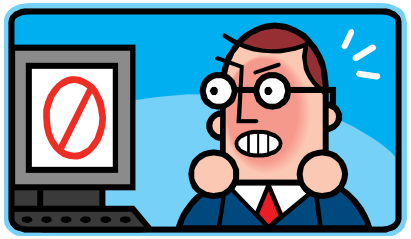
CS 1
Introduction to Computer Programming

Lecture 16: November 11, 2015

Exception Handling, part 1



Last time

- Classes and graphical objects
  - building a `Square` class
  - methods for scaling, moving, deleting squares

Today

- Exception handling
  - How to recover from errors
  - The `try`/`except` statement
  - Exception objects
  - Raising and catching exceptions
  - Catch-all exception handlers

Disclaimer

- This is not a "fun" lecture
- We won't see cool new ways to accomplish neat stuff in a few lines of code
- Instead, we will do two things:
  1. learn how to handle errors correctly
  2. understand better how Python works internally

Errors

- Many possible errors can occur while running a Python program
  - division by zero
  - accessing a non-existent index of a list
  - accessing a non-existent key in a dictionary
  - trying to store a new value into a tuple
  - trying to open a file (for reading) that does not exist
  - *etc.*

Errors

- How should errors be handled?
- Consider:

```
>>> a = 1 / 0
```

- This will not give a legal value
  - Python doesn't have infinitely large integers
- How should this error be handled?
  - Many possibilities...

Handling errors: 1st attempt

- Could ignore the error and just continue
  - Called *"silent failure"*
  - Problem: might cause the program to crash
    - (common in the C and JavaScript programming languages)
  - Not an effective way to proceed!
    - Don't know *why* error occurred or *where* it occurs

Handling errors: 2nd attempt

- Could halt the program
  - Advantage: prevents a crash
  - Problems:
    - Too drastic (some errors can be recovered from!)
    - Still don't know *why* error occurred or *where* it occurred

Handling errors: 3rd attempt

- Could print informative error message and halt the program
  - Advantages:
    - prevents a crash
    - know *why* error happened
  - Problems:
    - Too drastic (some errors can be recovered from!)
    - Still don't know *where* error occurred

Handling errors: 4th attempt

- Could print informative error message stating *why* and *where* the error occurred and halt the program
  - Advantages:
    - prevents a crash
    - know why and where error happened
  - Problems:
    - Too drastic (some errors can be recovered from!)
- This is what Python does by default

In Python

```
>>> a = 1 / 0
Traceback (most recent call first):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or
  modulo by zero
```

- Two components to this error message

In Python

```
>>> a = 1 / 0
Traceback (most recent call first):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or
  modulo by zero
```

- *Error message* states what *kind* of error occurred and more specifically *why* the error occurred in this particular case

In Python

```
>>> a = 1 / 0
Traceback (most recent call first):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or
  modulo by zero
```

- *Traceback* states *where* the error occurred
- We'll look into this more deeply next lecture

Error recovery

- Errors do not have to result in the termination of the entire program!
- Many kinds of errors can be recovered from
  - bad input → prompt for new input
  - nonexistent files → use a different file
  - *etc.*

Error recovery

- No general rule on how best to recover from errors
  - too specific to particular situation where error occurs
- Need a *general* mechanism
- It should say:
  - Under normal circumstances, do this
  - If error A happens, do A'
  - Else if error B happens, do B'
  - *etc.* until all possibilities are handled

Example

- Want to compute roots of a quadratic equation
- Solve this equation for x:

$a * x^2 + b * x + c = 0$

- Solution 1: $(-b + \text{sqrt}(b^2 - 4*a*c)) / (2 * a)$
- Solution 2: $(-b - \text{sqrt}(b^2 - 4*a*c)) / (2 * a)$
- *NOTE:* These solutions only work if a is not 0
- Let's write the Python code

Example

```
from math import sqrt

def solve_quadratic_equation(a, b, c):
    '''Solve the quadratic equation
    a*x**2 + b*x + c == 0 for x.'''
    sol1 = (-b + sqrt(b**2 - 4.0 * a * c)) / (2.0 * a)
    sol2 = (-b - sqrt(b**2 - 4.0 * a * c)) / (2.0 * a)
    return (sol1, sol2)
```

- Problem: What to do if `a == 0`?
- This makes `sol1` and `sol2` uncomputable (division by zero)

Example

- Could always use an `if` statement to check whether `a` was `0` before computing anything else
  - (Not a bad strategy)
  - However, usually `a` will not be `0`
  - Would rather have the code represent the typical case first, the exceptional cases only when the typical case fails
  - So we'll do this in a different way

`try` and `except`

- Python provides a special kind of statement for dealing with errors that can be recovered from: a `try`/`except` statement
- We'll show what this looks like in the context of our example, then explain it in detail
- Note that if a = 0, then the equation becomes

b * x + c = 0

- or:

x = -c / b (assuming b is nonzero)

Example using `try` and `except`

```
from math import sqrt

def solve_quadratic_equation(a, b, c):
  '''Solve the quadratic equation
  a*x**2 + b*x + c == 0 for x.'''
  try:
    sol1 = (-b + sqrt(b**2 - 4.0*a*c)) / (2.0*a)
    sol2 = (-b - sqrt(b**2 - 4.0*a*c)) / (2.0*a)
    return (sol1, sol2)
  except ZeroDivisionError:  # a == 0
    sol = -c / b   # assume b != 0
    return sol
```

`try` and `except`

- Structure of a `try`/`except` statement:

```
try:
   <some code which may result in an error>
except <name of the error>:
   <code to run if the error occurs>
```

- `try` and `except` say:
  - Try to execute this block of code (the `try` block)
  - If a particular error occurs, execute this other block of code (the `except` block)

`try` and `except`

- Structure of a `try`/`except` statement:

```
try:
   <some code which may result in an error>
except <name of the error>:
   <code to run if the error occurs>
```

- `try` and `except` are block-structured statements like `if`, `for`, and `while`
- Can have multiple statements in a `try` or `except` block, all indented the same

`try` and `except`

- Structure of a `try`/`except` statement:

```
try:
   <some code which may result in an error>
except <name of the error>:
   <code to run if the error occurs>
```

- OK to have a `try` block without `except` (though there's usually no point)
- Can't have `except` block without a preceding `try` block

`try` and `except`

- Structure of a `try`/`except` statement:

```
try:
   <some code which may result in an error>
except <error1>:
   <code to run if error1 occurs>
except <error2>:
   <code to run if error2 occurs>
```

- Can have multiple `except` blocks, each corresponding to a different kind of error

`try` and `except`

- Structure of a `try`/`except` statement:

```
try:
    <some code which may result in an error>
except:
    <code to run if any error occurs>
```

- Can have a "catch-all" `except` block, which will execute if *any* kind of error occurs
- Usually this is very poor design (not specific enough to the particular problem)

`try` and `except`

- Structure of a **`try`**/**`except`** statement:

```
try:
  <some code which may result in an error>
except <error1>:
  <code to run if error1 occurs>
except <error2>:
  <code to run if error2 occurs>
except:
  <code to run if any error occurs>
```

- But OK sometimes to use catch-all exception handler *after* more specific handlers

Terminology

- Errors that occur in code are called exceptions
  - They represent "exceptional conditions"
  - These don't *always* correspond to errors!
- Signaling an error is called raising an exception
  - or sometimes throwing an exception
- Handling an error is called catching an exception
- Therefore, exceptions are raised in `try` blocks and caught in `except` blocks

Exceptions

- Exceptions are actual Python objects
- They can have associated data
  - Often, an error message that indicates more precisely what went wrong
- Some examples of Python exceptions:
  - `ZeroDivisionError` (dividing by zero)
  - `IndexError` (list index out of bounds)
  - `IOError` (input/output error e.g. reading a non-existent file)
  - and many, many others

`try/except` again

- Let's look in detail at what happens when an exception is handled
- Sample function:

```
def print_reciprocal(x):
    try:
        recip = 1.0 / x
        print 'reciprocal of %g is %g' % \
          (x, recip)
    except ZeroDivisionError:
        print "Division by zero!"
```

try/except again

- Let's call this with `x = 5.0`

```
def print_reciprocal(x):
    try:
        recip = 1.0 / x
        print 'reciprocal of %g is %g' % \
          (x, recip)
    except ZeroDivisionError:
        print "Division by zero!"

>>> print_reciprocal(5.0)
```

`try/except` again

- Let's call this with `x = 5.0`

```
try:
    recip = 1.0 / x
    print 'reciprocal of %g is %g' % \
       (x, recip)
except ZeroDivisionError:
    print "Division by zero!"
```

`try/except` again

- Let's call this with `x = 5.0`

```
try:
    recip = 1.0 / x
    print 'reciprocal of %g is %g' % \
       (x, recip)
except ZeroDivisionError:
    print "Division by zero!"
```

try/except again

- Let's call this with `x = 5.0`

```
try:
    recip = 1.0 / 5.0
    print 'reciprocal of %g is %g' % \
      (x, recip)
except ZeroDivisionError:
    print "Division by zero!"
```

`try/except` again

- Let's call this with `x = 5.0`

```
try:
    recip = 0.2
    print 'reciprocal of %g is %g' % \
      (x, recip)
except ZeroDivisionError:
    print "Division by zero!"
```

`try/except` again

- Let's call this with `x = 5.0`

```
try:
    recip = 0.2
    print 'reciprocal of %g is %g' % \
      (5.0, 0.2)
except ZeroDivisionError:
    print "Division by zero!"
```

`try/except` again

- Let's call this with `x = 5.0`

```
try:
    recip = 0.2
    print 'reciprocal of %g is %g' % \
      (5.0, 0.2)
except ZeroDivisionError:
    print "Division by zero!"
```

- This prints:

```
reciprocal of 5 is 0.2
```

- and that's it (`except` block is never executed)

`try/except` again

- Let's call this with `x = 0.0`

```
def print_reciprocal(x):
    try:
        recip = 1.0 / x
        print 'reciprocal of %g is %g' % \
          (x, recip)
    except ZeroDivisionError:
        print "Division by zero!"

>>> print_reciprocal(0.0)
```

`try/except` again

- Let's call this with `x = 0.0`

```
try:
    recip = 1.0 / x
    print 'reciprocal of %g is %g' % \
       (x, recip)
except ZeroDivisionError:
    print "Division by zero!"
```

`try/except` again

- Let's call this with `x = 0.0`

```
try:
    recip = 1.0 / x
    print 'reciprocal of %g is %g' % \
      (x, recip)
except ZeroDivisionError:
    print "Division by zero!"
```

`try/except` again

- Let's call this with `x = 0.0`

```
try:
    recip = 1.0 / 0.0
    print 'reciprocal of %g is %g' % \
       (x, recip)
except ZeroDivisionError:
    print "Division by zero!"
```

try/except again

- Let's call this with `x = 0.0`

```
try:
    recip = 1.0 / 0.0
    print 'reciprocal of %g is %g' % \
      (x, recip)
except ZeroDivisionError:
    print "Division by zero!"
```

- Python can't compute `1.0 / 0.0`
- Instead, it raises a `ZeroDivisionError` exception at this point in the code

try/except again

- Let's call this with `x = 0.0`

```
try:
    recip = 1.0 / 0.0
    print 'reciprocal of %g is %g' % \
       (x, recip)
except ZeroDivisionError:
    print "Division by zero!"
```

- Python then jumps to the `except` block that handles `ZeroDivisionError` exceptions
- (skipping the `print` line entirely!)

`try/except` again

- Let's call this with `x = 0.0`

```
try:
    recip = 1.0 / 0.0
    print 'reciprocal of %g is %g' % \
       (x, recip)
except ZeroDivisionError:
    print "Division by zero!"
```

- Python then jumps to the `except` block that handles `ZeroDivisionError` exceptions
- (skipping the `print` line entirely!)

`try/except` again

- Let's call this with `x = 0.0`

```
try:
    recip = 1.0 / 0.0
    print 'reciprocal of %g is %g' % \
      (x, recip)
except ZeroDivisionError:
    print "Division by zero!"
```

- This line prints

```
Division by zero!
```

- and that's it (the exception has been handled)

Recapping

- As soon as an exception is raised in a `try` block, the program jumps out of the `try` block and looks for an `except` block that corresponds to that type of exception
- If it finds one, it executes the code in that block
- If not...?

Interlude

- More computer history!

Multiple `except` blocks

- A `try` block can be followed by multiple `except` blocks
  - much like an `if` block can be followed by multiple `elif`/`else` blocks
- Each `except` block should correspond to a different type of exception
- Can have a catch-all `except` block at the end (analogous to an `else` block in an `if` statement)
  - Though usually better to just leave this out

Example

```
values = [-1, 0, 1]
for i in range(5):
    try:
        r = 1.0 / values[i]
        print 'reciprocal of %g at %d is %g' % \
          (values[i], i, r)
    except IndexError:    # when i > 2
        print 'index %d out of range' % i
    except ZeroDivisionError:
        print 'division by zero'
```

Example: i == 0

```
values = [-1, 0, 1]
for i in range(5):
    try:
        r = 1.0 / values[i]
        print 'reciprocal of %g at %d is %g' % \
          (values[i], i, r)
    except IndexError:   # when i > 2
        print 'index %d out of range' % i
    except ZeroDivisionError:
        print 'division by zero'
```

execute this code

- This prints:

```
reciprocal of -1 at 0 is -1
```

Example: i == 1

```
values = [-1, 0, 1]
for i in range(5):
    try:
        r = 1.0 / values[i]
        print 'reciprocal of %g at %d is %g' % \
          (values[i], i, r)
    except IndexError:    # when i > 2
        print 'index %d out of range' % i
    except ZeroDivisionError:
        print 'division by zero'
```

execute this code

then execute this code

- This prints:

```
division by zero
```

Example: i == 3

```
values = [-1, 0, 1]
for i in range(5):
    try:
        r = 1.0 / values[i]
        print 'reciprocal of %g at %d is %g' % \
          (values[i], i, r)
    except IndexError:    # when i > 2
        print 'index %d out of range' % i
    except ZeroDivisionError:
        print 'division by zero'
```

execute this code

then execute this code

- This prints:

```
index 3 out of range
```

Note:

- When evaluating a `try`/`except` statement, only *one* `except` statement's code can be executed
- Even if there is a catch-all `except` block at the end, it's not executed if a previous `except` block's code was executed
- To illustrate, let's change the previous code a bit
  - add a catch-all `except` block

Example: i == 3

```
values = [-1, 0, 1]
for i in range(5):
    try:
        r = 1.0 / values[i]
        print 'reciprocal of %g at %d is %g' % \
          (values[i], i, r)
    except IndexError:    # when i > 2
        print 'index %d out of range' % i
    except ZeroDivisionError:
        print 'division by zero'
    except:
        print 'some other exception happened'
```

Catch-all **except** block

Note:

- In previous example:
  - if an `IndexError` exception occurs, only the code in the `except IndexError:` block gets executed
  - if a `ZeroDivisionError` exception occurs, only the code in the `except ZeroDivisionError:` block gets executed
  - if some other exception occurs, only the code in the catch-all `except:` block gets executed
- Different `except` blocks are mutually exclusive!

Final example

- When a file is opened for reading, but the file does not exist, an `IOError` exception is raised
- This is very common when *e.g.* a user inputs the wrong file name
- How can we handle this in our code?
- Assume our files contain numbers (one per line)
- We want to add them all up, print their sum

First attempt

```
def sum_numbers_in_file(filename):
    sum_nums = 0
    file = open(filename, 'r')
    for line in file:
        line = line.strip()
        sum_nums += int(line)
    file.close()
    return sum_nums
```

- OK as long as file with given name exists
- If not, get an **`IOError`** which halts the program
- Can we do better?

Second attempt

```
def sum_numbers_in_file(filename):
    sum_nums = 0
    found_file = False
    while not found_file:
        try:
            file = open(filename, 'r')
            found_file = True
        except IOError:
            print 'File not found!'
            filename = \
              raw_input('Enter another filename: ')
    for line in file: ... # (as before)
```

Second attempt

```
found_file = False
while not found_file:
    try:
        file = open(filename, 'r')
        found_file = True
    except IOError:
        print 'File not found!'
        filename = \
          raw_input('Enter another filename: ')
```

- **`open`** will raise an ***`IOError`*** if the file isn't found
- Code in the **`except`** block will be executed next
  - user prompted for new filename, then **`try`** again

Second attempt

```
found_file = False
while not found_file:
    try:
        file = open(filename, 'r')
        found_file = True
    except IOError:
        print 'File not found!'
        filename = \
          raw_input('Enter another filename: ')
```

- If `open` succeeds, no exception will be raised
- `file` will be a file object, `found_file` will be set to `True`, and the `while` loop will terminate

Second attempt

- This approach is still pretty ugly
- What if you had a very long list of filenames and wanted to sum all the numbers in all the files?
- Not practical to prompt user every time a file doesn't exist if there are lots of filenames
- Would be nice if could do error handling *outside* of this function
- This is the subject of the next lecture

Friday lecture

- "How to think about debugging"
- Tips, approaches, ideas to make debugging easier and more productive

Monday

- More exception handling
  - Catching an exception outside of the function it was raised in
- The runtime stack
  - and what a "traceback" is all about