

파이썬 프로그래밍 강의노트 #18

코루틴, 정규표현식

코루틴(coroutin) 활용

코루틴 사용하기

```
def add(a, b):
    c = a + b      # add 함수가 끝나면 변수와 계산식은 사라짐
    print(c)
    print('add 함수')

def calc():
    add(1, 2)      # add 함수가 끝나면 다시 calc 함수로 돌아옴
    print('calc 함수')

calc()
```

- calc가 메인 루틴(main routine)이면 add는 calc의 서브 루틴(sub routine)임

코루틴(coroutine) 활용

▼ 그림 41-1 메인 루틴과 서브 루틴의 동작 과정



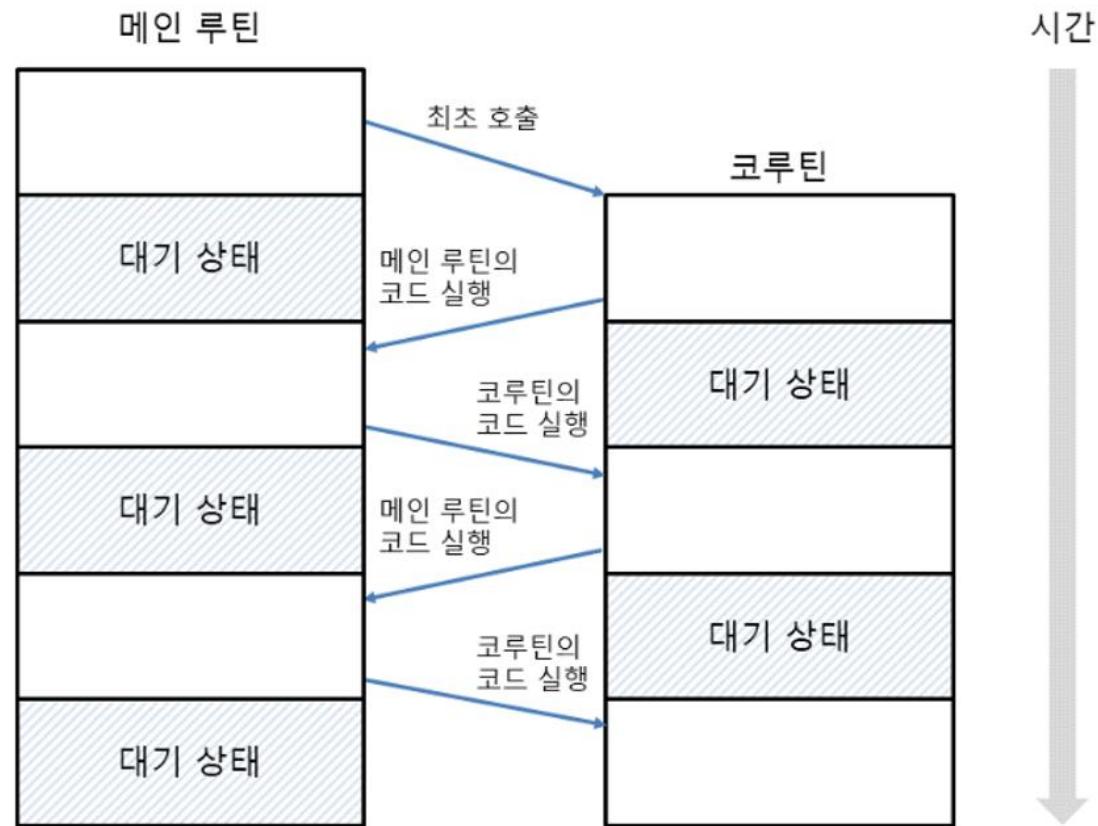
코루틴(coroutin) 활용

코루틴 사용하기

- 메인 루틴에서 서브 루틴을 호출하면 서브 루틴의 코드를 실행한 뒤 다시 메인 루틴으로 돌아옴
- 서브 루틴이 끝나면 서브 루틴의 내용은 모두 사라짐
- 서브 루틴은 메인 루틴에 종속된 관계임
- 코루틴은 방식이 조금 다른데 코루틴 coroutine은 cooperative routine를 의미하는데 서로 협력하는 루틴이라는 뜻임
- 메인 루틴과 서브 루틴처럼 종속된 관계가 아니라 서로 대등한 관계이며 특정 시점에 상대방의 코드를 실행함

코루틴(coroutine) 활용

▼ 그림 41-2 코루틴의 동작 과정



코루틴(coroutine) 활용

코루틴 사용하기

- 코루틴은 함수가 종료되지 않은 상태에서 메인 루틴의 코드를 실행한 뒤 다시 돌아와서 코루틴의 코드를 실행함
- 코루틴이 종료되지 않았으므로 코루틴의 내용도 계속 유지됨
- 일반 함수를 호출하면 코드를 한 번만 실행할 수 있지만, 코루틴은 코드를 여러 번 실행할 수 있음
- 참고로 함수의 코드를 실행하는 지점을 진입점(entry point)이라고 하는데, 코루틴은 진입점이 여러 개인 함수임

코루틴(coroutin) 활용

코루틴에 값 보내기

- 코루틴은 제너레이터의 특별한 형태임
- 제너레이터는 yield로 값을 발생시켰지만 코루틴은 yield로 값을 받아올 수 있음
- 다음과 같이 코루틴에 값을 보내면서 코드를 실행할 때는 send 메서드를 사용함
- send 메서드가 보낸 값을 받아오려면 (yield) 형식으로 yield를 괄호로 묶어준 뒤 변수에 저장함
 - 코루틴객체.send(값)
 - 변수 = (yield)

코루틴(coroutin) 활용

코루틴에 값 보내기

coroutine_consumer.py

```
def number_coroutine():
    while True:          # 코루틴을 계속 유지하기 위해 무한 루프 사용
        x = (yield)      # 코루틴 바깥에서 값을 받아옴, yield를 괄호로 묶어야 함
        print(x)

co = number_coroutine()
next(co)      # 코루틴 안의 yield까지 코드 실행(최초 실행)

co.send(1)    # 코루틴에 숫자 1을 보냄
co.send(2)    # 코루틴에 숫자 2를 보냄
co.send(3)    # 코루틴에 숫자 3을 보냄
```

실행 결과

```
1
2
3
```

코루틴(coroutin) 활용

코루틴에 값 보내기

```
def number_coroutine():
    while True:      # 코루틴을 계속 유지하기 위해 무한 루프 사용
        x = (yield)  # 코루틴 바깥에서 값을 받아옴, yield를 괄호로 묶어야 함
        print(x)
```

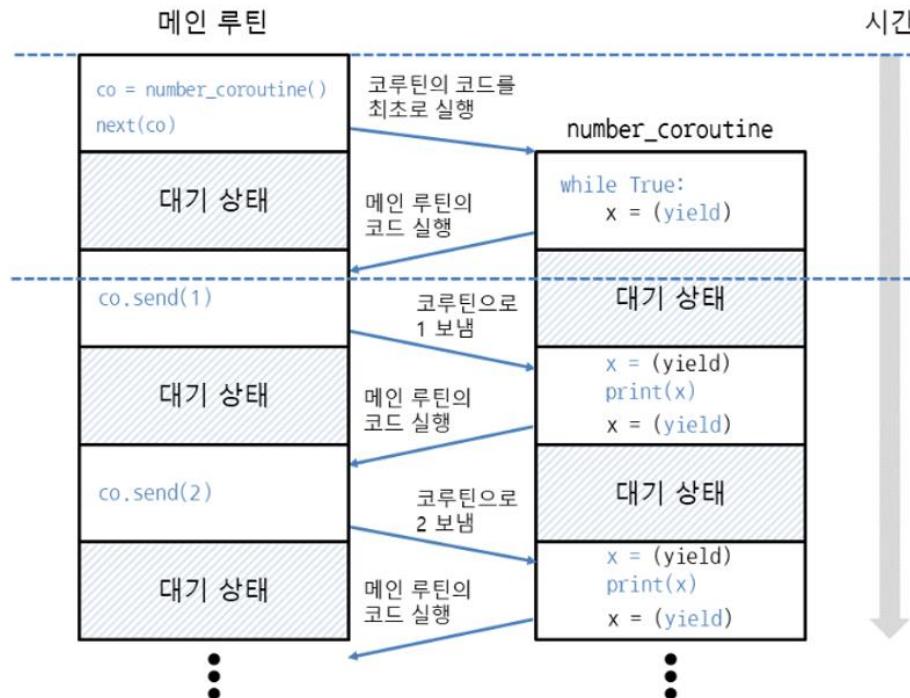
- next(코루틴객체)

```
co = number_coroutine()
next(co)      # 코루틴 안의 yield까지 코드 실행(최초 실행)
```

```
co.send(1)    # 코루틴에 숫자 1을 보냄
co.send(2)    # 코루틴에 숫자 2를 보냄
co.send(3)    # 코루틴에 숫자 3를 보냄
```

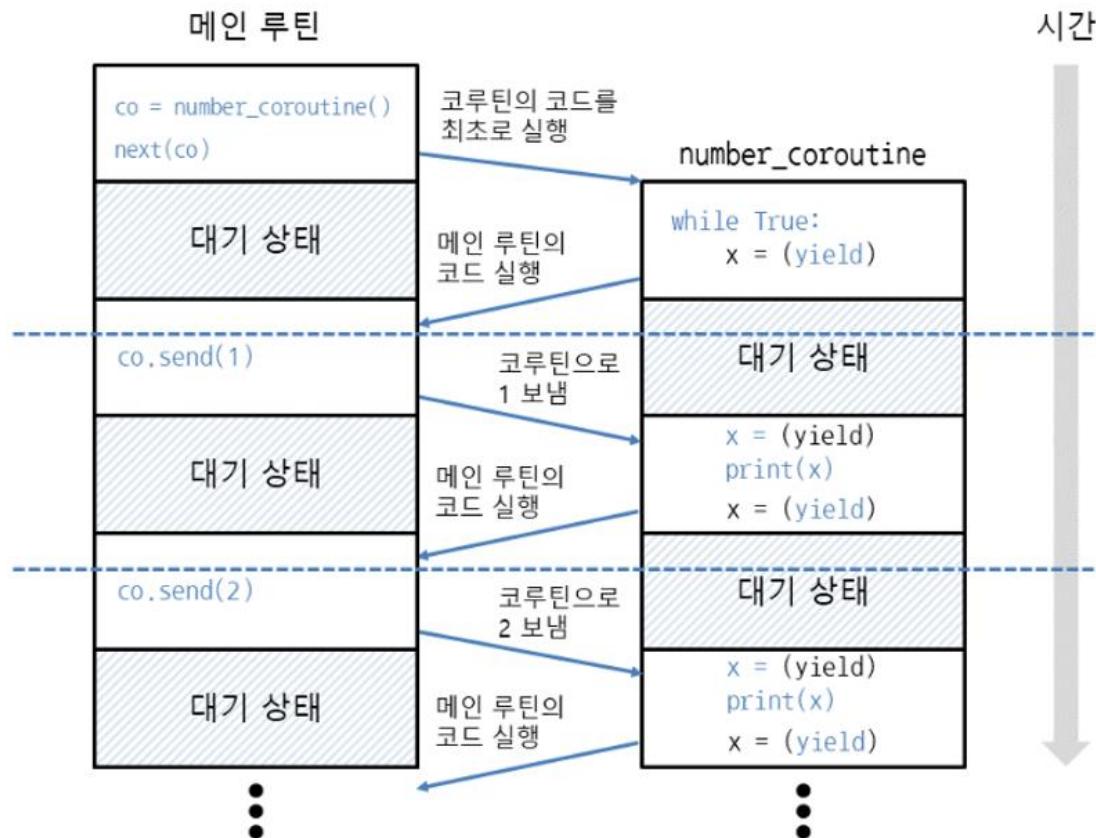
코루틴(coroutin) 활용

▼ 그림 41-3 코루틴의 코드를 최초로 실행한 뒤 메인 루틴으로 돌아옴



코루틴(coroutine) 활용

▼ 그림 41-4 send로 값을 보내면 코루틴에서 값을 받아서 출력



코루틴(coroutin) 활용

코루틴 바깥으로 값 전달하기

- 다음과 같이 (yield 변수) 형식으로 yield에 변수를 지정한 뒤 괄호로 묶어주면 값을 받아오면서 바깥으로 값을 전달함
- 변수 = (yield 변수)
- 변수 = next(코루틴객체)
- 변수 = 코루틴객체.send(값)

coroutine_producer_consumer.py

```
def sum_coroutine():
    total = 0
    while True:
        x = (yield total)    # 코루틴 바깥에서 값을 받아오면서 바깥으로 값을 전달
        total += x

co = sum_coroutine()
print(next(co))      # 0: 코루틴 안의 yield까지 코드를 실행하고 코루틴에서 나온 값 출력

print(co.send(1))    # 1: 코루틴에 숫자 1을 보내고 코루틴에서 나온 값 출력
print(co.send(2))    # 3: 코루틴에 숫자 2를 보내고 코루틴에서 나온 값 출력
print(co.send(3))    # 6: 코루틴에 숫자 3를 보내고 코루틴에서 나온 값 출력
```

실행 결과

```
0
1
3
6
```

코루틴(coroutin) 활용

코루틴 바깥으로 값 전달하기

- 그다음에 total += x와 같이 받은 값을 누적해줌

```
def sum_coroutine():
    total = 0
    while True:
        x = (yield total)      # 코루틴 바깥에서 값을 받아오면서 바깥으로 값을 전달
        total += n
```

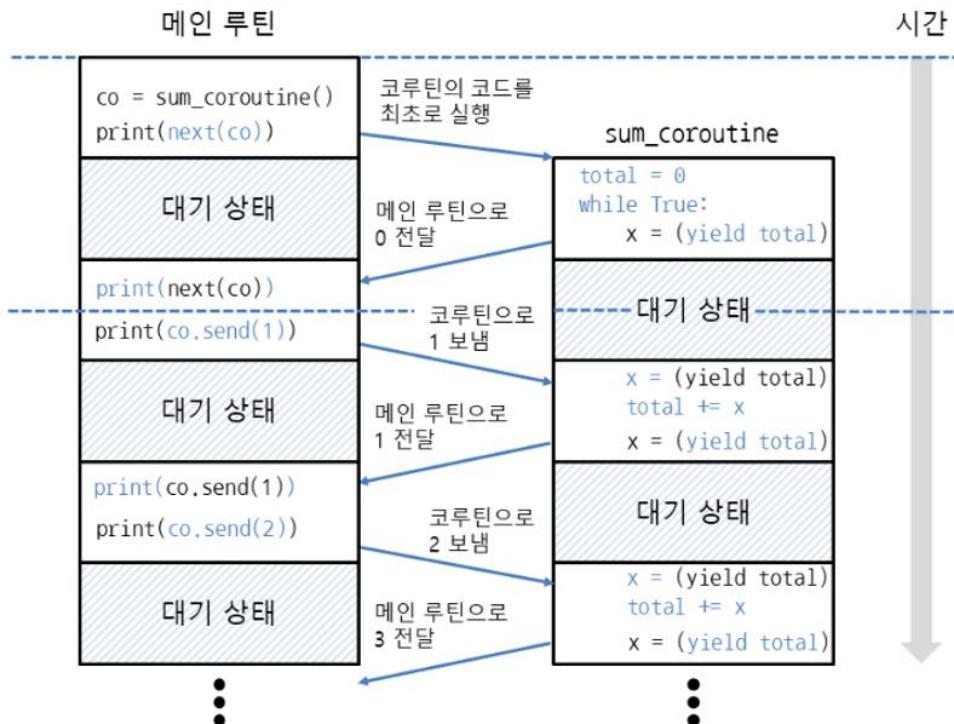
```
co = sum_coroutine()
print(next(co))      # 0: 코루틴 안의 yield까지 코드를 실행하고 코루틴에서 나온 값 출력

print(co.send(1))    # 1: 코루틴에 숫자 1을 보내고 코루틴에서 나온 값 출력
print(co.send(2))    # 3: 코루틴에 숫자 2를 보내고 코루틴에서 나온 값 출력
print(co.send(3))    # 6: 코루틴에 숫자 3을 보내고 코루틴에서 나온 값 출력
```

- 참고로 next와 send의 차이를 살펴보면 next는 코루틴의 코드를 실행하지만 값을 보내지 않을 때 사용하고, send는 값을 보내면서 코루틴의 코드를 실행할 때 사용함

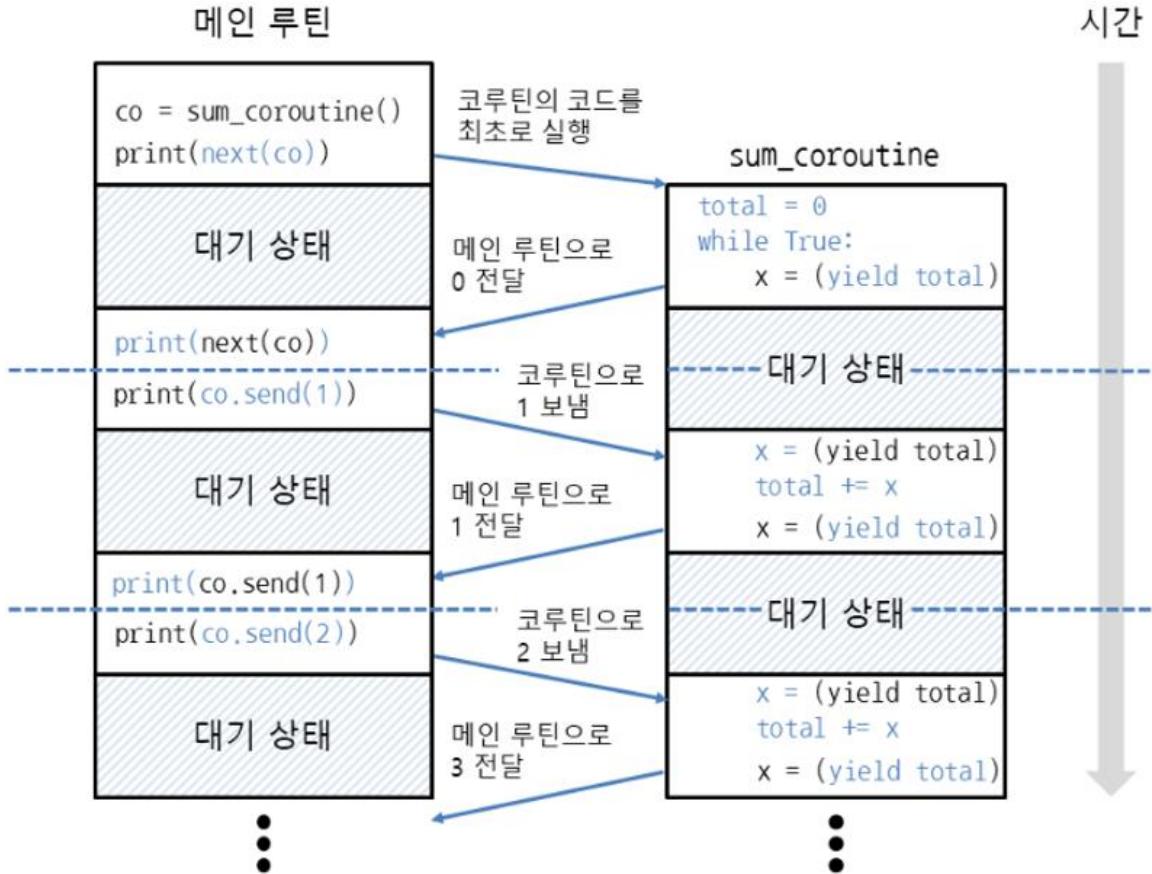
코루틴(coroutine) 활용

▼ 그림 41-5 코루틴의 코드를 최초로 실행한 뒤 yield의 값을 받아와서 출력



코루틴(coroutine) 활용

▼ 그림 41-6 코루틴에 값을 보낸 뒤 결과를 받아서 출력



코루틴(coroutin) 활용

코루틴 바깥으로 값 전달하기

- 제너레이터와 코루틴의 차이점을 정리해보자
 - 제너레이터는 `next` 함수(`__next__` 메서드)를 반복 호출하여 값을 얻어내는 방식
 - 코루틴은 `next` 함수(`__next__` 메서드)를 한 번만 호출한 뒤 `send`로 값을 주고 받는 방식

코루틴(coroutin) 활용

코루틴을 종료하고 예외 처리하기

- 코루틴은 실행 상태를 유지하기 위해 while True:를 사용해서 끝나지 않는 무한 루프로 동작함
- 코루틴을 강제로 종료하고 싶다면 close 메서드를 사용함
 - 코루틴객체.close()

coroutine_close.py

```
def number_coroutine():
    while True:
        x = (yield)
        print(x, end=' ')

co = number_coroutine()
next(co)

for i in range(20):
    co.send(i)

co.close()    # 코루틴 종료
```

실행 결과

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

코루틴(coroutin) 활용

GeneratorExit 예외 처리하기

- 코루틴 객체에서 close 메서드를 호출하면 코루틴이 종료될 때 GeneratorExit 예외가 발생함
- 이 예외를 처리하면 코루틴의 종료 시점을 알 수 있음

coroutine_generator_exit.py

```
def number_coroutine():
    try:
        while True:
            x = (yield)
            print(x, end=' ')
    except GeneratorExit:    # 코루틴이 종료 될 때 GeneratorExit 예외 발생
        print()
        print('코루틴 종료')

co = number_coroutine()
next(co)

for i in range(20):
    co.send(i)

co.close()
```

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
코루틴 종료

코루틴(coroutin) 활용

GeneratorExit 예외 처리하기

- 코루틴 안에서 try except로 GeneratorExit 예외가 발생하면 '코루틴 종료'가 출력되도록 만들었음
- close 메서드로 코루틴을 종료할 때 원하는 코드를 실행할 수 있음

```
except GeneratorExit:    # 코루틴이 종료 될 때 GeneratorExit 예외 발생
    print()
    print('코루틴 종료')
```

코루틴(coroutin) 활용

코루틴 안에서 예외 발생시키기

- 코루틴 안에 예외를 발생 시킬 때는 throw 메서드를 사용함
- throw는 말그대로 던지다라는 뜻인데 예외를 코루틴 안으로 던짐
- throw 메서드에 지정한 에러 메시지는 except as의 변수에 들어감
- 코루틴객체.throw(예외이름, 에러메시지)

코루틴(coroutin) 활용

코루틴 안에서 예외 발생시키기

- 다음은 코루틴에 숫자를 보내서 누적하다가 RuntimeError 예외가 발생하면 에러 메시지를 출력하고 누적된 값을 코루틴 바깥으로 전달함

coroutine_throw.py

```
def sum_coroutine():
    try:
        total = 0
        while True:
            x = (yield)
            total += x
    except RuntimeError as e:
        print(e)
        yield total    # 코루틴 바깥으로 값 전달

co = sum_coroutine()
next(co)

for i in range(20):
    co.send(i)

print(co.throw(RuntimeError, '예외로 코루틴 끝내기')) # 190
                                                # 코루틴의 except에서 yield로 전달받은 값
```

실행 결과

예외로 코루틴 끝내기

190

코루틴(coroutin) 활용

코루틴 안에서 예외 발생시키기

```
except RuntimeError as e:  
    print(e)  
    yield total    # 코루틴 바깥으로 값 전달
```

코루틴(coroutin) 활용

하위 코루틴의 반환값 가져오기

- yield from에 코루틴을 지정하면 해당 코루틴에서 return으로 반환한 값을 가져옴(yield from은 파이썬 3.3 이상부터 사용 가능)
 - 변수 = `yield from 코루틴()`

코루틴(coroutin) 활용

하위 코루틴의 반환값 가져오기

- 다음은 코루틴에서 숫자를 누적한 뒤 합계를 yield from으로 가져옴

coroutine_yield_from.py

```
def accumulate():
    total = 0
    while True:
        x = (yield)      # 코루틴 바깥에서 값을 받아옴
        if x is None:   # 받아온 값이 None이면
            return total # 함께 total을 반환
        total += x

def sum_coroutine():
    while True:
        total = yield from accumulate()    # accumulate의 반환값을 가져옴
        print(total)

co = sum_coroutine()
next(co)

for i in range(1, 11):    # 1부터 10)까지 반복
    co.send(i)            # 코루틴 accumulate에 숫자를 보냄
co.send(None)             # 코루틴 accumulate에 None을 보내서 숫자 누적을 끝냄

for i in range(1, 101):   # 1부터 100)까지 반복
    co.send(i)            # 코루틴 accumulate에 숫자를 보냄
co.send(None)             # 코루틴 accumulate에 None을 보내서 숫자 누적을 끝냄
```

실행 결과

55

5050

코루틴(coroutin) 활용

하위 코루틴의 반환값 가져오기

- 먼저 숫자를 받아서 누적할 코루틴을 만듬

```
def accumulate():
    total = 0
    while True:
        x = (yield)          # 코루틴 바깥에서 값을 받아옴
        if x is None:        # 받아온 값이 None이면
            return total    # 합계 total을 반환, 코루틴을 끝냄
        total += x
```

- 이제 합계를 출력할 코루틴을 만듬

```
def sum_coroutine():
    while True:
        total = yield from accumulate()    # accumulate의 반환값을 가져옴
        print(total)
```

코루틴(coroutin) 활용

하위 코루틴의 반환값 가져오기

- 코루틴에서 `yield from`을 사용하면 코루틴 바깥에서 `send`로 하위 코루틴까지 값을 보낼 수 있음

```
co = sum_coroutine()
next(co)

for i in range(1, 11):    # 1부터 10까지 반복
    co.send(i)            # 코루틴 accumulate에 숫자를 보냄
```

```
co.send(None)             # 코루틴 accumulate에 None을 보내서 숫자 누적을 끝냄
```

```
def sum_coroutine():
    while True:
        total = yield from accumulate()    # accumulate가 끝나면 yield from으로 다시 실행
        print(total)
```

코루틴(coroutin) 활용

StopIteration 예외 발생시키기(파이썬 3.6)

- 코루틴도 제너레이터이므로 return을 사용하면 StopIteration이 발생함
- 코루틴에서 return 값은 raise StopIteration(값)과 동작이 같음
- raise로 예외를 직접 발생시키고 값을 지정하면 yield from으로 값을 가져올 수 있음
- `raise StopIteration(값)`

파이썬 3.7 변경 사항

- 제너레이터에서 사용되는 StopIteration을 Runtime Error로 변경했기 때문에 raise StopIteration을 사용할 수 없습니다.
- raise StopIteration 대신 return 사용으로 수정

코루틴(coroutin) 활용

StopIteration 예외 발생시키기

coroutine_stopiteration.py

```
def accumulate():
    total = 0
    while True:
        x = (yield)
                    # 코루틴 바깥에서 값을 받아옴
        if x is None:
                    # 받아온 값이 None이면
            raise StopIteration(total)      # StopIteration에 반환할 값을 지정
        total += x
```

파이썬 3.7에서는 다음과 같이 수정
return total

```
def sum_coroutine():
    while True:
        total = yield from accumulate()    # accumulate의 반환값을 가져옴
        print(total)
```

```
co = sum_coroutine()
next(co)

for i in range(1, 11):    # 1부터 10)까지 반복
    co.send(i)            # 코루틴 accumulate에 숫자를 보냄
co.send(None)             # 코루틴 accumulate에 None을 보내서 숫자 누적을 끝냄

for i in range(1, 101):   # 1부터 100)까지 반복
    co.send(i)            # 코루틴 accumulate에 숫자를 보냄
co.send(None)             # 코루틴 accumulate에 None을 보내서 숫자 누적을 끝냄
```

실행 결과

```
55
5050
```

정규표현식(Regular expression) 사용

정규표현식 사용하기

- 정규표현식(regular expression)은 일정한 규칙(패턴)을 가진 문자열을 표현하는 방법
- 복잡한 문자열 속에서 특정한 규칙으로 된 문자열을 검색한 뒤 추출하거나 바꿀 때 사용함
- 문자열이 정해진 규칙에 맞는지 판단할 때도 사용함

정규표현식(Regular expression) 사용

문자열 판단하기

- 정규표현식은 re 모듈을 가져와서 사용하며 match 함수에 정규표현식 패턴과 판단할 문자열을 넣음(re는 regular expression의 약자)
 - `re.match('패턴', '문자열')`
- 다음은 'Hello, world!' 문자열에 'Hello'와 'Python'이 있는지 판단함

```
>>> import re
>>> re.match('Hello', 'Hello, world!')      # 문자열이 있으므로 정규표현식 매치 객체가 반환됨
<sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> re.match('Python', 'Hello, world!')     # 문자열이 없으므로 아무것도 반환되지 않음
```

- 여기서는 'Hello'가 있으므로 match='Hello'와 같이 패턴에 매칭된 문자열이 표시됨
- 'Hello, world!.find('Hello')처럼 문자열 메서드로도 충분히 가능함

정규표현식(Regular expression) 사용

문자열이 맨 앞에 오는지 맨 뒤에 오는지 판단하기

- 문자열 앞에 ^를 붙이면 문자열이 맨 앞에 오는지 판단하고, 문자열 뒤에 \$를 붙이면 문자열이 맨 뒤에 오는지 판단함(특정 문자열로 끝나는지)
- ^문자열
- 문자열\$
- 이때는 match 대신 search 함수를 사용해야 함
- match 함수는 문자열 처음부터 매칭되는지 판단하지만, search는 문자열 일부분이 매칭되는지 판단함
- `re.search('패턴', '문자열')`

```
>>> re.search('^Hello', 'Hello, world!')      # Hello로 시작하므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> re.search('world!$', 'Hello, world!')    # world!로 끝나므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(7, 13), match='world!'>
```

정규표현식(Regular expression) 사용

지정된 문자열이 하나라도 포함되는지 판단하기

- 기본 개념은 OR 연산자와 같음
 - 문자열|문자열
 - 문자열|문자열|문자열|문자열
- 'hello|world'는 문자열에서 'hello' 또는 'world'가 포함되는지 판단함

```
>>> re.match('hello|world', 'hello')      # hello 또는 world가 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 5), match='hello'>
```

정규표현식(Regular expression) 사용

범위 판단하기

- 다음과 같이 [](대괄호) 안에 숫자 범위를 넣으며 * 또는 +를 붙임
- 숫자 범위는 0-9처럼 표현하며 *는 문자(숫자)가 0개 이상 있는지, +는 1개 이상 있는지 판단함
- `[0-9]+`

```
>>> re.match('[0-9]*', '1234')      # 1234는 0부터 9까지 숫자가 0개 이상 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 4), match='1234'>
>>> re.match('[0-9]+', '1234')      # 1234는 0부터 9까지 숫자가 1개 이상 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 4), match='1234'>
>>> re.match('[0-9]+', 'abcd')     # 1234는 0부터 9까지 숫자가 1개 이상 없으므로 패턴에 매칭되지 않음
```

- *와 + 활용은 다음과 같이 a*b와 a+b를 확인해보면 쉽게 알 수 있음

```
>>> re.match('a*b', 'b')        # b에는 a가 0개 이상 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 1), match='b'>
>>> re.match('a+b', 'b')        # b에는 a가 1개 이상 없으므로 패턴에 매칭되지 않음
>>> re.match('a*b', 'aab')      # aab에는 a가 0개 이상 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 3), match='aab'>
>>> re.match('a+b', 'aab')      # aab에는 a가 1개 이상 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 3), match='aab'>
```

정규표현식(Regular expression) 사용

문자가 한 개만 있는지 판단하기

- ?와 .을 사용함
- ?는 ? 앞의 문자(범위)가 0개 또는 1개인지 판단하고, .은 .이 있는 위치에 아무 문자(숫자)가 1개 있는지 판단합니다.
- 문자?
- [0-9]?
- .

```
>>> re.match('abc?d', 'abd')          # abd에서 c 위치에 c가 0개 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 3), match='abd'>
>>> re.match('ab[0-9]?c', 'ab3c')    # [0-9] 위치에 숫자가 1개 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 4), match='ab3c'>
>>> re.match('ab.d', 'abxd')        # .이 있는 위치에 문자가 1개 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 4), match='abxd'>
```

정규표현식(Regular expression) 사용

문자 개수 판단하기

- 문자열의 경우에는 문자열을 괄호로 묶고 뒤에 {개수} 형식을 지정함
 - 문자{개수}
 - (문자열){개수}
- h{3}은 h가 3개 있는지 판단하고, (hello){3}은 hello가 3개 있는지 판단함

```
>>> re.match('h{3}', 'hhhello')
<_sre.SRE_Match object; span=(0, 3), match='hhh'>
>>> re.match('(hello){3}', 'hellohellohelloworld')
<_sre.SRE_Match object; span=(0, 15), match='hellohellohello'>
```

정규표현식(Regular expression) 사용

문자 개수 판단하기

- 특정 범위의 문자(숫자)가 몇 개 있는지 판단할 수도 있음
- 이때는 범위 [] 뒤에 {개수} 형식을 지정함
- [0-9]{개수}**
- 다음은 휴대전화의 번호 형식에 맞는지 판단함

```
>>> re.match('[0-9]{3}-[0-9]{4}-[0-9]{4}', '010-1000-1000')      # 숫자 3개-4개-4개 패턴에 매칭됨
<sre.SRE_Match object; span=(0, 13), match='010-1000-1000'>
>>> re.match('[0-9]{3}-[0-9]{4}-[0-9]{4}', '010-1000-100')     # 숫자 3개-4개-4개 패턴에 매칭되지 않음
```

정규표현식(Regular expression) 사용

문자 개수 판단하기

- 이 기능은 문자(숫자)의 개수 범위도 지정할 수 있음
- {시작개수,끝개수} 형식으로 시작 개수와 끝 개수를 지정해주면 특정 개수 사이에 들어가는지 판단함
- (문자){시작개수,끝개수}
- (문자열){시작개수,끝개수}
- [0-9]{시작개수,끝개수}
- 다음은 일반전화의 번호 형식에 맞는지 판단함

```
>>> re.match('[0-9]{2,3}-[0-9]{3,4}-[0-9]{4}', '02-100-1000')      # 2~3개-3~4개-4개 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 11), match='02-100-1000'>
>>> re.match('[0-9]{2,3}-[0-9]{3,4}-[0-9]{4}', '02-10-1000')    # 2~3개-3~4개-4개 패턴에 매칭되지 않음
```

정규표현식(Regular expression) 사용

숫자와 영문 문자를 조합해서 판단하기

- 영문 문자 범위는 a-z, A-Z와 같이 표현함
 - a-z
 - A-Z

```
>>> re.match('[a-zA-Z0-9]+', 'Hello1234')    # a부터 z, A부터 Z, 0부터 9까지 1개 이상 있으므로
<_sre.SRE_Match object; span=(0, 9), match='Hello1234'>                      # 패턴에 매칭됨
>>> re.match('[A-Z0-9]+', 'hello')      # 대문자, 숫자는 없고 소문자만 있으므로 패턴에 매칭되지 않음
```

- 한글은 영문 문자와 방법이 같음
- 가-힝처럼 나올 수 있는 한글 조합을 정해주면 됨

정규표현식(Regular expression) 사용

특정 문자 범위에 포함되지 않는지 판단하기

- 특정 문자 범위에 포함되지 않는지 판단하려면 다음과 같이 문자(숫자) 범위 앞에 ^를 붙이면 해당 범위를 제외함
 - [^범위]*
 - [^범위]+
- '[^A-Z]+'는 대문자를 제외한 모든 문자(숫자)가 1개 이상 있는지 판단함

```
>>> re.match('[^A-Z]+', 'Hello')    # 대문자를 제외. 대문자가 있으므로 패턴에 매칭되지 않음
>>> re.match('[^A-Z]+', 'hello')   # 대문자를 제외. 대문자가 없으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 5), match='hello'>
```

정규표현식(Regular expression) 사용

특정 문자 범위에 포함되지 않는지 판단하기

- 범위를 제외할 때는 '[^A-Z]+'와 같이 [] 안에 넣어주고, 특정 문자 범위로 시작할 때는 '^A-Z+'와 같이 [] 앞에 붙여줌
- 다음과 같이 '^A-Z+'는 영문 대문자로 시작하는지 판단함
 - ^[범위]*
 - ^[범위]+

```
>>> re.search('^[A-Z]+', 'Hello')      # 대문자로 시작하므로 패턴에 매칭됨
<sre.SRE_Match object; span=(0, 1), match='H'>
```

정규표현식(Regular expression) 사용

특정 문자 범위에 포함되지 않는지 판단하기

- 특정 문자(숫자) 범위로 끝나는지 확인할 때는 정규표현식 뒤에 \$를 붙이면 됨
 - [범위]*\$
 - [범위]+\$

```
>>> re.search('[0-9]+$', 'Hello1234')      # 숫자로 끝나므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(5, 9), match='1234'>
```

정규표현식(Regular expression) 사용

특수 문자 판단하기

- 정규표현식에 사용하는 특수 문자 *, +, ?, ., ^, \$, (,), [,], - 등을 판단하려면 특수 문자를 판단할 때는 특수 문자 앞에 \#을 붙이면 됨
- 단, [] 안에서는 \#을 붙이지 않아도 되지만 에러가 발생하는 경우에는 \#을 붙임
- \특수문자

```
>>> re.search('\*+', '1 ** 2')                      # *이 들어있는지 판단
<_sre.SRE_Match object; span=(2, 4), match='**'>
>>> re.match('[$(()a-zA-Z0-9]+', '$(document)')    # $, (, )와 문자, 숫자가 들어있는지 판단
<_sre.SRE_Match object; span=(0, 11), match='$(document)'>
```

정규표현식(Regular expression) 사용

특수 문자 판단하기

- 단순히 숫자인지 문자인지 판단할 때는 \d, \D, \w, \W를 사용하면 편리함
 - \d: [0-9]와 같음. 모든 숫자
 - \D: [^0-9]와 같음. 숫자를 제외한 모든 문자
 - \w: [a-zA-Z0-9_]와 같음. 영문 대소문자, 숫자, 밑줄 문자
 - \W: [^a-zA-Z0-9_]와 같음. 영문 대소문자, 숫자, 밑줄 문자를 제외한 모든 문자

```
>>> re.match('\d+', '1234')          # 모든 숫자이므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 4), match='1234'>
>>> re.match('\D+', 'Hello')        # 숫자를 제외한 모든 문자이므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> re.match('\w+', 'Hello_1234')   # 영문 대소문자, 숫자, 밑줄 문자이므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 10), match='Hello_1234'>
>>> re.match('\W+', '(:)')         # 영문 대소문자, 숫자, 밑줄문자를 제외한 모든 문자이므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 3), match='(:)'>
```

정규표현식(Regular expression) 사용

공백 처리하기

- 공백은 ''처럼 공백 문자를 넣어도 되고, \s 또는 \S로 표현할 수도 있음
 - \s: [\t\n\r\f\v]와 같음. 공백(스페이스), \t(탭) \n(새 줄, 라인 피드), \r(캐리지 리턴), \f(폼피드), \v(수직 탭)을 포함
 - \S: [^\t\n\r\f\v]와 같음. 공백을 제외하고 \t, \n, \r, \f, \v만 포함

```
>>> re.match('[a-zA-Z0-9 ]+', 'Hello 1234')      # ''로 공백 표현
<_sre.SRE_Match object; span=(0, 10), match='Hello 1234'>
>>> re.match('[a-zA-Z0-9\s]+', 'Hello 1234')    # \s로 공백 표현
<_sre.SRE_Match object; span=(0, 10), match='Hello 1234'>
```

정규표현식(Regular expression) 사용

그룹 사용하기

- 정규표현식 그룹은 해당 그룹과 일치하는 문자열을 얻어올 때 사용함
 - (정규표현식) (정규표현식)
- 다음은 공백으로 구분된 숫자를 두 그룹으로 나누어서 찾은 뒤 각 그룹에 해당하는 문자열(숫자)을 가져옴
 - 매치객체.group(그룹숫자)

```
>>> m = re.match('([0-9]+) ([0-9]+)', '10 295')
>>> m.group(1)      # 첫 번째 그룹(그룹 1)에 매칭된 문자열을 반환
'10'
>>> m.group(2)      # 두 번째 그룹(그룹 2)에 매칭된 문자열을 반환
'295'
>>> m.group()       # 매칭된 문자열을 한꺼번에 반환
'10 295'
>>> m.group(0)       # 매칭된 문자열을 한꺼번에 반환
'10 295'
```

정규표현식(Regular expression) 사용

그룹 사용하기

- 매치객체.groups()

```
>>> m.groups()    # 각 그룹에 해당하는 문자열을 튜플 형태로 반환  
('10', '295')
```

- 그룹 개수가 많아지면 숫자로 그룹을 구분하기가 힘들어짐
- 그룹에 이름을 지으면 편리함
- (?P<이름>정규표현식)
- 다음은 함수를 호출하는 코드 print(1234)에서 함수의 이름 print와 인수 1234를
+ + +
 - 매치객체.group('그룹이름')

```
>>> m = re.match('(?P<func>[a-zA-Z_][a-zA-Z0-9_]+)\((?P<arg>\w+)\)', 'print(1234)')  
>>> m.group('func')    # 그룹 이름으로 매칭된 문자열 출력  
'print'  
>>> m.group('arg')    # 그룹 이름으로 매칭된 문자열 출력  
'1234'
```

정규표현식(Regular expression) 사용

패턴에 매칭되는 모든 문자열 가져오기

- 그룹 지정 없이 패턴에 매칭되는 모든 문자열을 가져오려면 `findall` 함수를 사용하며 매칭된 문자열을 리스트로 반환함
 - `re.findall('패턴', '문자열')`
 - 다음은 문자열에서 숫자만 가져옴

```
>>> re.findall('[0-9]+', '1 2 Fizz 4 Buzz Fizz 7 8')
['1', '2', '4', '7', '8']
```

정규식 패턴

패턴	설명	예제
^	이 패턴으로 시작해야 함	^abc : abc로 시작해야 함 (abcd, abc12 등)
\$	이 패턴으로 종료되어야 함	xyz\$: xyz로 종료되어야 함 (123xyz, strxyz 등)
[문자들]	문자들 중에 하나이어야 함. 가능한 문자들의 집합을 정의함.	[Pp]ython : "Python" 혹은 "python"
[^문자들]	[문자들]의 반대로 피해야 할 문자들의 집합을 정의함.	[^aeiou] : 소문자 모음이 아닌 문자들
	두 패턴 중 하나이어야 함 (OR 기능)	a b : a 또는 b 이어야 함
?	앞 패턴이 없거나 하나이어야 함 (Optional 패턴을 정의할 때 사용)	\d? : 숫자가 하나 있거나 없어야 함
+	앞 패턴이 하나 이상이어야 함	\d+ : 숫자가 하나 이상이어야 함
*	앞 패턴이 0개 이상이어야 함	\d* : 숫자가 없거나 하나 이상이어야 함
패턴{n}	앞 패턴이 n번 반복해서 나타나는 경우	\d{3} : 숫자가 3개 있어야 함
패턴{n, m}	앞 패턴이 최소 n번, 최대 m 번 반복해서 나타나는 경우 (n 또는 m은 생략 가능)	\d{3,5} : 숫자가 3개, 4개 혹은 5개 있어야 함
\d	숫자 0 ~ 9	\d\d\d : 0 ~ 9 범위의 숫자가 3개를 의미 (123, 000 등)
\w	문자를 의미	\w\w\w : 문자가 3개를 의미 (xyz, ABC 등)
\s	화이트 스페이스를 의미하는데, [\t\n\r\f] 와 동일	\s\s : 화이트 스페이스 문자 2개 의미 (\r\n, \t\t 등)
.	뉴라인(\n)을 제외한 모든 문자를 의미	.{3} : 문자 3개 (F15, 0x0 등)

특히, 숫자와 문자와 관련된 것

메타 문자	의미
\d	숫자를 의미한다. 그러므로 [0-9]와 동일한 의미이다.
\D	숫자를 제외한 것을 의미한다. 따라서 [^0-9]와 동일한 의미이다.
\s	공백문자(white space)를 의미한다.
\S	공백문자를 제외한 것을 의미한다.
\b	단어의 시작과 끝의 빈공백을 의미한다.
\B	단어의 시작과 끝의 빈공백이 아닌 빈공백을 의미한다.
\w	숫자와 알파벳 문자를 의미한다. 따라서 [a-zA-Z0-9]와 동일한 의미이다.
\W	숫자와 알파벳 문자를 제외한 것을 의미한다.

정규표현식(Regular expression) 사용

문자열 바꾸기

- 문자열을 바꿀 때는 sub 함수를 사용하며 패턴, 바꿀 문자열, 문자열, 바꿀 횟수를 넣어줌
- `re.sub('패턴', '바꿀문자열', '문자열', 바꿀횟수)` 바꿀 횟수를 생략하면 찾은

```
>>> re.sub('apple|orange', 'fruit', 'apple box orange tree')    # apple 또는 orange를 fruit로 바꿈  
'fruit box fruit tree'
```

```
>>> re.sub('[0-9]+', 'n', '1 2 Fizz 4 Buzz Fizz 7 8')    # 숫자만 찾아서 n으로 바꿈  
'n n Fizz n Buzz Fizz n n'
```

정규표현식(Regular expression) 사용

문자열 바꾸기

- `sub` 함수는 바꿀 문자열 대신 교체 함수를 지정할 수도 있음
- 교체 함수는 매개변수로 매치 객체를 받으며 바꿀 결과를 문자열로 반환하면 됨
- 다음은 문자열에서 숫자를 찾은 뒤 숫자를 10배로 만듬
 - 교체함수(매치객체)
 - `re.sub('패턴', 교체함수, '문자열', 바꿀횟수)`

```
>>> def multiple10(m):          # 매개변수로 매치 객체를 받음
...     n = int(m.group())       # 매칭된 문자열을 가져와서 정수로 변환
...     return str(n * 10)        # 숫자에 10을 곱한 뒤 문자열로 변환해서 반환
...
>>> re.sub('[0-9]+', multiple10, '1 2 Fizz 4 Buzz Fizz 7 8')
'10 20 Fizz 40 Buzz Fizz 70 80'
```

- 교체 함수의 내용이 간단하다면 다음과 같이 람다 표현식을 만들어서 넣어도 됨

```
>>> re.sub('[0-9]+', lambda m: str(int(m.group()) * 10), '1 2 Fizz 4 Buzz Fizz 7 8')
'10 20 Fizz 40 Buzz Fizz 70 80'
```

정규표현식(Regular expression) 사용

찾은 문자열을 결과에 다시 사용하기

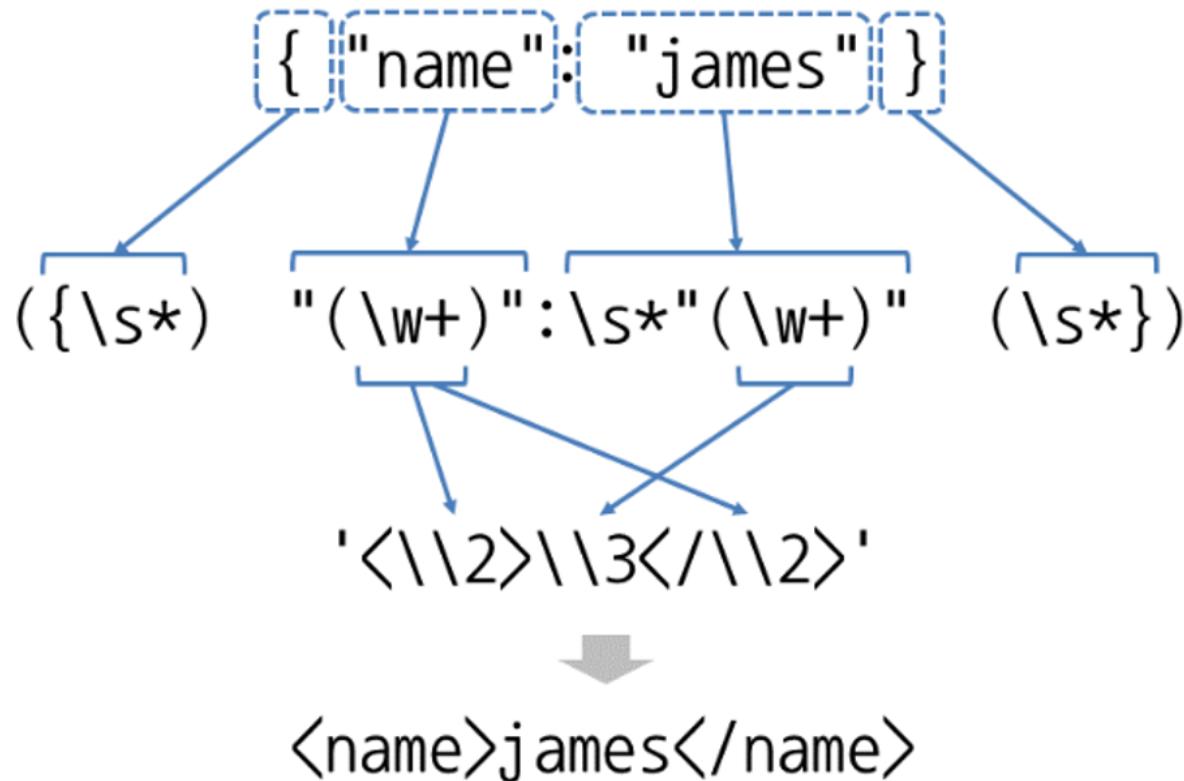
- 정규표현식을 그룹으로 묶음
- 바꿀 문자열에서 그룹수자 형식으로 매칭된 문자열을 가져와서 사용할 수 있음
 - \\\ 숫자

```
>>> re.sub('([a-z]+) ([0-9]+)', '\g<2> \g<1> \g<2> \g<1>', 'hello 1234')    # 그룹 2, 1, 2, 1 순으로 바꿈
'1234 hello 1234 hello'
```

```
>>> re.sub('({\s*})(\w+):\s*"(\\w+)"(\s*)', '<\g<2>>\g<3></\g<2>>', '{ "name": "james" }')
'<name>james</name>'
```

정규표현식(Regular expression) 사용

▼ 그림 43-1 정규 표현식으로 찾은 문자열을 사용



정규표현식(Regular expression) 사용

찾은 문자열을 결과에 다시 사용하기

- 만약 그룹에 이름을 지었다면 `\g<이름>` 형식으로 매칭된 문자열을 가져올 수 있음(`\g<숫자>` 형식으로 숫자를 지정해도 됨)
 - `\g<이름>`
 - `\g<숫자>`

```
>>> re.sub('(\s*)(?P<key>\w+):\s*(?P<value>\w+)(\s*)', '<\g<key>>\g<value></\g<key>>', '{ "name": "james" }')
'<name>james</name>'
```

문제1

▣ 간단한 카운터 코루틴

- 주어진 수에서 시작하여 받은 값을 더해 나가는 카운터 코루틴 counter()를 구현하세요.

문제1

```
def counter(start=0):
    current = start
    while True:
        increment = yield current
        if increment is None:
            increment = 1
        current += increment
```

```
# 코루틴 실행
count = counter(10)
print(next(count))  # 10 출력
print(count.send(5))  # 15 출력
print(count.send(3))  # 18 출력
```

문제2

▣ 반복적 요청 처리 코루틴

- 사용자로부터 입력을 받아 특정 작업을 반복적으로 수행하는 코루틴 repeat_task()를 구현하세요.

문제2

```
def repeat_task():
    while True:
        task = yield
        print(f"Task {task} completed!")
```

코루틴 실행

```
task_handler = repeat_task()
next(task_handler) # 코루틴 초기화
task_handler.send("Upload")
task_handler.send("Download")
task_handler.send("Sync")
```

문제3

▣ 이메일 주소 검증

- 주어진 문자열이 유효한 이메일 주소인지 확인하는 정규식을 작성하세요.

문제3

```
import re

def validate_email(email):
    pattern = r'^[a-zA-Z0-9._]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}\$'
    if re.match(pattern, email):
        return "Valid email"
    else:
        return "Invalid email"

# 함수 호출 및 출력
print(validate_email("example@email.com")) # Valid email
print(validate_email("example.email.com")) # Invalid email
```

문제4

▣ 전화번호 형식 확인

- 주어진 문자열이 "(xxx) xxx-xxxx" 형식의 전화번호인지 확인하는 정규식을 작성하세요.

문제4

```
import re

def validate_phone_number(number):
    pattern = r'^(\d{3}) (\d{3}-\d{4})$'
    if re.match(pattern, number):
        return "Valid phone number"
    else:
        return "Invalid phone number"

# 함수 호출 및 출력
print(validate_phone_number("(123) 456-7890")) # Valid phone number
print(validate_phone_number("123-456-7890")) # Invalid phone number
```

문제5

▣ HTML 태그 제거

- 주어진 HTML 문자열에서 모든 HTML 태그를 제거하고 텍스트 내용만 반환하는 정규식을 사용하세요.

문제5

```
import re

def remove_html_tags(html):
    pattern = r'<[^>]+>'
    text = re.sub(pattern, '', html)
    return text

# 함수 호출 및 출력
html_content = "<html><head><title>Test</title></head><body>Hello,  
world!</body></html>"
print(remove_html_tags(html_content)) # Hello, world!
```

문제6

▣ 비밀번호 복잡성 검사

- 주어진 비밀번호가 다음 조건을 모두 만족하는지 확인하세요: 최소 8자, 하나 이상의 대문자, 하나 이상의 소문자, 하나 이상의 숫자.

문제6

```
import re

def validate_password(password):
    pattern = r'^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[A-Za-z\d]{8,}\$'
    if re.match(pattern, password):
        return "Strong password"
    else:
        return "Weak password"

# 함수 호출 및 출력
print(validate_password("Password123")) # Strong password
print(validate_password("password"))     # Weak password
```

문제7

▣ URL 파싱

- 주어진 URL에서 프로토콜, 호스트, 옵션적으로 포트 번호를 추출하는 정규식을 작성하세요.

문제7

```
import re

def parse_url(url):
    pattern = r'^((https?://)?([^\/:/\s]+)(?::(\d+))?)'
    match = re.match(pattern, url)
    if match:
        return {"protocol": match.group(1), "host": match.group(2),
"port": match.group(3)}
    else:
        return "Invalid URL"

# 함수 호출 및 출력

print(parse_url("http://example.com:8080")) # {'protocol': 'http',
'host': 'example.com', 'port': '8080'}

print(parse_url("https://example.com"))      # {'protocol': 'https',
'host': 'example.com', 'port': None}
```

수고하셨습니다!!

