Lecture 9

Exceptions and Assertions

Burkay Genç, Ahmet Selman Bozkır, and Selma Dilek

03/05/2023

PREVIOUS LECTURE

- testing and debugging
  - how to write tests to find bugs
  - how to find sources of bugs to fix them

TODAY

- exceptions
- assertions

EXCEPTIONS

EXCEPTIONS

- what happens when procedure execution hits an **unexpected condition**?
- get an **exception…** to what was expected
  - trying to access beyond list limits

```
test = [1,7,4]
test[4]                    -> IndexError
```

  - trying to convert an inappropriate type

```
int(test)                  -> TypeError
```

  - referencing a non-existing variable

```
a                          -> NameError
```

  - mixing data types without coercion

```
'a'/4                      -> TypeError
```

OTHER TYPES OF EXCEPTIONS

- already seen common error types:
  - `SyntaxError`: Python can't parse program
  - `NameError`: local or global name not found
  - `AttributeError`: attribute reference fails
  - `TypeError`: operand does not have correct type
  - `ValueError`: operand type okay, but value is illegal
  - `IOError`: IO system reports malfunction (e.g. file not found)

DEALING WITH EXCEPTIONS

- Python code can provide **handlers** for exceptions

```
try:
  a = int("burkay")
  b = int(5)
  print(a/b)
except:
  print("Bug in user input.")
```

```
## Bug in user input.
```

- exceptions **raised** by any statement in body of **try** are **handled** by the **except** statement and execution continues with the body of the **except** statement

CODE FLOW

- Code gets executed until exception occurs
  - after exception, except block starts executing

```
try:
  print(1)                  # (1) executes
  print("burkay")           # (2) executes
  print(int("genc"))        # (!) creates exception
  print(2)                  # (-) this is never executed
  print("genc")             # (-) this is never executed
except:
  print("Bug found!")       # (3) executes
  print("Please fix bug!")  # (4) executes
```

```
## 1
## burkay
## Bug found!
## Please fix bug!
```

CODE FLOW

- When exception occurs the rest of the try block is omitted
  - You shouldn't simply put everything into a try block!

```
try:
  for i in range(-2, 3):
    print("For i=",i, " 1/i=", 1/i)
except:
  print("Bug found!")
```

```
## For i= -2  1/i= -0.5
## For i= -1  1/i= -1.0
## Bug found!
```

```
for i in range(-2, 3):
  try:
    print("For i=",i, " 1/i=", 1/i)
  except:
    print("Bug found!")
```

```
## For i= -2  1/i= -0.5
## For i= -1  1/i= -1.0
## Bug found!
## For i= 1  1/i= 1.0
## For i= 2  1/i= 0.5
```

CATCHING A BUG

- You can use clever coding to pinpoint bugs

```
try:
  for i in range(-5, 5):
    for j in range(-5, 5):
      k = 2 * i + j
      l = 10 / k
except:
  print("Bug at i=", i, "and j=", j)
```

```
## Bug at i= -2 and j= 4
```

HANDLING SPECIFIC EXCEPTIONS

- have **separate except clauses** to deal with a particular type of exception

```
try:
  a = int(input("Tell me one number: "))
  b = int(input("Tell me another number: "))
  print("a/b = ", a/b)
  print("a+b = ", a+b)
except ValueError:
  print("Could not convert to a number.")
except ZeroDivisionError:
  print("Can't divide by zero")
except:
  print("Something went very wrong.")
```

OTHER EXCEPTIONS

- `else`:
  - body of this is executed when execution of associated `try` body **completes with no exceptions**
- `finally`:
  - body of this is `always executed` after `try, else` and `except` clauses, even if they raised another error or executed a `break, continue` or `return`
  - useful for clean-up code that should be run no matter what else happened (e.g. close a file)

BIG PICTURE

```
try:
  # First try to run the regular code
except:
  # If there is AN exception in try, then run this
else:
  # If there is NO exception in try, then run this
finally:
  # No matter what, at the end, run this
```

EXAMPLE

```
try:
  i = 3
  print(i / x)       # is x defined?
except:
  print("There is an error in the code.")
else:
  print("There is no error in the code.")
finally:
  print("procedure ended.")
```

```
## There is an error in the code.
## procedure ended.
```

VS.

```
try:
  i, x = 3, 4
  print(i / x)       # x is defined
except:
  print("There is an error in the code.")
else:
  print("There is no error in the code.")
finally:
  print("procedure ended.")
```

```
## 0.75
## There is no error in the code.
## procedure ended.
```

WHAT TO DO WITH ERRORS?

- what to do when encounter an error?
- **fail silently**:
  - substitute default values or just continue
  - bad idea! user gets no warning
- return an **"error" value**
  - what value to choose?
  - complicates code having to check for a special value
- stop execution, **signal error** condition
  - in Python: **raise an exception** `raise Exception("descriptive string")`

EXAMPLE

- write a function that takes two lists of numbers and returns a list of sum of items

```
def listSum(li1, li2):
  li3 = []
  for i in range(len(li1)):
    li3.append(li1[i] + li2[i])
  return li3

print(listSum([1,2,3], [4,5,6]))
```

```
## [5, 7, 9]
```

- can you see where this function can fail?

EXAMPLE

- write a function that takes two lists of numbers and returns a list of sum of items

```
def listSum(li1, li2):
  li3 = []
  for i in range(len(li1)):
    li3.append(li1[i] + li2[i])
  return li3

print(listSum([1,2,3,4,5], [1,2,3]))
```

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

EXAMPLE

- Solution 1: use the shorter length

```
def listSum(li1, li2):
  li3 = []
  for i in range(min(len(li1), len(li2))):
    li3.append(li1[i] + li2[i])
  return li3

print(listSum([1,2,3,4,5], [1,2,3]))
```

```
## [2, 4, 6]
```

EXAMPLE

- Solution 2: use try/except to control output

```
def listSum(li1, li2):
  li3 = []
  for i in range(max(len(li1), len(li2))):
    try:
      li3.append(li1[i] + li2[i])
    except:
      if i >= len(li1):
        li3.append(li2[i])
      else:
        li3.append(li1[i])

  return li3

print(listSum([1,2,3,4,5], [1,2,3]))
```

```
## [2, 4, 6, 4, 5]
```

EXAMPLE

- Solution 3: raise an exception at the beginning

```
def listSum(li1, li2):
  if (len(li1) != len(li2)):
    raise Exception("Length of list1 is different than length of list2")

  li3 = []
  for i in range(len(li1)):
    li3.append(li1[i] + li2[i])
  return li3

print(listSum([1,2,3,4,5], [1,2,3]))
```

EXAMPLE

- Solutions 1 and 2 makes it possible to continue execution
  - But they may cause unpredictable results
- Solution 3 stops execution
  - But it avoids confusing results

EXAMPLE

- assume we are **given a class list** for a subject: each entry is a list of two parts
  - a list of first and last name for a student
  - a list of grades on assignments

```
test_grades = [[['peter', 'parker'], [80.0, 70.0, 85.0]],
               [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

- create a **new class list**, with name, grades, and an average

```
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],
  [['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.666667]]
```

EXAMPLE

- example `class_list` object

```
[[['peter', 'parker'], [80.0, 70.0, 85.0]],
  [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

- possible solution

```
def get_stats(class_list):
  new_stats = []
  for elt in class_list:
    new_stats.append([elt[0], elt[1], avg(elt[1])])
  return new_stats

def avg(grades):
  return sum(grades)/len(grades)
```

ERROR IF NO GRADE

- if one or more students **don't have any grades**, we get an error

```
test_grades = [[['peter', 'parker'], [10.0, 5.0, 85.0]],
               [['bruce', 'wayne'], [10.0, 8.0, 74.0]],
               [['captain', 'america'], [8.0,10.0,96.0]],
               [['deadpool'], []]]
```

- get `ZeroDivisionError: float division by zero`
  - because of: `return sum(grades)/len(grades)`

OPTION 1: FLAG THE ERROR BY PRINTING A MESSAGE

- decide to **notify** that something went wrong with a message

```
def avg(grades):
  try:
    return sum(grades)/len(grades)
  except ZeroDivisionError:
    print('warning: no grades data')
```

- running on test data gives

```
get_stats(test_grades)
```

```
## warning: no grades data
## [[['peter', 'parker'], [10.0, 5.0, 85.0], 33.333333333333336], [['bruce', 'wayne'], [10.0, 8.0,
74.0], 30.666666666666668], [['captain', 'america'], [8.0, 10.0, 96.0], 38.0], [['deadpool'], [],
None]]
```

- `None`, because `avg()` didn't return anything in except block.

OPTION 2: CHANGE THE POLICY

- decide that a student with no grades gets a zero

```
def avg(grades):
  try:
    return sum(grades)/len(grades)
  except ZeroDivisionError:
    print('warning: no grades data')
    return 0.0
```

- running on test data gives

```
get_stats(test_grades)
```

```
## warning: no grades data
## [[['peter', 'parker'], [10.0, 5.0, 85.0], 33.333333333333336], [['bruce', 'wayne'], [10.0, 8.0,
74.0], 30.666666666666668], [['captain', 'america'], [8.0, 10.0, 96.0], 38.0], [['deadpool'], [], 0.0]]
```

- after warning, we return `0.0`
  - nicer solution

ASSERTIONS

ASSERTIONS

- want to be sure that **assumptions** on state of computation are as expected
- use an **assert statement** to raise an `AssertionError` exception if assumptions not met
- an example of good **defensive programming**

EXAMPLE

```
def avg(grades):
  assert len(grades) != 0, 'no grades data'
  return sum(grades)/len(grades)
```

- raises an **AssertionError** if it is given an empty list for grades
  - otherwise runs ok

```
## AssertionError: no grades data
```

ASSERTIONS AS DEFENSIVE PROGRAMMING

- assertions don't allow a programmer to control response to unexpected conditions
- ensure that **execution halts** whenever an expected condition is not met
- typically used to **check inputs** to functions, but can be used anywhere
- can be used to **check outputs** of a function to avoid propagating bad values
- can make it easier to locate a source of a bug

WHERE TO USE ASSERTIONS?

- goal is to spot bugs as soon as introduced and make clear where they happened
- use as a **supplement** to testing
- raise **exceptions** if users supplies **bad data input**
- use **assertions** to
  - check **types** of arguments or values
  - check that **invariants** on data structures are met
  - check **constraints** on return values
  - check for **violations** of constraints on procedure (e.g. no duplicates in a list)

Copyright Information

These slides are a direct adaptation of the slides used for MIT 6.0001 course present (as of February 2020) on MIT OCW web site.

Original work by:

Ana Bell, Eric Grimson, and John Guttag. 6.0001 Introduction to Computer Science and Programming in Python. Fall 2016. Massachusetts Institute of Technology: MIT OpenCourseWare. License: Creative Commons BY-NC-SA.

Adapted by and for:

Asst. Prof. Dr. Burkay Genç. MUH101 Introduction to Programming, Spring 2020. Hacettepe University, Computer Engineering Department.