CSC148H Lecture 2

Dan Zingaro
OISE/UT

September 15, 2008

Testing with Nose

- Nose: unit testing framework for Python
- Searches through a directory for tests, and runs them
- If a test is in a Python class, it instantiates the class and runs the test
- Nose runs functions that have the word **test** at the beginning of the name or at a word boundary
- e.g. **testPush**, **fun_test**
- For this course: use it to automatically test your code and report on successes and failures

Testing with Nose: Example

```
import nose
from stack import *

def testPush():
  s = Stack()
  s.push (8)
  assert s.size() == 1, \
    'Stack size is ' + str(s.size()) + '; should be 1!'

nose.runmodule()
```

How can we Handle Errors?

- The C way: check return value from function call
- But what if all return values are valid? How would we know if the value -1 means error or not?
- What if error is rare? We still have to pollute our code with checks
- What if more than one error can occur from the same source?
- Do we know what to do when an error happens? (i.e. library code)
- The Python (and Java) way: exception objects

What are Exceptions?

- Exceptions allow you to structure code in a more natural way so that error handling and recovery is isolated from the regular flow of your program
- An exception is an object that indicates an exceptional situation (not necessarily a problem)
- An exception gets raised during program execution and interrupts regular program flow
- Exceptions must be handled somewhere along the line, or your program will crash

Examples of Exceptions

```
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
```

Handling Exceptions

- If a piece of code can raise an exception, you can put it in a **try** block
- If there is an associated **except** block that matches the kind of exception raised, it is handled by this block
- There may be more than one **except** block that matches; in this case, the first is used
- An **except** block "matches" a raised exception if the **except** block names the same class or a superclass

Example

```
import sys
try:
  f = open('myfile.txt')
  s = f.readline()
  i = int(s.strip())
except IOError:
  print "Input/Output error."
except ValueError:
  print "Could not convert data to an integer."
```

- Four execution paths: no exception, `IOError`, `ValueError`, unhandled exception

Execution with Exceptions

- If no exception occurs, no **except** block is executed
- If an exception occurs and an **except** block handles it, execution continues following the enclosing **try** block
- If an **except** block is present that does not specify the type of exception it handles, it will handle any exception (dangerous!)
- If an exception occurs and no **except** block handles it, the exception propagates until it is handled

Where is This Handled?

```
def badStuff():
  return 2 / 0

def callBad():
  try:
    badStuff()
  except ValueError:
    print "Caught exception in callBad!"

def entry():
  try:
    callBad()
  except ZeroDivisionError:
    print "Caught exception in entry"
```

The call stack is unwound, canceling functions until one of them can handle it.

Slight Change: Now Where?

```
def badStuff():
  return 2 / 0

def callBad():
  try:
    badStuff()
  except ArithmeticError:
    print "Caught exception in callBad!"

def entry():
  try:
    callBad()
  except ZeroDivisionError:
    print "Caught exception in entry"
```

Remember: **except** clauses catch subclasses (and **ArithmeticError** is a superclass of **ZeroDivisionError**)

Other Features of Exceptions

- You can name multiple exceptions as a tuple in an except clause
- You can use the **except** keyword without naming any exceptions, but it must be the last **except** listed (why?)
- After any **except** clauses, you can add a **finally** clause that always executes, regardless of whether an exception was raised and regardless of whether any raised exceptions were handled
- These **finally** blocks are often used for performing cleanup actions that must always occur

Exception Variables

You can get access to the exception object when handling an exception by adding it after the exception in the **except** statement:

```
try:
  ... do stuff
except ZeroDivisionError, detail:
  print 'Handling exception: ', detail
```

User-Defined Exceptions

- A user-defined exception has Exception as a superclass
- Can define it to have any instance variables or methods you want, just like a regular class
- Should include a `__str__` method

```
class MyException(Exception):
  def __init__(self, value):
    self.value = value
  def __str__(self):
    return str(self.value)
```

Raising Exceptions

Use the raise keyword to generate exceptions:

```
>>> raise NameError, 'my error message'
Traceback (most recent call last):
File "<string>", line 1, in <string>
NameError: my error message
```

Reraising Exceptions

You can use the **raise** keyword to reraise exceptions that you've already caught and (perhaps only partially) handled:

```
try:
  x = 1/0
except ZeroDivisionError, detail:
  print 'Caught the following error: ', detail
  raise
```

Exceptions: Strings or Objects?

- Python previously use strings to carry exception information instead of instances of a class
- Example: `raise "serious problem!"`
- You can optionally include an object to pass along with the string
- This will give you a deprecation warning in new Python versions
- TPS: But why objects? What are the benefits over strings? Drawbacks?

Uses of Exceptions

Now that we understand the exception execution model, let's discuss how exceptions can be used for (Lutz, 2007):

- Error Handling
- Event notification
- Special-case handling
- Termination actions
- Unusual control flow

Object-Oriented Programming

- The tetralogy of object-oriented programming (Meyer, 1997):
  - Dynamic binding
  - Redefinition
  - Polymorphism
  - Static typing
- What do these mean in a dynamically typed language like Python?

Object-Oriented Programming...

- Dynamic binding: the method called is based on the actual type of an object, not on a variable type declaration
- But in Python, we have no type declarations anyway, so this distinction goes away
- We also get polymorphism "for free" because we can store arbitrary objects in aggregates
- Redefinition is accomplished in Python through inheritance (just like in Java)

Dynamic Typing

- In Python, types of variables are not declared
- This is different than how it works in C, C++, Java, Pascal, etc.
- Python is still strongly typed, however, since the objects associated with variables do have types
- This prevents you from doing stuff like adding an integer to a string
- A variable name refers to an object at runtime, so it can be associated with objects of different types at different times

Inheritance

- Open-closed principle: we want code modules to be closed (so we can use them in our programs), but also open (so we can extend them to suit new purposes)
- This is a paradox for traditional modules: only way to change a module is to change its source code
- But what if we don't have access to the code? What if we break other parts of the program relying on its original behavior?
- With inheritance, base classes remain untouched, yet the class remains open because we can use inheritance to change its behavior
- Inheritance also lets us avoid code duplication by taking advantage of existing commonalities

Inheritance: Textbook Example

- Let's model different types of geometric shapes for a graphics library
- Points, segments, ellipses, triangles, rectangles, squares
- Consider a class **polygon** that includes a method for finding the perimeter (among other methods)
- Finding the perimeter in the general case requires adding the distance between pairs of adjacent vertices
- But to find the perimeter of a square, we have a more efficient method (how?)
- Technique: inherit **square** from **polygon**, redefining **perimeter** to take advantage of properties of a square

Let's Talk About Inheritance

TPS: is the following a valid use of inheritance?

- We are modeling cars and people who drive cars
- We define classes: **car** and **person**, inheriting **person** from **car**