**About super()**

- `super()` is very limited
  - Can only go one level
  - BAD: `super().super()`

- Need arguments for more
  - `super(class, self)`

---

**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!

**Error Types in Python**

- All errors are instances of class `BaseException`
- This allows us to organize them in a hierarchy

- `BaseException`
  - `Exception`
    - `AssertionError`
      - `AssertionError` means "extends" or "is an instance of"

---

**A Problem with 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"

- `class Executive(object):
  _name 'Fred'
  _salary 0.0
  _bonus 0.0

- `class Employee(object):
  _name 'Fred'
  _salary 0.0
  _bonus 0.0`

---

**Fixing Multiplication**

- `class Fraction(object):
  """Returns: Product of self, q
  Makes a new Fraction; does not modify contents of self or q
  Precondition: q a Fraction"

- `def __mul__(self, q):
  """Make fraction k/2^n"
  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
  >>> r` # OKAY
  - Python converts to
  - Can multiply so long as it has numerator, denominator
Handling Errors by Type

- try-except blocks can be restricted to specific errors
  - Do except if error is an instance of that type
  - If error not an instance, do not recover
- Example:
  ```python
  try:
    val = input()  # get number from user
    x = float(val)  # convert string to float
    print('The next number is ' + str(x+1))
  except ValueError:
    print('Hey! That is not a number!')
  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

Creating Your Own Exceptions

- try-except can put the error in a variable
- Example:
  ```python
  try:
    val = input()  # get number from user
    x = float(val)  # convert string to float
    print('The next number is ' + str(x+1))
  except ValueError as e:
    print(e.args[0])
  print('Hey! That is not a number!')
  ```

Accessing Attributes with Strings

- `hasattr(<obj>,<name>)`
  - Checks if attribute exists
- `getattr(<obj>,<name>)`
  - Reads contents of attribute
- `delattr(<obj>,<name>)`
  - Deletes the given attribute
- `setattr(<obj>,<name>,<val>)`
  - Sets the attribute value
- `<obj>.__dict__`
  - List all attributes of object

Typing Philosophy in Python

- `Duck Typing`:
  - "Type" object is determined by its methods and properties
  - Not the same as type()
  - 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(self, 'numerator') and
            hasattr(self, 'denominator') and
            hasattr(other, 'numerator') and
            hasattr(other, 'denominator')):
            return False
        left = self.numerator * q.denominator
        right = q.numerator * self.denominator
        return left == right
```