Recall: Overloading Multiplication

```
class Fraction(object):
    """Instance attributes:
        numerator   [int]:     top
        denominator [int > 0]: bottom """

    def __mul__(self,q):
        """Returns: Product of self, q
        Makes a new Fraction; does not
        modify contents of self or q
        Precondition: q a Fraction"""
        assert type(q) == Fraction
        top = self.numerator*q.numerator
        bot = self.denominator*q.denominator
        return Fraction(top,bot)
```

```
>>> p = Fraction(1,2)
>>> q = 2 # an int
>>> r = p*q
```

Python converts to

```
>>> r = p.__mul__(q) # ERROR
```

Can only multiply fractions. But ints "make sense" too.

Dispatch on Type

- Types determine behavior
  - Diff types = diff behavior
  - **Example**: + (plus)
    - Addition for numbers
    - Concatenation for strings
- Can implement with ifs
  - Main method checks type
  - "Dispatches" to right helper
- **How all operators work**
  - Checks (class) type on left
  - Dispatches to that method

```
class Fraction(object):
    ...
    def __mul__(self,q):
        """Returns: Product of self, q
        Precondition: q a Fraction or int"""
        if type(q) == Fraction:
            return self._mulFrac(q)
        elif type(q) == int:
            return self._mulInt(q)
    ...
    def _mulInt(self,q): # Hidden method
        return Fraction(self.numerator*q,
                        self.denominator)
```

Another Problem: Subclasses

```
class Fraction(object):
    """Instances are normal fractions n/d
    Instance attributes:
        numerator   [int]:     top
        denominator [int > 0]: bottom """

class BinaryFraction(Fraction):
    """Instances are fractions k/2^n
    Instance attributes are same, BUT:
        numerator   [int]:          top
        denominator [= 2^n, n ≥ 0]: bottom"""
    def __init__(self,k,n):
        """Make fraction k/2^n """
        assert type(n) == int and n >= 0
        Fraction.__init__(k,2 ** n)
```

```
>>> p = Fraction(1,2)
>>> q = BinaryFraction(1,2) # 1/4
>>> r = p*q
```

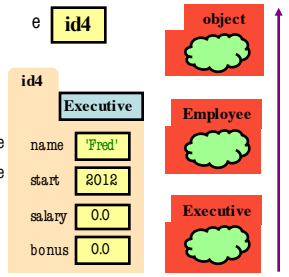
Python converts to

```
>>> r = p.__mul__(q) # ERROR
```

__mul__ has precondition type(q) == Fraction

The isinstance Function

- isinstance(<obj>,<class>)
  - True if <obj>'s class is same as or a subclass of <class>
  - False otherwise
- **Example**:
  - isinstance(e,Executive) is True
  - isinstance(e,Employee) is True
  - isinstance(e,object) is True
  - isinstance(e,str) is False
- Generally preferable to type
  - Works with base types too!



Fixing Multiplication

```
class Fraction(object):
    """Instance attributes:
        numerator   [int]:     top
        denominator [int > 0]: bottom"""

    def __mul__(self,q):
        """Returns: Product of self, q
        Makes a new Fraction; does not
        modify contents of self or q
        Precondition: q a Fraction"""
        assert isinstance(q, Fraction)
        top = self.numerator*q.numerator
        bot = self.denominator*q.denominator
        return Fraction(top,bot)
```

```
>>> p = Fraction(1,2)
>>> q = BinaryFraction(1,2) # 1/4
>>> r = p*q
```

Python converts to

```
>>> r = p.__mul__(q) # OKAY
```

Can multiply so long as it has numerator, denominator

Error Types in Python

```
def foo():
    assert 1 == 2, 'My error'
    ...
```

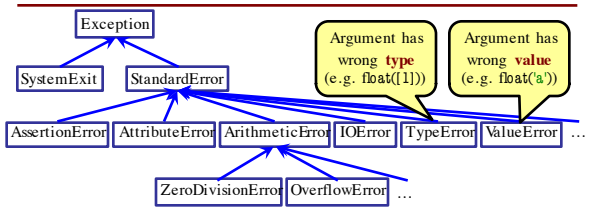
```
>>> foo()
AssertionError: My error
```

```
def foo():
    x = 5 / 0
    ...
```

```
>>> foo()
ZeroDivisionError: integer
division or modulo by zero
```

Class Names

Python Error Type Hierarchy



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

Why so many error types?

Errors and Dispatch on Type

- try-except blocks can be restricted to **specific** errors
  - Doe except if error is **an instance** of that type
  - If error not an instance, do not recover
- **Example**:

```
try:
    input = raw_input() # get number from user
    x = float(input)    # convert string to float
    print 'The next number is '+str(x+1)
except ValueError:
    print 'Hey! That is not a number!'
```

May have IOError

May have ValueError

Only recovers ValueError. Other errors ignored.

Creating Errors in Python

- Create errors with raise
  - **Usage**: raise <exp>
  - exp evaluates to an object
  - An instance of Exception
- Tailor your error types
  - **ValueError**: Bad value
  - **TypeError**: Bad type
- Still prefer **asserts** for preconditions, however
  - Compact and easy to read

```
def foo(x):
    assert x < 2, 'My error'
    ...
```

Identical

```
def foo(x):
    if x >= 2:
        m = 'My error'
        raise AssertionError(m)
    ...
```

Creating Your Own Exceptions

```
class CustomError(StandardError):
    """An instance is a custom exception"""
    pass
```

This is all you need
- No extra fields
- No extra methods
- No constructors

Inherit everything

Only issues is choice of parent Exception class. Use StandardError if you are unsure what.

Errors and Dispatch on Type

- try-except can put the error in a variable
- **Example**:

```
try:
    input = raw_input() # get number from user
    x = float(input)    # convert string to float
    print 'The next number is '+str(x+1)
except ValueError as e:
    print e.message
    print 'Hey! That is not a number!'
```

Some Error subclasses have more attributes

Typing Philosophy in Python

- **Duck Typing**:
  - "Type" object is determined by its methods and properties
  - Not the same as type() value
  - Preferred by Python experts
- Implement with hasattr()
  - hasattr(<object>,<string>)
  - Returns true if object has an attribute/method of that name
- This has many problems
  - The name tells you nothing about its specification

```
class Fraction(object):
    """Instance attributes:
        numerator   [int]:    top
        denominator [int > 0]: bottom"""
    ...
    def __eq__(self,q):
        """Returns: True if self, q equal,
        False if not, or q not a Fraction"""
        if (not (hasattr(other,'numerator') and
                 hasattr(other,'denomenator')):
            return False
        left = self.numerator*q.denominator
        right = self.denominator*q.numerator
        return left == right
```