Data Structures in Python Chapter 1

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University

Agenda

- Topics: Python Review
  - Exception Handling
- Learning outcomes
  - Understand the flow of control that occurs with exceptions.
    - **try, except, 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

Introduction

- **Errors occur in software programs**.
  - However, if you handle errors properly,
    you will greatly improve your program's **readability, reliability and maintainability**.
  - Python uses exceptions for error handling.
- Exception examples:
  - Attempt to divide by ZERO
  - Couldn't find the specific file to read
- The run-time system will attempt to handle the exception (default exception handler), usually by displaying an error message and terminating the program.

Divide by zero error

- Check for valid input first
  - Only accept input where the divisor is non-zero

```
def divide(a, b):
    if b == 0:
        result = 'Error: cannot divide by zero'
    else:
        result = a / b
    return result
```

Divide by zero error

- Check for valid input first
  - Only accept input where the divisor is non-zero

```
def divide(a, b):
    if b == 0:
        result = 'Error: cannot divide by zero'
    else:
        result = a / b
    return result
```

- What if “b” is not a number

```
def divide(a, b):
    if (type(b) is not int and type(b) is not float):
        result = "Error: divisor is not a number"
    elif b == 0:
        result = 'Error: cannot divide by zero'
...
```

Handling input error

- Check for valid input first
  - What if “a” is not a number?

```
def divide(a, b):
    if (type(b) is not int and type(b) is not float or
        type(a) is not int and type(a) is not float):
        result = ('Error: one or more operands' +
                       ' is not a number')
    elif b == 0:
        result = 'Error: cannot divide by zero'
    else:
        result = a / b
    return result

x = divide(5, 'hello')
print(x)
```

What is an Exception?

- An exception is **an event** that occurs during the execution of a program that **disrupts** the normal flow of instructions during the execution of a program.
- When an error occurs within a method, the method creates an **exception object** and hands it off to the runtime system.
- The **exception object** contains
  - **information** about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called **throwing** an exception.

Handling exceptions

- Code that might create a runtime error is enclosed in a **try** block.
  - Statements are executed sequentially as normal.
  - If an error occurs then the remainder of the code is skipped.
  - The code starts executing again at the **except** clause.
    - The exception is "caught“.

```
try:
    statement block
    statement block
except:
    exception handling statements
    exception handling statements
```

- Advantages of catching exceptions:
  - It allows you to fix the error.
  - It prevents the program from automatically terminating.

Handling exceptions - Case 1

```
def divide(a, b):
    try:
        result = a / b
        print ("try-block")
    except:
        result = 'Error in input data'
        print ("except-block")
    return result
```

- Case 1: No error
  - `divide(5, 5)`

```
x = divide(5, 5)
print ("Program continues here...")
print(x)
```

```
try-block
Program continue here...
1.0
```

Handling exceptions - Case 2

```
def divide(a, b):
    try:
        result = a / b ✘
        print ("try-block")
    except:
        result = 'Error in input data'
  ➡     print ("except-block")
    return result
```

- Case 2: Invalid input
  - `divide(5, 0)`
  - `divide(5, 'Hello')`

```
➡ x = divide(5, 0)
➡ print ("Program continues here...")
  print(x)
```

```
except-block
Program continues here...
Error in input data
```

- But what is the error in each situation?
  1. `5/0` → `ZeroDivisionError:division by zero`
  2. `5/'hello'`→`TypeError:unsupported operand type(s) for /: 'int' and 'str'`

Exercise 1

- What is the output of the following?

```
def divide(dividend, divisor):
    try:
        quotient = dividend / divisor
    except:
        quotient = 'Error in input data'
    return quotient

x = divide(5, 0)
print(x)
x = divide('hello', 'world')
print(x)
x = divide(5, 5)
print(x)
```

Exercise 1

- What is the output of the following?

```
def divide(dividend, divisor):
    try:
        quotient = dividend / divisor
    except:
        quotient = 'Error in input data'
    return quotient

x = divide(5, 0)
print(x)
x = divide('hello', 'world')
print(x)
x = divide(5, 5)
print(x)
```

```
Error in input data

Error in input data

1.0
```

Danger in catching all exceptions

- The general **except** clause catching all runtime errors.
  - Sometimes that can hide problems.
- You can put two or more except clauses, each except block is an exception handler and handles the type of exception indicated by its argument in a program.
  - The runtime system invokes the exception handler when the handler is the FIRST ONE matches the **type** of the exception thrown.
    - It executes the statement inside the matched except block, the other except blocks are bypassed and continues after the try-except block.

Specifying the exceptions

```
def divide(a, b):
    try:
        result = a / b
    except TypeError:
        result = 'Type of operands is incorrect'
    except ZeroDivisionError:
        result = 'Divided by zero'
    return result
```

- Case 1:
  - No error

```
x = divide(5, 5)
print(x)
```

```
1.0
```

Specifying the exceptions

```
def divide(a, b):
    try:
        result = a / b
    except TypeError:
        result = 'Type of operands is incorrect'
    except ZeroDivisionError:
        result = 'Divided by zero'
    return result
```

- Case 2:
  - is not a number

```
x = divide('hello', 5)
print(x)
```

```
Type of operands is incorrect
```

Specifying the exceptions

```
def divide(a, b):
    try:
        result = a / b
    except TypeError:
        result = 'Type of operands is incorrect'
    except ZeroDivisionError:
        result = 'Divided by zero'
    return result
```

- Case 3:
  - is not a number

```
x = divide(5, 0)
print(x)
```

```
Divided by zero
```

Specifying the exceptions - Exception not Matched

- If no matching except block is found, the run-time system will attempt to handle the exception, by terminating the program.

```
def divide(a, b):
    try:
        result = a / b ✗
    except IndexError:
        result = 'Type of operands is incorrect'
    except ZeroDivisionError:
        result = 'Divided by zero'
    return result
```

- Case 4:
  - Exception not matched

```
x = divide('abc', 0)
print(x)
```

```
TypeError                            Traceback (most recent call last):
<ipython-input-2-da1852461651> in <module>
----> 1 x = divide('abc', 0)
      2 print(x)result = a / b TypeError: unsupported operand
<ipython-input-1-5faae4bc1dfe> in divide(a, b)
      1 def divide(a, b):
      2     try:
----> 3         result = a / b
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Specifying the exceptions - Order of except clauses

- Specific exception block must come **before** any of their general exception block.

✘

```
def divide(a, b):
    try:
        result = a / b
    except:
        result = 'Type of operands is incorrect'
    except ZeroDivisionError:
        result = 'Divided by zero'
    return result
```

general exception block

This code will **never** catch ZeroDivisionError exception.

✓

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        result = 'Divided by zero'
    except:
        # handle all other exceptions
        result = ...
    ...
```

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 0
    - `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 (EOF) 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 2

- Consider the following code:

```
my_list = [1, 2, 3]
num = int(input('Enter an index: '))
print(my_list[num])
```

  - Sample Run:

```
Enter an index: 1
2
```

```
Enter an index: 6
...
IndexError: list index out of range
```

- Rewrite it using try-except block to handle the general error.
  - Sample Run:

```
Enter an index: 1
2
```

```
Enter an index: 6
DSpy joyful error: list index out of range
```

your own message

system's error message

Exercise 2 - solution

```python
my_list = [1, 2, 3]
num = int(input('Enter an index: '))
try:
    print(my_list[num])
except Exception as e:
    print('DSpy joyful error: ' + str(e))
```

- Rewrite it using try-except block to handle the general error.
  - Sample Run:

```
Enter an index: 1
2
```

```
Enter an index: 6
DSpy joyful error: list index out of range
```

your own message

system's error message

Exercise 3

- Consider the following code:

```
rgb = {'red': 1, 'green': 2, 'blue': 3 }
num = input('Enter a key: ')
print(rgb[num])
```

  - Sample Run:

```
Enter a key: red
1
```

```
Enter a key: orange
...
KeyError: 'orange'
```

- Rewrite it using try-except block to handle the **KeyError**.
  - Sample Run:

```
Enter a key: red
2
```

```
Enter a key: orange
DSpy joyful error: Invalid key!
```

your own message

system's error message

Exercise 3 - solution

```python
rgb = {'red': 1, 'green': 2, 'blue': 3 }
num = input('Enter a key: ')

try:
    print(rgb[num])
except KeyError:
    print('DSpy joyful error: ' + 'Invalid Key!')
except Exception as e:
    print('DSpy joyful error: ' + str(e))
```

- Rewrite it using try-except block to handle the **KeyError**.
  - Sample Run:

```
Enter a key: red       2
Enter a key: orange    DSpy joyful error: Invalid key!
```

your own message

your own message

Data Structures in Python Chapter 1

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University