Lecture 21

Programming with Subclasses

Announcements for Today

Reading

- Today: See reading online
- Tuesday: Chapter 7

- **Prelim, Nov 9th 7:30-9:00**
  - Material up to **Today**
  - Review has been posted
  - Recursion + Loops + Classes
- **S/U Students are exempt**
- **Conflict with Prelim time?**
  - **LAST DAY TO SUBMIT**

Assignments

- A4 is still being graded
  - Will be done tomorrow
- But I looked at surveys
  - People generally liked it
  - **Avg Time**: 8.8 hrs
  - **Median**: 8, **STDev**: 4.6
- A5 is due tonight at midnight
- Continue working on A6
  - Finish Task 3 by Sunday

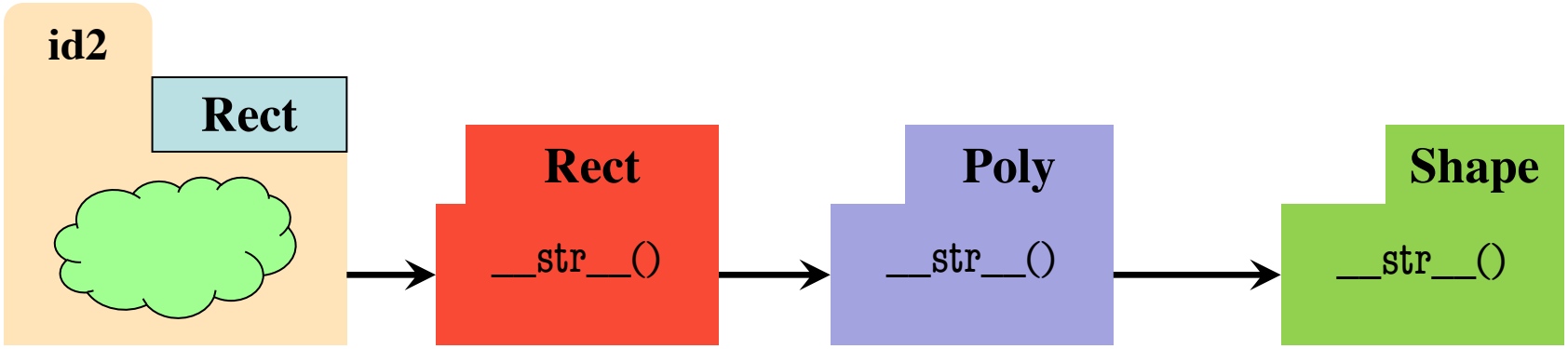
About super()

- super() is very limited
  - Can only go one level
  - **BAD**: super().super()
- Need arguments for more
  - super(class,self)

The **subclass**

Object in the method

p **id2**



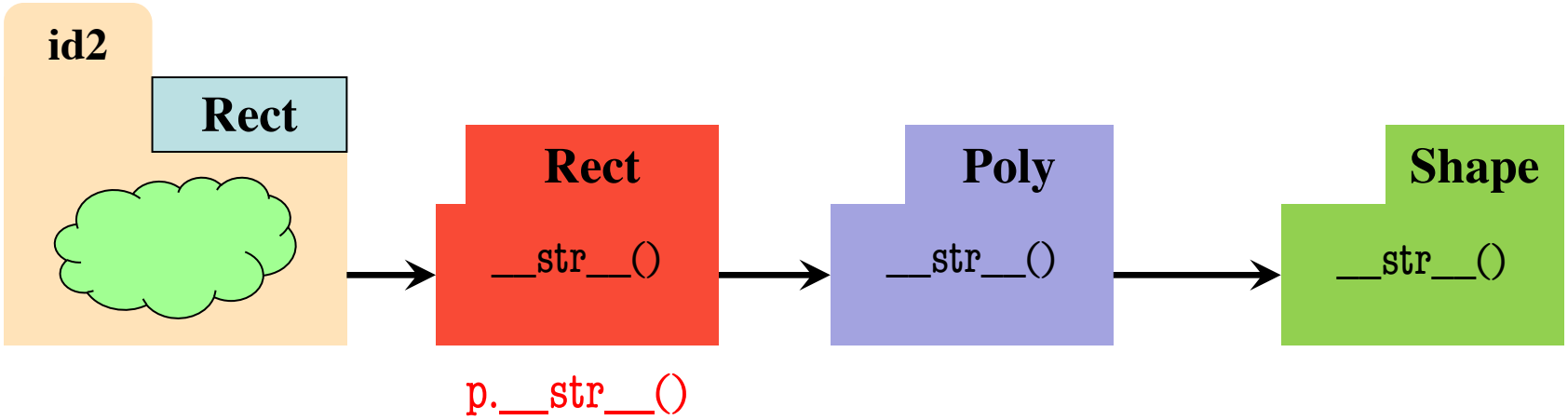
About super()

- super() is very limited
  - Can only go one level
  - **BAD**: super().super()
- Need arguments for more
  - super(class,self)

The **subclass**

Object in the method

p **id2**



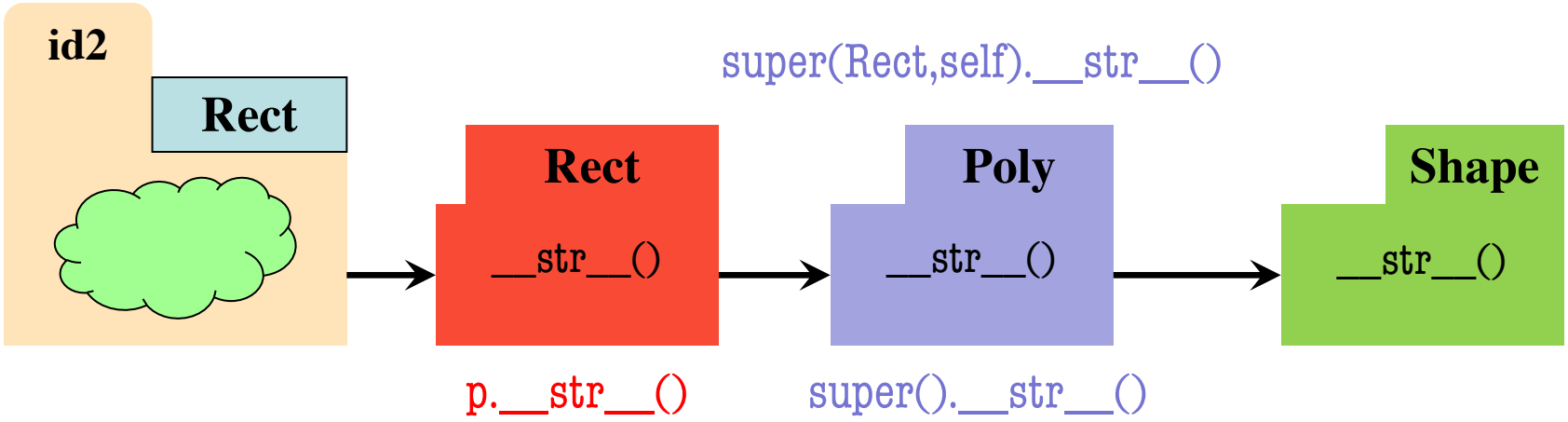
About super()

- super() is very limited
  - Can only go one level
  - **BAD**: super().super()
- Need arguments for more
  - super(class,self)

The **subclass**

Object in the method

p **id2**



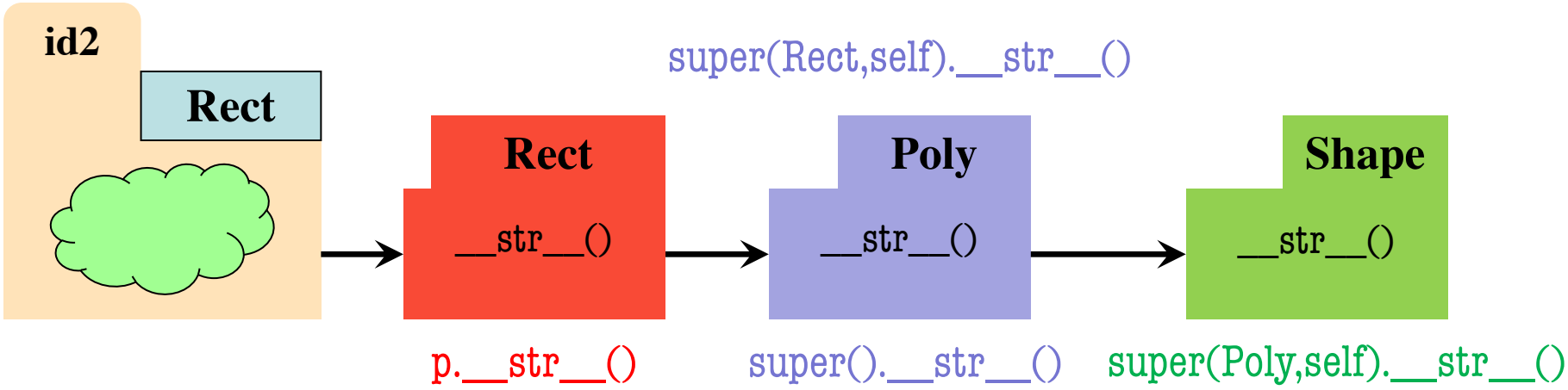
About super()

- super() is very limited
  - Can only go one level
  - **BAD**: super().super()
- Need arguments for more
  - super(class,self)

The **subclass**

Object in the method

p **id2**

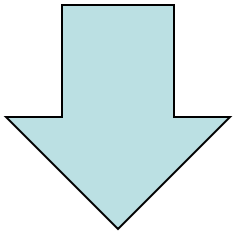


A Problem with Subclasses

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

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

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



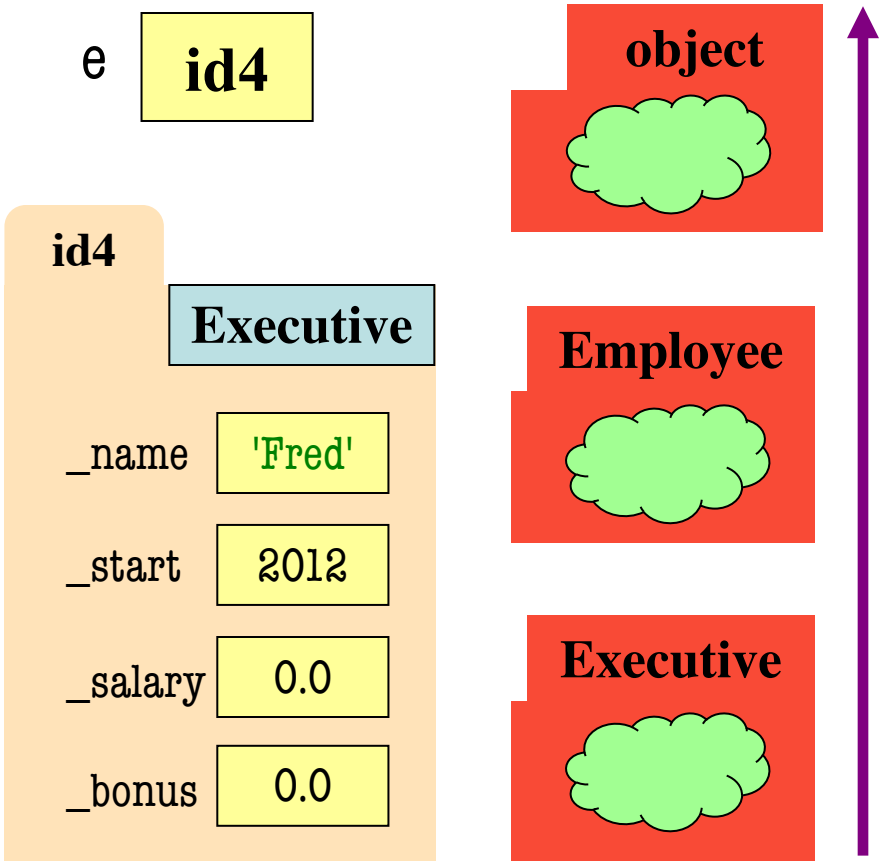
Python converts to

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

__mul__ has precondition type(q) == Fraction

The isinstance Function

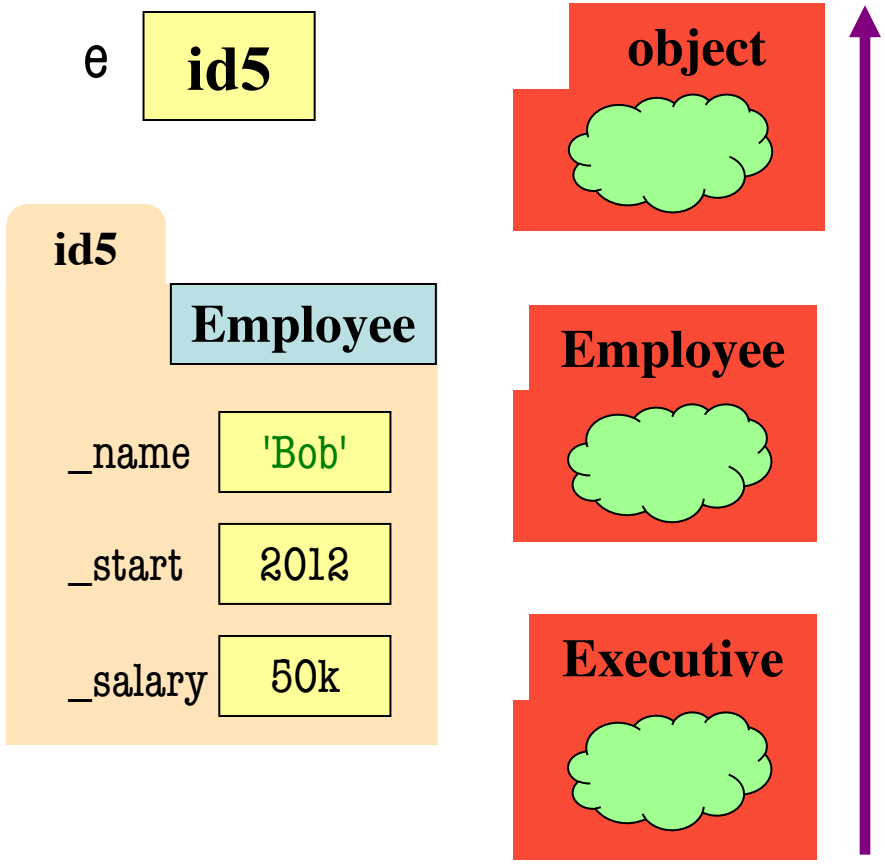
- isinstance(<obj>,<class>)
  - True if <obj>'s class is same as or a subclass of <class>
  - False otherwise
- **Example**:
  - isinstance(e,Executive) is True
  - isinstance(e,Employee) is True
  - isinstance(e,object) is True
  - isinstance(e,str) is False
- Generally preferable to type
  - Works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



isinstance and Subclasses

```
>>> e = Employee('Bob',2011)
>>> isinstance(e,Executive)
???
```

A: True
B: False
C: Error
D: I don't know



isinstance and Subclasses

```
>>> e = Employee('Bob',2011)
>>> isinstance(e,Executive)
???
```

A: True
B: False Correct
C: Error
D: I don't know

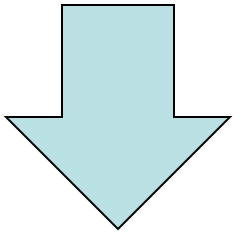
→ means "extends"
or "is an instance of"

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

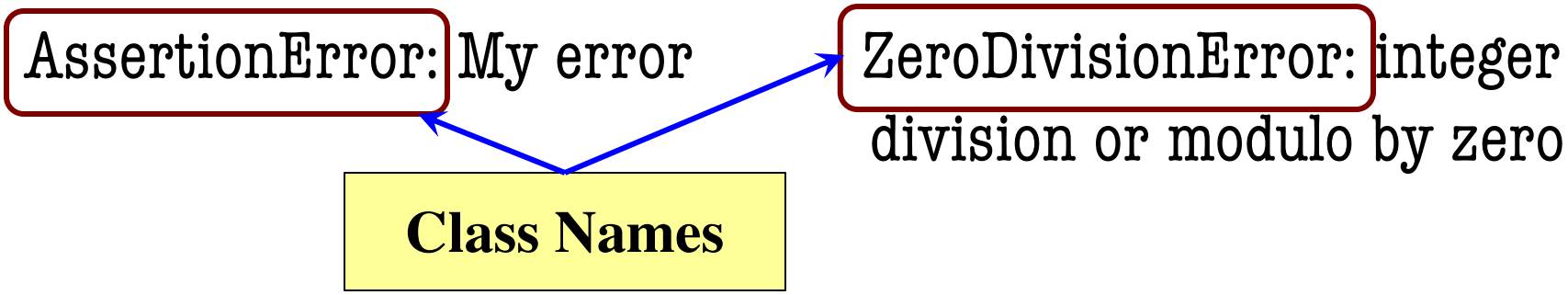
Error Types in Python

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

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

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

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



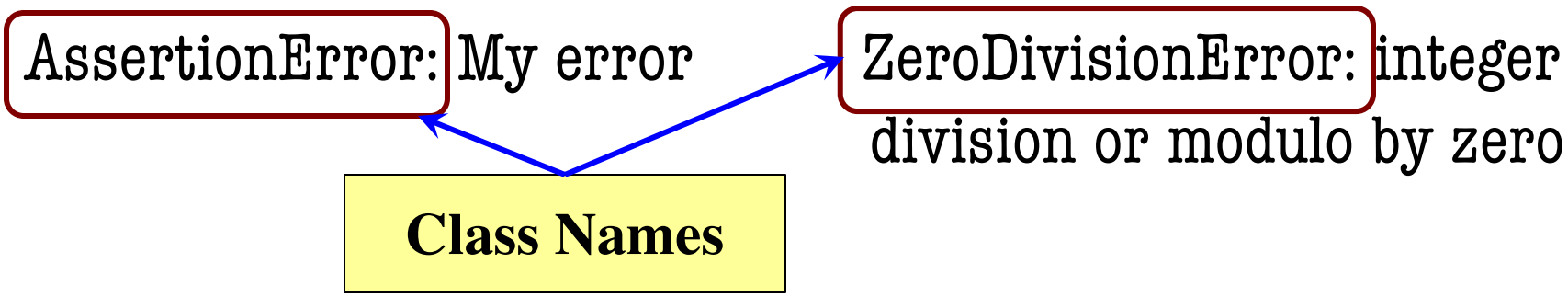
Error Types in Python

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

Information about an error is stored inside an **object**. The error type is the **class** of the error object.

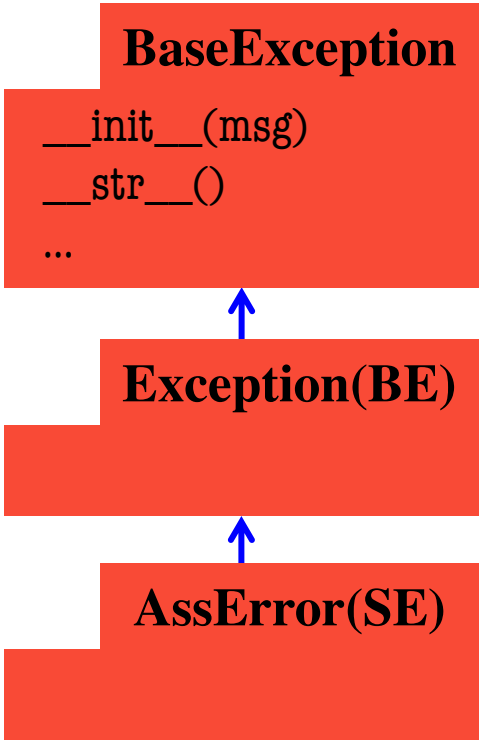
```
>>> foo()
AssertionError: My error
```

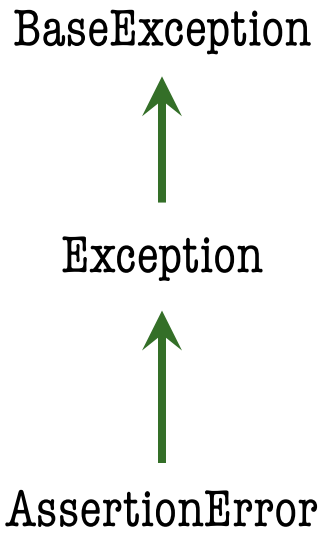
```
>>> foo()
ZeroDivisionError: integer
division or modulo by zero
```

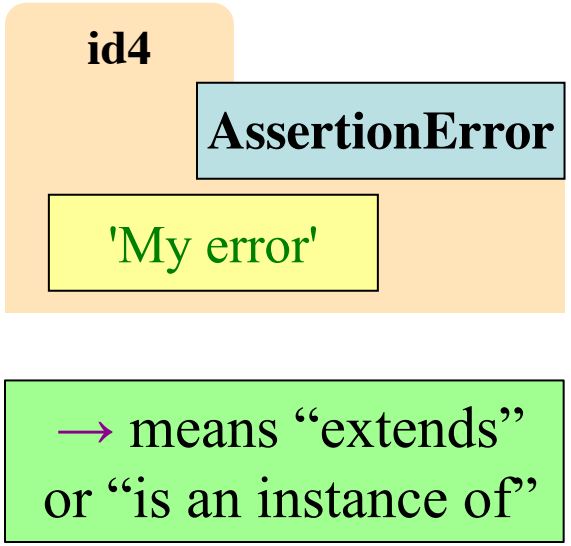


Error Types in Python

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

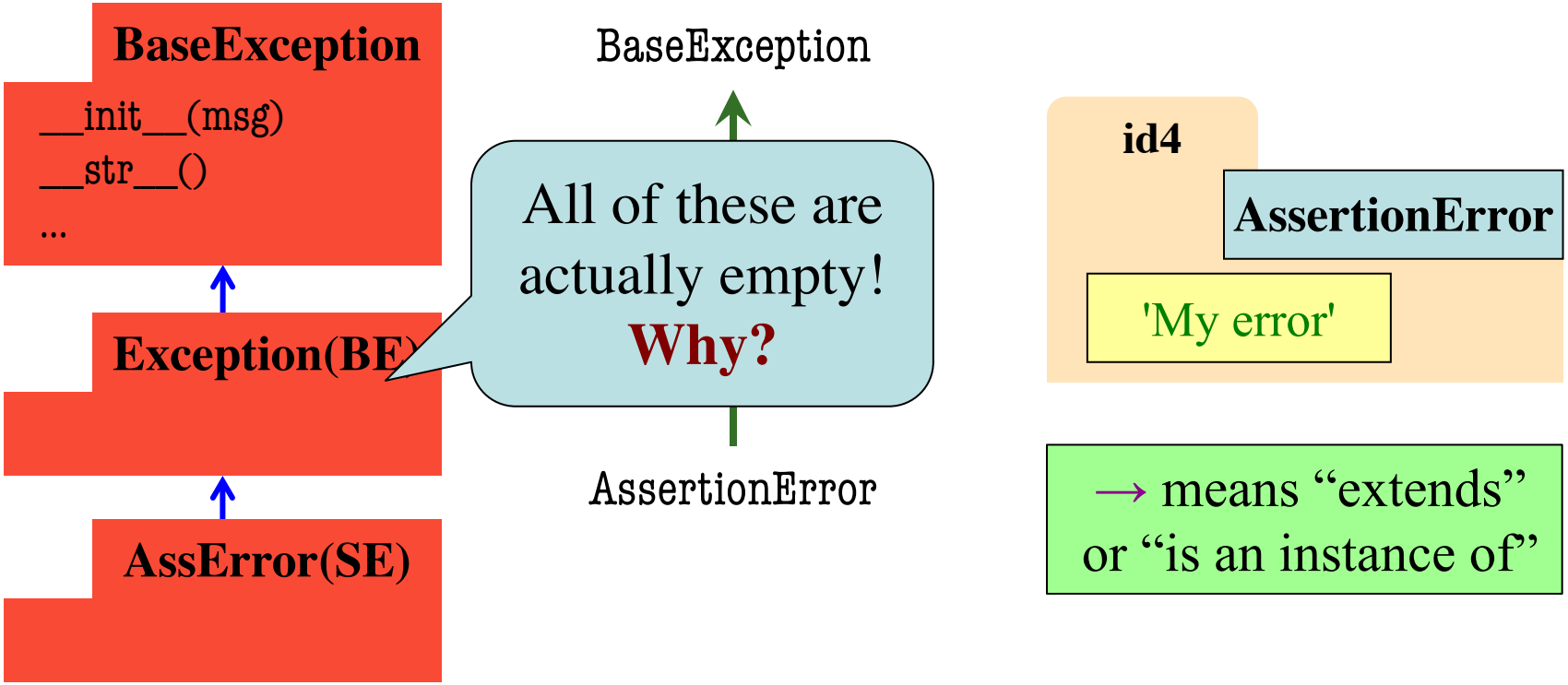




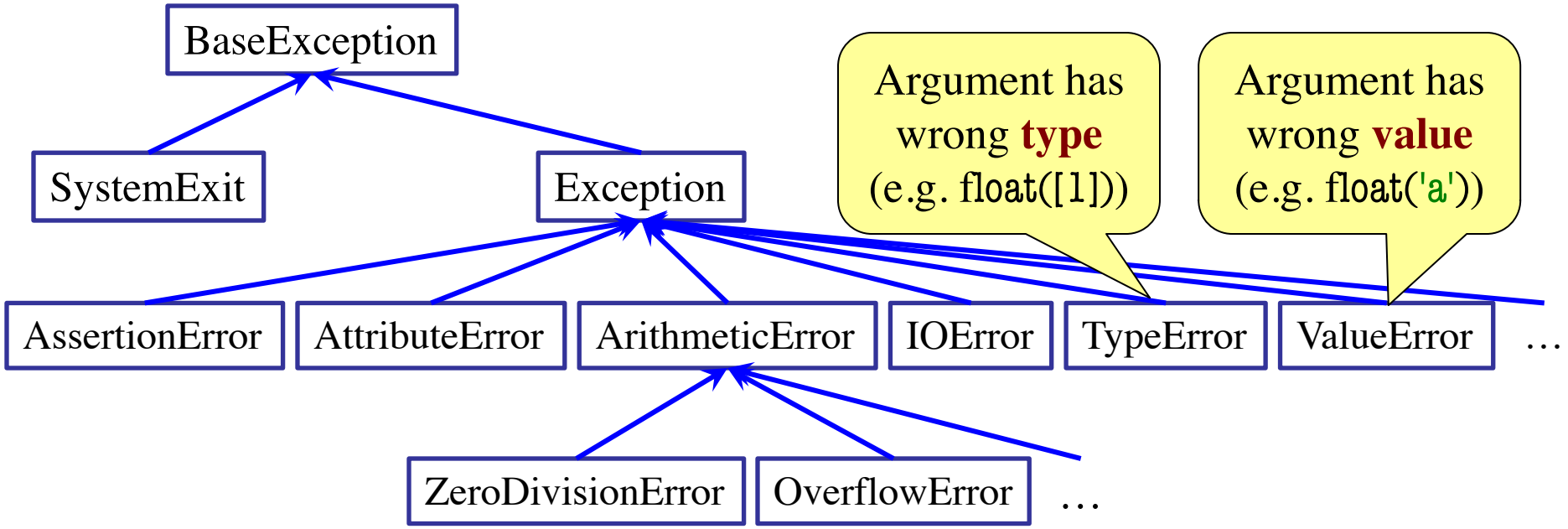


Error Types in Python

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



Python Error Type Hierarchy



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

Why so many error types?

Recall: Recovering from Errors

- try-except blocks allow us to recover from errors
  - Do the code that is in the try-block
  - Once an error occurs, jump to the catch
- **Example**:

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

might have an error

executes if have an error

Handling Errors by 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:
    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!')
```

May have IOError

May have ValueError

Only recovers ValueError.
Other errors ignored.

Handling Errors by 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:
    val = input()          # get number from user
    x = float(val)         # convert string to float
    print('The next number is '+str(x+1))
except IOError:
    print('Check your keyboard!')
```

May have IOError

May have ValueError

Only recovers IOError.
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'
        err = AssertionError(m)
        raise err
```

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'
        err = TypeError(m)
        raise err
```

Raising and Try-Except

```
def foo():
    x = 0
    try:
        raise Exception()
        x  = 2
    except Exception:
        x = 3
    return x
```

- The value of foo()?

A: 0
B: 2
C: 3
D: No value. It stops!
E: I don't know

Raising and Try-Except

```
def foo():
    x = 0
    try:
        raise Exception()
        x  = 2
    except Exception:
        x = 3
    return x
```

- The value of foo()?

A: 0
B: 2
C: 3 Correct
D: No value. It stops!
E: I don't know

Raising and Try-Except

```
def foo():
    x = 0
    try:
        raise Exception()
        x  = 2
    except BaseException:
        x = 3
    return x
```

- The value of foo()?

A: 0
B: 2
C: 3
D: No value. It stops!
E: I don't know

Raising and Try-Except

```
def foo():
    x = 0
    try:
        raise Exception()
        x  = 2
    except BaseException:
        x = 3
    return x
```

- The value of foo()?

A: 0
B: 2
C: 3 Correct
D: No value. It stops!
E: I don't know

Raising and Try-Except

```
def foo():
    x = 0
    try:
        raise Exception()
        x  = 2
    except AssertionError:
        x = 3
    return x
```

- The value of foo()?

A: 0
B: 2
C: 3
D: No value. It stops!
E: I don't know

Raising and Try-Except

```
def foo():
    x = 0
    try:
        raise Exception()
        x  = 2
    except AssertionError:
        x = 3
    return x
```

- The value of foo()?

A: 0
B: 2
C: 3
D: No value. Correct
E: I don't know

Python uses isinstance to match Error types

Creating Your Own Exceptions

```
class CustomError(Exception):
    """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 error class. Use `Exception` if you are unsure what.

Handling Errors by Type

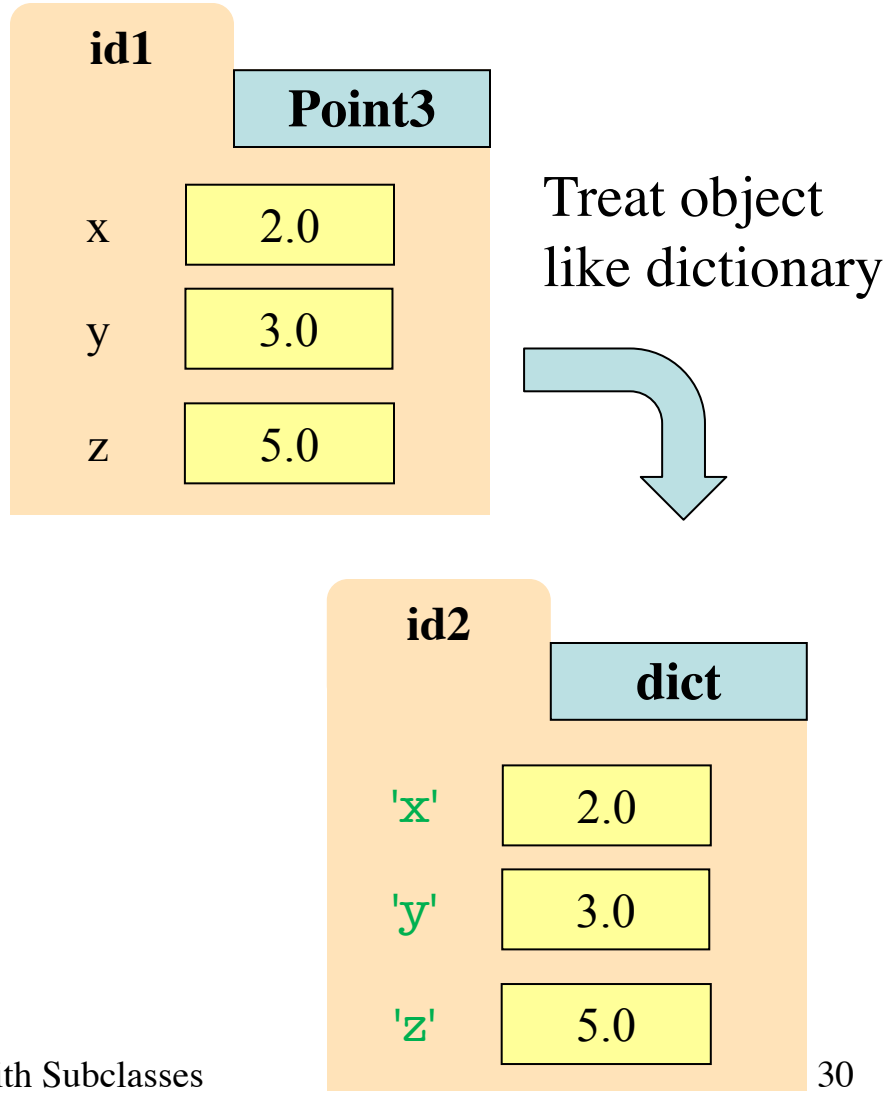
- try-except can put the error in a variable
- **Example**:

```
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!')
```

Some Error subclasses have more attributes

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() 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 type(q) != Fraction:
            return False
        left = self.numerator*q.denominator
        rght = self.denominator*q.numerator
        return left == rght
```

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(q,'numerator') and
                 hasattr(q,'denomenator')):
            return False
        left = self.numerator*q.denominator
        rght = self.denominator*q.numerator
        return left == rght
```

Typing Philosophy in Python

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

Compares **anything** with numerator & denominator

```
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(q,'numerator') and
                 hasattr(q,'denomenator')):
            return False
        left = self.numerator*q.denominator
        rght = self.denominator*q.numerator
        return left == rght
```

Typing Philosophy in Python

- **Duck Typing**:
  - "Type" ob… by its m…
  - Not the…
  - Preferre…
- Implemen…
  - hasattr(…
  - Returns … attribut…
- This has ma… problems
  - The name tells you nothing about its specification

How to properly implement/use typing is a major debate in language design

- What we really care about is **specifications** (and **invariants**)
- Types are a "shorthand" for this

Different typing styles trade ease-of-use with overall program robustness/safety

```
class Fraction(object):
    """Instance attributes:
                                    top
                                    ottom"""
                                    equal,
                                    ction"""
                                    tor') and
                                    nator')):
                                    .denominator
    rght = self.denominator*q.numerator
    return left == rght
```

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 Employee(object):
    """An Employee with a salary"""
    ...
    def __eq__(self,other):
        if (not (hasattr(other,'name') and
                 hasattr(other,'start') and
                 hasattr(other,'salary'))
            return False
        return (self.name == other.name and
                self.start == other.start and
                self.salary == other.salary)
```