COMPSCI 105 S1 2017
Principles of Computer Science

Exceptions

MCQ

- The __________statement causes the __str__ method to be invoked.
  A. print(objectOfClass).
  B. print("object“)
  C. objectOfClass.print().
  D. None of the others.

```
x = Fraction(2, 3)
```

MCQ Exericse

- Consider the following code:

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

    def getY(self):
        return self.__y

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

A. The program has an error because x is private and cannot be access outside of the class.
B. The program has an error because y is private and cannot be access outside of the class.
C. The program has an error because you cannot name a variable using __y.
D. The program runs fine and prints 1.
E. The program runs fine and prints 0.

Learning outcomes

- Understand the flow of control that occurs with exceptions
  - try, except, finally
- Use exceptions to handle unexpected runtime errors gracefully
  - 'catching' an exception of the appropriate type
- Generate exceptions when appropriate
  - raise an exception

- Resources:
  - Errors and Exceptions — Python 3.4.2 documentation
    - https://docs.python.org/3/tutorial/errors.html
  - Python3 Tutorial: Exception Handling
    - http://www.python-course.eu/python3_exception_handling.php

Introduction

- Errors occur in software programs. However, if you handle errors properly, you'll 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.

`Example01.py`

Handling unexpected input values

- What if the function is passed a value that causes a divide by zero?
  - Error caused at runtime
  - Error occurs within the function
  - Problem is with the input
- What can we do?
  - You can try to check for valid input first

```
def divide(a, b):
    result = a / b
    return result

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

where

```
Traceback (most recent call last):
  File "...", line 5, in <module> x = divide(5, 0)
  File "...", line 2, in divide  result = a / b
ZeroDivisionError: division by zero
```

reason

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

Case 1

Example02.py

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

1 ✓

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

```
x = divide(5, 5)
print ("Program can continue to run...")
print(x)
```

2 ✓

3 ✓

```
try-block
Program can continue to run...
1.0
```

Case 2

```
def divide(a, b):                                   Example02.py
    try:
        result = a / b ✗
        print ("try-block")
    except:
        result = 'Error in input data'
  1 ✓   print ("Error")
    return result
```

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

```
  x = divide(5, 'hello')
2 ✓ print ("Program can continue to run...")
3 ✓ print (x)
```

```
Error
Program can continue to run...
Error in input data
```

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

Exercise 01

- What is the output of the following?

```
def divide(dividend, divisor):
    try:
        quotient = dividend / divsor
    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)
```

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.

Example03.py

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
```

Example03.py

Specifying the exceptions

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

- Case 2:
  - is not a number

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

2 ✓

```
Type of operands is incorrect
```

Example03.py

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:
  - Division Error

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

2 ✓

```
Divided by zero
```

Example04.py

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
```

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

```
Traceback (most recent call last):
  File "...", line 11, in <module> x =
divide('abc', 0)File "...", line 3, in divide
result = a / b TypeError: unsupported operand
type(s) for /: 'str' and 'int'
```

Example05.py

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
```

```
result = a / b
SyntaxError: default 'except:' must be last
```

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

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
    - 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
    - …

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
```

Exercise 2

- Consider the following code:

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

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

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

- Rewrite it using try-except block to handle the IndexError

```
Enter an index: 6
Invalid index!
```

Exercise 3

- Consider the following code:

```
my_dict = {'test1':1,'test2':2}
num = input('Enter a key: ')
print(my_dict[num])
```

```
Enter a key: test1
1
```

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

- Rewrite it using try-except block to handle the KeyError

```
Enter a key: test
Invalid Key!
```

Example06.py

More specific feedback

- If you want to give the user more specific feedback about which input was wrong

```
try:
    dividend = int(input("Please enter the dividend: "))
    divisor = int(input("Please enter the divisor: "))
    print("%d / %d = %f" % (dividend, divisor, dividend/divisor))
except ValueError:
    print("The divisor and dividend have to be numbers!")
except ZeroDivisionError:
    print("The dividend may not be zero!")
```

```
try:
    dividend = int(input("Please enter the dividend: "))
except ValueError:
    print("The dividend has to be a number!")

try:
    divisor = int(input("Please enter the divisor: "))
except ValueError:
    print("The divisor has to be a number!")

try:
    print("%d / %d = %f" % (dividend, divisor, dividend/divisor))
except ZeroDivisionError:
    print("The dividend may not be zero!")
```