EXCEPTIONS, ASSERTIONS

(download slides and .py files to follow along)

6.100L Lecture 13

Ana Bell
EXCEPTIONS
UNEXPECTED CONDITIONS

- What happens when procedure execution hits an unexpected condition?
- Get an exception... to what was expected
  - Trying to access beyond list limits
    \[
    \text{test} = [1, 7, 4]\\
    \text{test}[4] \\
    \Rightarrow \text{IndexError}
    \]
  - Trying to convert an inappropriate type
    \[
    \text{int(test)} \\
    \Rightarrow \text{TypeError}
    \]
  - Referencing a non-existing variable
    \[a\]
    \Rightarrow \text{NameError}
  - Mixing data types without coercion
    \[\text{'a'}/4\]
    \Rightarrow \text{TypeError}
HANDLING EXCEPTIONS

- Typically, exception causes an error to occur and execution to stop
- Python code can provide handlers for exceptions

```python
try:
    # do some potentially problematic code
except:
    # do something to handle the problem
```

- If expressions in try block all succeed
  - Evaluation continues with code after except block
- Exceptions raised by any statement in body of try are handled by the except statement
  - Execution continues with the body of the except statement
  - Then other expressions after that block of code

```python
if <all potentially problematic code succeeds>:
    # great, all that code just ran fine!
else:
    # do something to handle the problem
```
EXAMPLE with CODE YOU MIGHT HAVE ALREADY SEEN

- A function that sums digits in a string

CODE YOU’VE SEEN

```python
def sum_digits(s):
    """ s is a non-empty string containing digits.
    Returns sum of all chars that are digits """
    total = 0
    for char in s:
        if char in '0123456789':
            val = int(char)
            total += val
    return total
```

CODE WITH EXCEPTIONS

```python
def sum_digits(s):
    """ s is a non-empty string containing digits.
    Returns sum of all chars that are digits """
    total = 0
    for char in s:
        try:
            val = int(char)
            total += val
        except:
            print("can't convert", char)
    return total
```
USER INPUT CAN LEAD TO EXCEPTIONS

- User might input a character :(  
- User might make b be 0 :(  

```python
a = int(input("Tell me one number:"))
b = int(input("Tell me another number:"))
print(a/b)
```

- Use try/except around the problematic code

```python
try:
a = int(input("Tell me one number:"))
b = int(input("Tell me another number:"))
print(a/b)
except:
    print("Bug in user input.")
```
HANDLING SPECIFIC EXCEPTIONS

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

```python
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")
    print("a/b = infinity")
    print("a+b =", a+b)
except:
    print("Something went very wrong.")
```

only execute if these errors come up

for all other errors
OTHER BLOCKS ASSOCIATED WITH A TRY BLOCK

- **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)

- Nice to know these exist, but we don’t really use these in this class
WHAT TO DO WITH EXCEPTIONS?

- 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 ValueError(“something is wrong”)`
EXAMPLE with SOMETHING YOU’VE ALREADY SEEN

- A function that sums digits in a string
- Execution stopping means a bad result is not propagated

```python
def sum_digits(s):
    """ s is a non-empty string containing digits.
    Returns sum of all chars that are digits """
    total = 0
    for char in s:
        try:
            val = int(char)
            total += val
        except:
            raise ValueError("string contained a character")
    return total
```

Halt execution as soon as you see a non-digit with our own informative message. Does not go on to next char!
def pairwise_div(Lnum, Ldenom):
    """Lnum and Ldenom are non-empty lists of equal lengths containing numbers
    Returns a new list whose elements are the pairwise
    division of an element in Lnum by an element in Ldenom.
    Raise a ValueError if Ldenom contains 0. """
    # your code here

    # For example:
    L1 = [4,5,6]
    L2 = [1,2,3]
    # print(pairwise_div(L1, L2))    # prints [4.0,2.5,2.0]

    L1 = [4,5,6]
    L2 = [1,0,3]
    # print(pairwise_div(L1, L2))    # raises a ValueError
ASSERTIONS
ASSERTIONS: DEFENSIVE PROGRAMMING TOOL

- 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

```python
assert <statement that should be true>, "message if not true"
```

- An example of good **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
A function that sums digits in a **NON-EMPTY** string

Execution stopping means a bad result is not propagated

```python
def sum_digits(s):
    """ s is a non-empty string containing digits. Returns sum of all chars that are digits """
    assert len(s) != 0, "s is empty"
    total = 0
    for char in s:
        try:
            val = int(char)
            total += val
        except:
            raise ValueError("string contained a character")
```

Halt execution when specification is not met
YOU TRY IT!

def pairwise_div(Lnum, Ldenom):
    """ Lnum and Ldenom are non-empty lists of equal lengths containing numbers
    Returns a new list whose elements are the pairwise division of an element in Lnum by an element in Ldenom.
    Raise a ValueError if Ldenom contains 0. """
    # add an assert line here
ANOTHER EXAMPLE
LONGER EXAMPLE OF EXCEPTIONS and ASSERTIONS

- 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

```python
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 added at the end

```python
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333333333333],
 ['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.66666666666667]]
```
def get_stats(class_list):
    new_stats = []
    for stu in class_list:
        new_stats.append([stu[0], stu[1], avg(stu[1])])
    return new_stats

def avg(grades):
    return sum(grades)/len(grades)
ERROR IF NO GRADE FOR A STUDENT

- If one or more students don’t have any grades, get an error

```python
test_grades = [[["peter", "parker"], [10.0, 55.0, 85.0]],
                [["bruce", "wayne"], [10.0, 80.0, 75.0]],
                [["captain", "america"], [80.0, 10.0, 96.0]],
                [["deadpool"], []]]
```

- Get `ZeroDivisionError`: float division by zero because try to
  return `sum(grades)/len(grades)`
OPTION 1: FLAG THE ERROR BY PRINTING A MESSAGE

- Decide to **notify** that something went wrong with a msg

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

- Running on same test data gives

```python
[[['peter', 'parker'], [10.0, 55.0, 85.0], 50.0],
[['bruce', 'wayne'], [10.0, 80.0, 75.0], 55.0],
[['captain', 'america'], [80.0, 10.0, 96.0], 62.0],
[['deadpool'], [], None]]
```

flagged the error

because `avg` did not return anything in the except
OPTION 2: CHANGE THE POLICY

- Decide that a student with no grades gets a **zero**

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

- Running on same test data gives:

```
[[['peter', 'parker'], [10.0, 55.0, 85.0], 50.0],
 [['bruce', 'wayne'], [10.0, 80.0, 75.0], 55.0],
 [['captain', 'america'], [80.0, 10.0, 96.0], 62]
 [['deadpool'], [], 0.0]]
```

still flag the error
OPTION 3: HALT EXECUTION IF ASSERT IS NOT MET

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

- Raises an \texttt{AssertionError} if it is given an empty list for grades, prints out string message; stops execution
- Otherwise runs as normal
ASSERTIONS vs. EXCEPTIONS

- Goal is to **spot bugs as soon as introduced** and make clear where they happened

- Exceptions provide a way of **handling unexpected input**
  - Use when you don’t need to halt program execution
  - Raise exceptions if users supplies bad data input

- Use **assertions:**
  - Enforce conditions on a “contract” between a coder and a user
  - As a **supplement** to testing
  - 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)
6.100L Introduction to Computer Science and Programming Using Python
Fall 2022

For information about citing these materials or our Terms of Use, visit: https://ocw.mit.edu/terms.