COMPSCI 107
Computer Science Fundamentals

Lecture 07 – Exceptions

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

Example

- Typical code with no errors

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

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

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?

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

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

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

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

Handling input error

- Check for valid input first
  - 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'
    else:
        result = a / b
    return result

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

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 = a / b
    else:
        result = 'Error: cannot divide by zero'
    return result

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

try

- 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
```

Alternative method of handling input error

- Assume input is correct
  - Deal with invalid input an an exceptional case

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

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

Exercise

- 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

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

x = divide(5, 2.5)
print(x)
```

Specifying the exceptions

- Can choose to catch only some 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

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

Exceptions

- Any kind of built-in error can be caught
  - Check the Python documentation for the complete list

```
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
```

Some other useful techniques

- *raise*
  - Creates a runtime error – by default, the most recent runtime error

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

- *raise Error('Error message goes here')*

```
raise ValueError("My very own runtime error")
```

Exercise

- Given the following function that converts a day number into the name of a day of the week, write code that will generate an informative exception if the name is outside the desired range.

```
def weekday(n):
    names = ['Sunday', 'Monday', 'Tuesday', 'Wednesday',
             'Thursday', 'Friday', 'Saturday']
    return names[n]
```

Example

- Given the following function that converts a day number into the name of a day of the week, write code that will generate an informative exception if the name is outside the desired range.

```
def weekday(n):
    if n not in range(0, 8):
        raise ValueError('Data must be between 0 and 7 inclusive')

    names = ['Sunday', 'Monday', 'Tuesday', 'Wednesday',
             'Thursday', 'Friday', 'Saturday']
    return names[n]
```

Exercise

- 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 element in data:
        sum += element
    mean = sum / len(data)
    return mean
```

Some other useful techniques

- *finally*
  - Executed after the try and except blocks, but before the entire try-except ends
  - Often used with files to close the file

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

- *else*
  - Executed only if the try clause completes with no errors

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

Example

```
try:
    file = open('test.txt')
    file.write('Trying to write to a read-only file')
except IOError:
    print('Error: cant find file or read data')
else:
    print("Written content in the file successfully")
finally:
    file.close()
```

Exercise

- *What is the output of the following program when x is 1, 0 and '0'?*

```
try:
    print('Trying some code')
    2 / x
except ZeroDivisionError:
    print('ZeroDivisionError raised here')
except:
    print('Error raised here')
    raise
else:
    print('Else clause')
finally:
    print('Finally')
```

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
- raise
  - Creates a runtime exception
- finally
  - Executes code after the exception handling code