Recall: Overloading Multiplication

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

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

Another Problem: Subclasses

```python
class BinaryFraction(Fraction):
    """Instances are fractions n/d
    Instance attributes are same, BUT:
    numerator (int) top
    denominator (>0, n > 0) int
    """
def __mul__(self, n):
    """Make fraction k/2""
    assert n >= 0
    return Fraction(self.numerator * n, self.denominator)
```

Fixing Multiplication

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

Can multiply so long as it has numerator, denominator

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

The `isinstance` Function

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

Error Types in Python

```python
def foo():
    assert 1 == 2, 'My error'
    x = 5 / 0
    ...?
```

- `foo()`
  - `foo()`
    - `AssertionError: My error`
    - `ZeroDivisionError: integer division or modulo by zero`
**Python Error Type Hierarchy**

- Exception
  - SystemExit
  - StandardError
  - AssertionError
  - AttributeError
  - ValueError
  - TypeError
  - IOError
  - ... (additional error types)
  - ZeroDivisionError
  - OverflowError

![Why so many error types?](http://docs.python.org/library/exceptions.html)

**Errors and Dispatch on Type**

- try-except blocks can be restricted to specific errors
- **Example:**
  ```python
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`:
  ```python
  def foo(x):
    assert x < 2, 'My error'
  ...
  ```

- Tailor your error types:
  ```python
  def foo(x):
    if x >= 2:
      m = 'My error'
      raise AssertionError(m)
  ...
  ```

- Create errors with `raise`:
  ```python
  def foo(x):
    raise <exp>
  ````
  ```
  ```
  exp evaluates to an object
  An instance of Exception
  ```

**Errors and Dispatch on Type**

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

**Creating Your Own Exceptions**

- This is all you need:
  ```python
  class CustomError(StandardError):
    """An instance is a custom exception""
    pass
  
  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"
    ...
    return False
  
  def __str__(self):
    """Returns string representation
    of Fraction"
    return str(self.numerator) + '/' + str(self.denominator)
  ```

**Typing Philosophy in Python**

- **Duck Typing:**
  ```python
  def __eq__(self, q):
    return self.numerator == q or self.denominator == q
  ```

- This has many problems:
  ```python
  def __str__(self):
    return str(self.numerator) + '/' + str(self.denominator)
  ```

- The name tells you nothing about its specification

**Creating Your Own Exceptions**

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

- This is all you need:
  ```python
  class CustomError(StandardError):
    """An instance is a custom exception""
    pass
  ```

- This is all you need:
  ```python
  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"
    ...
    return False
  
  def __str__(self):
    """Returns string representation
    of Fraction"
    return str(self.numerator) + '/' + str(self.denominator)
  ```

- This has many problems:
  ```python
  def __str__(self):
    return str(self.numerator) + '/' + str(self.denominator)
  ```

- The name tells you nothing about its specification

**Typing Philosophy in Python**

- **Duck Typing:**
  ```python
  def __eq__(self, q):
    return self.numerator == q or self.denominator == q
  ```

- This has many problems:
  ```python
  def __str__(self):
    return str(self.numerator) + '/' + str(self.denominator)
  ```

- The name tells you nothing about its specification