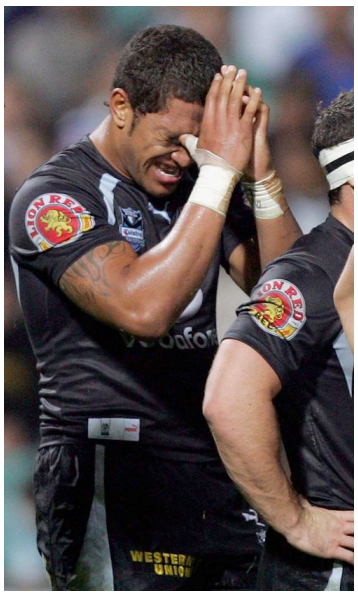
math 480 -- april 7, 2008

Errors and Exceptions: Recovering from problems gracefully



Reference: Chapter 8 of the Python Tutorial

Syntax Errors

```
while True print 'Hello world'
```

```
    Syntax Error:
        while True print 'Hello world'
```

Exceptions

Python has excellent very fast support for exception handling, which makes writing code much much cleaner. Here is an example. The *last line* of the error message indicates what happened!

```
10 * (1/0)
```

This page http://docs.python.org/lib/module-exceptions.html lists all the standard builtin exceptions along with what each means. Some common exceptions that appear in mathematical programming include

1. TypeError
2. ZeroDivisionError
3. ArithmeticError
4. ValueError
5. RuntimeError
6. NotImplementedError
7. OverflowError
8. IndexError

```
''.join([1,2])
```

```
Exception (click to the left for traceback):
...
TypeError: sequence item 0: expected string, sage.rings.integer.Int
```

```
1/0
```

```
Exception (click to the left for traceback):
...
ZeroDivisionError: Rational division by zero
```

```
factor(0)
```

```
Exception (click to the left for traceback):
...
ArithmeticError: Prime factorization of 0 not defined.
```

```
CRT(2, 1, 3, 3)
```

```
Exception (click to the left for traceback):
...
ValueError: arguments a and b must be coprime
```

```
find_root(SR(1), 0, 5)
```

```
Exception (click to the left for traceback):
...
RuntimeError: no zero in the interval, since constant expression is
```

```
brun.str(50)
```

```
Exception (click to the left for traceback):
...
NotImplementedError: Brun's constant only available up to 41 bits
```

```
float(5)^float(902830982304982)
```

```
Exception (click to the left for traceback):
...
OverflowError: (34, 'Result too large')
```

```
v = [1,2,3]
v[10]
```

```
Exception (click to the left for traceback):
...
IndexError: list index out of range
```

Handling Exceptions

You can often **handle** specific exceptions by doing a specific action only if the exception occurs.

```
try:
    1/0
except ZeroDivisionError:
    print "A zero division error occured."
```

```
A zero division error occured.
```

The `ArithmeticError` exception is the base class for those built-in exceptions that are raised for various arithmetic errors: OverflowError, ZeroDivisionError, FloatingPointError. Thus the following also works.

```
try:
    1/0
except ArithmeticError:
    print "something went wrong"
```

```
something went wrong
```

```
try:
    1/0
except RuntimeError:
    print "a runtime error!"
```

```
Exception (click to the left for traceback):        print "a runti
...
ZeroDivisionError: Rational division by zero
```

```
```

```
try:
   a = 5
   b = 0
   c = a/b
   d = 10
except RuntimeError:
   print "a runtime error"
else:
   print "Everything ran fine!"  # only executed if no exception
raised.
finally:
   print "This always gets executed."
```

```
This always gets executed.
Traceback (most recent call last):    except RuntimeError:
  File "element.pyx", line 1480, in
sage.structure.element.RingElement.__div__
  File "coerce.pxi", line 138, in sage.structure.element._div_c
  File "integer.pyx", line 1095, in
sage.rings.integer.Integer._div_c_impl
  File "integer_ring.pyx", line 204, in
sage.rings.integer_ring.IntegerRing_class._div
ZeroDivisionError: Rational division by zero
```

```
try:
   a = 5
   b = 1
   c = a/b
   d = 10
except RuntimeError:
   print "a runtime error"
else:
   print "Everything ran fine!"  # only executed if no exception
raised.
finally:
   print "This always gets executed."
```

```
Everything ran fine!
This always gets executed.
```

```
```

This is an example use of exception handling in a function. We make a division function that returns +infinity instead upon division by 0.

```
def mydiv(a,b):
    return a/b
```

```
mydiv(2, 0)
```

```
Exception (click to the left for traceback):
...
ZeroDivisionError: Rational division by zero
```

```
def mydiv(a,b):
    try:
        return a/b
    except ZeroDivisionError:
        print "You tried to divide by 0, but that's OK, I'll give
you infinity back."
        return infinity
```

```
mydiv(2,3)
```

```
2/3
```

```
mydiv(2,0)
```

```
You tried to divide by 0, but that's OK, I'll give you infinity
back.
+Infinity
```

```
```

Raising exceptions

Use the `raise` keyword to raise an exception.

```
def mydiv(a,b):
    if b == 0:
        raise ZeroDivisionError, "Oops -- you can't divide by 0"
    return a/b
```

```
mydiv(2,3)
```

```
2/3
```

```
mydiv(2,0)
```

```
Exception (click to the left for traceback):
...
ZeroDivisionError: Oops -- you can't divide by 0
```

```
try:
    mydiv(2,0)
except ZeroDivisionError, msg:
    print "an error occured"
    print "error: ", msg
```

```
an error occured
error:  Oops -- you can't divide by 0
```

```
```

WARNING: Handling multiple exceptions!

We define a function that can raise three different types of exceptions.

```
def mydiv(a,b):
    if b == 0:
        raise ZeroDivisionError, "Oops -- you can't divide by 0"
    if a == 0:
        raise NotImplementedError, "dividing 0 by something is too
difficult!"
    if a == b:
        raise ValueError, "dividing equal things not allowed for
no good reason"
    return a/b
```

This is a *very* common and painful mistake people (=me many many times) make:

The code looks fine. What is wrong?

```
try:
    mydiv(0,4)
except NotImplementedError, ZeroDivisionError:
    print "An error occured"
```

```
An error occured
```

```
ZeroDivisionError
```

```
NotImplementedError('dividing 0 by something is too difficult!',)
```

```
reset('ZeroDivisionError')   # reset to default state at startup.
```

Instead give a tuple of different exception types, and catch the message as the second output:

```
try:
    mydiv(2,0)
except (NotImplementedError, ZeroDivisionError, ValueError), msg:
    print "An error occured:", msg
```

```
An error occured: Oops -- you can't divide by 0
```

Classes: Defining your own new data types

Reference: Chapter 9 of the Python Tutorial

The Python class construction allows you to define your own new data types. It is modeled on C++ classes, though Python classes are simpler and easier to use. They support both single and multiple inheritance and one can derive from builtin classes.

Defining a new class

```
# You can define any new class you want very easily at any point,
even inside the body of a function, etc.
# It's very nice. Here are some examples.
```

```
class NaturalNumber:
    pass
```

```
# A class itself is a Python object, just like anything else

print NaturalNumber
```

```
__main__.NaturalNumber
```

```
type(NaturalNumber)
```

```
<type 'classobj'>
```

```
# This is how to make instances of a class
n = NaturalNumber()
```

```
n
```

```
<__main__.NaturalNumber instance at 0x834ae68>
```

```
type(n)
```

```
<type 'instance'>
```

```
# The above class is very boring.  Let's add printing capabilities
and a value
class NaturalNumber:
    def __init__(self, n):
        self.__n = n
    def __repr__(self):
        return str(self.__n)
```

```
n = NaturalNumber(5)
```

```
n
```

```
5
```

```
# Lets add a little error handling and a set function.
class NaturalNumber:
    def __init__(self, n):
        if n < 0:
            raise ValueError, "n must be nonnegative"
        self.__n = n

    def number(self):
        return self.__n

    def __repr__(self):
        return str(self.__n)

    def set(self):
        # The set corr. to n is {n-1} union n-1; this is how the
integers
        # are built up using set theory in Axiom Set Theory.
        if self.__n > 0:
            z = NaturalNumber(self.__n-1).set()
            return Set([z]).union(z)
        return Set([]) # empty set
```

```
n = NaturalNumber(-1)
```

```
Exception (click to the left for traceback):
...
ValueError: n must be nonnegative
```

```
n = NaturalNumber(3); n
```

```
3
```

```
n.set()
```

```
{{{{}}, {}}, {{}}, {}}
```

```
for n in [0..4]:
    print n, NaturalNumber(n).set()
```

```
0 {}
1 {{}}
2 {{{}}, {}}
3 {{{{}}, {}}, {{}}, {}}
4 {{{{}}, {}}, {{}}, {}, {{{{}}, {}}, {{}}, {}}}
```

```

```

Single Inheritance

```
# In Python a class B inherits from another class A by simply
putting the A in parenthesis.
# This makes all the methods of class A available for instances of
B.
# However, if methods are defined in B with the same name as
methods in A
```

```
class PositiveNatural(NaturalNumber):
     def __init__(self, n):
         if n <= 0:
             raise ValueError, "n must be positive"
         # Call the base class constructor
         NaturalNumber.__init__(self, n)
     def inverse(self): # a new function
         return 1/self.number()
     def set(self):     # refine function from base class
         return 'we redefined set'
```

```
n = PositiveNatural(0)
```

```
Exception (click to the left for traceback):
...
ValueError: n must be positive
```

```
n = PositiveNatural(3)
n
```

```
3
```

```
n.set()
```

```
'we redefined set'
```

```
n.inverse()
```

```
1/3
```

```
# IMPORTANT: instances of derived class should always satisfy an
"is a" relationship, are you
# are doing something seriously wrong.
```

```
```

Multiple Inheritance

```
# We can also list several class to derive from.
```

```
class PositiveNatural(NaturalNumber, Rational):
     def __init__(self, n):
         if n <= 0:
             raise ValueError, "n must be positive"
         # Call the base class constructor
         NaturalNumber.__init__(self, n)
         Rational.__init__(self, n)
     def inverse(self):
         "Return the inverse of this positive natural number"
         return 1/self.number()
```

```
n = PositiveNatural(10); n
```

```
10
```

```
n.inverse()
```

```
1/10
```

```
n.factor()
```

```
2 * 5
```

```
# Method resolution order.
#   Check to see whether the first (left-most) class defines the
function; if so, use it.
#   If not, try the next class.
```

```
class X:
    def foo(self):
        print "X"
class A(X):
    pass
class B:
    def foo(self):
        print "B"
class C(A,B):
    pass

c = C()
c.foo()
```

```
X
```

```
# Class corresponding to the mathematical objects you are working
with,
# e.g., a Matrix class for matrices, a DifferentialEquations class
for
# differential equations, etc.
# This works very very nicely for expressing mathematics, and is
# much different and conceptually superior to
# what you get in Mathematica and Matlab
```

Object-Oriented Programming