Exception Handling

BBM 101 - Introduction to Programming I

Hacettepe University
Fall 2016

Fuat Akal, Aykut Erdem, Erkut Erdem

What is an Exception?

- An exception is an abnormal condition (and thus rare) that arises in a code sequence at runtime. For instance:
  - Dividing a number by zero
  - Accessing an element that is out of bounds of an array
  - Attempting to open a file which does not exist

- When an exceptional condition arises, an object representing that exception is created and thrown in the code that caused the error

- An exception can be caught to handle it or pass it on

- Exceptions can be generated by the run-time system, or they can be manually generated by your code

What is an Exception?

```
test = [1,2,3]
test[3]
```

```
IndexError: list index out of range
```

What is an Exception?

```
successFailureRatio = numSuccesses/numFailures
print('The success/failure ratio is',
successFailureRatio)
print('Now here')
```

```
ZeroDivisionError: integer division or
modulo by zero
```

What is an Exception?

```
val = int(input('Enter an integer: '))
print('The square of the number you entered is', val**2)
```

```
> Enter an integer: asd
```

```
ValueError: invalid literal for int() with base 10: 
'asd'
```

Handling Exceptions

- Exception mechanism gives the programmer a chance to do something against an abnormal condition.

- Exception handling is performing an action in response to an exception.

- This action may be:
  - Exiting the program
  - Retrying the action with or without alternative data
  - Displaying an error message and warning user to do something
  - ....

Handling Exceptions

```
try:
    successFailureRatio = numSuccesses/numFailures
    print('The success/failure ratio is', successFailureRatio)
Except ZeroDivisionError:
    print('No failures, so the success/failure is undefined.')
print('Now here')
```

- Upon entering the **`try`** block, the interpreter attempts to evaluate the expression **`numSuccesses/numFailures`**.
- If expression evaluation is successful, the assignment is done and the result is printed.
- If, however, a **`ZeroDivisionError`** exception is raised, the print statement in the **`except`** block is executed.

Handling Exceptions

```
while True:
    val = input('Enter an integer: ')
    try:
        val = int(val)
        print('The square of the number you entered is', val**2)
        break #to exit the while loop
    except ValueError:
        print(val, 'is not an integer')
```

Checks for whether `ValueError` exception is raised or not

Keywords of Exception Handling

- There are five keywords in Python to deal with exceptions: **`try`**, **`except`**, **`else`**, **`raise`** and **`finally`**.

- **`try`**: Creates a block to monitor if any exception occurs.

- **`except`**: Follows the try block and catches any exception which is thrown within it.

Are There Many Exceptions in Python?

- Yes, some of them are…
  - `Exception`
  - `ArithmeticError`
  - `OverflowError`
  - `ZeroDivisonError`
  - `EOFError`
  - `NameError`
  - `IOError`
  - `SyntaxError`

List of all exceptions (errors):
http://docs.python.org/3/library/exceptions.html#bltin-exceptions

Multiple except Statements

- It is possible that more than one exception can be thrown in a code block.
  - We can use multiple **`except`** clauses
- When an exception is thrown, each **`except`** statement is inspected in order, and the first one whose type *matches* that of the exception is executed.
  - Type matching means that the exception thrown must be an object of the same class or a sub-class of the declared class in the **`except`** statement
- After one **`except`** statement executes, the others are bypassed.

Multiple except Statements

```
try:
        You do your operations here;
except Exception-1:
        Execute this block.
except Exception-2:
        Execute this block.
except (Exception-3[, Exception-4[,...ExceptionN]]]):
        If there is any exception from the given exception list,
        then execute this block.
```

```
except (ValueError, TypeError):
        …
```

The except block will be entered if <u>any of the listed exceptions</u> is raised within the try block

Multiple except Statements

```
try:
    f = open('outfile.dat', 'w')
    dividend = 5
    divisor = 0
    division = dividend / divisor
    f.write(str(division))
except IOError:
    print("I can't open the file!")
except ZeroDivisionError:
    print("You can't divide by zero!")
```

```
You can't divide by zero!
```

Multiple except Statements

```
try:
    f = open('outfile.dat', 'w')
    dividend = 5
    divisor = 0
    division = dividend / divisor
    f.write(str(division))
except Exception:
    print("Exception occured and handled!")
except IOError:
    print("I can't open the file!")
except ZeroDivisionError:
    print("You can't divide by zero!")
```

```
Exception occured and handled!
```

Multiple except Statements

```
try:
    f = open('outfile.dat', 'w')
    dividend = 5
    divisor = 0
    division = dividend / divisor
    f.write(str(division))
except:
    print("Exception occured and handled!")
except IOError:
    print("I can't open the file!")
except ZeroDivisionError:
    print("You can't divide by zero!")
```

```
SyntaxError: default 'except:' must be last
```

except-else Statements

```
try:
      You do your operations here
except:
      Execute this block.
else:
      If there is no exception then execute this block.
```

```
try:
      f = open(arg, 'r')
except IOError:
      print('cannot open', arg)
else:
      print(arg, 'has', len(f.readlines()), 'lines')
```

finally Statement

- **`finally`** creates a block of code that will be executed after a **`try/except`** block has completed and before the code following the **`try/except`** block
- **`finally`** block is executed whether or not exception is thrown
- **`finally`** block is executed whether or not exception is caught
- It is used to gurantee that a code block will be executed in any condition.

finally Statement

You can use it to clean up files, database connections, etc.

```
try:
       You do your operations here
except:
       Execute this block.
finally:
       This block will definitely be executed.
```

```
try:
       file = open('out.txt', 'w')
       do something…
finally:
       file.close()
       os.path.remove('out.txt')
```

Nested try Blocks

- When an exception occurs inside a `try` block;
  - If the `try` block does not have a matching except, then the outer `try` statement's except clauses are inspected for a match
  - If a matching except is found, that except block is executed
  - If no matching except exists, execution flow continues to find a matching except by inspecting the outer try statements
  - If a matching except cannot be found at all, the exception will be caught by Python's exception handler.

- Execution flow never returns to the line that exception was thrown. This means, an exception is caught and except block is executed, the flow will continue with the lines following this except block

Let’s clarify it on various scenarios

```
try:
    statement1
    try:
            statement2
    except Exception1:
            statement3
    except Exception2:
            statement4;
    try:
            statement5
    except Exception3:
            statement6
    statement7;
except Exception3:
    statement8
statement9;
```

Information: Exception1 and Exception2 are subclasses of Exception3

Question: Which statements are executed if

1- statement1 throws Exception1

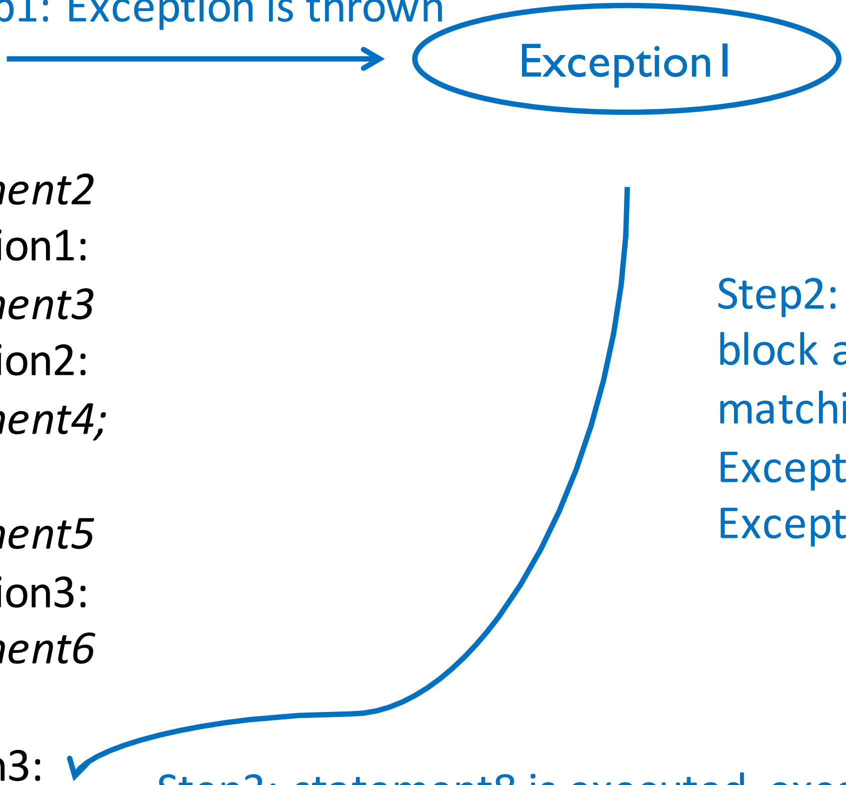
2- statement2 throws Exception1

3- statement2 throws Exception3

4- statement2 throws Exception1 and statement3 throws Exception2

Scenario: statement1 throws Exception1

```
try:
    statement1
    try:
            statement2
    except Exception1:
            statement3
    except Exception2:
            statement4;
    try:
            statement5
    except Exception3:
            statement6
    statement7;
except Exception3:
    statement8
statement9;
```



Step1: Exception is thrown

Step2: except clauses of the try block are inspected for a matching except statement. Exception3 is super class of Exception1, so it matches.

Step3: statement8 is executed, exception is handled and execution flow will continue bypassing the following except clauses

Step4: statement9 is executed

Scenario: statement2 throws Exception1

```
try:
   statement1
   try:
         statement2
   except Exception1:
         statement3
   except Exception2:
         statement4;
   try:
         statement5
   except Exception3:
         statement6
   statement7;
except Exception3:
   statement8
statement9;
```

Step1: Exception is thrown

Exception1

Step2: except clauses of the try block are inspected for a matching except statement. First clause catches the exception

Step3: statement3 is executed, exception is handled

Step4: execution flow will continue bypassing the following except clauses. statement5 is executed.

Step5: Assuming no exception is thrown by statement5, program continues with statement7 and statement9.

Scenario: statement2 throws Exception3

```
try:
   statement1
   try:
         statement2
   except Exception1:
         statement3
   except Exception2:
         statement4;
   try:
         statement5
   except Exception3:
         statement6
   statement7;
except Exception3:
   statement8
statement9;
```

Step1: Exception is thrown

Exception3

Step2: except clauses of the try block are inspected for a matching except statement. None of these except clauses match Exception3

Step3: except clauses of the outer try statement are inspected for a matching except . Exception3 is catched and statement8 is executed

Step4: statement9 is executed

Scenario: statement2 throws Exception1 and statement3 throws Exception2

```
try:
    statement1
    try:
            statement2
    except Exception1:
            statement3
    except Exception2:
            statement4;
    try:
            statement5
    except Exception3:
            statement6
    statement7;
except Exception3:
    statement8
statement9;
```

Step1: Exception is thrown

Exception1

Step2: Exception is catched and statement3 is executed.

Step3: statement3 throws a new exception

Exception2

Step4: Except clauses of the outer try statement are inspected for a matching except. Exception2 is catched and statement8 is executed

Step5: statement9 is executed

`raise` Statement

- You can raise exceptions by using the **`raise`** statement.

- The syntax is as follows:

```
raise exceptionName(arguments)
```

raise Statement

```
def getRatios(vect1, vect2):
    """Assumes: vect1 and vect2 are equal length lists of numbers
       Returns: a list containing the meaningful values of vect1[i]/vect2[i]
"""
    ratios = []
    for index in range(len(vect1)):
        try:
            ratios.append(vect1[index]/vect2[index])
        except ZeroDivisionError:
            ratios.append(float('nan')) #nan = Not a Number
        except:
            raise ValueError('getRatios called with bad arguments')
     return ratios

try:
     print(getRatios([1.0, 2.0, 7.0, 6.0], [1.0,2.0,0.0,3.0]))
     print(getRatios([], []))
     print(getRatios([1.0, 2.0], [3.0]))
except ValueError as msg:
     print(msg)
```

```
[1.0, 1.0, nan, 2.0]
[]
getRatios called with bad arguments
```

`raise` Statement

- Avoid raising a generic **`Exception`**! To catch it, you'll have to catch all other more specific exceptions that subclass it..

```
def demo_bad_catch():
  try:
    raise ValueError('a hidden bug, do not catch this')
    raise Exception('This is the exception you expect to handle')
  except Exception as error:
    print('caught this error: ' + repr(error))

>>> demo_bad_catch()
caught this error: ValueError('a hidden bug, do not catch this',)
```

`raise` Statement

- and more specific catches won't catch the general exception:..

```
def demo_no_catch():
  try:
    raise Exception('general exceptions not caught by specific handling')
  except ValueError as e:
    print('we will not catch e')

>>> demo_no_catch()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in demo_no_catch
Exception: general exceptions not caught by specific handling
```

Custom Exceptions

- Users can define their own exception by creating a new class in Python.
- This exception class has to be derived, either directly or indirectly, from Exception class.
- Most of the built-in exceptions are also derived form this class.

Custom Exceptions

```
class ValueTooSmallError(Exception):
   """Raised when the input value is too small"""
   pass

class ValueTooLargeError(Exception):
   """Raised when the input value is too large"""
   pass

number = 10       # you need to guess this number

while True:
   try:
       i_num = int(input("Enter a number: "))
       if i_num < number:
           raise ValueTooSmallError
       elif i_num > number:
           raise ValueTooLargeError
       break
   except ValueTooSmallError:
       print("This value is too small, try again!")
   except ValueTooLargeError:
       print("This value is too large, try again!")

print("Congratulations! You guessed it correctly.")
```