

8. 예외 처리와 내장함수

contents

- ▶ 예회처리
- ▶ 내장함수



1. 예외 처리

▶ 오류 예외 처리 기법

▶ try-except 문

- ▶ 오류 처리를 위한 구문
- ▶ try 블록 수행 중 오류가 발생하면 except 블록 수행
try 블록에서 오류가 발생하지 않으면 except 블록 미수행

▶ except 구문

```
except [발생_오류 [as 오류_변수]]:
```

□ [] 기호

- 괄호 안의 내용을 생략할 수 있다는 관례 표기 기법

```
try:  
    ...  
except [발생_오류 [as 오류_변수]]:  
    ...
```



1. 예외 처리

▶ 오류 예외 처리 기법

▶ try-except 문

▶ try-except만 쓰는 방법

- 오류 종류에 상관없이
오류가 발생하면 except 블록 수행

```
try:  
    ...  
except:  
    ...
```

▶ 발생 오류만 포함한 except 문

- 오류가 발생했을 때 except 문에
미리 정해 놓은 오류와 동일할 때만
except 블록을 수행한다는 뜻

```
try:  
    ...  
except 발생_오류:  
    ...
```



1. 예외 처리

▶ 오류 예외 처리 기법

▶ try-except 문

- ▶ 발생 오류와 오류 변수까지 포함한 except 문
 - 오류가 발생했을 때 except 문에 미리 정해 놓은 오류와 동일할 때만 except 블록을 수행하고, 오류 메시지의 내용까지 알고 싶을 때 사용하는 방법

```
try:  
    4 / 0  
except ZeroDivisionError as e:  
    print(e)
```

```
try:  
    ...  
except 발생_오류 as 오류_변수:  
    ...
```

실행 결과

division by zero

▶ 오류 예외 처리 기법

▶ try-finally 문

- ▶ finally 절은 try 문 수행 도중 예외 발생 여부에 상관없이 항상 수행됨
- ▶ 보통 finally 절은 사용한 리소스를 close해야 할 때 많이 사용
 - 예) foo.txt 파일을 쓰기 모드로 열어 try 문을 수행한 후
예외 발생 여부와 상관없이 finally 절에서 f.close()로 열린 파일을 닫을 수 있음

```
try:
    f = open('foo.txt', 'w')
    # 무언가를 수행

    (...생략...)

finally:
    f.close()    # 중간에 오류가 발생하더라도 무조건 실행
```



1. 예외 처리

▶ 오류 예외 처리 기법

▶ 여러 개의 오류 처리하기

- ▶ try문 안에서
여러 개의 오류를 처리하기 위한 방법

```
try:
    ...
except 발생_오류1:
    ...
except 발생_오류2:
    ...
```

■ 0으로 나누는 오류와 인덱싱 오류 처리

```
try:
    a = [1, 2]
    print(a[3])
    4 / 0
except ZeroDivisionError:
    print("0으로 나눌 수 없습니다.")
except IndexError:
    print("인덱싱할 수 없습니다.")
```



1. 예외 처리

▶ 오류 예외 처리 기법

- ▶ 여러 개의 오류 처리하기
 - ▶ 오류 메시지 가져오기

```
try:
    a = [1, 2]
    print(a[3])
    4 / 0
except ZeroDivisionError as e:
    print(e)
except IndexError as e:
    print(e)
```

- 2개 이상의 오류를 동일하게 처리하기 위해 괄호를 사용하여 함께 묶어 처리

```
try:
    a = [1, 2]
    print(a[3])
    4 / 0
except (ZeroDivisionError, IndexError) as e:
    print(e)
```



▶ 오류 예외 처리 기법

- ▶ try-else 문
 - ▶ 오류 메시지 가져오기

```
try:
    ...
except [발생_오류 [as 오류_변수]]:
    ...
else:    ← 오류가 없을 경우에만 수행
    ...
```

- try 문에 else 절을 사용한 예제

```
try:
    age = int(input('나이를 입력하세요: '))
except:
    print('입력이 정확하지 않습니다.')
else:
    if age <= 18:
        print('미성년자는 출입금지입니다.')
    else:
        print('환영합니다.')
```



▶ 오류 회피하기

- ▶ 특정 오류가 발생할 경우 그냥 통과시키는 방법

```
try:  
    f = open("나없는파일", 'r')  
except FileNotFoundError:    # 파일이 없더라도 오류가 발생하지 않고 통과  
    pass
```

- ▶ try 문 안에서 FileNotFoundError가 발생할 경우, pass를 사용하여 오류를 그냥 회피하도록 함



1. 예외 처리

▶ 오류 일부러 발생시키기

- ▶ raise 명령어를 사용해 오류를 강제로 발생시킬 수 있음

- ▶ 예) Bird 클래스를 상속받는 자식 클래스가 반드시 fly라는 함수를 구현하도록 만들고 싶은 경우

```
class Bird:
    def fly(self):
        raise NotImplementedError
```

- 파이썬 내장 오류 NotImplementedError와 raise 문 활용
 - fly 함수를 구현하지 않은 상태로 fly 함수 호출 시 NotImplementedError 오류 발생

```
class Eagle(Bird):
    pass

eagle = Eagle()
eagle.fly()
```



실행 결과

```
Traceback (most recent call last):
  File "...", line 33, in <module>
    eagle.fly()
  File "...", line 26, in fly
    raise NotImplementedError
NotImplementedError
```

1. 예외 처리

▶ 오류 일부러 발생시키기

- ▶ raise 명령어를 사용해 오류를 강제로 발생시킬 수 있음
 - ▶ NotImplementedError가 발생하지 않게 하려면

```
class Eagle(Bird):  
    def fly(self):  
        print("very fast")
```

```
eagle = Eagle()  
eagle.fly()
```

실행 결과

very fast

- Eagle 클래스에 fly 함수 구현



▶ 예외 만들기

- ▶ 파이썬 내장 클래스인 Exception 클래스를 상속하여 생성 가능

```
class MyError(Exception):  
    pass
```

- ▶ 예) 별명을 출력해주는 함수에서 MyError 사용하기

```
def say_nick(nick):  
    if nick == '바보':  
        raise MyError()  
    print(nick)
```

```
say_nick("천사")  
say_nick("바보")
```



실행 결과

천사

```
Traceback (most recent call last):  
  File "...", line 11, in <module>  
    say_nick("바보")  
  File "...", line 7, in say_nick  
    raise MyError()  
__main__.MyError
```

1. 예외 처리

▶ 예외 만들기

- ▶ 파이썬 내장 클래스인 Exception 클래스를 상속하여 생성 가능

```
class MyError(Exception):  
    pass
```

- ▶ 예외처리 기법을 사용하여 MyError 발생 예외 처리

```
try:  
    say_nick("천사")  
    say_nick("바보")  
except MyError:  
    print("허용되지 않는 별명입니다.")
```

실행 결과

천사

허용되지 않는 별명입니다.



1. 예외 처리

▶ 예외 만들기

- ▶ 파이썬 내장 클래스인 Exception 클래스를 상속하여 생성 가능

```
class MyError(Exception):  
    pass
```

- ▶ MyError에서 __str__ 메서드 구현하여 오류 메시지 사용하기

```
try:  
    say_nick("천사")  
    say_nick("바보")  
except MyError as e:  
    print(e)
```

```
class MyError(Exception):  
    def __str__(self):  
        return "허용되지 않는 별명입니다."
```



2. 내장 함수

▶ 파이썬 내장(built-in) 함수

- ▶ 파이썬 모듈과 달리 import가 필요 없기 때문에 아무런 설정 없이 바로 사용 가능

Don't Reinvent
the Wheel!



이미 있는 것을 다시 만드느라 시간을 낭비하지 말라.



2. 내장 함수

▶ **abs(x)**

- ▶ 어떤 숫자를 입력받았을 때 그 숫자의 절댓값을 돌려주는 함수

```
>>> abs(3)
3
>>> abs(-3)
3
>>> abs(-1.2)
1.2
```

■ **all(x)**

- 반복 가능한 데이터 x를 입력값으로 받으며 x의 요소가 모두 참이면 True, 하나라도 거짓이면 False를 리턴

```
>>> all([1, 2, 3])
True
```

```
>>> all([1, 2, 3, 0])
False
```

```
>>> all([])
True
```



2. 내장 함수

▶ any(x)

- ▶ 반복 가능한 데이터 x를 입력으로 받아 x의 요소 중 하나라도 참이면 True, x가 모두 거짓일 때만 False를 리턴

```
>>> any([1, 2, 3, 0])  
True
```

```
>>> any([0, ""])  
False
```

```
>>> any([])  
False
```

▶ chr(i)

- ▶ 유니코드 숫자 값을 입력받아 그 코드에 해당하는 문자를 리턴

```
>>> chr(97)  
'a'  
>>> chr(44032)  
'가'
```



2. 내장 함수

▶ **dir(x)**

- ▶ 객체가 지닌 변수나 함수를 보여 주는 함수

```
>>> dir([1, 2, 3])  
['append', 'count', 'extend', 'index', 'insert', 'pop',...]  
>>> dir({'1':'a'})  
['clear', 'copy', 'get', 'has_key', 'items', 'keys',...]
```

- ▶ 리스트와 딕셔너리가 지닌 함수(메서드)를 보여 주는 예

▶ **divmod(a, b)**

- ▶ a를 b로 나눈 몫과 나머지를 튜플로 리턴

```
>>> divmod(7, 3)  
(2, 1)
```

- ▶ 몫을 구하는 연산자 //와 나머지를 구하는 연산자 %를 각각 사용한 결과와 비교

```
>>> 7 // 3  
2  
>>> 7 % 3  
1
```

2. 내장 함수

▶ enumerate(x)

- ▶ '열거하다'라는 뜻
- ▶ 순서가 있는 데이터(리스트, 튜플, 문자열)를 입력으로 받아 인덱스 값을 포함하는 enumerate 객체 리턴

```
>>> for i, name in enumerate(['body', 'foo', 'bar']):  
...     print(i, name)  
...  
0 body  
1 foo  
2 bar
```

▶ eval(expression)

- ▶ 문자열로 구성된 표현식(expression)을 입력으로 받아 해당 문자열을 실행한 결과값을 리턴

```
>>> eval('1 + 2')  
3  
>>> eval("'hi' + 'a'")  
'hia'  
>>> eval('divmod(4, 3)')  
(1, 1)
```



2. 내장 함수

▶ filter(f, iterable)

- ▶ ‘무엇인가를 걸러 낸다’는 뜻

filter(함수, 반복_가능한_데이터)

- ▶ 반복 가능한 데이터의 요소
순서대로 함수를 호출했을 때
리턴값이 참인 것만 묶어서
(걸러 내서) 리턴

```
def positive(l):  
    result = []           # 양수만 걸러 내서 저장할 변수  
    for i in l:  
        if i > 0:  
            result.append(i)  # 리스트에 i 추가  
    return result  
  
print(positive([1, -3, 2, 0, -5, 6]))
```

실행 결과

[1, 2, 6]

2. 내장 함수

▶ filter(f, iterable)

- ▶ filter 함수를 사용해 간단하게 작성

```
def positive(x):  
    return x > 0  
  
print(list(filter(positive, [1, -3, 2, 0, -5, 6])))
```

실행 결과

[1, 2, 6]

- ▶ lambda를 사용해 더욱 간단하게 작성

```
>>> list(filter(lambda x: x > 0, [1, -3, 2, 0, -5, 6]))  
[1, 2, 6]
```



2. 내장 함수

▶ hex(x)

- ▶ 정수 값을 입력받아 16진수(hexadecimal) 문자열로 리턴

```
>>> hex(234)
'0xea'
>>> hex(3)
'0x3'
```

▶ id(object)

- ▶ 객체(object)를 입력받아 고유 주소값(레퍼런스)을 리턴

```
>>> a = 3
>>> id(3)
135072304
>>> id(a)
135072304
>>> b = a
>>> id(b)
135072304
```

```
>>> id(4)
135072292
```



2. 내장 함수

▶ input([prompt])

- ▶ 사용자 입력을 받는 함수
- ▶ 입력 인수로 문자열을 전달하면 그 문자열은 프롬프트가 됨

```
>>> a = input()  ← 사용자가 입력한 정보를 변수 a에 저장
hi  ← hi 입력
>>> a
'hi'  ← 사용자 입력으로 받은 'hi' 출력
>>> b = input("Enter: ")  ← 입력 인수로 "Enter: " 문자열 전달
Enter: hi  ← Enter: 프롬프트를 띄우고 사용자 입력을 받음.
>>> b
'hi'  ← 사용자 입력으로 받은 'hi' 출력
```

▶ int(x)

- ▶ 문자열 형태의 숫자나 소수점이 있는 숫자를 정수로 리턴

```
>>> int('3')  ← 문자열 '3'
3
>>> int(3.4)  ← 소수점이 있는 숫자 3.4
3
```

- radix 진수로 표현된 문자열 x를 10진수로 변환하여 리턴

```
>>> int('11', 2)
3
```

```
>>> int('1A', 16)
26
```


2. 내장 함수

▶ `isinstance(object, class)`

- ▶ `isinstance(object, class)` 함수는 첫 번째 인수로 객체, 두 번째 인수로 클래스를 받음
- ▶ 입력으로 받은 객체가 그 클래스의 인스턴스인지를 판단하여 참이면 `True`, 거짓이면 `False`를 리턴

```
>>> class Person: pass  ← 아무런 기능이 없는 Person 클래스 생성
...
>>> a = Person()  ← Person 클래스의 인스턴스 a 생성
>>> isinstance(a, Person)  ← a가 Person 클래스의 인스턴스인지 확인
True
```

```
>>> b = 3
>>> isinstance(b, Person)  ← b가 Person 클래스의 인스턴스인지 확인
False
```



2. 내장 함수

▶ len(s)

- ▶ 입력값 s의 길이(요소의 전체 개수)를 리턴

```
>>> len("python")
6
>>> len([1, 2, 3])
3
>>> len((1, 'a'))
2
```

▶ list(iterable)

- ▶ 반복 가능한 데이터를 입력받아 리스트로 만들어 리턴

```
>>> list("python")
['p', 'y', 't', 'h', 'o', 'n']
>>> list((1, 2, 3))
[1, 2, 3]
```

```
>>> a = [1, 2, 3]
>>> b = list(a)
>>> b
[1, 2, 3]
```



2. 내장 함수

▶ `map(f, iterable)`

- ▶ 함수(f)와 반복 가능한 데이터를 입력으로 받음
- ▶ 입력받은 데이터의 각 요소에 함수 f를 적용한 결과를 리턴

```
>>> def two_times(x):  
...     return x * 2  
...  
>>> list(map(two_times, [1, 2, 3, 4]))  
[2, 4, 6, 8]
```

- ▶ lambda 활용 가능

```
>>> list(map(lambda a: a*2, [1, 2, 3, 4]))  
[2, 4, 6, 8]
```

▶ `max(iterable)`

- ▶ 반복 가능한 데이터를 입력받아 그 최댓값을 리턴

```
>>> max([1, 2, 3])  
3  
>>> max("python") ← 문자열의 경우, 유니코드 값이 가장 큰 문자를 리턴  
'y'
```

▶ `min(iterable)`

- ▶ 반복 가능한 데이터를 입력받아 그 최솟값을 리턴

```
>>> min([1, 2, 3])  
1  
>>> min("python")  
'h'
```

2. 내장 함수

▶ oct(x)

- ▶ 정수를 8진수 문자열로 바꾸어 리턴

```
>>> oct(34)
'0o42'
>>> oct(12345)
'0o30071'
```

▶ open(filename, [mode])

- ▶ '파일 이름'과 '읽기 방법'을 입력받아 파일 객체를 리턴
- ▶ mode를 생략하면 기본값인 읽기 모드(r)로 파일 객체를 리턴
- ▶ b는 w,r,a와 함께 사용

mode	설명
w	쓰기 모드로 파일 열기
r	읽기 모드로 파일 열기
a	추가 모드로 파일 열기
b	바이너리 모드로 파일 열기

```
>>> f = open("binary_file", "rb")
```

2. 내장 함수

▶ ord(c)

- ▶ 문자의 유니코드 숫자 값을 리턴

```
>>> ord('a')  
97  
>>> ord('가')  
44032
```

ord 함수는 앞에서
배운 chr 함수와 반대로
동작하는구나!



▶ pow(x, y)

- ▶ x를 y제곱한 결과값을 리턴

```
>>> pow(2, 4)  
16  
>>> pow(3, 3)  
27
```

2. 내장 함수

▶ `range([start,] stop [,step])`

▶ 입력받은 숫자에 해당하는 범위 값을 반복 가능한 객체로 만들어 리턴

1) 인수가 하나일 경우

▶ 시작 숫자를 지정해 주지 않으면 `range` 함수는 0부터 시작

```
>>> list(range(5))  
[0, 1, 2, 3, 4]
```

2) 인수가 2개일 경우

- 시작 숫자와 끝 숫자
- 끝 숫자는 해당 범위에 포함되지 않음

```
>>> list(range(5, 10))  
[5, 6, 7, 8, 9] ← 끝 숫자 10은 포함되지 않음.
```

3) 인수가 3개일 경우

- 세 번째 인수는 숫자 사이의 거리

```
>>> list(range(1, 10, 2))  
[1, 3, 5, 7, 9] ← 1부터 9까지, 숫자 사이의 거리는 2  
>>> list(range(0, -10, -1))  
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9] ← 0부터 -9까지, 숫자 사이의 거리는 -1
```

2. 내장 함수

▶ `round(number[, ndigits])`

- ▶ 숫자를 입력받아 반올림해 리턴하는 함수

```
>>> round(4.6)
5
>>> round(4.2)
4
```

- ▶ ,ndigits는 반올림하여 표시하고 싶은 소수점의 자릿수를 의미

▶ `sorted(iterable)`

- ▶ 입력 데이터를 정렬한 후 그 결과를 리스트로 리턴하는 함수

```
>>> sorted([3, 1, 2])
[1, 2, 3]
>>> sorted(['a', 'c', 'b'])
['a', 'b', 'c']
>>> sorted("zero")
['e', 'o', 'r', 'z']
>>> sorted((3, 2, 1))
[1, 2, 3]
```



2. 내장 함수

▶ **str(object)**

- ▶ 문자열 형태로 객체를 변환하여 리턴하는 함수

```
>>> str(3)
'3'
>>> str('hi')
'hi'
```

▶ **sum(iterable)**

- ▶ 입력 데이터의 합을 리턴하는 함수

```
>>> sum([1, 2, 3])
6
>>> sum((4, 5, 6))
15
```



2. 내장 함수

▶ tuple(iterable)

- ▶ 반복 가능한 데이터를 튜플로 바꾸어 리턴하는 함수
- ▶ 입력이 튜플인 경우 그대로 리턴

```
>>> tuple("abc")
('a', 'b', 'c')
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> tuple((1, 2, 3))
(1, 2, 3)
```

▶ type(object)

- ▶ 입력값의 자료형이 무엇인지 알려 주는 함수

```
>>> type("abc")
<class 'str'> ← "abc"는 문자열 자료형
>>> type([])
<class 'list'> ← []는 리스트 자료형
>>> type(open("test", 'w'))
<class '_io.TextIOWrapper'> ← 파일 자료형
```



2. 내장 함수

▶ **zip(*iterable)**

- ▶ 동일한 개수로 이루어진 데이터들을 묶어서 리턴하는 함수

```
>>> list(zip([1, 2, 3], [4, 5, 6]))  
[(1, 4), (2, 5), (3, 6)]  
>>> list(zip([1, 2, 3], [4, 5, 6], [7, 8, 9]))  
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]  
>>> list(zip("abc", "def"))  
[('a', 'd'), ('b', 'e'), ('c', 'f')]
```

