Classes and Objects II

Source:
"Think Python", 2nd edition, by Allen B. Downey
"www.python-course.eu" by Bernd Klein

Instructor: Paruj Ratanaworabhan

Sameness

We have to define precisely what it means for two objects to be the same

For example, if two Points are the same, does that mean they contain the same data (coordinates) or that they are actually the same object?

If we use the ‘is’ operator, sameness means the same object

```
>>> p1 = Point(3, 4)
>>> p2 = Point(3, 4)
>>> p1 is p2
False
>>> p3 = p1
>>> p1 is p3
True
```

If we assign p1 to p3, then the two variables are **aliases** of the same object.

Shallow VS Deep Equality

Shallow equality compares only the references, not the contents of the objects

To compare the contents of the objects — **deep equality** — we can write a function called same_coordinates:

```
def same_coordinates(p1, p2):
    return (p1.x == p2.x) and (p1.y == p2.y)
```

Now if we create two different objects that contain the same data, we can use same_point to find out if they represent points with the same coordinates

```
>>> p1 = Point(3, 4)
>>> p2 = Point(3, 4)
>>> same_coordinates(p1, p2)
True
```

Of course, if the two variables refer to the same object, they have both shallow and deep equality

def __eq__(self, other)

You can define equality for 2 objects from a given class with def __eq__

For point objects, here is the implementation of def __eq__

```
def __eq__(self, other):
    return (self.x == other.x) and (self.y == other.y)

>>> p1 = Point(3, 4)
>>> p2 = Point(3, 4)
>>> p1 == p2
True
```

Of course, if the two variables refer to the same object, they have both shallow and deep equality

Copying I

With aliasing, changes made in one place might have unexpected effects in another place

It is hard to keep track of all the variables that might refer to a given object

Copying an object is often an alternative to aliasing

The copy module contains a function called copy that can duplicate any object:

```
>>> import copy
>>> p1 = Point(3, 4)
>>> p2 = copy.copy(p1)
>>> p1 is p2
False
>>> same_coordinates(p1, p2)
True
>>> p1 == p2
>>> True
```

Once we import the copy module, we can use the copy function to make a new Point

p1 and p2 are not the same point, but they contain the same data

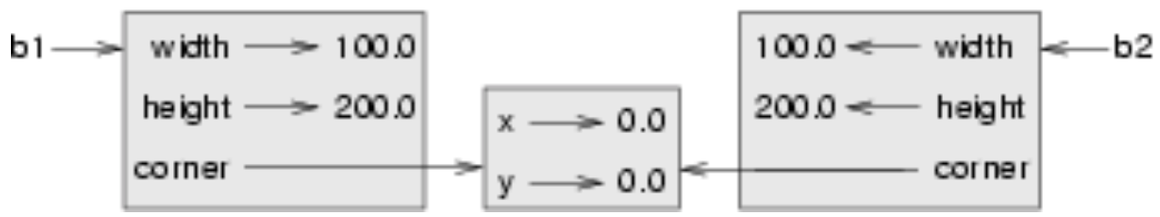
Copying II

To copy a simple object like a Point, which doesn't contain any embedded objects, copy is sufficient. This is called **shallow copying**

Rectangle contains a reference to a Point, copy doesn't do quite the right thing

It copies the reference to the Point object, so both the old Rectangle and the new one refer to a single Point

If we create a box, b1, in the usual way and then make a copy, b2, using copy, the resulting state diagram looks like this:



Copying III

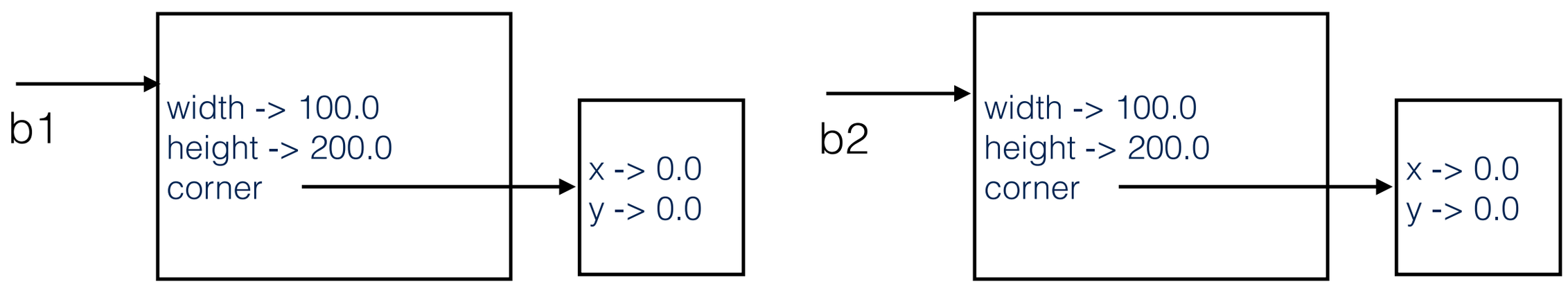
In this case, invoking the grow method on one of the Rectangle objects would not affect the other, but invoking move on either would affect both

This behavior is confusing and error-prone. The shallow copy has created an alias to the Point that represents the corner.

The copy module contains a function named **deepcopy** that copies not only the object but also any embedded objects.

```
>>> b2 = copy.deepcopy(b1)
```

Now b1 and b2 are completely separate objects.



Operator Overloading

- It is possible to have different meanings for the same operator when applied to different types

- For example, + in Python means quite different things for integers and for strings

- This feature is called operator overloading

Operator Overloading in class Fraction

```
class Fraction:
    def __init__(self, num, den):
        self.num = num
        self.den = den

    def reduce(self):
        """Returns a reduced form of this fraction
        """
        import math
        g = math.gcd(self.num, self.den)
        return Fraction(self.num//g, self.den//g)

    def add(self, m):
        """Returns a new fraction in reduced form that results from
adding this fraction with the m fraction
        """
        f_num = self.num*m.den + m.num*self.den
        f_den = self.den*m.den
        f = Fraction(f_num, f_den)
        return f.reduce()

    def __add__(self, other):
        return self.add(other)


f1 = Fraction(1, 2)
f2 = Fraction(1, 4)
print(f1 + f2) # 3/4
```

Exercise

1. Overload the plus (+) operator so that you can do fraction addition with it; test your code

2. Overload the multiplication (*) operator so that you can do fraction multiplication with it; test your code

3. Overload the subtraction (-) operator so that you can doe fraction subtraction with it; test your code

Once you are done with 1. and 2., you should be able to execute run_OO_fraction.py without any errors

4. Add def __eq__ method to class Fraction so that you can compare if the two fraction objects are equal; test your code

5. Complete OO_complex.py and run_OO_complex.py

Attribute Types

Naming	Type	Meaning
name	Public	These attributes can be freely used inside or outside a class definition.
_name	Protected	Protected attributes should not be used outside the class definition, unless inside a subclass definition.
__name	Private	This kind of attribute is inaccessible and invisible. It's neither possible to read nor write to those attributes, except inside the class definition itself.

An Example

```
class A():

    def __init__(self):
        self.__priv = "I am private"
        self.pub = "I am public"


x = A()
x.pub
# Output:'I am public'

x.pub = x.pub + " and my value can be changed"
x.pub
# Output:’I am public and my value can be changed'

x.__priv

# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# AttributeError: 'A' object has no attribute '__priv'
```

An Information Hiding Class

```
class Robot:

    def __init__(self, name=None, build_year=2000):
        self.__name = name                  # private attribute
        self.__build_year = build_year # private attribute

    # getter and setter methods to access the attributes
    def set_name(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_build_year(self, by):
        self.__build_year = by

    def get_build_year(self):
        return self.__build_year

    # method to print the string representation of this class
    def __str__(self):
        return "Name: " + self.__name + ", Build Year: " +
str(self.__build_year)
```

Not so Pythonic
(not the style widely adopted by mainstream Python programmers)

Interacting with Objects from an Information Hiding Class

```
x = Robot("Marvin", 1979)
y = Robot("Caliban", 1943)
robo_list = [x, y]
for rob in robo_list:
     print(rob)
     if rob.get_name() == "Caliban":
          rob.set_name("Caliban Limited")
          rob.set_build_year(1993)
          print(rob)
```

```
# Output:
# Name: Marvin, Build Year: 1979
# Name: Caliban, Build Year: 1943
# Name: Caliban Limited, Build Year: 1993
```

Not so Pythonic
(not the style widely adopted by mainstream Python programmers)

What We Have Learned

- Object equality
- Operator overloading
- Attribute types and information hiding