Lecture 21

Typing and Subclasses

Announcements for This Lecture

Assignments

- A4 is now graded
  - **Mean**: 90.4 **Median**: 93
  - **Std Dev**: 10.6
  - **Mean**: 8.5 hrs **Median**: 8 hrs
  - **Std Dev**: 3.5 hrs
- A5 is also graded
  - **Mean**: 47.2 **Median**: 49
  - **A**: 47 (75%), **B**: 40 (20%)
  - Solutions posted in CMS

Prelim 2

- **Prelim, Nov 21st at 7:30**
  - Same rooms as last time
- **Material up to TODAY**
  - Recursion + Loops + Classes
  - Study guide is now posted
  - Review Sun. 5pm in Statler
- **Conflict with Prelim?**
  - Prelim 2 Conflict on CMS
  - SDS students must submit!

What is Typing?

- We know what a (Python) type is
  - All values in Python have a type
  - **Typing:** act of finding the type of a value
  - **Example:** `type(x) == int`
- Commonly used in **preconditions**
  - Definition assumes certain operations
  - If operations are missing, def may crash
  - So we use assert to check for operations

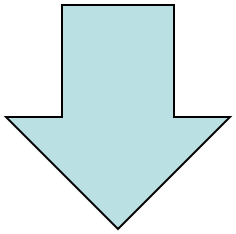
A Problem with Subclasses

```
class Fraction(object):
    """Instances are normal fractions n/d"""
    # INSTANCE ATTRIBUTES
    # _numerator:    int
    # _denominator: int > 0

class BinaryFraction(Fraction):
    """Instances are fractions k/2^n """
    # INSTANCE ATTRIBUTES same but
    # _denominator: int = 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
```



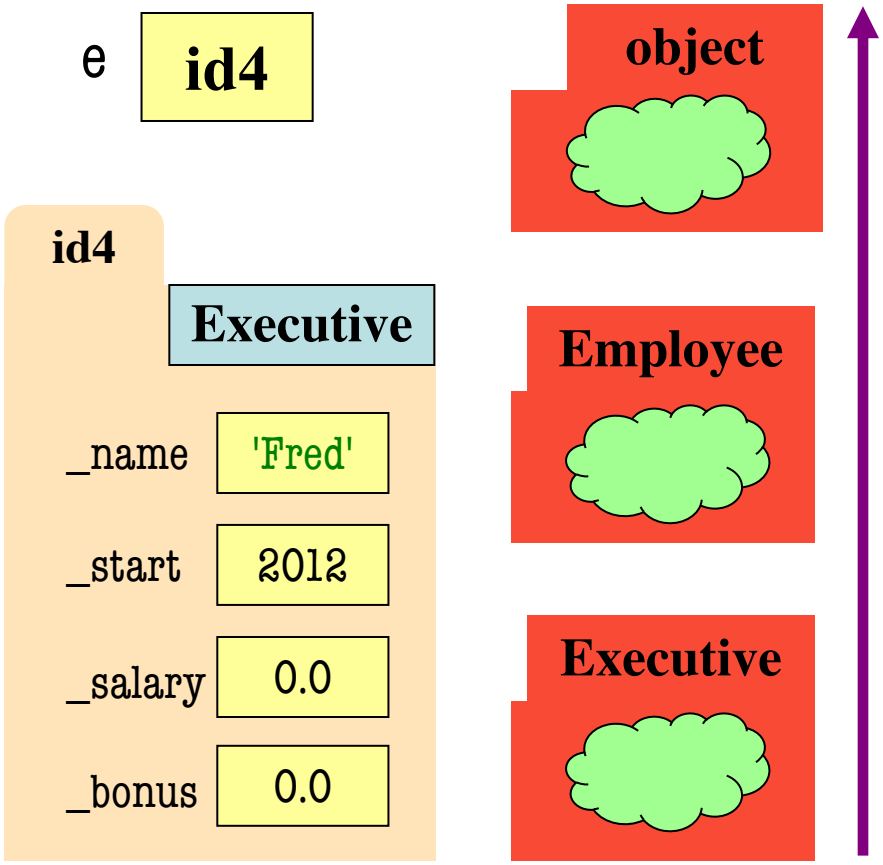
Python converts to

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

__mul__ has precondition type(q) == Fraction

The isinstance Function

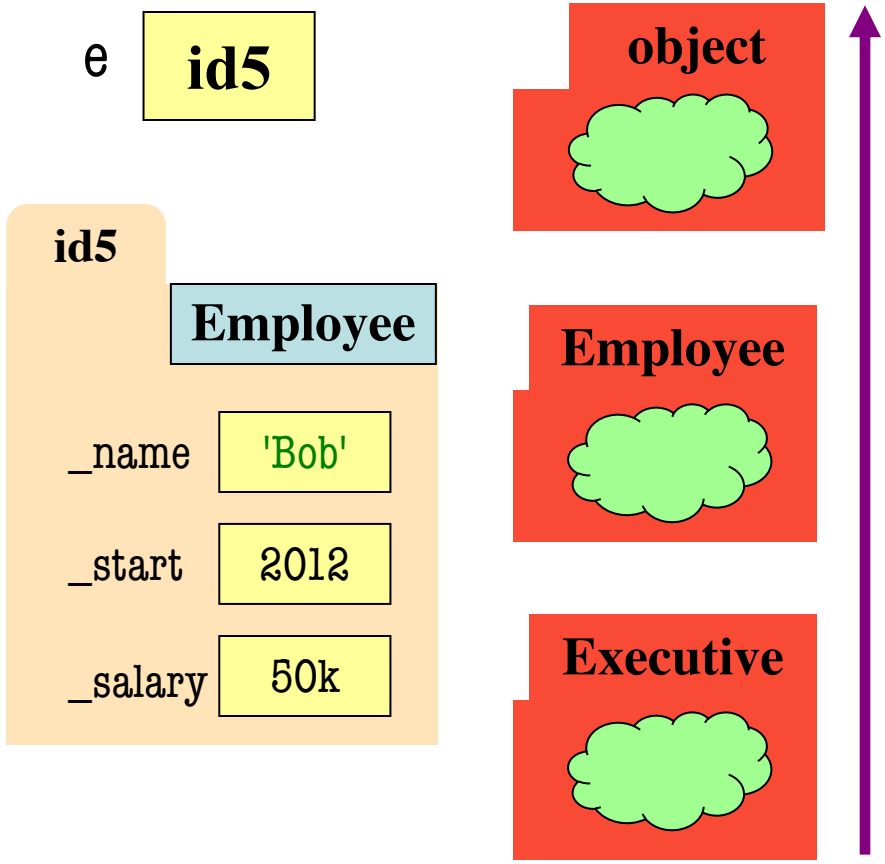
- isinstance(<obj>,<class>)
  - True if <obj>'s class is same as or a subclass of <class>
  - False otherwise
- **Example**:
  - isinstance(e,Executive) is True
  - isinstance(e,Employee) is True
  - isinstance(e,object) is True
  - isinstance(e,str) is False
- Generally preferable to type
  - Works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

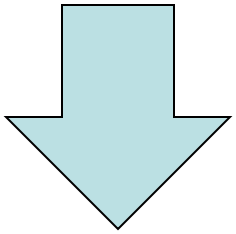
→ means "extends"
or "is an instance of"

Fixing Multiplication

```
class Fraction(object):
    """Instances are fractions n/d"""
    # _numerator:    int
    # _denominator: int > 0

    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

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.

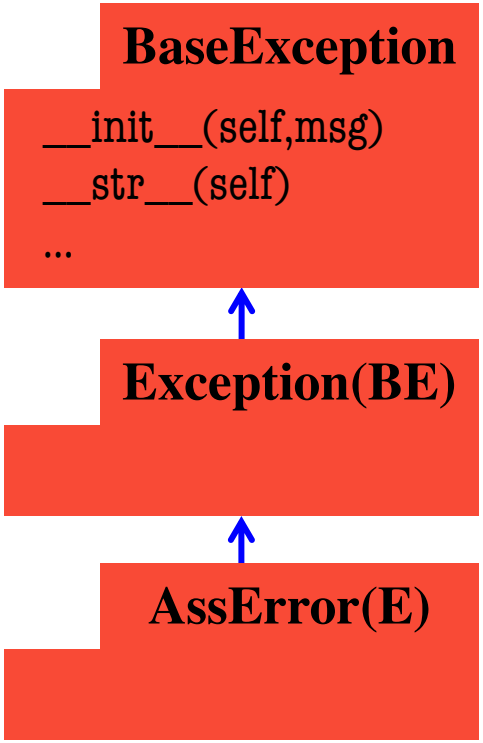
```
>>> foo()
AssertionError: My error
```

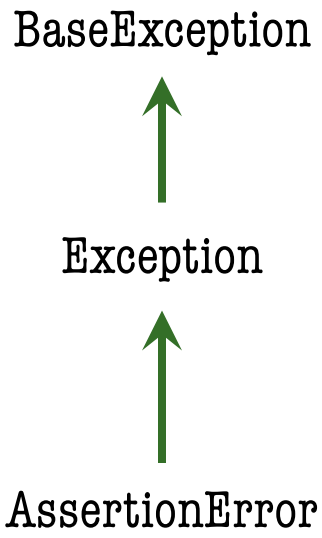
```
>>> foo()
ZeroDivisionError: integer division or modulo by zero
```

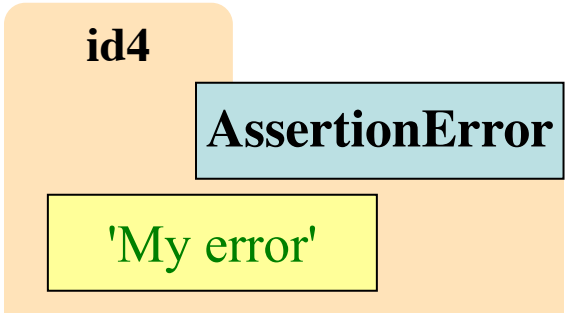
Class Names

Error Types in Python

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



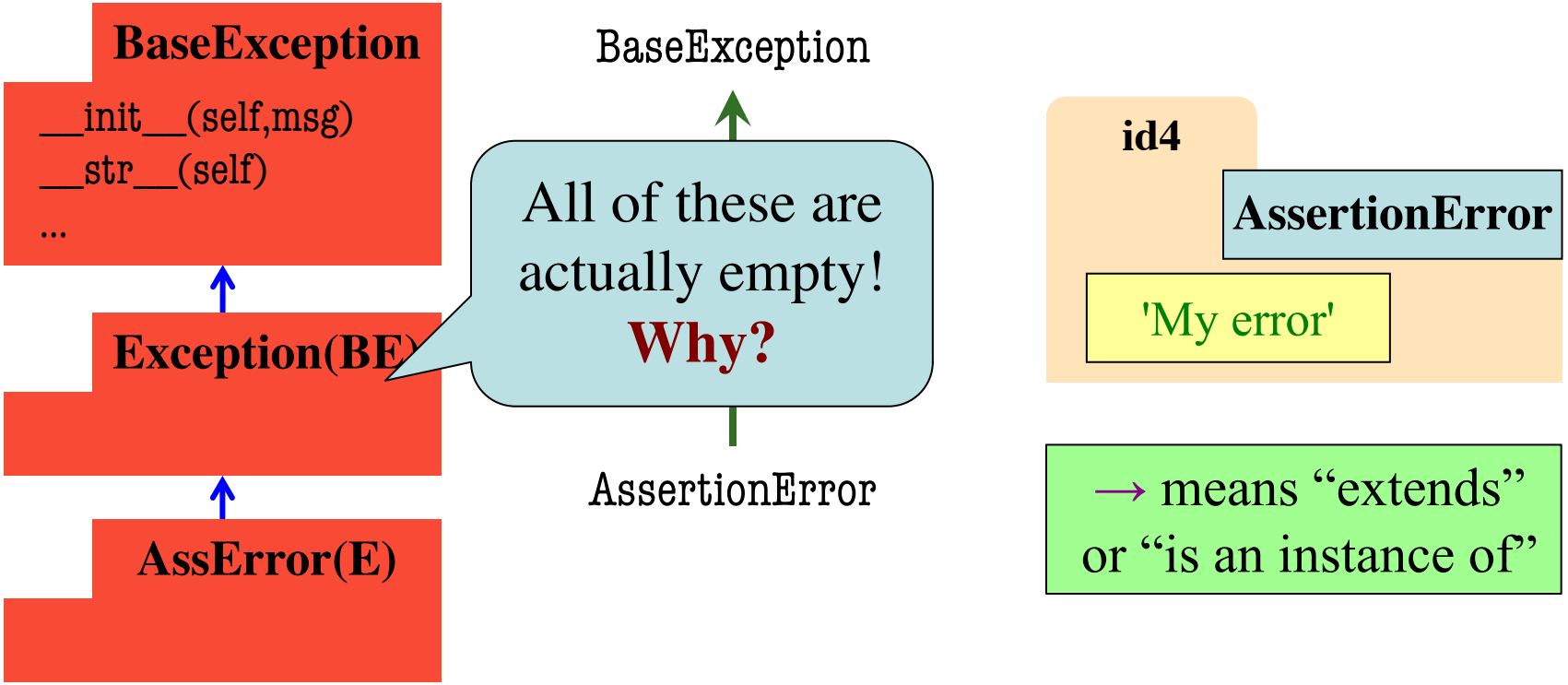




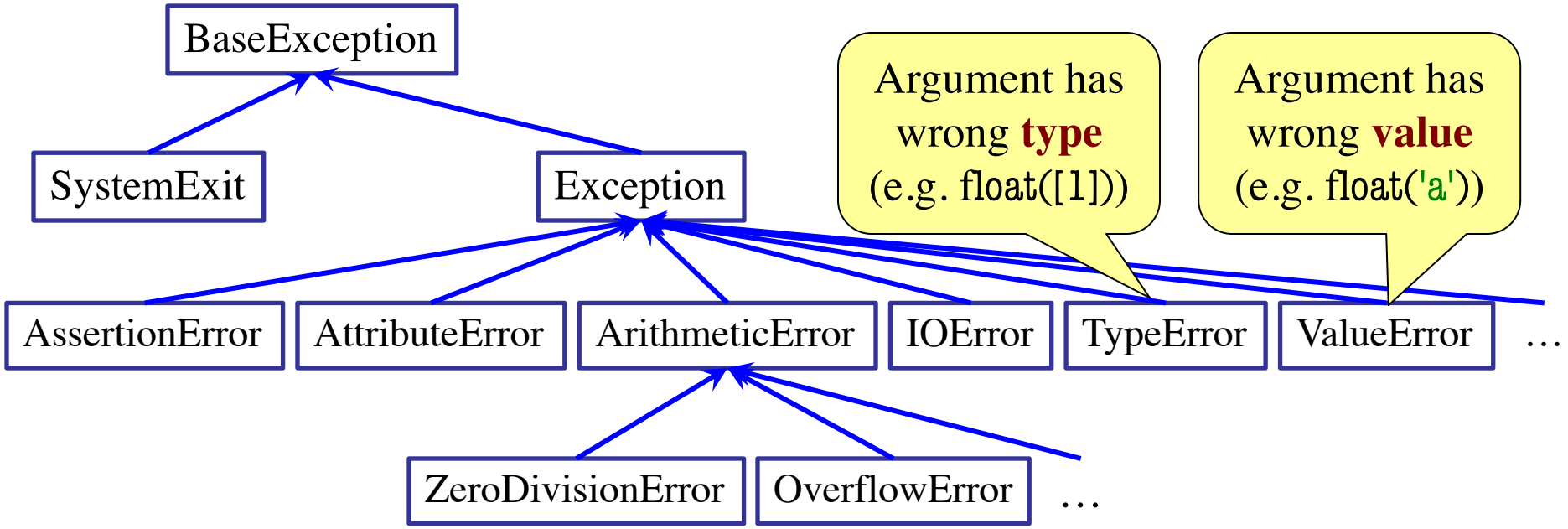
→ means "extends"
or "is an instance of"

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 = ValueError(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

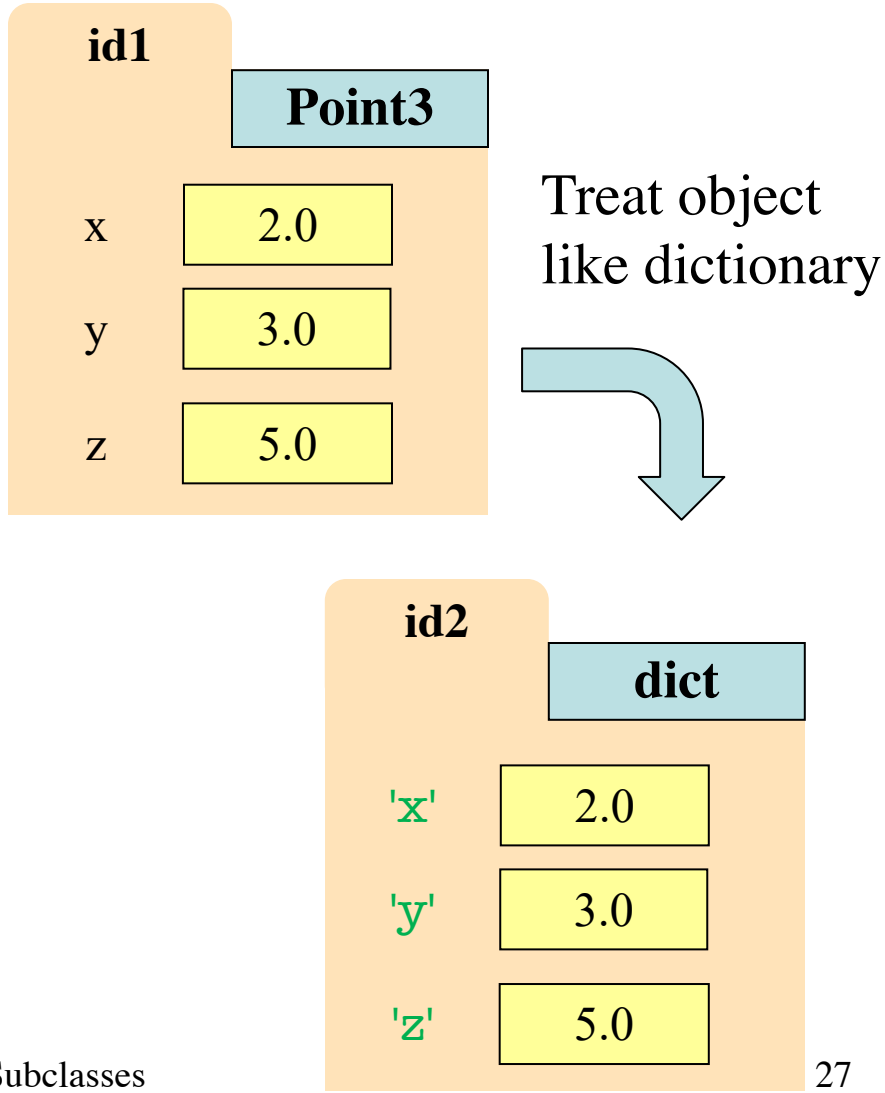
- try-except can put the error in a variable
- **Example**:

```
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):
    """Instances are fractions n/d"""
    # numerator:    int
    # denominator: int > 0
    ...
    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):
    """Instances are fractions n/d"""
    # numerator:    int
    # denominator: int > 0
    ...
    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):
    """Instances are fractions n/d"""
    # numerator:    int
    # denominator: int > 0
    ...
    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
```

Final Word on Typing

- How to implement/use typing is **controversial**
  - Major focus in **designing new languages**
  - Some langs have no types; others complex types
- Trade-of between **ease-of-use** and **robustness**
  - Complex types allow automated bug finding
  - But make they also make code harder to write
- What we really care about is **specifications**
  - **Duck Typing:** we *think* the value meets a spec
  - Types **guarantee** that a specification is met