Exceptions

Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

October 2, 2015

Overview

Exceptions

Errors detected during execution are called **exceptions**.
An exception is *raised* at the point where the error is detected:

- Raised by the python interpreter
- Raised by the code to signal exceptional situation (broken precondition)

Exceptions - example


```
>>> x=0
>>> print 10/x

Trace back (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print 10/x
ZeroDivisionError: integer division or modulo by zero
```

```
def rational_add(a1, a2, b1, b2):
    """
    Return the sum of two rational numbers.
    a1,a2,b1,b2 integer numbers, a2<>0 and b2<>0
    return a list with 2 int, representing a rational number a1/b2 + b1/b2
    Raise ValueError if the denominators are zero.
    """
    if a2 == 0 or b2 == 0:
        raise ValueError("0 denominator not allowed")
    c = [a1 * b2 + a2 * b1, a2 * b2]
    d = gcd(c[0], c[1])
    c[0] = c[0] / d
    c[1] = c[1] / d
    return c
```

Execution flow

Exceptions are a means of breaking out of the normal flow of execution in order to handle errors or other exceptional conditions.

```
def compute(a,b):
    print "compute :start "
    aux = a/b
    print "compute:after division"
    rez = aux*10
    print "compute: return"
    return rez

def main():
    print "main:start"
    a = 40
    b = 1
    c = compute(a, b)
    print "main:after compute"
    print "result:",c*c
    print "main:finish"
```

Exception handling

Is the process of handling error conditions in a program systematically by taking the necessary action.

```
try:
    #code that may raise exceptions
    pass
except ValueError:
    #code that handle the error
    pass
```

Exception handling

Exceptions need to be *handled* by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred otherwise the program will crash **raise**, **try-except** statements:

```
try:
      calc_add (int(m), int(n))
      printCurrent()
except ValueError:
      print ("Enter integers for m, n, with n!=0")
```

Exception handling


```
def f():
#    x = 1/0
    raise ValueError("Error Message")

try:
    f()
except ValueError as msg:
    print "handle value error:", msg
except KeyError:
    print "handle key error"
except:
    print "handle any other errors"
finally:
    print ("Clean-up code here")
```

Exception handling

- Multiple except cases
- Propagate information about the exception
- **finally** always runs (even when no exceptions)
- You can raise exception using ... **raise**

Exception handling

Only use exceptions to:

- Signal an exceptional situation - the function is unable to perform the promised situation
- Enforce preconditions

Do not use exception just to control the execution flow!

Function specification

Is a way for abstracting **functions** that will only work if we provide:

- Meaningful name for the function
- Short description of the function (the problem solved by the function)
- Type and meaning of each input parameter
- Conditions imposed over the input parameters (preconditions)
- Type and meaning of each output parameter
- Relation between the input and output parameters (post condition)
- **Exceptions** that the function may raise

Function specification

- **Precondition** - a condition that must be true just prior to the execution of some section of code.
- **Post condition** - a condition that must be true just after the execution of some section of code.

```
def gcd(a, b):
    """
    Return the greatest common divisor of two positive integers.
    a,b integer numbers
    return an integer number, the  greatest common divisor of a and b
    Raise ValueError if a<=0 or b<=0
    """
```

Test case for exceptions - example


```
def test_rational_add():
    """
      Test function for rational_add
    """
    assert rational_add(1, 2, 1, 3) == [5, 6]
    assert rational_add(1, 2, 1, 2) == [1, 1]
    try:
        rational_add(2, 0, 1, 2)
        assert False
    except ValueError:
        assert True
    try:
        rational_add(2, 3, 1, 0)
        assert False
    except ValueError:
        assert True
```

Test case for exceptions - example


```
def rational_add(a1, a2, b1, b2):
    """
    Return the sum of two rational numbers.
    a1,a2,b1,b2 integer numbers, a2<>0 and b2<>0
    return a list with 2 ints, representing a rational number a1/b2 + b1/b2
    Raise ValueError if the denominators are zero.
    """
    if a2 == 0 or b2 == 0:
        raise ValueError("0 denominator not allowed")
    c = [a1 * b2 + a2 * b1, a2 * b2]
    d = gcd(c[0], c[1])
    c[0] = c[0] / d
    c[1] = c[1] / d
    return c
```