2026.01.23.금

생성일	@2026년 1월 23일 오전 9:01
태그	Python

목차

▼ 1) 예외 처리

- try-except

```
# 예외 발생 예시들
try:
    # 1. 사용자 입력 오류
    # 숫자가 아닌 문자열을 입력할 경우 ValueError가 발생합니다.
    age = int(input("나이를 입력하세요: "))  # 문자열 입력 시 ValueError

    # 2. 파일 처리 오류
    # 경로에 해당 파일이 없는 경우 FileNotFoundError가 발생합니다.
    with open("존재하지_않는_파일.txt", "r") as file:  # FileNotFoundError
        content = file.read()

    # 3. 계산 오류
    # 어떤 수를 0으로 나누려고 하면 ZeroDivisionError가 발생합니다.
    result = 10 / 0  # ZeroDivisionError

except Exception as e:
    # 발생한 예외를 e라는 변수로 받아와 오류 메시지를 출력합니다.
    print(f"오류가 발생했습니다: {e}")
```

→ 모든 에러를 하나로 처리하는 것은 좋지 않음

- try-except-else-finally

```
try:
    print("파일을 처리합니다.")
    # 1. 파일 열기 및 읽기
    file = open("data.txt", "r")
    content = file.read()

    # 2. 데이터 변환 (문자열 -> 정수)
    value = int(content)

except FileNotFoundError:
    # 파일이 존재하지 않을 때 실행됩니다.
    print("파일을 찾을 수 없습니다.")

except ValueError:
    # 파일 내용은 있으나 숫자로 변환할 수 없는 형식일 때 실행됩니다.
    print("파일 내용을 숫자로 변환할 수 없습니다.")
```

```
except Exception as e:
        print("예상치 못한 에러가 발생했습니다.: " {e})

else:
    # 예외가 발생하지 않고 모든 과정이 성공했을 때만 실행됩니다.
    print(f"파일에서 읽은 숫자: {value}")

    # 추가 연산 수행 (0으로 나누기 오류가 발생할 수 있음)
    result = 100 / value
    print(f"100을 이 숫자로 나눈 결과: {result}")

finally:
    # 예외 발생 여부와 상관없이 항상 실행됩니다.
    print("파일 처리가 완료되었습니다.")

    # 파일 객체가 생성되었는지 확인 후 안전하게 닫습니다.
    if 'file' in locals():
        file.close()
```

- 에러 종류
  - Exception: 예외 기본 클래스
  - RuntimeError: 일반적인 실행 시간 오류
  - TypeError: 타입이 맞지 않을 때 발생
  - ValueError: 타입은 맞지만 값이 적절하지 않을 때 발생
  - ZeroDivisionError: 0으로 나누기를 시도할 때 발생
  - OverflowError: 연산의 결과가 표현 가능한 범위를 초과할 때
  - IndexError: 리스트나 튜플과 같은 시퀀스의 범위를 벗어난 인덱스에 접근할 때 발생
  - KeyError: 딕셔너리에 존재하지 않는 키를 사용할 때 발생
  - NameError: 정의되지 않은 변수를 사용할 때 사용
  - AttributeError: 객체에 존재하지 않는 속성이나 메서드에 접근할 때 발생
  - FileNotFoundError: 존재하지 않는 파일에 접근하려고 할 때 발생
  - PermissionError: 파일 접근 권한이 없을 때 발생
  - SystemExit: sys.exit() 함수가 호출될 때 발생
  - KeyboardInterrupt: 키보드 인터럽트 키(보통 ctrl + c)를 누를 때 발생
- 사용자 정의 예외 처리

```
class InsufficientFundsError(Exception):
    """계좌에 잔액이 부족할 때 발생하는 예외"""

    def __init__(self, balance, amount, message=None):
        # 예외 발생 시점의 잔액과 요청된 출금 금액을 속성으로 저장합니다.
        self.balance = balance
        self.amount = amount
        # 메시지가 제공되지 않았을 경우 기본 안내 메시지를 생성합니다.
        self.message = message or f"잔액 부족: 현재 잔액 {balance}원, 요청 금액 {amount}원"
        # 부모 클래스인 Exception의 생성자를 호출하여 메시지를 전달합니다.
        super().__init__(self.message)

    def get_deficit(self):
        """부족한 금액 계산"""
        # 출금 요청 금액에서 현재 잔액을 뺀 '실제 부족한 액수'를 반환합니다.
        return self.amount - self.balance
```

```
    def withdraw(balance, amount):
        if amount > balance:
            # 커스텀 예외를 발생시키며 현재 상태 데이터를 넘겨줍니다.
            raise InsufficientFundsError(balance, amount)
        return balance - amount

    try:
        current_balance = 5000
        withdraw(current_balance, 8000)
    except InsufficientFundsError as e:
        print(e)  # "잔액 부족: 현재 잔액 5000원, 요청 금액 8000원" 출력
        print(f"부족한 금액: {e.get_deficit()}원")  # "부족한 금액: 3000원" 출력
```

- 예외 체이닝
  - raise: 에러를 발생 시키고 코드 흐름을 중단하는 역할

```
def fetch_data(url):
    try:
        # 데이터 가져오기 시도
        response = make_request(url)
        data = parse_response(response)
        return data
    except RequestError as e:
        # 원래 예외(RequestError)를 유지하면서 새 예외(DataFetchError) 발생
        raise DataFetchError(f"URL {url}에서 데이터를 가져오는 데 실패했습니다.") from e
    except ParseError as e:
        # 원래 예외(ParseError)를 유지하면서 새 예외(DataFetchError) 발생
        raise DataFetchError(f"응답 파싱에 실패했습니다.") from e
```

- assert
  - 내부 검증을 위한 개발자용 코드
  - 최적화 모드에서 제거됨
  - 내부 동작 (`assert 참이어야하는 조건, "에러 메세지"`)

    ```
    if __debug__:
        if not condition:
            raise AssertionError("메시지")
    ```

```
def divide_numbers(a, b):
    # b가 0이 아님을 확인 (나누는 수가 0이면 안 되는 수학적 조건)
    assert b != 0, "0으로 나눌 수 없습니다"

    result = a / b

    # 결과가 음수가 아님을 확인 (비즈니스 로직상 결과가 반드시 양수여야 할 때)
    assert result >= 0, f"결과가 음수입니다: {result}"

    return result

# 사용 예
try:
    print(divide_numbers(10, 2))   # 결과: 5

    # 다음 줄들은 AssertionError를 발생시킵니다.
    print(divide_numbers(10, 0))   # AssertionError: 0으로 나눌 수 없습니다
    print(divide_numbers(-10, 2))  # AssertionError: 결과가 음수입니다: -5.0
```

```
except AssertionError as e:
    # 단언문에서 설정한 에러 메시지를 출력합니다.
    print(f"단언문 오류: {e}")
```

▼ 2) 함수형 프로그래밍 기법

- 함수형 프로그래밍 특징
  - 불변성(Immutability): 데이터가 생성된 후 변경되지 않음

```
# 불변성을 지키는 방식
def add_item(items, item):
    # 원본 리스트를 변경하지 않고 새 리스트 반환
    return items + [item]

original = [1, 2, 3]
new_list = add_item(original, 4)

print(original)  # 출력: [1, 2, 3] - 원본 유지
print(new_list)  # 출력: [1, 2, 3, 4] - 새 객체
```

  - 순수 함수(Pure Functions): 동일 입력에 항상 동일 출력, 부작용 없음
  - 고차 함수(Higher-order Functions): 함수를 인자로 받거나 함수를 반환

```
# 함수를 인자로 받는 고차 함수
def apply_twice(func, value):
    # 전달받은 함수(func)를 두 번 연속으로 적용하여 결과를 반환합니다.
    return func(func(value))

def add_five(x):
    # 입력값에 5를 더하는 간단한 함수입니다.
    return x + 5

# apply_twice에 add_five 함수와 초기값 10을 인자로 전달합니다.
result = apply_twice(add_five, 10)
print(result)  # 출력: 20 (10+5+5)
```

  - 선언적 프로그래밍(Declarative): '어떻게'보다 '무엇을' 할지 표현
- 일급 함수 (First-Class Functions)
  - 함수를 변수에 할당하거나 다른 함수에 전달할 수 있음
  - 함수를 자료구조에 저장하거나 함수에서 반환 가능

```
# 함수를 변수에 할당
def greet(name):
    return f"Hello, {name}!"

# greet 함수 자체를 greeting 이라는 변수에 할당합니다.
greeting = greet
print(greeting("Alice"))  # 출력: Hello, Alice!

# 함수를 리스트에 저장
# 파이썬 내장 함수(len)와 메서드(str.upper, str.lower)를 리스트에 담습니다.
function_list = [len, str.upper, str.lower]
text = "Python"

# 리스트를 순회하며 각 함수에 text를 인자로 전달하여 실행합니다.
for func in function_list:
    print(func(text))  # 출력: 6, PYTHON, python
```

- 내장 함수형 프로그래밍 도구
  - lambda functions
  - map
  - filter
  - reduce
- 고급 함수형 프로그래밍 기법
  - Closure

```
def make_counter():
    # 외부 함수의 지역 변수
    count = 0

    def counter():
        # 외부 함수의 변수 count를 수정하기 위해 nonlocal 키워드 사용
        nonlocal count
        count += 1
        return count

    # 내부 함수 객체 자체를 반환 (일급 객체 특징)
    return counter

# 서로 독립적인 상태를 가진 카운터 객체 생성
counter1 = make_counter()
counter2 = make_counter()

print(counter1())  # 출력: 1
print(counter1())  # 출력: 2
print(counter2())  # 출력: 1 (독립적인 카운터)
```

  - Currying

```
# 3개 인자를 가진 커링 함수
def curry_mul(x):
    def mul_y(y):
        def mul_z(z):
            # 최종적으로 세 인자 x, y, z를 곱한 값을 반환합니다.
            return x * y * z
        return mul_z
    return mul_y

# 1단계: x=2, y=3이 고정된 부분 함수를 생성합니다.
mul_2_3 = curry_mul(2)(3)

# 2단계: 마지막 인자 z=4를 전달하여 최종 결과를 얻습니다.
print(mul_2_3(4)) # 출력: 24 (2*3*4)
```

  - 함수 합성 (Function Composition)

```
# 함수 합성 예시
def compose(f, g):
    # 두 함수 f와 g를 인자로 받아 f(g(x)) 형태의 새로운 익명 함수를 반환합니다.
    return lambda x: f(g(x))

def add_one(x):
    # 입력값에 1을 더합니다.
    return x + 1
```

```
def square(x):
    # 입력값을 제곱합니다.
    return x * x

# 먼저 제곱한 후 1을 더하는 합성 함수를 생성합니다.
# g(x) = square, f(x) = add_one → f(g(x))
square_then_add_one = compose(add_one, square)

# (5^2) + 1 = 26
print(square_then_add_one(5))  # 출력: 26
```

- 부분 적용 (Partial Application)

```
from functools import partial

# 인자가 3개인 기본 함수 정의
def power(base, exponent, multiplier):
    # 거듭제곱 후 multiplier를 곱하는 계산식입니다.
    return base ** exponent * multiplier

# 1. 첫 번째 인자(base)와 세 번째 인자(multiplier)를 고정한 예시
# base=2, multiplier=2로 고정된 새로운 함수를 생성합니다.
square_and_double = partial(power, 2, multiplier=2)
# 고정되지 않은 exponent인자 '3'만 전달하여 실행합니다.
print(square_and_double(3))  # 출력: 16 (2^3 * 2)

# 2. 특정 키워드 인자(exponent)만 고정한 예시
# exponent=3으로 고정된 '세제곱' 함수를 생성합니다.
cube = partial(power, exponent=3)
# base=2와 multiplier=1을 각각 전달하여 실행합니다.
print(cube(2, multiplier=1))  # 출력: 8 (2^3 * 1)
```

▼ 3) 이터레이터와 제너레이터

- 이터레이터
  - 반복 가능한 객체를 의미 (리스트, 튜플, 문자열, 세트, 딕셔너리)
  - `__iter__()` 및 `__next__()` 메서드를 구현한 객체
  - `next()` 함수로 다음 요소를 가져올 수 있음
  - 내부 동작을 직접 구현한 코드

```
# 1부터 n까지 카운트하는 이터레이터
class Counter:
    def __init__(self, low, high):
        # 시작 값과 끝 값을 초기화합니다.
        self.current = low    # 현재 값
        self.high = high      # 최대 값

    def __iter__(self):
        # 객체 자신을 이터레이터로 반환합니다.
        return self

    def __next__(self):
        # 현재 값이 최대 값보다 크면 반복을 중단합니다.
        if self.current > self.high:
            # 요소가 더 이상 없음을 알리는 표준 예외를 발생시킵니다.
            raise StopIteration
        else:
            # 다음 값으로 이동하고 현재 값을 반환합니다.
```

```
            self.current += 1
            return self.current - 1 # 현재 값 반환

# 사용 예시
counter = Counter(1, 5)
print(next(counter))  # 출력: 1
print(next(counter))  # 출력: 2
```

- 제너레이터
  - 이터레이터를 생성하는 간단한 방법
  - `yield` 키워드를 사용하는 특별한 함수
  - 호출 시 이터레이터를 반환
  - 상태를 유지하면서 값을 생성하고 일시 중지 가능
  - `next()` 호출 시 다음 `yield` 까지 실행 후 일시 중지

```
class Counter:
    def __init__(self, low, high):
        # 시작 값(low)과 최대 값(high)을 인스턴스 변수로 저장합니다.
        self.current = low    # 현재 값
        self.high = high      # 최대 값

    def __iter__(self):
        # iter() 함수 호출 시 객체 자신을 반환하도록 설정합니다.
        return self

    def __next__(self):
        # 현재 값이 최대 값을 초과하면 반복을 중단합니다.
        if self.current > self.high:
            # StopIteration 예외를 발생시켜 반복의 끝을 알립니다.
            raise StopIteration
        else:
            # 현재 값을 저장한 후 1을 증가시키고, 이전 현재 값을 반환합니다.
            self.current += 1
            return self.current - 1 # 현재 값 반환

# 사용 예시
counter = Counter(1, 5)
print(next(counter))  # 출력: 1
print(next(counter))  # 출력: 2
```

구분	리스트 컴프리헨션	제너레이터 표현식
기호[ ]	(대괄호)( )	(소괄호)
메모리 할당	즉시 전체 할당 (Eager)	필요 시 순차 생성 (Lazy)
메모리 효율	낮음 (데이터가 많을수록 불리)	매우 높음 (대용량 처리에 유리)
주요 용도	데이터를 여러 번 재사용해야 할 때	한 번 순회하며 처리하고 끝낼 때

```
# 리스트 컴프리헨션 vs 제너레이터 표현식
squares_list = [x**2 for x in range(10)]  # 리스트 컴프리헨션: [] 사용
squares_gen = (x**2 for x in range(10))   # 제너레이터 표현식: () 사용

print(squares_list)  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81] 출력
print(squares_gen)   # <generator object <genexpr> at 0x...> (객체 정보 출력)
```

```
# 제너레이터 표현식은 필요할 때만 값을 계산합니다.
for num in squares_gen:
    print(num)  # 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 순차적 출력
```

- yield 키워드 사용

```
# 제너레이터 함수의 상태 관리 예시
def stateful_generator():
    print("첫 번째 값 생성")
    yield 1

    print("두 번째 값 생성")
    yield 2

    print("세 번째 값 생성")
    yield 3

# 제너레이터 객체 생성
gen = stateful_generator()

# next() 호출 시마다 다음 yield 지점까지 코드가 실행됩니다.
print(next(gen))  # 출력: 첫 번째 값 생성, 1 반환
print("중간 작업 수행") # 제너레이터가 멈춘 사이 다른 로직 수행 가능
print(next(gen))  # 출력: 두 번째 값 생성, 2 반환
print(next(gen))  # 출력: 세 번째 값 생성, 3 반환
```

▼ 4) 로그 출력 실습

```
log_data = """2023-11-20 10:15:32 INFO: 사용자 로그인 성공
2023-11-20 10:16:45 ERROR: 데이터베이스 연결 실패
2023-11-20 10:17:23 WARNING: 비정상적인 접근 시도
2023-11-20 10:18:01 INFO: 세션 갱신 완료
2023-11-20 10:20:14 INFO: 사용자 로그아웃
2023-11-20 10:25:43 WARNING: 디스크 사용량 80% 초과
2023-11-20 10:30:12 INFO: 백업 작업 시작
2023-11-20 10:31:55 INFO: 백업 작업 완료
2023-11-20 10:45:07 ERROR: 캐시 서버 응답 없음
2023-11-20 10:50:33 INFO: 서비스 재시작 시도
2023-11-20 10:52:10 INFO: 서비스 재시작 완료
2023-11-20 11:05:11 ERROR: 메모리 부족 오류
2023-11-20 11:06:45 WARNING: 메모리 사용량 90% 초과
2023-11-20 11:08:22 INFO: 가비지 컬렉션 수행
2023-11-20 11:10:55 INFO: 시스템 상태 정상
2023-11-20 11:20:03 ERROR: 외부 API 호출 실패
2023-11-20 11:25:19 INFO: 외부 API 재시도 성공
2023-11-20 11:40:41 WARNING: 네트워크 지연 감지
2023-11-20 11:45:30 INFO: 네트워크 상태 복구
2023-11-20 12:00:00 INFO: 정기 점검 시작
2023-11-20 12:30:00 INFO: 정기 점검 종료"""

def log_reader(logs):
  for line in logs.strip().split('\n'):
    yield line

def error_filter(log_stream, state):
  for line in log_stream:
    if state in line:
      yield line
```

```
print("\n======= 전체 로그 출력 =======")
raw_logs = log_reader(log_data)
for error in raw_logs:
  print(error)

states = ["ERROR", "WARNING", "INFO"]
for state in states:
  raw_logs = log_reader(log_data)
  errors_only = error_filter(raw_logs, state)
  print(f"\n======= {state} 필터링된 로그 출력 =======")
  for error in errors_only:
    print(error)
```

▼ 5) 동시성, 병렬 처리

- 동시성: 여러 작업을 동일한 시간 범위 안에서 실행 → 논리적, 물리적
- 병렬성: 여러 작업을 실제로 동시에 실행 (여러 코어가 작업) → 물리적
- 프로그램 실행 → 프로세스 생성 → 스레드 스케쥴링 → 코어가 실행

> 💡 물리적 병렬: 하드웨어 시점 (cpu 여러개가 동시에 작업)
> 논리적 병렬: 소프트웨어적 시점 (하나의 cpu 작업에서 대기 중인 스레드 없이 동작-스케쥴링)
>
> - 코어 수 > 스레드 수: 물리적 병렬
> - 코어 수 < 스레드 수: 논리적 병렬
>
> 보통의 경우 물리적 병렬이 논리적 병렬보다 빠르다. 항상은 아님.

- 멀티스레딩: 하나의 프로세스 내에서 여러 스레드 실행, 프로세스는 하나의 메모리를 공유
- 멀티프로세싱: 서로 독립된 프로세스를 여러 코어에 분산 실행하는 방식, 각 프로세스는 독립된 메모리를 가짐.
- 비동기 프로그래밍: 이벤트 루프 기반의 비동기 처리, 하나의 스레드가 여러 작업을 관리
- 파이썬의 GIL(Global Interpreter Lock): 한 번에 하나의 스레드만 파이썬 인터프리터를 사용할 수 있게 제한하는 메커니즘