하는 함수로 re.split() 함수가 있습니다. 문자열의 split() 함수와 비슷한데, 정규 표현식을 사용해 문자열을 자를 수 있는 함수입니다.

코드 C-11 re.split( ) 함수

```
# 모듈을 읽어 들입니다.
import re

# re.split() 함수
input_a = "Coffee and Tea"
output = re.split("\W", input_a)
print(output)
```

\W는 '글자가 아닌 것'을 의미하는 정규 표현식입니다. 따라서 공백, 쉼표, 탭 등으로 문자열을 자를 때 활용할 수 있습니다.

실행 결과

```
['Coffee', 'and', 'Tea']
```

정규 표현식과 함께 사용하는 함수들을 살펴보았습니다. 이러한 함수와 함께 사용할 수 있다는 것만 기억해주세요.

Match 객체

이전 절에서 살펴보았던 finditer() 함수와 match() 함수는 실행 결과가 <_sre.SRE_Match object; span=(0, 6), match='coffee'>와 같은 형태로 나왔습니다. 이를 간단하게 Match 객체라고 부르는데, 이번 절에서는 이러한 Match 객체가 가지고 있는 기능을 살펴보겠습니다.

Match 객체는 다음과 같은 4가지 함수를 가지고 있습니다(group() 함수와 관련된 추가적인 기능은 이후에 다시 살펴보겠습니다).

표 C-1 | Match 객체의 함수

함수	설명
group( )	단어를 리턴합니다.
start( )	시작 위치를 숫자로 리턴합니다.
end( )	끝 위치를 숫자로 리턴합니다.
span( )	(〈시작 위치〉, 〈끝 위치〉) 형태의 튜플을 리턴합니다.

굉장히 간단한 내용이므로 곧바로 예제 코드를 살펴봅시다.

코드 C-12 Match 객체

```
# 모듈을 읽어 들입니다.
import re

# 정규 표현식 객체를 생성합니다.
regexp = re.compile("\w*e\w*")

# finditer() 함수
input_a = "coffee and tea"
output = regexp.finditer(input_a)

# 출력합니다.
print("# coffee and tea에서 e가 들어가는 단어")
```

```
for item in output:
    print("group():", item.group())
    print("start():", item.start())
    print("end():", item.end())
    print("span():", item.span())
    print("----------------")
```

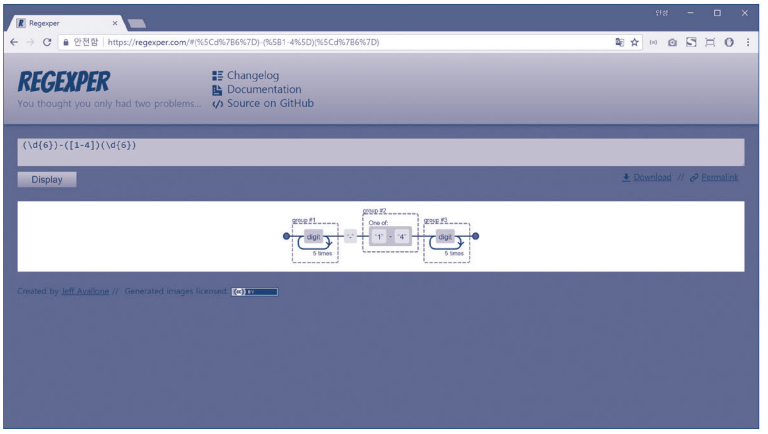
실행하면 다음과 같이 출력합니다. 이러한 값들을 문자열 처리와 함께 활용하면 다양한 작업들을 할 수 있습니다.

실행 결과

```
# coffee and tea에서 e가 들어가는 단어
group(): coffee
start(): 0
end(): 6
span(): (0, 6)
----------------
group(): tea
start(): 11
end(): 14
span(): (11, 14)
----------------
```

04 정규 표현식 기본

지금부터 정규 표현식 자체에 대해서 살펴볼텐데, 정규 표현식을 처음 공부하는 것이라면 굉장히 어렵게 느껴질 것입니다. 이럴 때는 다음 사이트를 참고해주세요.



https://regexper.com/

그림 C-1 | Regexper

자바스크립트 스타일의 정규 표현식을 입력하면, 이를 다음과 같은 그림으로 표현해주는 사이트입니다. 물론 파이썬의 정규 표현식과 자바스크립트의 정규 표현식이 약간의 차이를 가지고 있으므로 이미지로 출력할 수 없는 경우가 있을 수도 있는데, 그런 점을 감안하더라도 충분히 도움이 될 것이라고 생각합니다.

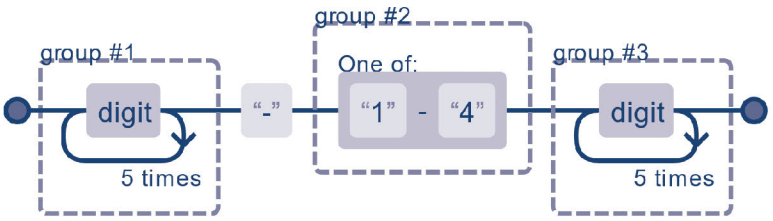


그림 C-2 | 그림으로 표현된 정규 표현식

그럼 본격적으로 정규 표현식에 대해 살펴봅시다.

단순 문자

일단 정규 표현식에는 단순한 문자를 입력할 수 있습니다. 내부에 "abc"라는 문자가 있는지 확인하고 싶으면, 그냥 "abc"를 입력하면 됩니다. 다음 코드는 "coffee and tea"라는 문자열에서 "tea"라는 글자를 찾습니다.

코드 C-13 단순 문자

```
# 모듈을 읽어 들입니다.
import re

# 변수를 선언합니다.
input_a = "coffee and tea"
regexp = re.compile("tea")

# 단순 문자 찾기
print("# tea를 coffee and tea에 적용하기")
output = regexp.finditer(input_a)
for item in output:
    print("keyword:", item.group())
    print("start:", item.start())
```

실행하면 다음과 같이 출력합니다.

실행 결과

```
# tea를 coffee and tea에 적용하기
keyword: tea
start: 11
```

기본 메타 문자

a, b, c와 같은 확실한 글자가 아니라, '숫자', '알파벳', '한글'처럼 추상화된 형태를 입력하고 싶을 때는 메타 문자를 사용합니다. 정규 표현식에는 '.(마침표)'로 모든 값을 표현할 수 있으며, [<시작>-<끝>]으로 값의 범위를 한정해서 표현할 수 있습니다.

표 C-2 | 기본 메타 문자

기호	설명
.	모든 글자
[abc]	괄호 안의 글자
[^abc]	괄호 안의 글자 제외
[a-z]	알파벳 a부터 z까지
[A-Z]	알파벳 A부터 Z까지
[0-9]	숫자 0부터 9까지

그럼 [0-9]를 사용해서 주민등록번호를 표현해봅시다. 주민등록번호는 대시(-)를 기준으로 앞에 숫자 6개, 뒤에 숫자 7개가 배치되는 형태입니다. 따라서 다음과 같이 표현할 수 있습니다.

코드 C-14 기본 메타 문자

```
# 모듈을 읽어 들입니다.
import re

# 변수를 선언합니다.
input_a = "coffee and tea"
input_b = "911209-0000000"
regexp = re.compile("[0-9][0-9][0-9][0-9][0-9][0-9]-[0-9][0-9][0-9][0-9][0-9][0-9][0-9]")

# 단순 문자 찾기
print("# 주민등록번호 형태 찾기")
print("{}: {}".format(input_a, regexp.match(input_a)))
print("{}: {}".format(input_b, regexp.match(input_b)))
print()

# 활용
print("# match 객체 활용 예")
if regexp.match(input_b):
    # 주민등록번호 형식이라면
    print("input_b는 주민등록번호 형식입니다.")
```

```
else:
    # 주민등록번호 형식이라면
    print("input_b는 주민등록번호 형식이 아닙니다.")
```

코드를 실행하면 다음과 같이 출력합니다. "coffee and tea"라는 글자는 주민등록번호 형식이 아니므로 None을 출력하며, "911209-0000000"은 주민등록번호 형식이므로 Match 객체를 출력합니다.

실행 결과

```
# 주민등록번호 형태 찾기
coffee and tea: None
911209-0000000: <_sre.SRE_Match object; span=(0, 14), match='911209-0000000'>

# match 객체 활용 예
input_b는 주민등록번호 형식입니다.
```

TIP 주민등록번호 정규 표현식의 활용 예

어떤 경우에 활용될까요? 예를 들어, 금융 업무를 할 때(최근에는 주민등록번호 수집이 금지되었지만) 일시적으로 확인하는 경우가 있습니다. 이때 다음과 같은 순서로 주민등록번호를 확인합니다.

1 사용자가 입력한 것이 주민등록번호 형식이 맞는지 확인한다(숫자로만 구성되어 있는지 등).

2 이를 주민등록번호 인증 기관에 요청해서 진짜 주민등록번호인지 확인한다.

정규 표현식으로 처리하는 부분은 1번입니다. 만약 1번 과정 없이 2번으로 넘어가면, 악의적인 사용자가 인증 기관에 무차별적으로 인증을 요청해서 비용이 낭비되는 등의 일이 발생할 수 있습니다. 따라서 일단 정규 표현식을 활용해서 형식이 맞는지부터 확인하고, 인증을 요청합니다.

특수 메타 문자

자주 사용되는 메타 문자는 다음과 같이 쉽게 사용할 수 있도록 정의되어 있습니다.

표 C-3 | 특수 메타문자

기호	설명
\d	숫자[0-9]
\w	모든 글자(숫자 포함)[a-zA-Z0-9]
\s	공백 문자(탭, 띄어쓰기, 줄바꿈)
\D	숫자 아님
\W	모든 글자 아님
\S	공백 문자 아님

이를 활용하면 이전의 정규 표현식을 조금 더 쉽게 표현할 수 있습니다.

코드 C-15 특수 메타문자

```
# 모듈을 읽어 들입니다.
import re

# 변수를 선언합니다.
input_a = "coffee and tea"
input_b = "911209-0000000"
regexp = re.compile("\d\d\d\d\d\d-\d\d\d\d\d\d\d")

# 단순 문자 찾기
print("# 주민등록번호 형태 찾기")
print("{}: {}".format(input_a, regexp.match(input_a)))
print("{}: {}".format(input_b, regexp.match(input_b)))
```

실행 결과

```
# 주민등록번호 형태 찾기
coffee and tea: None
911209-0000000: <_sre.SRE_Match object; span=(0, 14), match='911209-0000000'>
```

수량 문자

방금 전의 주민등록번호를 확인하는 [코드 C-15]를 보면 "\d"가 여러 번 사용되어 몇 번인지 세기도 힘듭니다. 그래서 이런 형태를 쉽게 작성하고자 수량 문자가 제공됩니다.

표 C-4 | 수량 문자

기호	설명
a+	a가 1개 이상
a*	a가 0개 또는 여러 개
a?	a가 0개 또는 1개
a{5}	a가 5개
a{2,5}	a가 2~5개
a{2,}	a가 2개 이상
a{,2}	a가 2개 이하

수량 문자를 활용하면 다음과 같이 주민등록번호를 표현할 수 있습니다.

코드 C-16 수량 문자

```
# 모듈을 읽어 들입니다.
import re

# 변수를 선언합니다.
input_a = "911209-0000000"
regexp = re.compile("\d{6}-\d{7}")

# 단순 문자 찾기
print("# 주민등록번호 형태 찾기")
print("{}: {}".format(input_a, regexp.match(input_a)))
```

실행 결과

```
# 주민등록번호 형태 찾기
911209-0000000: <_sre.SRE_Match object; span=(0, 14), match='911209-0000000'>
```

그룹

정규 표현식 내부의 값들은 특정한 의미를 가지고 있을 가능성이 크며, 이러한 의미를 추출해 활용하고 싶은 경우가 생길 수 있습니다. 예를 들어 주민등록번호는 앞부분이 생년월일이며, 뒷부분은 성별과 추가적인 구분으로 나뉩니다.

이러한 것들은 그룹이라는 기능으로 묶으면 다양하게 활용할 수 있습니다. 그룹은 그룹으로 묶고 싶은 부분을 소괄호(( ))로 묶어 만듭니다.

코드 C-17 그룹

```
# 모듈을 읽어 들입니다.
import re

# 변수를 선언합니다.
input_a = "911209-1000000"
regexp = re.compile("(\d{6})-([1-4])(\d{6})")

# 단순 문자 찾기
output = regexp.match(input_a)
group_a = output.group(1)
group_b = output.group(2)
group_c = output.group(3)

print("# 주민등록번호 형태 찾기")
print("group(1):", group_a)
print("group(2):", group_b)
print("group(3):", group_c)
```

위의 정규 표현식은 이전에 그림으로 소개했던 정규 표현식입니다. 다음과 같이 3개의 그룹으로 나뉘며, 각각의 그룹은 숫자가 6번 출현하는지 또는 1~4 사이의 숫자인지 등으로 구분됩니다.

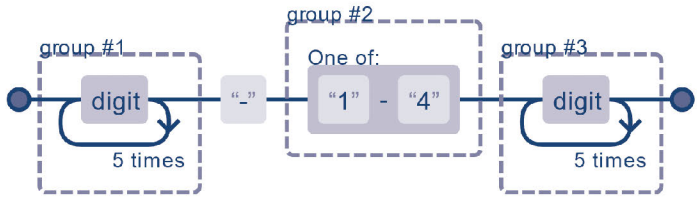


그림 C-3 | 현재 정규 표현식을 그림으로 표현한 상태

참고로 그림에 5 times라고 표현되어 있어서 5번이라고 생각할 수 있는데, 처음 digit(숫자)을 만나는 횟수는 제외되어 있습니다. 따라서 1+5 = 6번이 됩니다. 이런 표현을 많이 사용하므로 항상 주의하고 기억하기 바랍니다.

실행 결과

```
# 주민등록번호 형태 찾기
group(1): 911209
group(2): 1
group(3): 000000
```

파이썬의 정규 표현식은 약간 특이한 기능들을 더 가지고 있습니다. 예로 그룹 앞에 ?P<그룹 이름> 등을 붙이면, 그룹에 이름을 붙일 수 있습니다. 다음과 같이 앞의 부분에 birthday, 중간 부분에 gender라는 이름을 붙일 수 있습니다.

코드 C-18 그룹에 이름 붙이기

```
# 모듈을 읽어 들입니다.
import re

# 변수를 선언합니다.
input_a = "911209-1000000"
regexp = re.compile("(?P<birthday>\d{6})-(?P<gender>[1-4])(\d{6})")

# 단순 문자 찾기
output = regexp.match(input_a)
group_a = output.group("birthday")
group_b = output.group("gender")
group_c = output.group(1)
group_d = output.group(2)
```

이름을 붙인 그룹은 그룹을 이름으로도 꺼낼 수 있고, 숫자로도 꺼낼 수 있습니다.

```
print("# 주민등록번호 형태 찾기")
print("group(birthday):", group_a)
print("group(gender):", group_b)
print("group(1):", group_c)
print("group(2):", group_d)
```

실행하면 다음과 같은 결과가 나옵니다.

실행 결과

```
# 주민등록번호 형태 찾기
group(birthday): 911209
group(gender): 1
group(1): 911209
group(2): 1
```

다만 파이썬의 정규 표현식에서만 제공되는 기능이라는 것에 주의하기 바랍니다.

정규 표현식 옵션

compile() 함수에는 두 번째 매개변수로 옵션을 넘길 수 있습니다. 이때 지정할 수 있는 옵션은 다음과 같습니다(물론 더 있지만 자주 사용되는 것만 정리했습니다).

표 C-5 | 정규 표현식 옵션

단축 옵션	옵션	설명
re.I	re.IGNORECASE	대소문자 무시하도록 설정
re.M	re.MULTILINE	여러 줄 매치하도록 설정
re.S	re.DOTALL	메타 문자 "."이 줄바꿈 기호와도 매치되도록 설정
re.X	re.VERBOSE	주석을 허용하도록 설정

이러한 옵션은 다음과 같이 사용합니다. 두 번째 매개변수에 단축 형식 또는 기본 형식을 입력하며, 두 개 이상을 적용하고 싶을 때는 | 기호(한국어 키보드에서 [shift]키를 누르고 ₩를 입력)로 연결합니다.

정규 표현식 옵션 사용 방법

```
# 입력 방법
re.compile("", re.I)                # 단축 옵션 입력
re.compile("", re.IGNORECASE)       # 옵션 입력
re.compile("", re.I | re.MULTILINE) # 여러 개 입력
```

그럼 각각의 옵션에 대해서 살펴보겠습니다.

대소문자 무시

대소문자 무시라는 의미는 단어만 봐도 이해할 수 있을 것입니다. 바로 예제를 살펴봅시다.

코드 C-19 대소문자 무시

```
# 모듈을 읽어 들입니다.
import re

# 변수를 선언합니다.
input_a = "Coffee and Tea"
regexp = re.compile("tea", re.I)

# 단순 문자 찾기
output = regexp.finditer(input_a)
for item in output:
    print("keyword:", item.group())
    print("start:", item.start())
```

Coffee와 Tea가 대문자입니다.

코드를 실행하면 다음과 같이 출력합니다. 소문자 "tea"로 검색했지만, 대소문자 무시 옵션을 넣었으므로, "Tea"를 검색해주는 것입니다.

실행 결과

```
keyword: Tea
start: 11
```

여러 줄 처리

정규 표현식 내부에서는 다음과 같은 위치 메타 기호를 사용할 수 있습니다.

표 C-6 | 위치 메타 기호

기호	설명
^	문자열 한 줄의 시작을 나타냅니다(여러 줄 인정).
$	문자열 한 줄의 끝을 나타냅니다(여러 줄 인정).
\A	문자열의 가장 앞을 나타냅니다(여러 줄 인정하지 않음).
\Z	문자열의 가장 뒤를 나타냅니다(여러 줄 인정하지 않음).

이때 여러 줄 처리 옵션 re.M이 켜져있지 않으면 ^와 \A, $와 \Z는 차이가 없어집니다. 무슨 의미인지 일단 코드로 살펴봅시다.

다음 코드는 "^ABC"라는 정규 표현식을 사용해서, '문자열 한 줄의 시작과 함께 ABC가 나올 때'를 표현했습니다. 이렇게 표현하고 re.M 옵션을 비활성화, 활성화한 상태로 실행해봅시다.

코드 C-20 여러 줄 처리

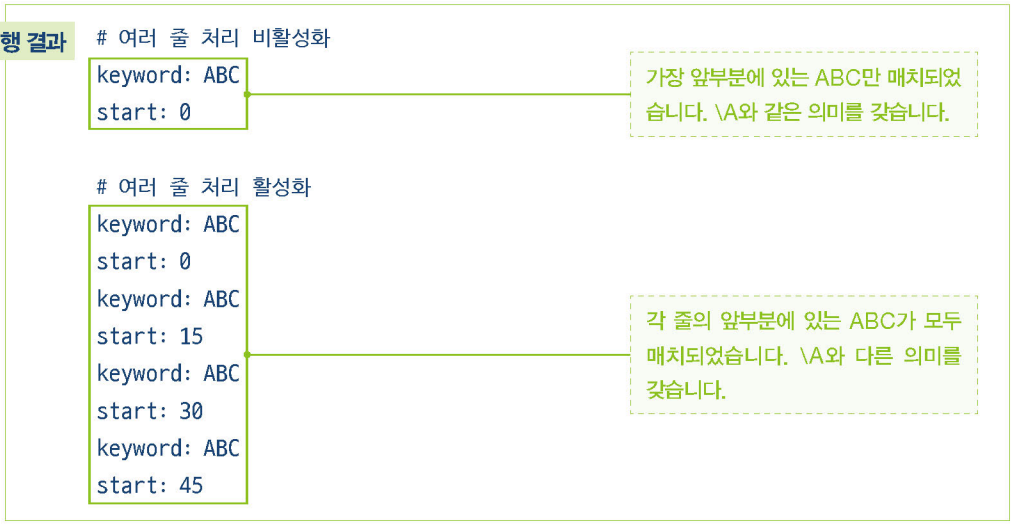
```
# 모듈을 읽어 들입니다.
import re

# 변수를 선언합니다.
input_a = """\
ABCDEFGABCDEFG
ABCDEFGABCDEFG
ABCDEFGABCDEFG
ABCDEFGABCDEFG"""

# 정규 표현식을 사용합니다.
print("# 여러 줄 처리 비활성화")
regexp = re.compile("^ABC")
output = regexp.finditer(input_a)
for item in output:
    print("keyword:", item.group())
    print("start:", item.start())
print()

print("# 여러 줄 처리 활성화")
regexp = re.compile("^ABC", re.M)
output = regexp.finditer(input_a)
for item in output:
    print("keyword:", item.group())
    print("start:", item.start())
```

코드를 실행하면 다음과 같이 출력합니다.



위치 메타 기호는 굉장히 많이 사용되므로 꼭 기억하기 바랍니다.

주석

re.X와 re.VERBOSE는 정규 표현식 내부에 주석을 사용하고 싶을 때 사용하는 옵션입니다. 주석이라는 말이 약간 이상하게 들릴 수 있는데, 정규 표현식을 다음과 같이 작성할 수 있다는 의미입니다. 추가로 re.X와 re.VERBOSE를 적용하게 되면 정규 표현식 내부의 모든 띄어쓰기가 무시됩니다.

코드 C-21 정규 표현식과 주석

```
# 모듈을 읽어 들입니다.
import re

# 변수를 선언합니다.
input_a = "911209-1000000"
```

```
regexp = re.compile("""
    (?P<birthday>\d{6}) # 생년월일을 나타내는 6개의 숫자
    -                   # 대시 1개
    (?P<gender>[1-4])   # 성별을 나타내는 숫자 하나
    (\d{6})             # 이외의 숫자
""", re.X)

# 단순 문자 찾기
output = regexp.match(input_a)
group_a = output.group("birthday")
group_b = output.group("gender")
group_c = output.group(3)

print("# 주민등록번호 형태 찾기")
print("group(1):", group_a)
print("group(2):", group_b)
print("group(3):", group_c)
```

정규 표현식에 설명을 달았습니다.

NOTE 다만 대괄호([ ]) 내부에서 공백 앞에 역슬래시(\)가 오는 경우는 공백으로 인정됩니다.

정규 표현식이 복잡하다면

지금까지 정규 표현식과 관련된 기본적인 내용을 살펴보았습니다. 사실 정규 표현식은 이 외에도 더 많은 기능이 있지만, 이 정도만 알고 있으면 활용할 때 큰 문제 없을 것입니다.

참고로 과거에는 '프로그램을 최대한 짧게 작성해야 좋은거야!'라는 인식이 많았습니다. 하지만 본문에서 언급했던 것처럼 최근에는 '모두 함께 작업하려면 최대한 쉽게 풀어서 작성하는 것이 좋은 거야!'라는 인식이 많습니다.

이때 개발자들 사이에서 논란이 되는 것이 바로 '정규 표현식'입니다. 한 문장에 너무나도 많은 내용이 집약되므로, 이해할 때 힘들 수 있다는 것이지요. 정규 표현식을 너무 복잡하게 사용하면 다른 동료는 물론이고, 미래의 나도 이해하기 힘들 수가 있습니다. 따라서 `re.VERBOSE(re.X)`로 주석을 적절하게 달거나, 너무 복잡하다고 생각되면 차라리 다른 코드로 풀어서 작성하는 것이 좋을 수도 있습니다.

예를 들어 다음 코드는 이미지 파일을 찾고, 이름만 추출하는 예입니다.

코드 C-22 정규 표현식으로 작성한 이미지 파일 확인 후 출력

```
# 모듈을 읽어 들입니다.
import os
import re

# 이미지 파일인지 확인하는 함수
def check_image_file(filename):
    if re.match(r"[-\w]+\.(jpg|gif|png)", filename):
        return True
    return False

# 폴더를 읽어 들이는 함수
def read_folder(path):
    output = os.listdir(path)
    for item in output:
        if os.path.isdir(item):
```

```
            read_folder(item)
        else:
            if check_image_file(item):
                print("이미지 파일:", item)

# 현재 폴더의 파일/폴더를 출력합니다.
read_folder(".")
```

사실 이 정도의 정규 표현식은 조금 살펴보면 이해할 수 있지만, 생각하는 데 잠시 시간이 걸리는 것이 사실입니다. 또한 정규 표현식을 설명할 때도 의문이 계속 발생할 수 있습니다.

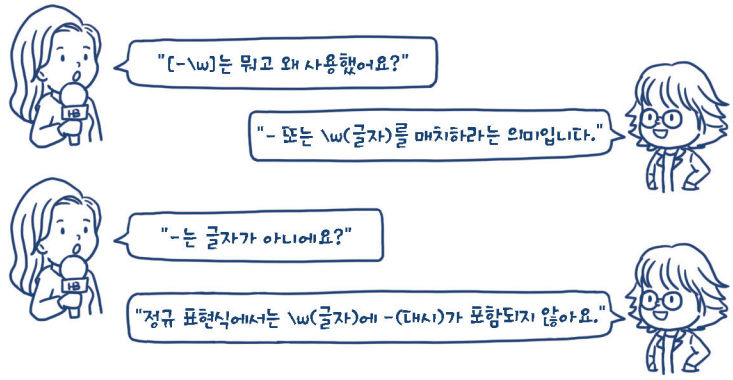


그림 C-4 | 어려운 정규 표현식으로 인한 소통 문제

따라서 다음과 같이 풀어서 작성하는 것도 나쁘지 않습니다. 다음 코드는 파일 이름을 "."으로 자르고, 마지막에 있는 글자를 추출한 뒤 ["jpg", "gif", "png"] 중에 있는지 확인하는 예입니다. 이 코드는 누가 봐도 쉽고 명확하게 이해할 수 있을 것입니다.

코드 C-23 다른 기능들로 작성한 이미지 파일 확인 후 출력

```
# 이미지 파일인지 확인하는 함수
def check_image_file(filename):
    ext = filename.split(".")[-1]
    if ext in ["jpg", "gif", "png"]:
        return True
    return False
```

참고로 '이해하기 어려우니까, 정규 표현식을 아예 사용하지 말자'라는 의미는 아닙니다. 정규 표현식은 정말 많은 곳에 사용되고 있으며, 큰 텍스트 파일에서 필요한 부분을 추출할 때 코드를 쉽게 작성할 수 있도록 해주는 좋은 기능입니다.

또한, `check_image_file()` 함수는 이름만 보아도 '이미지 파일인지 확인하는 함수겠구나'라고 이해할 수 있으며, 내부의 내용은 블랙박스(내부의 내용을 몰라도 되는 것)처럼 취급할 수 있습니다. 따라서 '다른 사람이 이해하지 못해도 상관없는 블랙박스 코드' 등에는 적극적으로 활용해도 무관합니다.

자신이 처한 상황(팀원 중에 정규 표현식을 모르는 사람이 있는 경우 등)에 따라 적절하게 활용하기 바랍니다.

appendix D
데코레이터

부록 D에서는

파이썬에는 '데코레이터Decorator'라는 특이한 기능이 있습니다. 사실 처음 보면 굉장히 이해하기 힘든 개념입니다. 본문에서 Flask 모듈을 간단하게 살펴보면서, @app.route( ) 형태의 코드를 보았습니다. 그리고 클래스 메서드를 살펴보면서도 @classmethod 형태의 코드를 보았습니다.

이렇게 @로 시작하는 것을 파이썬에서는 '데코레이터'라고 부릅니다. 데코레이터는 '꾸며주는 것'이라는 의미인데, 무엇을 꾸민다는 의미인지 차근차근 알아보겠습니다.

함수 데코레이터

데코레이터는 만드는 방법에 따라 크게 '함수 데코레이터'와 '클래스 데코레이터' 두 가지로 나눌 수 있습니다. 일단 함수 데코레이터가 더 쉬우므로 함수 데코레이터부터 살펴보겠습니다.

데코레이터 기본

함수 데코레이터는 함수를 사용해서 만드는 데코레이터입니다. 굉장히 기본적인 형태인데, 다음과 같은 형태로 만듭니다.

코드 D-1 함수 데코레이터 기본

```
# 함수로 데코레이터를 생성합니다.
def test(function):
    return "Decorator"

# 데코레이터를 붙여 함수를 만듭니다.
@test
def hello():
    pass

# 데코레이터를 붙였던 함수를 출력해봅니다.
print(hello)
```

매개변수로 함수를 받는 test라는 이름의 함수를 선언하면, @test라는 구문을 사용할 수 있습니다. 이는 다음과 같은 형태로 변환된다고 생각하면 쉽습니다.

쉬운 이해를 위한 데코레이터 변환

```
@test
def hello():
    pass
```

→

```
hello = test(<함수 내용>)
```

현재 코드를 보면 test() 함수를 실행했을 때 최종 결과가 "Decorator"입니다. 따라서 hello를 출력했을 때 "Decorator"가 출력됩니다.

실행 결과

```
Decorator
```

그런데 현재 코드처럼 사용하면 '대체 데코레이터를 왜 사용하지?'라는 느낌밖에 들지 않을 것입니다. 그럼 데코레이터를 어떻게 활용하는지 조금 더 자세하게 살펴봅시다.

데코레이터 활용

데코레이터의 가장 기본적인 활용 예시는 다음과 같습니다. 이렇게 사용하면, 함수를 호출하기 전과 후에 특정한 처리를 할 수 있게 됩니다.

코드 D-2 함수 데코레이터 활용

```
# 함수로 데코레이터를 생성합니다.
def test(function):
    def wrapper():
        print("wrapper()가 시작되었습니다.")
        function()
        print("wrapper()가 종료되었습니다.")
    return wrapper

# 데코레이터를 붙여 함수를 만듭니다.
@test
def hello():
    print("hello")

# 함수를 호출합니다.
hello()
```

test() 함수에서 wrapper() 함수를 리턴하므로, 최종적으로 hello에는 함수가 들어갑니다. 따라서 hello() 형태로 호출할 수도 있습니다.

실행 결과

```
wrapper()가 시작되었습니다.
hello
wrapper()가 종료되었습니다.
```

이러한 형태로 사용하기 위해 functools라는 모듈이 제공됩니다. fuctools 모듈을 사용하면, 함수 데코레이터를 사용할 때 매개변수 등을 전달할 수 있게 됩니다. 일반적으로 데코레이터는 이러한 형태를 기본적으로 사용합니다. 따라서 데코레이터가 필요할 때는 이 코드를 보고 활용해주세요.

코드 D-3 안전 장치와 매개변수 전달을 적용한 데코레이터

```
# 모듈을 가져옵니다.
from functools import wraps

# 함수로 데코레이터를 생성합니다.
def test(function):
    @wraps(function)
    def wrapper(*arg, **kwargs):
        print("wrapper()가 시작되었습니다.")
        function(*arg, **kwargs)
        print("wrapper()가 종료되었습니다.")
    return wrapper
```

NOTE 다른 프로그래밍 언어를 해보았던 분이라면 @라는 기호 때문에 '어노테이션Annotation'과 비슷한 것이라고 생각할 수 있는데, 작성 방법만 비슷하지 기능이 크게 다르므로 구분해주세요.

02 데코레이터 중첩

그럼 데코레이터를 활용하는 간단한 예를 살펴봅시다. 다음 코드는 @p_tag와 @h1_tag라는 데코레이터를 만듭니다. 이 데코레이터들은 함수의 리턴값을 기반으로 HTML 태그를 씌어 줍니다.

코드 D-4 데코레이터 중첩 예

```
# 모듈을 가져옵니다.
from functools import wraps

# 함수로 데코레이터를 생성합니다.
def p_tag(function):
    @wraps(function)
    def wrapper(*arg, **kwargs):
        return "<p>" + str(function(*arg, **kwargs)) + "</p>"
    return wrapper

def h1_tag(function):
    @wraps(function)
    def wrapper(*arg, **kwargs):
        return "<h1>" + str(function(*arg, **kwargs)) + "</h1>"
    return wrapper

# 데코레이터를 붙여 함수를 만듭니다.
@p_tag
@h1_tag
def hello(string):
    return "Hello" + string

# 함수를 호출합니다.
output = hello("Python Programming")
print(output)
```

실행 결과

```
<p><h1>HelloPython Programming</h1></p>
```

클래스 데코레이터

클래스로 데코레이터를 만들면 '클래스 데코레이터'라고 부릅니다. 기능은 함수 데코레이터와 같으며, 만드는 방법은 다음과 같습니다. 거의 비슷하므로 실행 결과를 미리 예측해보기 바랍니다.

코드 D-5 클래스 데코레이터

```
# 클래스로 데코레이터를 생성합니다.
class Test:
    def __init__(self, function):
        print("__init__이 호출되었습니다.")
        self.function = function
    def __call__(self):
        print("__call__이 시작되었습니다.")
        self.function()
        print("__call__이 종료되었습니다.")

# 데코레이터를 붙여 함수를 만듭니다.
@Test
def hello():
    print("hello")

# 함수를 호출합니다.
print()
print("hello() 호출 전")
hello()
print("hello() 호출 후")
```

실행하면 다음과 같이 출력합니다.

실행 결과

```
__init__이 호출되었습니다.

hello() 호출 전
__call__이 시작되었습니다.
hello
__call__이 종료되었습니다.
hello() 호출 후
```

사실 데코레이터를 직접 만들 일은 거의 없을 것입니다. 하지만 데코레이터가 사용된 코드를 보며 '이건 어떤 코드를 사용해서 만든거지?'라는 궁금증이 들 수 있을 것 같아 부록에서 간단하게 다뤄보았습니다.