Module 22

Dyanmic Typing

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

What Happened Here?

- Our typing precondition is too strict
  - Only allow Fractions, not subclasses
  - But subclasses still make sense!
- **Recall:** Why put types in preconditions?
  - To guarantee that we have a set of operations
  - But subclasses inherit all operations!
- In this video series, we will revisit typing
  - Act of checking the (current) type of a variable
  - How do we adapt this to handle subclasses?

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"""
        # q is Fraction or a subclass
        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
```

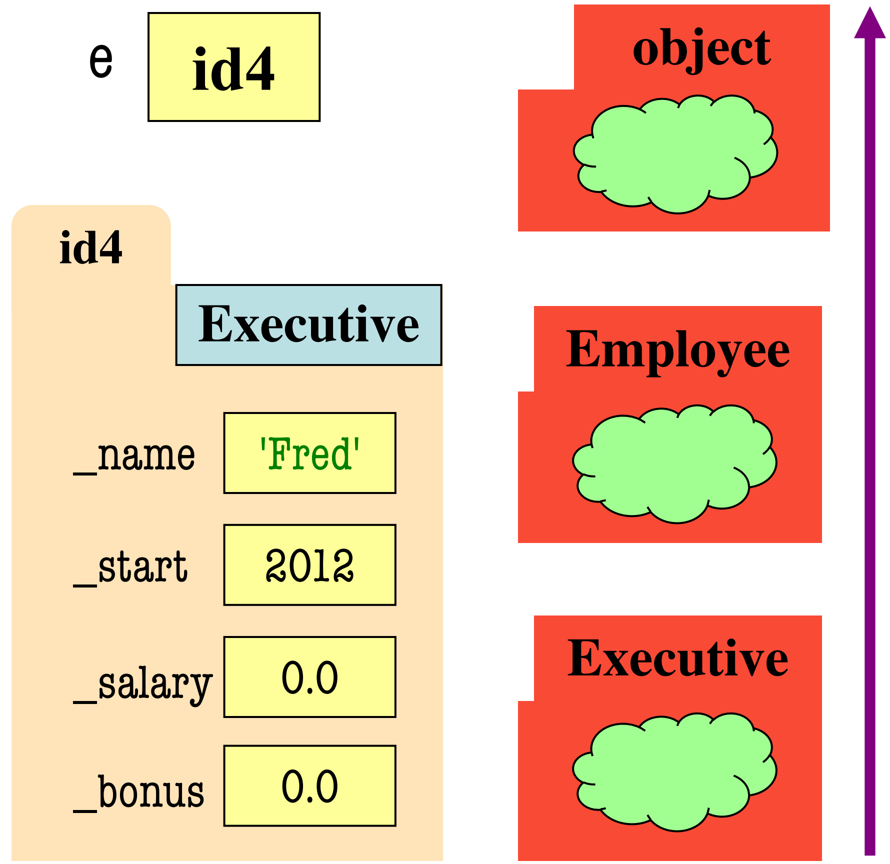
Python converts to

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

Can multiply so long as it has numerator, denominator

The isinstance Function

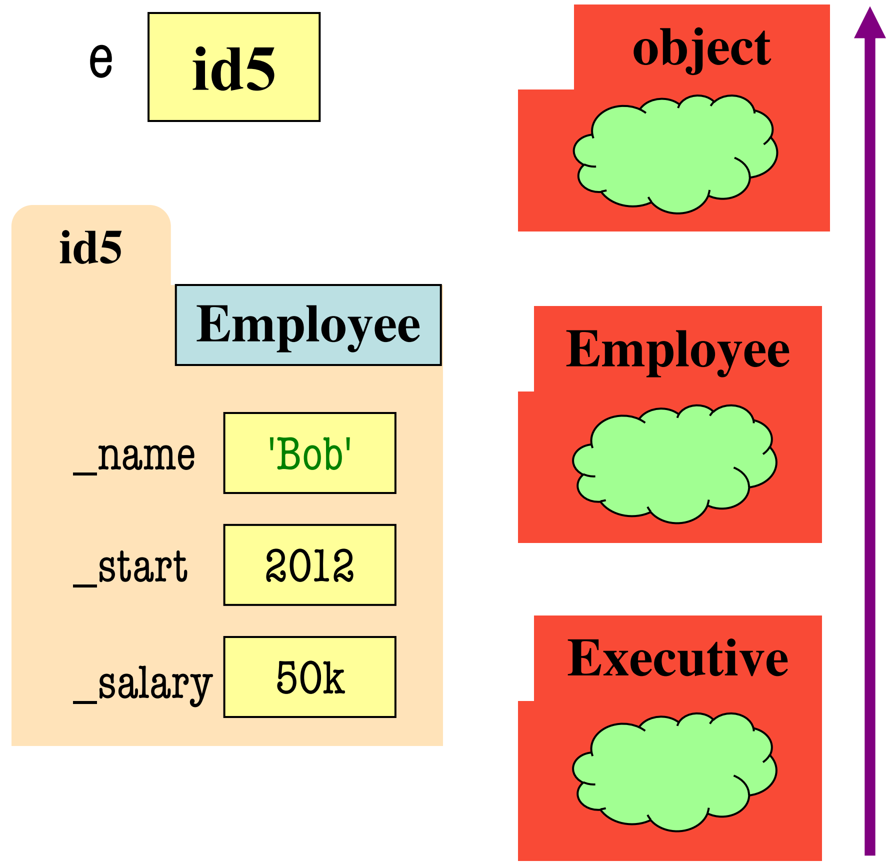
- isinstance(<obj>,<class>)
  - True if <obj>'s class is same as or a subclass of <class>
  - False otherwise
- **Example**:
  - isinstance(e,Executive) is True
  - isinstance(e,Employee) is True
  - isinstance(e,object) is True
  - isinstance(e,str) is False
- Generally preferable to type
  - Works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

object

Employee

Executive

→ 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
```

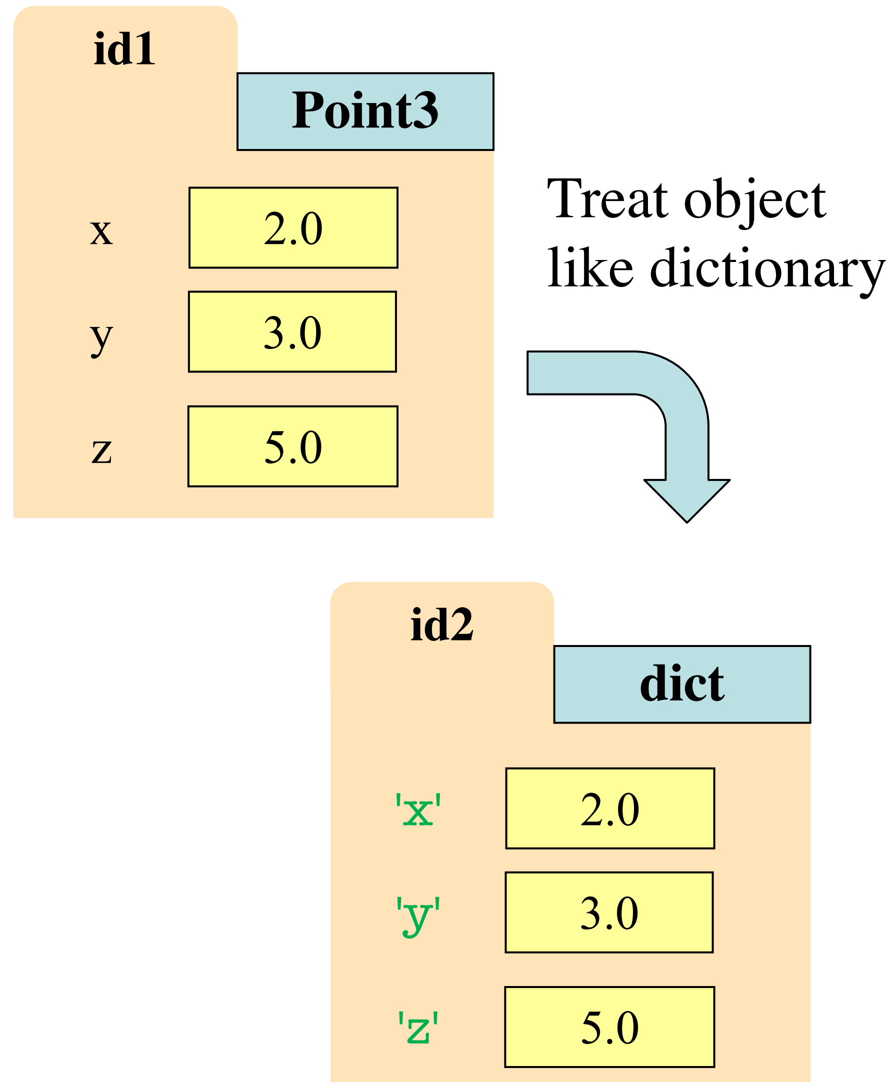
Can multiply so long as it has numerator, denominator

Accessing Attributes

- Typing guarantees certain attributes exists
  - `RGB` object? It has red, green, and blue
  - `Point3` object? It has x, y, and z
- What if you are unsure an attribute exists?
  - Is there a way to ask Python?
  - …other than crashing inside of a try-except
- Remember that all objects are dictionaries
  - (or at least are backed by dictionaries)
  - We can use this to our advantage

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



Why Is This Useful?

- This is useful in interactive scripts
  - User types in an attribute to access
  - That value is a string
  - Can now turn that string into attribute!
- **Demo:** dynamic.py
- Used in very advanced applications
  - A way to separate responsibilities
  - User does not need to know all attributes
  - Can write code filling in with strings later

Why Is This Useful?

- This is useful in interactive scripts
  - User types in an attribute to access
  - That value is a string
  - Can now turn that string into attribute!
- **Demo:** dynamic.py
- Used in very advanced applications

Far beyond scope of this course

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