17. 예외 처리

17.1 예외처리란

우리는 지금까지 정상적인 프로그램의 흐름만을 가정했다. 프로그램을 수행하다 보면 문법은 맞으나 실행중 더 이상 진행할수 없는 상황이 발생한다. 이것을 예외라고 한다. 예외상황에는 여러 가지 경우가 있을수 있다. 예를 들어, 0으로 숫자 나누기, 문자열과 숫자 더하기, 참조범위를 넘어서 인덱스 참조하기 등록되어 있지 않은 키로 사전검색하기 등이다.
다음 나누기 연산은 ZeroDivisionError 에러를 발생시킨다.

```
>>> a, b = 5, 0
>>> c = a / b
Traceback (most recent call last)
<ipython-input-130-249b22ba83f7> in <module>
----> 1 c = a / b

ZeroDivisionError: division by zero
```

또한 정의되지 않은 변수를 사용하면 NameError 에러가 발생한다.

```
>>> 4 + spam * 3
  Traceback (most recent call last)
<ipython-input-131-c8ef1c0a2ff8> in <module>
----> 1 4 + spam * 3

NameError: name 'spam' is not defined
```

연산중에 인수의 타입이 맞지 않으면 TypeError 에러가 발생한다.

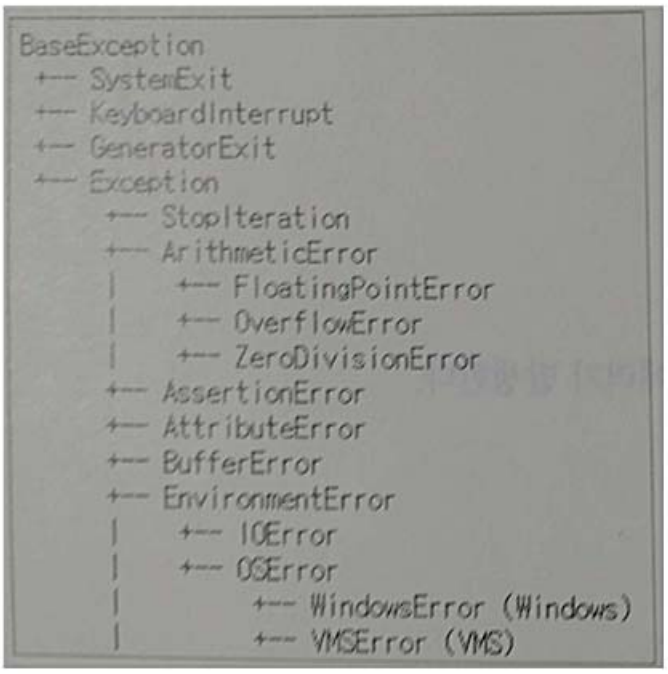
```
>>>'2' + 2
Traceback (most recent call last)
<ipython-input-132-7e1e50100c0b> in <module>
----> 1 '2'+2

TypeError: can only concatenate str (not "int") to str
```

이 외에는 사전에 없는 키를 참조하면 KeyError 에러가, 없는 파일을 읽으려고 하면 IOError 에러가, 리스트 참조에서 범위를 넘은 인덱스를 부여하면 IndexError 에러가 발생한다. 예외가 발생하면 에러 메시지가 나온다. 스택 추적 형태로 상황을 알려주며, 마지막 부분에 최종적으로 에러가 발생한 정보를 표시해 준다.

모든 예외는 클래스로 표현된다. 최상위에 있는 클래스는 BaseException 클래스이다. 예외 클래스는 종류에 따라서 여러 층으로 분류되어 있다. 예를 들어, ZeroDivisionError 클래스는 BaseException / Exception / ArithmeticError / ZeroDivisionError로 계층화되어 있다. 이에대한 그림과 설명은 라이브러리 레퍼런스의

Built-in Exceptions 장을 참고하기 바란다.

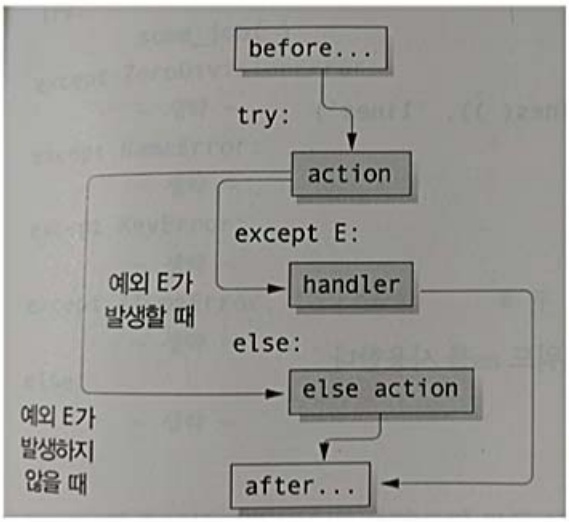


17.2 try~~except~else 문을 사용하기

예외가 발생하면 프로그램에서 try~ except 문을 사용하여 예외를 잡아낼수 있다. 사용하는 구문의 형식은 다음과 같다.

```
try :
    <문들1>
except <예외 종류1>:
    <문들2>     ------ <문들1>을 수행하는 중 <예외 종류1>이 발생하면 <문들2>가 수행된다.
else:           ------ else 이하는 생략할 수 있다.
    <문들3>     ------ 예외가 발생하지 않으면 <문들3>이 수행된다.
```

처리순서는, 우선 try 절(try와 except 사이)이 실행된다. 예외가 발생하지 않으면 else절을 수행한다. 예외가 발생하면 except 절을 수행하고 try ~ except 문을 마친다. else 절은 생략할수 있다. 이것을 그림으로 나타내면 그림과 같다.



(그림) try ~ except ~ else 문

before는 try 절 이전의 문들이고, after는 try ~ except ~ else 문 이후의 문들이 실제로 사용하는 예를 들면 다음과 같다.

```
>>> x = 0
>>> try:
...     print(1.0 / x)
... except ZeroDivisionError:
...     print('has no inverse')
...
has no inverse
```

다른 예로, 파일을 열 때 파일이 없으면 IOError 에러가 발생한다. 이것을 다음과 같이 처리할 수 있다.

```
>>> name = 'notexistingfile'
>>> try:
```

```
...     f = open(name, 'r')
... except IOError:
...     print('cannot open', name)
... else:
...     print(name, 'has', len(f.readlines()), 'lines')
...     f.close()
...
cannot open notexistingfile
```

예외를 발생시킨 객체를 변수로 받을수 있다. 키워드 as를 사용한다.

```
>>> try:
...     spam()
... except NameError as x:              # NameError 객체를 x로 받는다.
...     print(x)
...
name 'spam' is not defined
```

try:는 try 절에서 발생하는 예외뿐 아니라, 간접적으로 호출한 함수의 내부 예외도 처리한다.

```
>>> def this_fails():
...     x = 1 / 0

>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Runtime error:', err)
...
Runtime error: division by zero
```

여러개의 예외가 발생할수 있고, 각 경우마다 별도의 처리를 원한다면 다음과 같이 except 절을 여러번 사용할수 있다. except 괄호안에 예외이름을 사용하면, 해당 예외들이 발생했을 때 같은 루틴을 공유한다.

```
try:
        some_job()
except ZeroDivisionError:
        ~ 생략 ~
except NameError:
        ~ 생략 ~
except keyError:
        ~ 생략 ~
except (TypeError, IOError):    # 두가지 예외를 다 잡아낸다.
```

```
        ~ 생략 ~
else:
        ~ 생략 ~
```

except 절에 아무것도 기술하지 않으면 모든 예외를 다 잡아낸다.

```
try:
        some_job()
except:
        ~ 생략 ~
```

예외 클래스는 같은 종류의 예외 간에 계층적인 관계를 가지고 있으므로 이를 이용하면 여러 가지의 예외를 한꺼번에 검출해 낼수 있다. except 절에 기술된 예외 클래스는 하위의 파생 클래스의 예외까지 함께 잡아낸다. 예를들어, ArithmeticError 예외는 하위 클래스인 OverflowError와 ZeroDivisionError, FloatingPointError 예외를 모두 잡아낸다.

```
>>> def dosomething():
...     a = 1 / 0               # ZeroDivisionError
>>> try:
...     dosomething()
... except ArithmeticError:     print('Exception occurred')
```

만일 좀더 구체적으로 어떤 예외가 발생하는지 알고 싶다면 sys 모듈의 exc_info() 함수를 사용할수 있다. exc_info() 함수는(예외 클래스, 예외 인스턴스 객체, trackback 객체)를 반환한다. 예외 정보를 출력하려면 traceback 모듈의 pring_exception()나 print_exc() 함수를 사용할수 있다.

```
>>> import sys
>>> import traceback
>>> x = 0
>>> try:
...     1 / x
... except:
...     etype, evalue, tb = sys.exc_info()
...     print('Exception class = ', etype)
...     print('value=', evalue)
...     print('trackback object=',tb)
...     traceback.print_exception(etype, evalue, tb)    # 예외정보를 출력한다.
...     traceback.print_exc()                           # 위의 문과 동일하다.
...
Exception class=<class  'ZeroDivisionError' >
value = division by zero
trackback object= <traceback object at 0x02C7FD00>
```

```
Traceback (most recent call last):
  File "<pyshell#34>", line 2, in <module>
ZeroDivisionError: division by zero
Traceback (most recent call last):
  File "<pyshell#34>", line 2, in <module>
ZeroDivisionError: division by zero
```

17.3 try 문에서 finally 절을 사용하기

try 문에서 finally 절을 사용할수 있는데, finally 절에 포함된 문들은 예외 발생 여부에 관계없이 모두 수행된다.

```
f = open(filename, "w")
try:
        do_something_with(f)
finally:
        f.close()
```

do_something_with(f) 문에서 예외가 발생하지 않으녀 finally 절의 f.close()를 수행하고, 예외가 발생해도 f.close()를 수행한다. 이 상황은 어떤 경우에도 파일을 닫아야할 경우에 유용하게 사용된다.

다음은 try 문에서 except 와 finally 절을 사용한 예이다.

```
>>> x = 0
>>> try:
...     1 / x
... except:
...     print(sys.exc_info()[1])
... finally:
...     print('어떤 경우에도 호출이 된다')
...
division by zero
어떤 경우에도 호출이 된다.
```

17.4 raise 문으로 예외 발생시키기

- 내장 예외의 발생

이미 시스템에서 사용하고 있는 예외를 raise문을 이요하여 발생시킬수 있다. raise 문은 예외 클래스의 인스턴스 객체를 매개 변수로 받아들인다.

```
>>> raise IndexError("범위 오류")
Traceback (most recent call last):
  File "<pyshell#116>", line 1, in <module>
    raise IndexError("범위 오류")
IndexError: 범위 오류
```

만일 raise 문이 인수없이 사용된다면 가장 최근에 발생했던 예외가 다시 발생한다. 만일 최근에 발생한 에러가 없다면 RuntimeError 에러가 발생한다.

```
>>> try:
...     raise IndexError('a')
... except:
...     print('in except..')
...     raise   # 예외 다시 발생
...
in except..
Traceback (most recent call last):
  File "<pyshell#125>", line 2, in <module>
    raise IndexError( 'a' )
IndexError: a
```

이름 공간을 이용하면 예외 객체에 정보를넘기는 것도 가능하다.

```
>>> ie = IndexError('a')
>>> ie.value = 10
>>> ie.count = 3
>>> try:
...     raise ie
... except IndexError as x:     # ie를 x로 받는다.
...     print(x, x.value, x.count)
...
a 10 3
```

- 사용자 정의 예외

사용자 예외를 발생시키는 표준 방법은 BaseException 클래스의 하위 클래스를 이요하는 것이다. 상속받을 수 있는 클래스는 앞서 설명한 내장 예외의 어떤 것일수도 있다. 다음예는 Exception 클래스로 상속받은 Big 과

그 Big으로부터 상속받은 Small 클래스를 정의한다.

```
# user_defined_exception.py

import sys

class Big(Exception):
    pass

class Small(Big):
    pass

def dosomething1():
    raise Big('big excpetion')          # 예외가 발생한다.

def dosomething2():
    raise Small('small exception')

for f in (dosomething1, dosomething2):
    try:
        f()
    except Big:
        print(sys.exc_info())
```

코드를 실행한 결과는 다음과 같다.

```
(<class '__main__.Big'>, Big('big excpetion'), <traceback object at 0x0000019B45991D48>)
(<class '__main__.Small'>, Small('small exception'), <traceback object at 0x0000019B45991F08>)
```

앞의 예에서 except Big이 검출하는 예외는 Big과 Small이다. 왜냐하면 클래스 Small은 클래스 Big 으로부터 상속받은 클래스이기 때문이다.

sys.exc_info() 함수는 발생한 예외의 종류와 예외 객체, traceback 정보 세가지를 튜플로 반환한다. 따라서 발생한 예외에 대해서 정보를 얻고자 할 때 유용하게 사용된다.

예외를 발생시킬 때 예외 정보를 함께 전달하고 싶을때가 있다. 다음예는 인스턴스 객체에 값을 넣어서 예외를 발생시키는 예이다.

```
# user_defined_exception2.py

class MessageException(Exception):      # 사용자 예외 클래스를 정의한다.
    def __init__(self, message, dur):
```

```
        self.message = message
        self.duration = dur
    def __str__(self):
        return '{}: message={}, duration={}'.format(
            self.__class__.__name__,
            self.message, self.duration)

def f():
    raise MessageException('message', 10)       # 예외 객체를 전달한다.

try:
    f()
except MessageException as a:            # 예외 객체를 a란 이름으로..
    print(a)
    print(a.message, a.duration)
```

코드를 실행한 결과는 다음과 같다.

```
MessageException: message=message, duration=10
message 10
```

17.5 assert 문으로 예외발생시키기

예외를 발생시키는 특수한 경우로 assert문에 의한 AssertionError 에러가 있다. assert 문은 주로 디거깅할 때 많이 사용한다. 프로그램이 바르게 진행하는지를 시험할 때 사용한다. assert 문의 형식은 다음과 같다.

asset <시험 코드>, <데이터>

만일 <시험코드>가 거짓이면 'raise AssertionError, 데이터' 예외를 발생시킨다. <시험 코드>가 참이면 그냥 통과한다. 데이터는 사용하지 않아도 좋은 옵션이다. 사용하는 예를 보자.

```
# asserttest.py
a = 30
margin = 2 * 0.2
assert margin > 10, 'not enough margin'
```

코드를 실행한 결과는 다음과 같다.

```
C:₩Python32>python asserttest.py
Traceback (most recent call last):
  File "test.py" ,line 36, in <module>
    assert margin > 10, 'not enough margin'
AssertionError: not enough margin
```

파이썬을 -O 옵션으로 실행하면 assert 문은 컴파일된 바이트 코드로부터 자동으로 삭제된다. 또한, -O 옵션은 _debug_ 클래그가 0이 되어 디버깅 상태가 아님을 알린다.

```
C:₩Python32>python -O asserttest.py
0.4
```