

파이썬으로 배우는 데이터 구조



한동대학교
전산전자공학부
김영섭 교수



학습 목표

try, except, else, finally를 사용해 다양하게
exception 구문을 사용하고 필요한 에러를 발생할 수 있다

Data Structures

Chapter 1 - 2

- Object-Oriented Programming
- OOP in Python
- OOP - Fraction Example
- OOP - Classes
- OOP - In-Place Operators
- Exceptions
- **Exception Clauses**

Quiz 1

The _____ statement causes the `__str__()` method to be invoked.

1. `print("object")`
2. `print(objectOfClass)`
3. `objectOfClass.print()`
4. None of the listed above

Quiz 2

Consider the following code:

```
class A:
    def __init__(self):
        self.x = 1
        self.__y = 1

    def get_y(self):
        return self.__y

a = A()
print(a.__y)
```

Select a correct statement.

1. The program has an error because x is private and cannot be access outside of the class.
2. The program has an error because y is private and cannot be access outside of the class.
3. The program has an error because you cannot name a variable using __y.
4. The program runs fine and prints 1.
5. The program runs fine and prints 0

Agenda

- Topics: Python Review
 - Exception Handling
- Learning outcomes
 - Understand the flow of control that occurs with exceptions.
 - **try, except, else, finally**
 - Use exceptions to handle unexpected runtime errors gracefully.
 - Catch an exception of the appropriate type.
 - Throw an exception.
 - Raise exceptions when appropriate.
- Resources
 - Errors and Exceptions — Python 3.9.6 documentation
 - <https://docs.python.org/3/tutorial/errors.html>
 - Python3 Tutorial: Exception Handling
 - https://www.python-course.eu/python3_exception_handling.php

The else clause

- Executed only if the **try** clause completes with no errors
 - It is useful for code that must be executed if the **try** clause does not raise an exception.

```
try:  
    statement block here  
except:  
    more statements here (undo operations)  
else:  
    more statements here (close operations)
```

The else clause - Examples

- Examples:

```
try:
    age = int(input("Please enter your age: "))
except ValueError:
    print("Hey, that wasn't a number!")
else:
    print("I see that you are %d years old." % age)
```

Please enter your age: **4**
I see that you are 4 years old.

Please enter your age: **hello**
Hey, that wasn't a number!

The else clause - Exercise 1

- What is the output of the following code snippet?

```
try:
    my_list = [1, 2, 3]
    num = int(input('Enter an index: '))
    value = my_list[num]
except IndexError:
    print("Invalid index!")
else:
    print(value)
print("DONE")
```

- Cases:

1. Enter an index: 1
2. Enter an index: 6

Enter an index: 1

Enter an index: 6

The Finally clause

- The **try** statement in Python can have an optional **finally** clause.
- It executes after the **try** and **except** blocks, but before the entire **try-except** ends.
- Code within a **finally** block **is guaranteed to be executed** if any part of the associated **try** block is executed regardless of an exception being thrown or not.
 - It allows for cleanup of actions that occurred in the try block but may remain undone if an exception is caught
 - Often used with files to close the file.

```
try:  
    statement block here  
except:  
    more statements here (undo operations)  
finally:  
    more statements here (close operations)
```

The Finally clause - Examples

```
def divide(a, b):  
    try:  
        result = a / b  
    except ZeroDivisionError:  
        result = 'Divided by zero'  
    else:  
        print("result is", result)  
    finally:  
        print("finally clause")  
    return result
```

- Case 1: No error

➡ `x = divide(2, 1)`
`print(x)`

result is 2.0
finally clause
2.0

- Case 2: Divided by zero

The Finally clause - Examples

```
def divide(a, b):  
    try:  
        result = a / b  
    except ZeroDivisionError:  
        result = 'Divided by zero'  
    else:  
        print("result is", result)  
    finally:  
        print("finally clause")  
    return result
```

- Case 1: No error

→ `x = divide(2, 1)`
`print(x)`

result is 2.0
finally clause
2.0

```
def divide(a, b):  
    try:  
        result = a / b  
    except ZeroDivisionError:  
        result = 'Divided by zero'  
    else:  
        print("result is", result)  
    finally:  
        print("finally clause")  
    return result
```


- Case 2: Divided by zero

`x = divide(2, 0)`
`print(x)`

finally clause
Divided by zero

The Finally clause - Examples

```
def divide(a, b):  
    try:  
        result = a / b  
    except ZeroDivisionError:  
        result = 'Divided by zero'  
    else:  
        print("result is", result)  
    finally:  
        print("finally clause")  
    return result
```



- Case 3: Other error

```
x = divide('2', '1')  
print(x)
```

```
finally clause  
Traceback (most ...  
TypeError: unsupported operand type(s) ...
```

Exercise 2

- What is the output of the following code snippet?

```
try:
    age = int(input("Please enter your age: "))
except ValueError:
    print("Hey, that wasn't a number!")
else:
    print("I see that you are %d years old." % age)
finally:
    print("It was really nice talking to you. Goodbye!")
```

- Cases:

- Please enter your age: a
- Please enter your age: -1
- Please enter your age: 4

Please enter your age: a

Please enter your age: -1

Please enter your age: 4

FileNotFoundError & IOError

- **Raised** when an input/ output operation fails, such as the print statement or the open function when trying to open a file that does not exist.
- Example:

```
input_file = open ("numbers1.txt", "r")

print ("Reading from file numbers.txt")
one_line = input_file.readline()
print(one_line)

print ("Completed reading of file input.txt")
input_file.close()
```

- It generates the following error:

FileNotFoundError

Traceback (most recent call last)

...

FileNotFoundError: [Errno 2] No such file or directory: 'numbers1.txt'

Handling With Exceptions for FileIO

- Basic structure of handling exceptions

```
try:
    # Attempt something where exception error may happen
    # (i.e. open a file and read the content)
except IOError:
    # React to the error
else:
    # What to do if no error is encountered
    # (i.e. close the file)
finally:
    # Actions that must always be performed
```


Handling With Exceptions for FileIO - Example

- Example

```
try:
    inputFileName = input("Enter name of input file: ")
    input_file = open(inputFileName, "r")
except IOError:
    print("File", inputFileName, "could not be opened")
else:
    one_line = input_file.readline()
    print(one_line)
    input_file.close()
print('Hello World')
```

- Case 1:

Enter name of input file: **_start_jupyter.bat**
rem -- start jupyter notebook here .bat file

Hello World
- Case 2:

Enter name of input file: test.txt
File test.txt could not be opened
Hello World

Raising an exception

- You can create an exception by using the raise statement.

```
raise Error('Error message goes here')
```

- The program stops immediately after the raise statement; and any subsequent statements are not executed.
- It is normally used in testing and debugging purpose.

- Example:

```
def checkLevel(level):  
    if level < 1:  
        raise ValueError('Invalid level!')  
    else:  
        print (level)
```

```
Traceback (most recent call last):  
...  
raise ValueError('Invalid level!')  
ValueError: Invalid level!
```

Using Exceptions

- Put code that might create a runtime error is enclosed in a try block.

```
def checkLevel(level):  
    try:  
        if level < 1:  
            raise ValueError('Invalid level!')  
        else:  
            print (level)  
            print ('This print statement will not be reached.')  
    except ValueError as e:  
        print ('Problem: {0}'.format(e))
```

Problem: Invalid level!

```
def checkLevel(level):  
    try:  
        if level < 1:  
            raise ValueError('Invalid level!')  
        ...  
    except ValueError as e:  
        pass
```

Optionally you may pass it (or exception object).

Using Exceptions

- When to use **try-except** blocks?
 - If you are executing statements that you know are unsafe and you want the code to **continue running anyway**.
- When to **raise an exception**?
 - When there is a problem that you cannot deal with at that point in the code, and you want to "pass the buck" so **the problem can be dealt with elsewhere**.
 - **"The buck stops here"**



Exceptions

- Any kind of built-in error can be caught
 - Check the Python documentation for the complete list
 - Some popular errors:
 - `ArithmeticError`: various arithmetic errors
 - `ZeroDivisionError`: dividing by zero
 - `IndexError`: a sequence subscript is out of range
 - `TypeError`: inappropriate type
 - `ValueError`: has the right type but an inappropriate value
 - `IOError`: Raised when an I/O operation
 - `EOFError`: hits an end-of-file condition without reading any data
 - ...
- Resources: Built-in Exceptions:
 - <https://docs.python.org/3/library/exceptions.html>
 - <https://docs.python.org/3/library/exceptions.html#exception-hierarchy>

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |   +-- FloatingPointError
    |   +-- OverflowError
    |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    |   +-- ModuleNotFoundError
    +-- LookupError
    |   +-- IndexError
    |   +-- KeyError
    +-- MemoryError
    +-- NameError
    |   +-- UnboundLocalError
    +-- OSError
    |   +-- BlockingIOError
    |   +-- ChildProcessError
    |   +-- ConnectionError
    |   |   +-- BrokenPipeError
    |   |   +-- ConnectionAbortedError
    |   |   +-- ConnectionRefusedError
    |   |   +-- ConnectionResetError
    |   +-- FileExistsError
    |   +-- FileNotFoundError
```

Exercise 3

- Modify the following function that calculates the mean value of a list of numbers to ensure that the function generates an informative exception when input is unexpected.

```
def mean(data):  
    sum = 0  
    for x in data:  
        sum += x  
    return sum / len(data)
```

- Sample Run:

```
a = mean([1,3,2,4,5])  
b = mean([1,3,'5',4,5])  
c = mean([])  
print(a, b, c)
```

```
Joyful error: The list must contain only numbers  
Joyful error: Divide by zero  
3.0 None None
```

Exercise 3

- Modify the following function that calculates the mean value of a list of numbers to ensure that the function generates an informative exception when input is unexpected.

```
def mean(data):  
    sum = 0  
    for x in data:  
        sum += x  
    return sum / len(data)
```

- Sample Run:

```
a = mean([1,3,2,4,5])  
b = mean([1,3,'5',4,5])  
c = mean([])  
print(a, b, c)
```

```
def mean(data):  
    sum = 0  
    try:  
        for x in data:  
            sum += x  
        return sum / len(data)  
    except ZeroDivisionError:  
        print('Joyful error: Divide by zero')  
    except TypeError:  
        print('Joyful error: The list must contain only numbers')
```

```
Joyful error: The list must contain only numbers  
Joyful error: Divide by zero  
3.0 None None
```

Exercise 4

- Modify the following function that calculates the sum of a list of numbers.
 - Ensure that it catches an informative exception when input is unexpected but continue the summation process until the end of the list and return the sum.
 - The list must have at least one element. Otherwise, raise a ValueError exception with a message "Joyful error: Have at least one element".

```
def getSum(data):  
    sum = 0  
    for x in data:  
        sum += x  
    return sum
```

- Sample Run:

```
try:  
    a = getSum([1,3,2,4,5])  
    b = getSum([1,3,'5',4,5])  
    c = getSum([])  
    print(a, b, c)  
except Exception as e:  
    print(e)
```

This message is generated by the system, but received and printed by in our code.

```
Joyful error: Invalid number found in the list  
unsupported operand type(s) for +=: 'int' and 'str'  
Joyful error: Have at least one element
```


Summary

- Exceptions alter the flow of control
 - When an exception is raised, execution stops
 - When the exception is caught, execution starts again
- try... except blocks are used to handle problem code
 - Can ensure that code fails gracefully
 - Can ensure input is acceptable
- finally
 - Executes code after the exception handling code

학습 정리

- 1) exception에서 else구문은 try구문에서 에러가 발생하지 않았을 때 실행된다
- 2) finally 구문은 에러 발생 유무에 관계없이 실행되는 구문이다
- 3) raise 구문을 이용하여 프로그래머가 원하는 에러를 직접 발생시킬 수 있다

파이썬으로 배우는 데이터 구조

수고했습니다
곧 다음 시간에
다시 뵙겠습니다

