오류와 예외 처리

프로그램 오류의 종류

문법 오류(syntax error)

대부분 오타나 잘못된 문법을 사용해서 발생하는 오류다.

오류가 발생한 코드 줄 번호와 오류 종류를 즉시 출력하기 때문에 오류를 발견하기도 수정하기도 쉽다.

논리 오류(logical error)

프로그램은 아무 문제없이 잘 실행되지만 예상한 결과가 나오지 않는 등 프로그램 작성 당시 논리적으로 잘못 설계되어 나타나는 오류다.

- 예) **1 / 2 * 3**

논리 오류를 사전에 발견하는 것은 쉽지 않다.

논리 오류를 방지하기 위해서는 오류가 발생할 수 있는 다양한 경우의 수와 시나리오로 프로그램을 테스트해보는 습관을 갖는 것이 중요하다.

실행 오류(runtime error)

프로그램을 실행하는 동안 발생하는 오류다.

예를 들어,

사용자가 잘못된 값을 입력할 때,

존재하지 않는 파일을 불러올 때,

파일을 저장하면서 디스크 용량이 부족할 때 등 발생하는 오류다.

예외 처리(exception handling)만 잘하면 피할 수 있는 오류다.

문법 오류는 컴파일 단계에서,

논리 오류와 실행 오류는 실행 단계에서 발생한다.

실행 오류가 발생하는 예를 살펴보자.

따라해보기

In [ ]:

```
# 숫자를 0으로 나눠서 오류가 난다.
15 / 0
```

In [ ]:

```
# 폴더에 없는 파일을 열려고 해서 오류가 난다.
f = open('hw1.py')
```

앞의 예처럼 예외 처리를 하지 않거나 예외 처리에 실패하면 프로그램을 실행하는 중 갑작스럽게 종료하게 된다.

예외가 발생하면 오류 메시지를 통해 예외가 발생한 줄의 번호를 확인할 수 있다.

뿐만 아니라 발생한 예외의 종류와 예외가 발생한 이유에 대한 간략한 설명을 화면으로 출력하기 때문에 그 원인을 확인할 수 있다.

따라서, 프로그램 작성할 때 주의만 잘 기울이면 대부분의 예외 발생을 막을 수 있다.

예를 들어, 사용자가 입력한 값이 유효한 입력 값인지 확인하는 등의 예외 처리 코드만 미리 작성해 놓아도 많은 실행 오류를 방지할 수 있다.

몰라도 우선 따라해보기

오류가 발생하면 예외 객체(exception object)가 생성된다.

예외 객체는 주로 해당 오류와 관련된 기본 오류 메시지를 담고 있다.

다음 프로그램을 작성해서 실행해보자.

먼저 **3**을 입력해서 실행해보고 그 다음에는 **0**을 입력해서 실행해보자.

In [ ]:

```
i = int(input('정수를 입력하세요: '))
print('15 / {} = {}'.format(i, 15 / i))
```

0을 입력하면 **ZeroDivisionError** 예외 객체를 생성한 것을 알 수 있다.

사용자가 **0**을 입력하면 **ZeroDivisionError**로 예외 처리를 하는 프로그램을 작성한 후 실행해서 **0**을 입력해보자.

In [ ]:

```
i = int(input('정수를 입력하세요: '))

try:  # 오류가 발생할 가능성이 있는 코드를 try문 안에 넣는다.
    print('15 / {} = {}'.format(i, 15 / i))
except ZeroDivisionError:
    print('0으로 나눌 수 없습니다. 다른 숫자를 입력하세요.')
```

0을 입력했지만 오류가 나지 않는다.

0을 입력하면 **ZeroDivisionError**로 예외 처리를 했기 때문이다.

그런데 다시 입력하려면 프로그램이 종료되었기 때문에 프로그램을 다시 실행해야 한다.

프로그램을 중단하지 않고 예외 처리를 할 수는 없을까?

무한 루프를 사용하면 된다.

이번에는 무한 루프를 사용해 사용자가 **0**을 입력하면 예외 처리를 하고, **0**이 아닌 숫자를 입력할 때까지 다시 입력을 요구하는 프로그램을 작성해보자.

In [ ]:

```
while True:
    try:  # 오류가 발생할 가능성이 있는 코드를 try문 안에 넣는다.
        i = int(input('정수를 입력하세요: '))
        print('15 / {} = {}'.format(i, 15 / i))
        break  # 유효한 값을 입력했으므로 처리한 후 무한 루프를 빠저나간다.
    except ZeroDivisionError:
        print('0으로 나눌 수 없습니다. 다른 숫자를 입력하세요.')
```

그런데 만약 숫자가 아닌 문자를 입력하면 어떻게 될까?

이번에는 앞서 작성한 프로그램을 실행해서 문자 '**a**'를 입력해보자.

In [ ]:

```
# --- 앞 코드와 같은 코드이다.
while True:
    try:  # 오류가 발생할 가능성이 있는 코드를 try문 안에 넣는다.
        i = int(input('정수를 입력하세요: '))
        print('15 / {} = {}'.format(i, 15 / i))
        break  # 유효한 값을 입력했으므로 처리한 후 무한 루프를 빠저나간다.
    except ZeroDivisionError:
        print('0으로 나눌 수 없습니다. 다른 숫자를 입력하세요.')
```

이번에는 숫자를 입력하지 않았기 때문에 **ValueError** 오류가 났다.

여러 종류의 예외를 한번에 처리할 수는 없을까?

여러 개의 예외를 한번에 처리할 수 있다.

이번에는 **ZeroDivisionError**와 **ValueError**를 한꺼번에 처리하는 프로그램을 작성해보자.

In [ ]:

```
# --- 복수의 except문
while True:
    try:  # 오류가 발생할 가능성이 있는 코드를 try문 안에 넣는다.
        i = int(input('정수를 입력하세요: '))
        print('15 / {} = {}'.format(i, 15 / i))
        break  # 유효한 값을 입력했으므로 처리한 후 무한 루프를 빠저나간다.
    except ZeroDivisionError:
        print('0으로 나눌 수 없습니다. 다른 숫자를 입력하세요.')
    except ValueError:
        print('숫자를 입력해야 합니다.')
```

여러 개의 예외를 하나의 그룹으로 묶어 처리할 수도 있다.

다음 프로그램을 작성해서 실행해보자.

In [ ]:

```
# --- 예외 그룹
while True:
    try:  # 오류가 발생할 가능성이 있는 코드를 try문 안에 넣는다.
        i = int(input('정수를 입력하세요: '))
        print('15 / {} = {}'.format(i, 15 / i))
        break  # 유효한 값을 입력했으므로 처리한 후 무한 루프를 빠저나간다.
    except (ZeroDivisionError, ValueError):  # 반드시 괄호로 묶어야 한다.
        print('잘못된 값이 입력되었습니다. 다시 입력하세요.')
```

예외 객체는 해당 오류와 관련된 기본 오류 메시지를 담고 있다.

따라서 예외 객체를 ***변수***로 참조하면 해당 오류가 무엇인지 확인할 수 있다.

다음 프로그램은 예외 그룹을 변수를 지정해 어떤 오류가 나는지 확인한다.

In [ ]:

```
# --- as 변수
while True:
    try:  # 오류가 발생할 가능성이 있는 코드를 try문 안에 넣는다.
        i = int(input('정수를 입력하세요: '))
        print('15 / {} = {}'.format(i, 15 / i))
        break  # 유효한 값을 입력했으므로 처리한 후 무한 루프를 빠저나간다.
    except (ZeroDivisionError, ValueError) as err:  # 반드시 괄호로 묶어야 한다.
        print('다음과 같은 예외가 발생했습니다:', err)
        print('다시 입력하세요.')
```

try-except-else-finally문

try-except-else-finally문은 파이썬에서 제기하는 예외(exception)를 통해 흐름을 제어하는 일종의 조건문이다.

- **if**문과 달리 오직 '예외 발생'만을 조건으로 설정할 수 있다.

예외 처리를 하려면 **try**문 블록 안에 오류가 발생할 가능성이 있는 코드를 넣고 실행하면 된다.

예외를 처리하는 코드의 일반적인 형식은 다음과 같다.

```
try:
    try-명령문
except 예외그룹-1 [as 변수-1]:
    예외처리-명령문-1
except 예외그룹-2 [as 변수-2]:
    예외처리-명령문-2
...
except 예외그룹-N [as 변수-N]:
    예외처리-명령문-N
else:
    else-명령문
finally:
    finally-명령문
```

특징은 다음과 같다.

try문이 오면 최소한 한 개 이상의 **except**문이 와야 한다.

- 단, **try ... finally**문에서는 **except**문을 사용하지 않는다.

except문은 **try**문 안의 코드에서 오류가 발생할 때 예외를 처리하는 코드를 포함한다.

as *변수-N*는 선택 사항으로 ***예외그룹-N***을 ***변수***로 참조할 수 있게 한다.

try문 안의 코드에서 오류가 발생하지 않고 정상적으로 종료했다면 **else**문은 반드시 실행된다.

하지만 **try**문 안의 코드에서 오류가 발생하면 **else** 문은 실행되지 않는다.

else문은 마지막 **except**문 다음에 와야 한다.

- **else**문은 선택 사항이라 **try-except**문 다음에 반드시 사용할 필요는 없다.

finally문은 예외 처리 블록의 맨 마지막에 와야 한다.

- **finally**문도 선택 사항이라 **try-except**문 다음에 반드시 사용할 필요는 없다.

finally문이 있다면 **try**문 안의 코드에서 오류가 발생했는지 여부와 상관없이 **항상 마지막에 실행**된다.

except문의 실행 순서는 다음과 같다.

(1)

try문에서 오류가 발생하면 각 **except**문은 **순서대로 실행**된다.

(2)

만약 발생한 오류가 ***예외그룹-N***에 지정한 **except**문에 도달하면

해당 **except**문의 ***예외처리-명령문-N***이 실행된다.

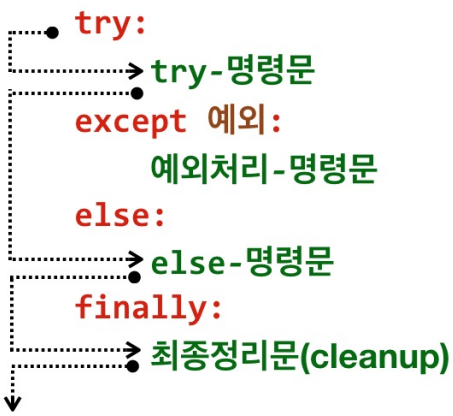
(3)

발생한 오류와 **except**문에 지정한 ***예외그룹-N***의 오류가 일치하는 경우는 두 가지가 있다.

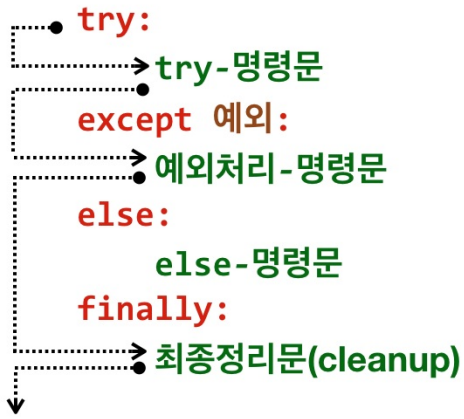
- 발생한 오류가 ***예외그룹-N***에 있는 오류와 **같은 예외 클래스**일 때
- 발생한 오류가 ***예외그룹-N***에 있는 오류의 **하위 예외 클래스**일 때

try-except-else-finally문 처리 순서

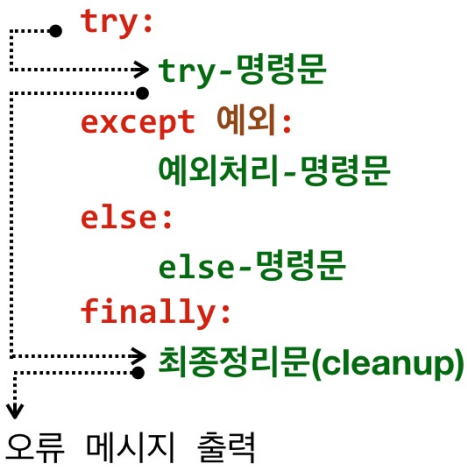
예외가 발생하지 않고 프로그램이 정상적으로 종료하는 경우



예외가 발생하고 해당 예외를 처리하는 경우



예외가 발생했지만 해당 예외를 지정하지 않아 예외 처리를 실패한 경우



예외처리

각 **except**문의 ***예외그룹***은 하나의 예외를 처리할 수도 있고, 튜플 형태의 예외 목록을 한번에 처리하는 것도 가능하다.

In [ ]:

```
# --- 한 개의 예외만 처리한다.
try:
    15 / 0
except ZeroDivisionError:
    print('0으로 나눌 수 없습니다.')
```

In [ ]:

```
# --- 복수의 예외를 처리한다.
try:
    15 / 'a'   # 0 외 다른 문자도 입력해본다.
except (ZeroDivisionError, ValueError, TypeError, RuntimeError):
    print('뭔가 잘못되었습니다.')
```

선택사항인 **as *변수***를 사용하면

발생한 예외 객체를 참조하기 때문에

해당 오류와 관련된 기본 오류 메시지를 확인할 수 있다.

In [ ]:

```
# --- 한 개의 예외만 변수로 참조해서 처리한다.
try:
    15 / 0
except ZeroDivisionError as err:
    print('다음과 같은 예외가 발생했습니다:', err)
```

as *변수*를 사용하면

튜플 형태의 ***예외그룹***도 하나의 **변수**로 참조할 수 있다.

In [ ]:

```
# --- 복수의 예외를 변수로 참조해서 처리한다.
try:
    15 / '0'  # 숫자 0도 입력해본다.
except (ZeroDivisionError, TypeError) as err:
    print('다음과 같은 예외가 발생했습니다:', err)
```

except:문

처리할 ***예외그룹***을 선언하지 않고 다음 예처럼 **except:**만 사용할 수 있다.

In [ ]:

```
# --- 모든 예외를 처리하기 때문에 권장하지 않는 방법이다.
try:
    15 / 0  # 문자도 입력해본다.
except:     # 모든 예외를 처리한다.
    print('뭔가 잘못된 것 같아요 ㅠㅠ')
```

In [ ]:

```
# --- 모든 예외를 처리하기 때문에 권장하지 않는 방법이다.
# Exception보다 ZeroDivisionError가 더 바람직하다.
try:
    15 / 0  # 문자도 입력해본다.
except Exception as err:
    print(err)
```

발생하는 모든 오류를 처리하지만 이것은 좋은 방법이 아니니 권장하지 않는다.

가급적이면 발생 가능한 모든 오류를 구체적으로 지정할 것을 권장한다.

발생 가능한 오류가 무엇인지 모를 때만 어쩔 수 없이 이 방법을 사용해야 한다.

except문 작성 순서

따라해보기

In [ ]:

```
try:
    15 / 0                # ZeroDivisionError가 발생한다.
except ArithmeticError as err:
    print('산술 예외가 발생했습니다:', err)
except ZeroDivisionError:  # 절대 실행되지 않는 except문이다.
    print('0으로 나눌 수 없습니다:', err)
```

0으로 나눴는데

왜 ZeroDivisionError가 아니라

ArithmeticError가 예외 처리를 할까?

except문을 작성할 때 주의할 점은

오류가 발생하면 각 **except**문이

순서대로 실행된다는 것이다.

뿐만 아니라 **except**문에 지정한

예외그룹에 속한 예외 뿐만 아니라

그 **하위 예외를 모두 처리**한다.

이처럼 예외는 **위계 관계**가 존재하기 때문에

범위가 더 넓고 포괄적인 상위 예외를 먼저 선언하면,

하위 예외는 절대로 실행되지 않는다.

따라서 프로그램을 작성할 때 다음과 같이

좀 더 세부적인 하위 예외를 먼저 작성해야 한다.

In [ ]:

```
try:
    15 / 0  # ZeroDivisionError가 발생한다.
except ZeroDivisionError as err:
    print('0으로 나눌 수 없습니다:', err)
except ArithmeticError as err:
    print('산술 예외가 발생했습니다:', err)
```

결론 :

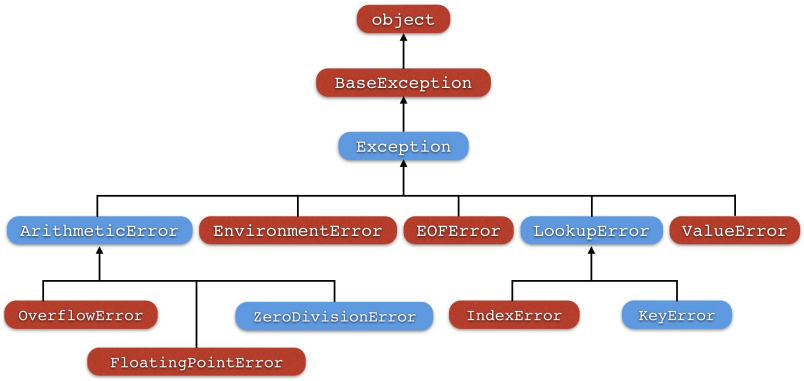
항상 하위 예외를 상위 예외보다 먼저 선언해야 한다!!!

이처럼 예외는

상위 예외와 하위 예외로 이루어진

계층 구조를 가진다.

다음 그림은 예외 계층 구조의 일부를 보여준다.



하위 예외로 갈수록 오류의 종류가 보다 구체적인 반면에

상위 예외는 자신의 아래에 있는 모든 하위 예외를 처리할 수 있다.

하지만 예외 처리를 할 때

해당 오류에 맞는 구체적인 예외 처리를 하는 것이 좋다.

따라서 가급적이면 해당 예외를 선언해서 처리하도록 하자.

파이썬이 다루는 모든 예외 종류와 계층 구조는 파이썬 문서 (https://docs.python.org/3/library/exceptions.html)를 확인하면 된다.

- https://docs.python.org/3/library/exceptions.html (https://docs.python.org/3/library/exceptions.html)

예외 회피

오류가 발생해도 처리하지 않고 그냥 통과시켜야 할 때도 종종 있다.

이런 상황에서는

except문에 예외 처리 코드를 넣지 않고

그냥 **pass**문을 추가하면 된다.

숫자를 **0**으로 나누었을 때 예외 처리를 하지 않고 그냥 통과시켜보자.

In [ ]:

```
try:
    15 / 0
except ZeroDivisionError:
    pass
```

아무 것도 일어나지 않는다.

try-finally문

오류가 발생해 작업을 처리하지 못하는 경우에도,

반드시 실행해야 하는 코드가 있을 때 사용한다.

컴퓨터나 프로그램이 충돌이 일어나 갑자기 멈추지 않는 한,

try문 안의 오류 발생과 상관없이

finally문은 마지막에 반드시 실행된다.

try-finally문을 작성하는 형식은 다음과 같다.

```
try:
    try-명령문
finally:
    finally-명령문
```

따라해보기

숫자를 **0**으로 나누었을 때 예외 처리를 하지 않아도 **finally**문이 실행되는지 확인해보자.

In [ ]:

```
try:
    15 / 0  # ZeroDivisionError가 발생한다.
finally:   # 반드시 처리해야 하는 코드는 finally문에 넣고 처리하면 된다.
    print('예외가 발생해도 finally문은 반드시 실행된다!!!')
```

try-finally문은 주로 **try**문을 종료한 후,

웹 사이트나 데이터베이스 연결을 끊거나

현재 처리 중인 파일을 닫는 등

최종 정리(clean-up)를 위한 명령문을 작성할 때 유용하다.