BIND SOFT

구문 오류와 예외

BIND SOFT ㈜바인드소프트

BIND SOFT

목차

시작하기 전에

[핵심 키워드] : 구문 오류, 예외 (런타임 에러), 기본 예외 처리, by except 구문

[핵심 포인트]

프로그램 활용 시 쉽게 예상치 못하는 상황들에 대해 알아보고 세분화해본다.

시작하기 전에

- 오류(error)와 예외

오류

```
Traceback (most recent call last):
  File "test.py", line 16, in <module>
    print(fibonacci(10))
  File "test.py", line 6, in fibonacci
    counter += 1
UnboundLocalError: local variable 'counter' referenced before assignment
```

오류의 종류

- 오류 (error)
  - 구문 오류 (syntax error)
    - 프로그램 실행 전에 발생하는 오류
  - 런타임 오류 (runtime error) / 예외 (exception)
    - 프로그램 실행 중에 발생하는 오류

- 구문 오류

구문 오류가 발생하는 코드

```
# 프로그램 시작
print("# 프로그램이 시작되었습니다!")

# 구문 오류 발생 코드
print("# 예외를 강제로 발생시켜 볼게요!)
```

닫는 따옴표로 문자열을 닫지 않았습니다.

오류의 종류

오류

```
SyntaxError: EOL while scanning string literal
```

- SyntaxError
  - 구문에 문제가 있어 프로그램 실행부터 불가능한 경우

구문 오류 해결

```
# 프로그램 시작
print("# 프로그램이 시작되었습니다!")

# 구문 오류 발생 코드
print("# 예외를 강제로 발생시켜 볼게요!")
```

닫는 따옴표로 문자열을 닫아서 해결합니다.

오류의 종류

- 예외 / 런타임 오류
  - 실행 중에 발생하는 오류

예외가 발생하는 코드

```
# 프로그램 시작
print("# 프로그램이 시작되었습니다!")

# 예외 발생 코드
list_a[1]
```

```
# 프로그램이 시작되었습니다!  ---> 여기까지는 프로그램이 정상으로 실행되었다는 것을 확인할 수 있습니다.
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    list_a[1]
NameError: name 'list_a' is not defined
```

오류의 종류

예외 해결

```
# 프로그램 시작
print("# 프로그램이 시작되었습니다!")


# 예외 발생 코드 해결
list_a = [1, 2, 3, 4, 5]  ⟶ 에러 메시지에서 정의하지 않았다고 하니 정의해 줍니다.
list_a[1]
```

기본 예외 처리

- 예외 처리 (exception handling)
  - 조건문을 사용하는 방법
    - 기본 예외 처리
  - try 구문을 사용하는 방법

- 예외 상황 확인하기

예외가 발생할 수 있는 코드

```
# 숫자를 입력받습니다.
number_input_a = int(input("정수 입력> "))

# 출력합니다.
print("원의 반지름:", number_input_a)
print("원의 둘레:", 2 * 3.14 * number_input_a)
print("원의 넓이:", 3.14 * number_input_a * number_input_a)
```

기본 예외 처리

- 정수를 입력하지 않았을 경우

```
정수 입력> 7센티미터 Enter ──> 정수로 변환할 수 없는 문자열을 입력했습니다.
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    number_input_a = int(input("정수 입력> "))
ValueError: invalid literal for int() with base 10: '7센티미터'
```

기본 예외 처리

- 조건문으로 예외 처리하기
  - 위 슬라이드의 경우

    isdigit() 함수 사용하여 숫자로만 구성된 글자인지 확인

```
01  # 숫자를 입력받습니다.
02  user_input_a = input("정수 입력> ")
03
04  # 사용자 입력이 숫자로만 구성되어 있을 때
05  if user_input_a.isdigit():
06      # 숫자로 변환합니다.
07      number_input_a = int(user_input_a)
08      # 출력합니다.
09      print("원의 반지름:", number_input_a)
10      print("원의 둘레:", 2 * 3.14 * number_input_a)
11      print("원의 넓이:", 3.14 * number_input_a * number_input_a)
12  else:
13      print("정수를 입력하지 않았습니다.")
```

기본 예외 처리

- 정수 입력하면 정상적인 값 출력

```
정수 입력> 8 Enter
원의 반지름: 8
원의 둘레: 50.24
원의 넓이: 200.96
```

- 정수로 변환할 수 없는 문자열 입력하는 경우

```
정수 입력> yes!! Enter
정수를 입력하지 않았습니다.
```

try except 구문

- try except 구문
  - 예외 처리할 수 있는 구문

```
try:
    예외가 발생할 가능성이 있는 코드
except:
    예외가 발생했을 때 실행할 코드
```

  - 어떤 상황에 예외가 발생하는지 완벽하게 이해하고 있지 않아도 프로그램이 강제로 죽어버리는 상황은 막을 수 있음

try except 구문

- 예시

```
01  # try except 구문으로 예외를 처리합니다.
02  try:
03      # 숫자로 변환합니다.
04      number_input_a = int(input("정수 입력> ")) ---> 예외가 발생할 가능성이 있는 구문
05      # 출력합니다.
06      print("원의 반지름:", number_input_a)
07      print("원의 둘레:", 2 * 3.14 * number_input_a)
08      print("원의 넓이:", 3.14 * number_input_a * number_input_a)
09  except:
10      print("무언가 잘못되었습니다.") ---> 예외가 발생했을 때 실행할 구문
```

```
정수 입력> yes!! Enter
무언가 잘못되었습니다.
```

try except 구문

- try except 구문과 pass 키워드 조합하기
  - 예외가 발생하면 일단 처리해야 하지만, 해당 코드가 딱히 중요한 부분이 아닌 경우 프로그램 강제 종료부터 막는 목적으로 except 구문에 아무 것도 넣지 않고 try 구문 사용
  - pass 키워드를 빈 except 구문에 넣음

```
try:
    예외가 발생할 가능성이 있는 코드
except:
    pass
```

try except 구문

- 예시 – 숫자로 변환되는 것들만 리스트에 넣기

```
01  # 변수를 선언합니다.
02  list_input_a = ["52", "273", "32", "스파이", "103"]
03
04  # 반복을 적용합니다.
05  list_number = []
06  for item in list_input_a:
07      # 숫자로 변환해서 리스트에 추가합니다.
08      try:
09          float(item) # 예외가 발생하면 알아서 다음으로 진행은 안 되겠지?
10          list_number.append(item)  # 예외 없이 통과했으면 리스트에 넣어줘!
11      except:
12          pass
13
14  # 출력합니다.
15  print("{} 내부에 있는 숫자는".format(list_input_a))
16  print("{}입니다.".format(list_number))
```

실행결과

```
['52', '273', '32', '스파이', '103'] 내부에 있는 숫자는
['52', '273', '32', '103']입니다.
```

try except else 구문

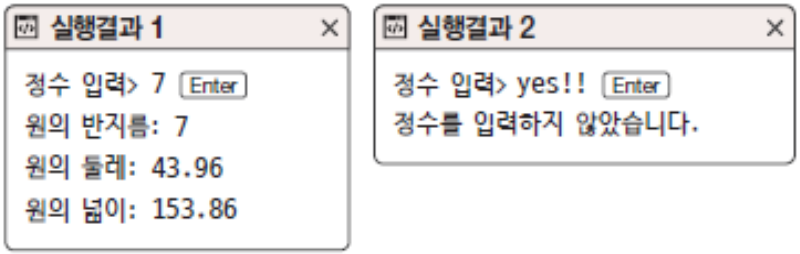
- try except 구문 뒤에 else 구문 붙여 사용하면
  예외가 발생하지 않았을 때 실행할 코드 지정할 수 있음

```
try:
    예외가 발생할 가능성이 있는 코드
except:
    예외가 발생했을 때 실행할 코드
else:
    예외가 발생하지 않았을 때 실행할 코드
```

try except else 구문

- 예시

```
01  # try except else 구문으로 예외를 처리합니다.
02  try:
03      # 숫자로 변환합니다.
04      number_input_a = int(input("정수 입력> "))
05  except:
06      print("정수를 입력하지 않았습니다.")
07  else:
08      # 출력합니다.
09      print("원의 반지름:", number_input_a)
10      print("원의 둘레:", 2 * 3.14 * number_input_a)
11      print("원의 넓이:", 3.14 * number_input_a * number_input_a)
```



finally 구문

- finally 구문
  - 예외 처리 구문에서 가장 마지막에 사용할 수 있는 구문
  - 예외 발생 여부와 관계없이 무조건 실행할 경우 사용

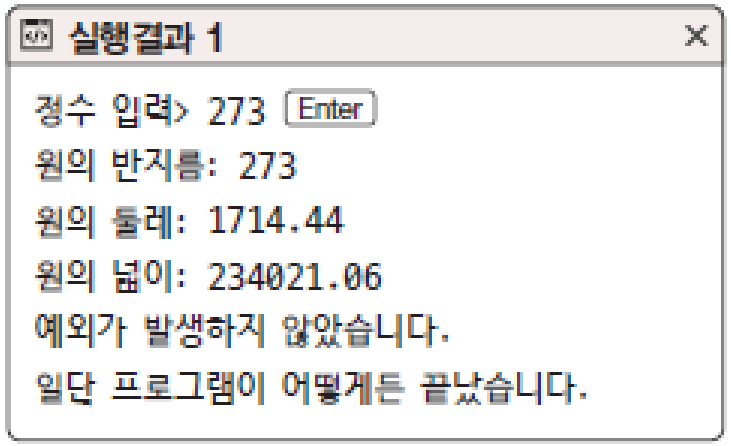
```
try:
    예외가 발생할 가능성이 있는 코드
except:
    예외가 발생했을 때 실행할 코드
else:
    예외가 발생하지 않았을 때 실행할 코드
finally:
    무조건 실행할 코드
```

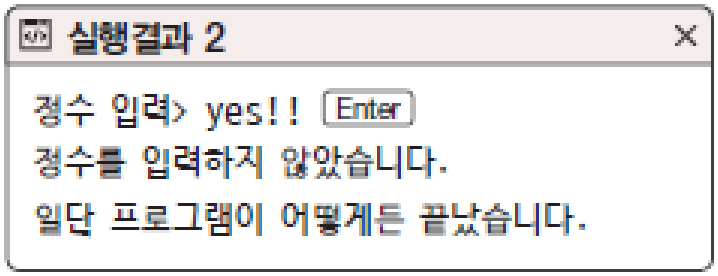
finally 구문

- finally 구문

```
01  # try except 구문으로 예외를 처리합니다.
02  try:
03      # 숫자로 변환합니다.
04      number_input_a = int(input("정수 입력> "))
05      # 출력합니다.
06      print("원의 반지름:", number_input_a)
07      print("원의 둘레:", 2 * 3.14 * number_input_a)
08      print("원의 넓이:", 3.14 * number_input_a * number_input_a)
09  except:
10      print("정수를 입력해달라고 했잖아요?!")
11  else:
12      print("예외가 발생하지 않았습니다.")
13  finally:
14      print("일단 프로그램이 어떻게든 끝났습니다.")
```

finally 구문





- tr

  - try 구문은 단독으로 사용할 수 없으며,

    반드시 except 구문 또는 finally 구문과 함께 사용해야 함

  - else 구문은 반드시 except 구문 뒤에 사용해야 함

finally 구문

- try + except 구문 조합
- try + except + else 구문 조합
- try + except + finally 구문 조합
- try + except + else + finally 구문 조합
- try + except 구문 조합

finally 구문

- 오류 경우

try + else 구문 조합

```
# try except 구문으로 예외를 처리합니다.
try:
    # 숫자로 변환합니다.
    number_input_a = int(input("정수 입력> "))
    # 출력합니다.
    print("원의 반지름:", number_input_a)
    print("원의 둘레:", 2 * 3.14 * number_input_a)
    print("원의 넓이:", 3.14 * number_input_a * number_input_a)
else:
    print("프로그램이 정상적으로 종료되었습니다.")
```

오류

```
SyntaxError: Invalid syntax
```

finally 구문

- finally에 대한 오해
  - finally 키워드 설명 예제로 '파일 처리'를 자주 사용하나, 실제 finally의 사용과는 사실 전혀 관련 없음
  - 파일 제대로 닫았는지는 파일 객체의 closed 속성으로 알 수 있음

```
01  # try except 구문을 사용합니다.
02  try:
03      # 파일을 엽니다.
04      file = open("info.txt", "w")
05      # 여러 가지 처리를 수행합니다.
06      # 파일을 닫습니다.
07      file.close()
08  except Exception as e:
09      print(e)
10
11  print("# 파일이 제대로 닫혔는지 확인하기")
12  print("file.closed:", file.closed)
```

실행결과

```
# 파일이 제대로 닫혔는지 확인하기
file.closed: True
```

finally 구문

- closed() 함수 사용 과정에서 예외 발생하여 try 구문 중간에 튕기는 경우

```
01  # try except 구문을 사용합니다.
02  try:
03      # 파일을 엽니다.
04      file = open("info.txt", "w")
05      # 여러 가지 처리를 수행합니다.
06      예외.발생해라()
07      # 파일을 닫습니다.
08      file.close()
09  except Exception as e:
10      print(e)
11
12  print("# 파일이 제대로 닫혔는지 확인하기")
13  print("file.closed:", file.closed)
```

실행결과

```
name '예외' is not defined
# 파일이 제대로 닫혔는지 확인하기
file.closed: False
```

finally 구문

- finally 구문 사용하여 파일 닫게 함

```
01  # try except 구문을 사용합니다.
02  try:
03      # 파일을 엽니다.
04      file = open("info.txt", "w")
05      # 여러 가지 처리를 수행합니다.
06      예외.발생해라()
07  except Exception as e:
08      print(e)
09  finally:
10      # 파일을 닫습니다.
11      file.close()
12
13  print("# 파일이 제대로 닫혔는지 확인하기")
14  print("file.closed:", file.closed)
```

실행결과

```
name '예외' is not defined
# 파일이 제대로 닫혔는지 확인하기
file.closed: True
```

finally 구문

- 예시 – try except 구문 끝난 후 파일 닫기

```
01  # try except 구문을 사용합니다.
02  try:
03      # 파일을 엽니다.
04      file = open("info.txt", "w")
05      # 여러 가지 처리를 수행합니다.
06      예외.발생해라()
07  except Exception as e:
08      print(e)
09
10  # 파일을 닫습니다.
11  file.close()
12  print("# 파일이 제대로 닫혔는지 확인하기")
13  print("file.closed:", file.closed)
```

실행결과

```
name '예외' is not defined
# 파일이 제대로 닫혔는지 확인하기
file.closed: True
```

finally 구문

- try 구문 내부에서 return 키워드를 사용하는 경우
  - 예시

```
01  # test() 함수를 선언합니다.
02  def test():
03      print("test() 함수의 첫 줄입니다.")
04      try:
05          print("try 구문이 실행되었습니다.")
06          return
07          print("try 구문의 return 키워드 뒤입니다.")
08      except:
09          print("except 구문이 실행되었습니다.")
10      else:
11          print("else 구문이 실행되었습니다.")
12      finally:
13          print("finally 구문이 실행되었습니다.")
14      print("test() 함수의 마지막 줄입니다.")
15
16  # test() 함수를 호출합니다.
17  test()
```

finally 구문은 무조건 실행됩니다. ←

실행결과

```
test() 함수의 첫 줄입니다.
try 구문이 실행되었습니다.
finally 구문이 실행되었습니다.
```

finally 구문

- try 구문 내부에 return 키워드 있음
  - try 구문 중간에서 탈출해도 finally 구문 무조건 실행됨

finally 구문

- 예시 – finally 키워드 활용

```
01  # 함수를 선언합니다.
02  def write_text_file(filename, text):
03      # try except 구문을 사용합니다.
04      try:
05          # 파일을 엽니다.
06          file = open(filename, "w")
07          # 여러 가지 처리를 수행합니다.
08          return
09          # 파일에 텍스트를 입력합니다.
10          file.write(text)
11      except Exception as e:
12          print(e)
13      finally:
14          # 파일을 닫습니다.
15          file.close()
16
17  # 함수를 호출합니다.
18  write_text_file("test.txt", "안녕하세요!")
```

finally 구문

- finally 구문에서 close() 함수 호출하도록 작성하면 코드 깔끔해짐

- 반복문과 함께 사용하는 경우

```
01  print("프로그램이 시작되었습니다.")
02
03  while True:
04      try:
05          print("try 구문이 실행되었습니다.")
06          break
07          print("try 구문의 break 키워드 뒤입니다.")
08      except:
09          print("except 구문이 실행되었습니다.")
10      finally:
11          print("finally 구문이 실행되었습니다.")
12      print("while 반복문의 마지막 줄입니다.")
13  print("프로그램이 종료되었습니다.")
```

실행결과

```
프로그램이 시작되었습니다.
try 구문이 실행되었습니다.
finally 구문이 실행되었습니다.
프로그램이 종료되었습니다.
```

키워드로 정리하는 핵심 포인트

- **구문 오류** : 프로그램의 문법적 오류로 프로그램이 실행조차 되지 않게 만드는 오류
- **예외 (런타임 에러)** : 프로그램 실행 중에 발생하는 오류. try except 구문 등으로 처리할 수 있다. 반대로 구문 오류는 실행 자체가 안 되므로 try except 구문으로 처리할 수 없다.
- **기본 예외 처리** : 조건문 등을 사용해 예외를 처리하는 기본적 방법
- **try except 구문** : 예외 처리에 특화된 구문

확인문제

- 구문 오류 (Syntax Error)와 예외(Exception)의 차이를 설명해보세요.
- 리스트 내부에서 특정 값이 어디 있는지 확인할 때는 리스트의 index() 함수를 아래처럼 사용합니다.

```
>>> numbers = [52, 273, 32, 103, 90, 10, 275]
>>> numbers.index(52)
0
>>> numbers.index(103)
3
```

  - 해당 값이 여러 개 있을 경우에는 다음과 같이 첫 번째 값의 위치를 리턴합니다.

```
>>> numbers = [1, 1, 1, 1, 1, 1, 1]
>>> numbers.index(1)
0
```

확인문제

- 그런데 이 함수는 리스트의 없는 값에 접근하려고 할 때 ValueError 예외가 발생합니다.

```
>>> numbers = [52, 273, 32, 103, 90, 10, 275]
>>> numbers.index(1000000)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    numbers.index(1000000)
ValueError: 1000000 is not in list
```

- 다음 코드의 빈칸을 ① 조건문을 사용한 코드, ② try except 구문을 사용한 코드로 채워서 예외가 발생하지 않고 코드가 실행결과처럼 출력되게 만들어주세요.

확인문제

```
numbers = [52, 273, 32, 103, 90, 10, 275]

print("# (1) 요소 내부에 있는 값 찾기")
print("- {}는 {} 위치에 있습니다.".format(52, numbers.index(52)))
print()

print("# (2) 요소 내부에 없는 값 찾기")
number = 10000
                ❶
  print("- {}는 {} 위치에 있습니다.".format(52, numbers.index(52)))
              ❷
  print("- 리스트 내부에 없는 값입니다.")
print()

print("--- 정상적으로 종료되었습니다. ---")
```

확인문제

실행결과

```
# (1) 요소 내부에 있는 값 찾기
- 52는 0 위치에 있습니다.

# (2) 요소 내부에 없는 값 찾기
- 리스트 내부에 없는 값입니다.

--- 정상적으로 종료되었습니다. ---
```