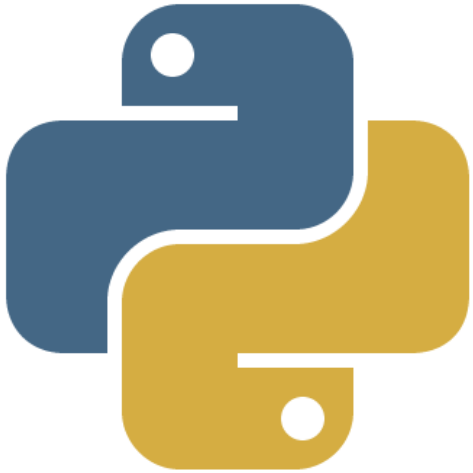
Programming in Python

14 – 에러와 예외 처리



python

2016년 8월, 국민대학교 컴퓨터공학부

예외 (Exceptions)

예외란? - 프로그램이 실행되면서 정상적으로 처리할 수 없거나 특별한 주의를 필요로 하는 사건이 발생한 상황

- 0 으로 나누는 나눗셈을 시도
  - 3 / 0 - ZeroDivisionError
- 리스트나 순서쌍의 인덱스 범위가 초과
  - L = [1, 2]
  - L[2] - IndexError
- 어떻게 처리할지가 정해져 있지 않은 연산을 시도
  - 1 + 'spam' - TypeError

- Python 에서 예외를 다루는 문장들
  - try / except
  - try / finally
  - raise
  - assert
  - with / as

예외의 역할

- 에러 처리
  - 프로그램 실행에서 발생할 수 있는 여러 가지 에러 상황에 대처하는 코드를 마련
- 이벤트의 알림
  - 특정한 사건이 발생하였음을 알릴 때 리턴 값을 이용하는 것보다 효과적인 경우가 있음
- 특수 경우를 처리
  - 발생할 가능성은 있으나 드물게 일어날 조건에 대한 코드를 예외 처리로 구현
- 종료 사건을 처리
  - 에러나 예외 상황의 발생에 불구하고 무조건 실행되어야 하는 코드를 정의
- 예외적인 프로그램 흐름
  - (드물지만) goto 대신 이용하여 특수한 프로그램 흐름을 제어하는 데 이용

예외 처리의 기본

```
>>> def fetcher(obj, index):
...     return obj[index]
...

>>> x = 'spam'
>>> fetcher(x, 3)
'm'

>>> fetcher(x, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
```

프로그래머에 의하여 정의되지 않은
(시스템에 의하여 이미 정의되어 있는)
디폴트 예외 핸들러에 의한 메시지

```
>>> try:
...     fetcher(x, 4)
... except IndexError:
...     print('got exception')
...
got exception
```

프로그래머에 의하여 정의된
예외 핸들러에 의한 메시지

try … except … else

```
try:
    <statements>                # 우선 이 문장들을 실행
except <name1>:
    <statements>                # try 블록의 실행 중 name1 의 예외가 발생한 경우 실행
except (name2, name3):
    <statements>                # name2 또는 name3 의 예외가 발생한 경우 실행
except <name4> as <data>:
    <statements>                # name4 의 예외가 발생한 경우, 예외 객체를 data 로 받아와서 실행
except:
    <statements>                # 그 외의 모든 예외의 경우 실행
else:
    <statements>                # 아무런 예외가 발생하지 않은 경우 실행
```

예외 발생시키기

```
>>> try:
...     raise IndexError
... except IndexError:
...     print('got exception')
...
got exception

>>> raise IndexError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError

>>> assert False, 'Nobody expects this!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Nobody expects this!
```

- raise 문장은 무조건 정해진 예외를 발생시킴
- assert 문장은 조건에 따라 AssertionError 예외를 발생시킴

사용자 정의 예외

Exception 클래스를 상속하여
사용자 정의 예외 클래스를 만들 수 있다.

```
>>> class Bad(Exception):
...     pass
...
>>>
>>> def doomed():
...     raise Bad()                    # raise 에서는 클래스 또는 인스턴스를
...

>>> try:
...     doomed()
... except Bad:                        # except 에는 클래스 이름을
...     print('god Bad')
...
god Bad
```

except … as … 에 의한 예외 인스턴스 취득

내장 예외 타입의 경우 아래 두 결과는 동일하다.

```
raise IndexError        # 클래스 (예외 인스턴스가 자동으로 생성됨)
raise IndexError()      # 인스턴스 (명시적으로 생성하였음)
```

사용자 정의 예외 타입의 경우

```
>>> class MyExc(Exception):
...     pass
...
>>> raise MyExc('spam')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyExc: spam
```

```
>>> try:
...     raise MyExc
... except MyExc as X:
...     print(X.args)
...
()

>>> try:
...     raise MyExc('msg')
... except MyExc as X:
...     print(X.args)
...
('msg',)
```

종료 사건의 처리 (Termination Action)

```
>>> def after(y):
...     try:
...         fetcher(x, y)
...     finally:
...         print('after fetch')
...     print('after try?')
...
>>> after(4)
after fetch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in after
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
>>>
>>> after(3)
after fetch
after try?
```

finally 절을 이용하여
예외가 발생하더라도
정해진 코드를 실행할 수 있다.

```
try:
    <statements>      # 우선 이 문장들을 실행
finally:
    <statements>      # 이 문장들은 예외 발생해도 실행
```

with … as 를 이용한 파일 처리

with *expression* [as *variable*]:
    with-block

context manager 라고 부름

- expression 은 context manager 프로토콜을 따르는 객체를 리턴할 것으로 기대됨
- 기대에 맞게 객체가 리턴되면, 이 객체에 variable 이라는 이름을 부여하고 with-block 을 실행
- 어떤 Python 내장형 객체들은 context manager 프로토콜을 따르도록 되어 있음
    - 예: 파일 객체는 with-block 의 실행이 끝난 후 close 됨을 보장

```
>>> with open('lumberjack.txt', 'r') as file:
...         for line in file:
                print(line)
                ...
```

〈파일이 닫힘을 보장할 수 있음〉

Context Management 프로토콜

with *expression* [as *variable*]:
with-block

1. with 다음의 표현식 (expression) 이 계산되고, 이것은 context manager 객체를 리턴
   - 이 객체는 __enter__ 와 __exit__ 메서드를 갖추고 있어야 함
2. Context manager 의 __enter__ 메서드가 호출되고, 이로부터 리턴되는 객체가 as 다음의 변수 (variable) 에 대입됨
3. with-block 에 포함된 문장들이 실행됨
4. 만약 with-block 의 실행에서 예외가 발생하면 __exit__(type, value, traceback) 메서드가 호출됨
   - 이 인자들은 발생한 예외에 대한 정보를 담고 있음
5. 만약 with-block 의 실행에서 아무런 예외가 발생하지 않으면 __exit__ 메서드는 모든 인자가 None 으로 대입되어 호출됨

Context Manager 예제

context.py

```
class TraceBlock:
    def message(self, arg):
        print('running', arg)
    def __enter__(self):
        print('starting with block')
        return self
    def __exit__(self, exc_type, exc_value, exc_tb):
        if exc_type is None:
            print('exited normally')
        else:
            print('raise an exception!', exc_type)
            return False
```

```
>>> from context import TraceBlock
>>> with TraceBlock() as action:
...     action.message('test 1')
...     print('reached')
...
starting with block
running test 1
reached
exited normally

>>> with TraceBlock() as action:
...     action.message('test 2')
...     raise TypeError
...     print('not reached')
...
starting with block
running test 2
raise an exception! <type 'exceptions.TypeError'>
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
TypeError
```

예외 클래스 (Exception Classes)

classexc.py

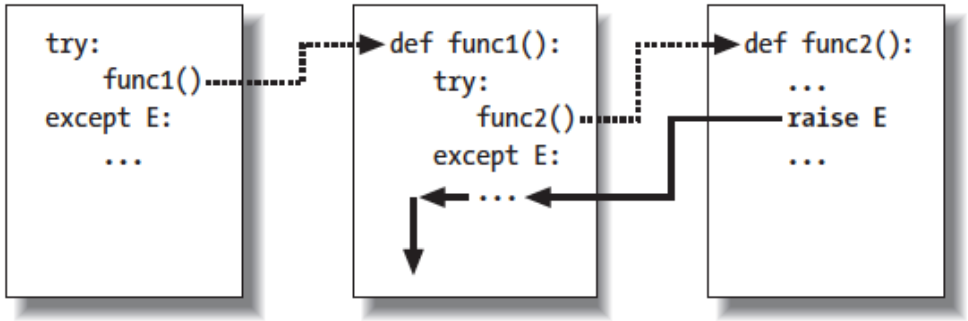
```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass

def raiser0():
    X = General(); raise X

def raiser1():
    X = Specific1(); raise X

def raiser2():
    X = Specific2(); raise X

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General:
        import sys
        print('caught: %s' % sys.exc_info()[0])
```

```
% python classexc.py
caught: <class '__main__.General'>
caught: <class '__main__.Specific1'>
caught: <class '__main__.Specific2'>
```

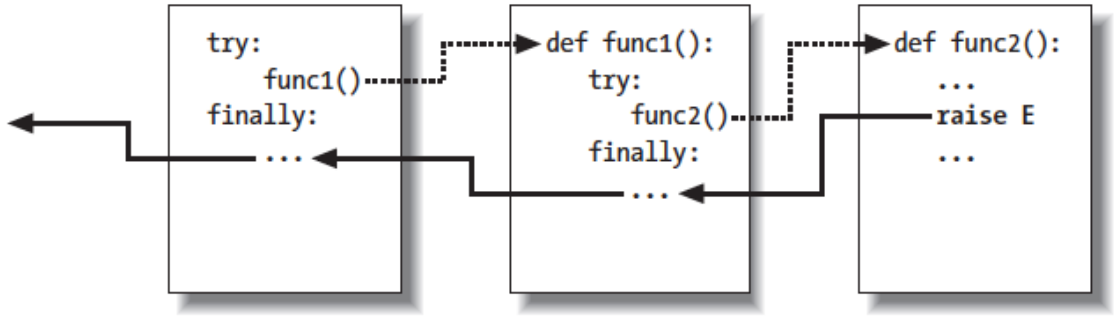
```
except General as X:
    print('caught: %s' % X.__class__)
```

로 바꾸어 써도 마찬가지

중첩된 예외 핸들러 (Nested Exception Handlers)



최근에 진입한 try 문의 except 절에서
예외 처리가 일어남
그 후로는 예외에서 복구하고 정상 실행을 재개



최근에 진입한 try 문의 finally 절로 진입하지만
예외는 그 이전의 try 문으로 전파됨

Quiz

- 예외란 무엇인가? 예외 처리는 어떤 쓸모가 있는가?
- 예외를 처리하는 방법에는 어떠한 것들이 있는가?
  - 처리되지 않은 예외는 어떤 일을 일으키는가?
- 프로그램 코드를 이용하여 예외를 발생시키는 두 가지 방법은 무엇인가?
- Context manager 란 무엇이며, 대표적인 용례는 어디에서 찾을 수 있는가?

Q & A