핵심 정리

예외 처리

예외란 코드 실행 중에 발생한 에러를 뜻합니다. 예외 처리를 하려면 `try`에 실행할 코드를 넣고 `except`에 예외가 발생했을 때 처리할 코드를 넣어줍니다. 그리고 `else`는 예외가 발생하지 않았을 때 코드를 실행하며 `finally`는 예외 발생 여부와 상관없이 항상 코드를 실행합니다.

```
try:
    실행할 코드
except:
    예외가 발생했을 때 처리하는 코드
else:
    예외가 발생하지 않았을 때 실행할 코드
finally:
    예외 발생 여부와 상관없이 항상 실행할 코드
```

`try`의 코드가 에러 없이 잘 실행되면 `except`의 코드는 실행되지 않으며 `try`의 코드에서 에러가 발생했을 때만 `except`의 코드가 실행됩니다.

`except`에 예외 이름을 지정하면 특정 예외가 발생했을 때만 처리 코드를 실행할 수 있습니다. 그리고 `except`에서 `as` 뒤에 변수를 지정하면 발생한 예외의 에러 메시지를 받아옵니다.

```
try:
    실행할 코드
except 예외이름:            # 특정 예외가 발생했을 때만 처리 코드를 실행
    예외가 발생했을 때 처리하는 코드

try:
    실행할 코드
except 예외이름 as 변수:    # 발생한 예외의 에러 메시지가 변수에 들어감
    예외가 발생했을 때 처리하는 코드
```

예외 발생시키기

예외를 발생시킬 때는 `raise`에 `Exception`을 지정하고 에러 메시지를 넣습니다.

```
try:
    raise Exception(에러메시지)    # 예외를 발생시킴
except Exception as e:             # 예외가 발생했을 때 실행됨
    print(e)                       # Exception에 지정한 에러 메시지가 e에 들어감
```

`except` 안에서 `raise`만 사용하면 현재 예외를 다시 상위 코드 블록으로 넘깁니다.

```
try:
    함수A()
except Exception as e:                # 하위 코드 블록에서 예외가 발생해도 실행됨
    print(e)
```

예외 만들기

예외를 만들 때는 `Exception`을 상속받아서 새로운 클래스를 만들고, `__init__` 메서드에서 기반 클래스의 `__init__` 메서드를 호출하면서 에러 메시지를 넣어줍니다. 예외를 발생시킬 때는 `raise`에 새로 만든 예외를 지정해줍니다.

```
class 예외이름(Exception):    # 예외 만들기
    def __init__(self):
        super().__init__('에러메시지')

raise 예외                    # 예외 발생 시키기
```

반복 가능한 객체와 이터레이터

반복 가능한 객체는 문자열, 리스트, 튜플, `range`, 딕셔너리, 세트 등이 있습니다. 반복 가능한 객체에서 `__iter__` 메서드 또는 `iter` 함수를 호출하면 이터레이터가 나옵니다. 이터레이터에서 `__next__` 메서드 또는 `next` 함수를 호출하면 반복 가능한 객체의 요소를 차례대로 꺼낼 수 있습니다.

```
이터레이터 = 반복가능한객체.__iter__()    # 반복가능한 객체에서 이터레이터를 얻음
이터레이터.__next__()                     # 반복 가능한 객체의 요소를 차례대로  꺼냄

이터레이터 = iter(반복가능한객체)         # iter 함수 사용
next(이터레이터)                          # next 함수 사용
```

반복 가능한 객체는 요소를 한 번에 하나씩 꺼낼 수 있는 객체이고, 이터레이터는 `__next__` 메서드를 사용해서 차례대로 값을 꺼낼 수 있는 객체입니다.

이터레이터 만들기

클래스에서 `__iter__`, `__next__` 메서드를 구현하면 이터레이터가 됩니다. 또한, 이렇게 만든 이터레이터는 반복 가능한 객체이면서 이터레이터입니다.

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

    def __next__(self):
        값 생성 코드, 반복을 끝내려면 StopIteration 예외를 발생시킴

이터레이터객체 = 이터레이터()    # 이터레이터 객체 생성
이터레이터.__next__()            # 이터레이터에서 값을 차례대로 꺼냄
next(이터레이터)                 # next 함수 사용

for i in 이터레이터():    # 이터레이터를 반복문에 사용
    pass
```

클래스에 `__getitem__` 메서드를 구현하면 인덱스로 접근할 수 있는 이터레이터가 됩니다. 이때는 `__iter__`와 `__next__` 메서드는 생략해도 됩니다.

```
class 이터레이터이름:
    def __getitem__(self, index):
        인덱스에 해당하는 값을 반환하는 코드, 지정된 범위를 벗어났다면 IndexError 예외를 발생시킴

이터레이터객체 = 이터레이터()    # 이터레이터 객체 생성
이터레이터객체[인덱스]           # 인덱스로 접근
```

이터레이터는 값을 미리 만들어 놓지 않고, 값이 필요한 시점이 되었을 때 값을 만드는 방식입니다.

이터레이터와 언패킹

이터레이터(제너레이터)는 변수 여러 개에 값을 저장하는 언패킹이 가능합니다.

```
변수1, 변수2, 변수3 = 이터레이터()
```

제너레이터

제너레이터는 이터레이터를 생성해주는 함수이며 함수 안에서 `yield` 키워드만 사용하면 됩니다. 제너레이터 함수를 호출하면 제너레이터 객체가 반환되고, 제너레이터 객체에서 `__next__` 메서드 또는 `next` 함수를 호출하면 `yield`까지 실행한 뒤 `yield`에 지정한 값이 반환값으로 나옵니다.

```
for i in 제너레이터():          # 제너레이터를 반복문에 사용
    pass
```

yield는 값을 함수 바깥으로 전달하면서 코드 실행을 함수 바깥에 양보합니다.

yield from을 사용하면 값을 여러 번 바깥으로 전달합니다.

```
yield from 반복가능한객체
yield from 이터레이터
yield from 제너레이터객체
```

제너레이터 표현식

리스트 표현식을 [ ](대괄호) 대신 ( )(괄호)로 묶으면 제너레이터 표현식이 됩니다.

```
(식 for 변수 in 반복가능한객체)
(i for i in range(100))
```

코루틴

코루틴은 두 루틴이 대등한 관계인 상태에서 특정 시점에 상대방의 코드를 실행하는 방식입니다. 또한, 코루틴은 제너레이터의 특별한 형태이며 yield로 값을 받아올 수 있습니다.

코루틴에 값을 보내면서 코드를 실행할 때는 send 메서드를 사용하고, send 메서드가 보낸 값을 받아오려면 (yield)와 같이 yield를 괄호로 묶어준 뒤 변수에 저장해줍니다.

```
def 코루틴이름():
    while True:
    변수 = (yield)    # 코루틴 바깥에서 값을 받아옴

코루틴객체 = 코루틴()
next(코루틴객체)          # 코루틴 안의 yield까지 코드 실행(최초 실행), __next__ 메서드도 같음
코루틴객체.send(값)       # 코루틴에 값을 보냄
```

코루틴 바깥으로 값을 전달할 때는 yield 뒤에 값(변수)을 지정한 뒤 괄호로 묶어줍니다. yield를 사용하여 바깥으로 전달한 값은 next 함수(__next__ 메서드)와 send 메서드의 반환값으로 나옵니다.

```
def 코루틴이름():
    while True:
    변수 = (yield 변수)    # 코루틴 바깥에서 값을 받아오면서 바깥으로 값을 전달

코루틴객체 = 코루틴()
변수 = next(코루틴객체)        # 코루틴 안의 yield까지 코드를 실행하고 코루틴에서 나온 값 반환
변수 = 코루틴객체.send(값)     # 코루틴에 값을 보내고 코루틴에서 나온 값 반환
```

코루틴 종료와 예외 처리

코루틴을 강제로 종료할 때는 코루틴 객체에서 close 메서드를 사용합니다. 그리고 close 메서드를 사용하면 코루틴이 종료 될 때 GeneratorExit 예외가 발생합니다.

```
def 코루틴이름():
    try:
        실행할 코드
    except GeneratorExit:    # 코루틴이 종료될 때 GeneratorExit 예외 발생
        예외가 발생했을 때 처리하는 코드

코루틴객체 = 코루틴()
next(코루틴객체)
코루틴객체.close()    # 코루틴 종료
```

만약 코루틴 안에 예외를 발생시킬 때는 throw 메서드를 사용합니다. 코루틴 안의 except에서 yield를 사용하여 바깥으로 전달한 값은 throw 메서드의 반환값으로 나옵니다.

```
코루틴객체 = 코루틴()
next(코루틴객체)
코루틴객체.throw(예외이름, 에러메지시)    # 코루틴 안에 예외를 발생시킴
```

하위 코루틴의 반환값 가져오기

`yield from`에 코루틴을 지정하면 해당 코루틴에서 `return`으로 반환한 값을 가져옵니다. 코루틴 안에서 `return 값`은 `raise StopIteration(값)`과 동작이 같습니다.

```
def 코루틴A():
    변수 = (yield)    # 코루틴 바깥에서 값을 받아옴
    return 값         # return으로 값을 반환, raise StopIteration(값)과 동작이 같음

def 코루틴B():
    변수 = yield from 코루틴A()    # 코루틴A의 반환값을 가져옴
```


◀ 41.7 심사문제: 사칙연산 코루틴 만들기

Q & A ▶

내비게이션

홈

코딩 도장

내 강좌

파이썬 코딩 도장

참여자

배지

역량

성적

일반

Unit 1. 소프트웨어 교육과 파이썬

Unit 2. 파이썬 설치하기

Unit 3. Hello, world!로 시작하기

Unit 4. 기본 문법 알아보기

핵심 정리

Unit 5. 숫자 계산하기

Unit 6. 변수와 입력 사용하기

Unit 7. 출력 방법 알아보기

핵심 정리

Unit 8. 불과 비교, 논리 연산자 알아보기

Unit 9. 문자열 사용하기