CS1110

Lecture 24: **Exceptions and Try-statements**

Announcements

Readings

Today: 14.5, A.2.3

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

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

When things go wrong (in Python)

Q1: What happens when an error causes a crash?

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

IndexError: list index out of range

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).

(Runtime) errors are exception objects

When various bad things happen, Python creates an *exception* object.

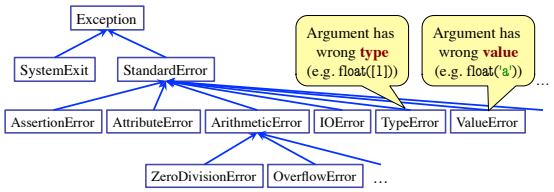
If that object is not otherwise "handled", the system halts, printing the stack trace and info about the exception object.

ZeroDivisionError: integer division or modulo by zero

name of the type of the exception object

string kept in the exception object

Hierarchy of exceptions



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

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.)

```
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

Recovering from specific error types

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

```
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

```
def recip4():
    """Return reciprocal of user input (we don't handle 0)"""
    prompt = "Pick a non-zero number: "
    while True:
        try:
            n = float(raw_input(prompt))
            return 1.0/n
        except ZeroDivisionError:
            print 'The number has to be non-zero; please try again.'
        except ValueError:
            print 'The input has to be a number; please try again.'
```

The only escape is if valid input is given in the try block, so the that return statement succeeds.

Creating exceptions: raise

You can signal errors by creating exceptions with raise.

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

As usual, creates a new object.

The type is informative to the user (or enclosing except-blocks)

You can choose an informative output message

Creating Your Own Class of Exceptions

```
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:

```
...
except StandardError:
    print 'Something is wrong, but proceeding anyway'
    # a SpeedError will trigger this except clause
```

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 AssertionErrors.)

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).