Exception Handling

Genome 559

Review - classes

Use your own classes to:

- package together related data
- conceptually organize your code
- force a user to conform to your expectations

Class constructor:

```
class MyClass:
   def __init__(self, arg1, arg2):
      self.var1 = arg1
      self.var2 = arg2
foo = MyClass('student', 'teacher')
```

Exception Handling

Sometimes you want your code to handle errors "gracefully", e.g. providing useful feedback.

Best approach is called exception handling (or error handling).

Especially useful for anything you will give to someone else to use (finished program or modules etc).

Example: command line arguments

```
import sys

intval = int(sys.argv[1])
```

How could you check that the user entered a valid argument?

```
import sys

try:
   intval = int(sys.argv[1])
except:
   print "first argument could not be parsed as an int value"
   sys.exit()
```

two new reserved key words - `try` and `except`

You can put `try-except` clauses anywhere.

Python provides several types of exceptions (each of which is of course a class!). Some common exception classes:

```
ZeroDivisionError # when you try to divide by zero
NameError # when a variable name can't be found
MemoryError # when program runs out of memory
ValueError # when int() or float() can't parse a value
IndexError # when a list or string index is out of range
KeyError # when a dictionary key isn't found
ImportError # when a module import fails
SyntaxError # when the code syntax is uninterpretable
```

You have seen most of these already when you had bugs in your code.

(note - each of these is actually an extension of the base Exception class - any code shared by all of them can be written once for the Exception class!)

Example - enforcing format in the `Date` class

```
class Date:
   def __init__(self, day, month, year):
      try:
         self.day = int(day)
      except ValueError:
         print 'Date constructor: day must be an int value'
      try:
         self.month = int(month)
      except ValueError:
         print 'Date constructor: month must be an int value'
      try:
         self.year = int(year)
      except ValueError:
         print 'Date constructor: year must be an int value'
```

indicates only catches this type of exception

FYI, if there are other types of exceptions, they will be reported by the default Python exception handler, with output you are very familiar with by now, e.g.:

```
Traceback (most recent call last):
  File <pathname>, line X, in <module>
    <code line>
<default exception report>
```

You may even want to force a program exit with information about the offending line of code:

```
import traceback
import sys

class Date:
   def __init__(self, day, month, year):
      try:
         self.day = int(day)
      except ValueError:
         print 'Date constructor: day must be an int value'
         traceback.print_exc()
         sys.exit()
```

special traceback function that prints other information for the exception (e.g. code line number)

Exceptions - when to use

- Any software that will be given to someone else, especially if they don't know Python.

- Private software that is complex enough to warrant.

- As with code comments, exceptions are a useful way of reminding yourself of what the program expects.

- They have NO computational cost (if no exception is thrown, nothing at all is computed).

Imagine some poor schmuck's frustration when they try to use your program:

```
import sys
val = int(sys.argv[1])
```

```
> parse_int.py hello
Traceback (most recent call last):
  File "C:\Documents and Settings\jht\My Documents\parse_int.py", line 3, in <module>
    val = int(sys.argv[1])
ValueError: invalid literal for int() with base 10: 'hello'
```

what the @!#&$!

by the way, notice that the ValueError is an exception class

```
import sys
try:
   val = int(sys.argv[1])
except ValueError:
   print "first argument '" + sys.argv[1] + "' is not a valid integer"
except IndexError:
   print "one integer argument required"
```

```
> parse_int.py
one integer argument required
> parse_int.py hello
first argument 'hello' is not a valid integer
```

hey, nice feedback!

Exercise 1

Write a program check_args.py that gets two command line arguments and checks that the first represents a valid int number and that the second represents a valid float number. Make useful feedback if they are not.

```
> python check_args.py 3 help!
'help!' is not a valid second argument, expected a
float value
> python check_args.py I_need_somebody 3.756453
'I_need_somebody' is not a valid first argument,
expected an int value
```

```
import sys

try:
   arg1 = int(sys.argv[1])
except ValueError:
   print "'sys.argv[1]' is not a valid first argument, \
expected an int value"
   sys.exit()
try:
   arg2 = float(sys.argv[2])
except ValueError:
   print "'sys.argv[2]' is not a valid second argument, \
expected a float value"
   sys.exit()

<do something with the arguments>
```

FYI - it is fine to nest try-except clauses:

```
import sys

try:
   try:
      arg1 = int(sys.argv[1])
   except ValueError:
      print "'sys.argv[1]' is not a valid first argument, \
expected an int value"
      sys.exit()
   try:
      arg2 = float(sys.argv[2])
   except ValueError:
      print "'sys.argv[2]' is not a valid second argument, \
expected a float value"
      sys.exit()
except:
   print 'something was wrong with arguments, expected two'
   sys.exit()
```

Exercise 2

Write a program that tries to read a file corresponding to the first command-line argument. Provide useful feedback if the file doesn't exist or anything goes wrong reading the file.

Hint - you can just use myFile.read() to read the file contents - in a real program you would do something with the file contents of course.

```
import sys

try:
   openFile = open(sys.argv[1])
   fileText = openFile.read()
except:
   print 'file not found or error reading file'
```

Challenge Exercise

Rewrite one of the several functions you have written, either using a direct test or exception handling to provide feedback if the function has been given an inappropriate argument.

e.g.

```
# expects a list of int or float numbers and a float or int
# value for the increment
def incrementValues(numberList, incVal):
   if type(numberList) is not list:
      print "function requires a list to increment"
      return None
   if (type(incVal) is not int) and (type(incVal) is not float):
      print "function requires an int or float for incrementing"
   for i in range(0, len(numberList)):
      try:
         numberList[i] += incVal
      except:
         print "list value is not a number"
         return None
```

You may not have learned the built-in `type` function - it returns the type of an object