CS1110
Lecture 24: Exceptions and Try-statements

Announcements
Today: 14.5, A.2.3
Next time: 17, 18 (especially 18.7). Our graphical notation will (once again) differ significantly from the book.

Readings
Slides by D. Gries, L. Lee, S. Marschner, W. White

1. When things go wrong (in Python)

Q1: What happens when an error causes a crash?

TypeError: unsupported operand type(s) for +: 'int' and 'list'

Understanding this helps you debug.

Q2: Can we use "problem-signalling" to handle unusual situations more smoothly?

Understanding this helps you write more flexible code.

It is sometimes better to warn and re-prompt the user than to have the program crash (even if the user didn't follow your exquisitely clear directions or preconditions).

2. (Runtime) errors are exception objects

When various bad things happen, Python creates an exception object.
If that object is not otherwise "handled", the system halts, printing the stack trace and info about the exception object.

ZeroDivisionError: integer division or modulo by zero

name of the type of the exception object

.string kept in the exception object

3. Recovering from errors: Try-except

Try-except blocks allow us to recover from errors
- Do the code that is in the try-block
- If an error occurs, jump to the except-block (skip it o.w.)

```python
def recip(x):
    """Return 1.0/x, or inf if x is 0. Pre: x is a number""
    try:
        return 1.0/x
    except:
        return float('Inf')
```

executes if an error occurs

4. Hierarchy of exceptions

http://docs.python.org/library/exceptions.html

5. Recovering from specific error types

You can have except-blocks that are executed only if the exception is an instance of a particular class.

```python
def recip(x):
    """Return 1.0/x, or inf if x is 0""
    try:
        return 1.0/x
    except ZeroDivisionError:
        return float('Inf')
```
**Template: Handling user-input problems**

```python
def recip4():
    '''Return reciprocal of user input (we don't handle 0)'''
    prompt = "Pick a non-zero number: ",
    n = float(input(prompt))
    return 1.0/n
```

**Creating exceptions: raise**

You can signal errors by creating exceptions with `raise`.

```python
def speed(x):
    if x > 3e8:
        raise ValueError("speed: input > light speed")
```

**Creating Your Own Class of Exceptions**

```python
class SpeedError(StandardError):
    '''An instance signals violation of a speed constraint.''
    pass
```

What's in parentheses is what you declare the `parent` class of the new class to be. Thus, all SpeedErrors are also StandardErrors, and inherit their characteristics:

```python
except StandardError:
    print 'Something is wrong, but proceeding anyway'
```

**Try-except vs. if-statements or asserts**

Rules of thumb:

For simple tests and "normal" situations, if-thens are usually better.

For precondition violations, asserts are more readable. (Note: asserts raise AssertionsErrors.)

For more "abnormal" situations, try-excepts are better.

There are some canonical try-except idioms, such as processing malformed user input (which we just saw).