Imperative programming with Python

January 2012 project: Class #6

Facundo Carreiro

ILLC, University of Amsterdam

January 17th, 2012

User-defined types

- Built-in types are the basic building blocks.
- With them we can build user-defined types called *classes*.
- As built-in types, classes abstract concepts.

```
class Color(object):
    """Represents a color"""
```

the `class` keyword defines a new class.

```
>>> print Color
<class '__main__.Color'>
```

- We create a `Color` object calling it as a function

```
>>> c = Color()
>>> print c
<__main__.Color object at 0x55f10>
```

c is called an *instance* of the `Color` class.

Classes

- We have to choose an internal representation for the color.
- We can use, for example, the red-green-blue values.

```
class Color(object):
    """Represents a color"""
    def __init__(self, r=0, g=0, b=0):
        self.r = r
        self.g = g
        self.b = b
```

  the `__init__` function gets called when creating a new object.

- It is called a *constructor:* creates (initializes) the r,g,b *attributes*.
- Fundamental: The internal representation should be *hidden*.

Classes: methods

- We define *methods* to interact with the class.

```
# inside the class definition
    def setRGB(self, r, g, b):
        self.r = r;
        self.g = g;
        self.b = b;

    def getRGB(self):
        return (self.r, self.g, self.b)
```

- They are used with the dot notation.

```
>>> c.setRGB(0.2, 0.75, 0.5)
>>> print c.getRGB()
(0.2, 0.75, 0.5)
```

Classes: methods

- The internal representation should be transparent to the user.
- For instance, we could have some method for the YIQ representation

```
# inside the class definition
    def setYIQ(self, y, i, q):
        self.r = y + 0.956*i + 0.621*q;
        self.g = y - 0.272*i - 0.647*q;
        self.b = y - 1.105*i + 1.702*q;

    def getYIQ(self):
        y = 0.299*self.r + 0.587*self.g + 0.114*self.b
        i = 0.596*self.r - 0.275*self.g - 0.321*self.b
        q = 0.212*self.r - 0.523*self.g - 0.311*self.b
        return (y, i, q)
```

Classes: pure and modifying methods

- So far we defined the following types of methods:
  1. `__init__`: the constructor.
  2. `set...`: the "*setters*".
  3. `get...`: the "*getters*".
- More generally we can make the following distinction
  1. *Modifying* methods: change the representation of the object.
  2. *Pure* methods: perform a calculation and/or side effect but leave the object unchanged.
- Note: Python doesn't have a way to specify pure methods but other languages (e.g. C++) do.

Classes: representation invariants

- In the previous slides we implicitly assumed $r, g, b \in [0, 1]$.
  This is called a *representation invariant* (RI).
- The idea is that
  1. The constructor creates an object satisfying the RI and,
  2. The modifying methods assume the invariant and should preserve it
- Therefore, if our methods depend on external data we should check it

```
# inside the class definition
    def setRGB(self, r, g, b):
        assert in_range(r) and in_range(g) and in_range(b)
        self.r = r; self.g = g; self.b = b;

# outside
def in_range(component):
    return component >= 0 and component <= 1
```

- In theory this shouldn't be necessary but in practice it helps find bugs.

Classes: string representation

- The default printing of an object is not very useful

```
>>> print c
<__main__.Color object at 0x55f10>
```

- The `__str__` method returns a string representation of the object
- By redefining it we say how the object should be printed

```
# inside the class definition
    def __str__(self):
        return 'red: %i, green: %i, blue: %i' % (self.r * 255,
        self.g * 255, self.b * 255)
```

```
>>> print c
red: 51, green: 191, blue: 128
```

Classes: comparing objects

- ```
  >>> c1 = Color(0.2, 0.75, 0.5)
  >>> c2 = Color(0.2, 0.75, 0.5)
  >>> print c1; print c2
  red: 51, green: 191, blue: 191
  red: 51, green: 191, blue: 191
  ```

  c1 and c2 are different objects representing the same color.

- Intuitively, `is` should return False and `==` return True.

  ```
  >>> c1 is c2
  False
  >>> c1 == c2
  False
  ```

  Watch out: The `==` operator is (by default) defined as `is` for user-defined types!
  ```

Classes: comparing objects

- The `__eq__` method computes the comparison
- We should redefine it to reflect the expected behaviour

```
# inside the class definition
    def __eq__(self, other):
        return (self.r == other.r and
                self.g == other.g and
                self.b == other.b)
```

```
>>> c1 == c2
True
```

Classes: copying objects

- Remember that we can't just use `=` to *copy* objects.

```
>> d = c1
>> d is c1
True
```

  it would only create an alias of the object.

- Python provides the `copy` module to do so.

```
>>> import copy
>>> d = copy.copy(c1)
```

- The `copy` function creates a new object and copies all the attributes.

```
>>> c1 is d
False
>>> c1 == d
True
```

A more complex example

```
class Car(object):
    def __init__(self, np=0, c=Color()):
        self.numberplate = np
        self.color = c

    def __eq__(s, o):
        return (s.numberplate == o.numberplate and
            s.color == o.color)
```

Observe that

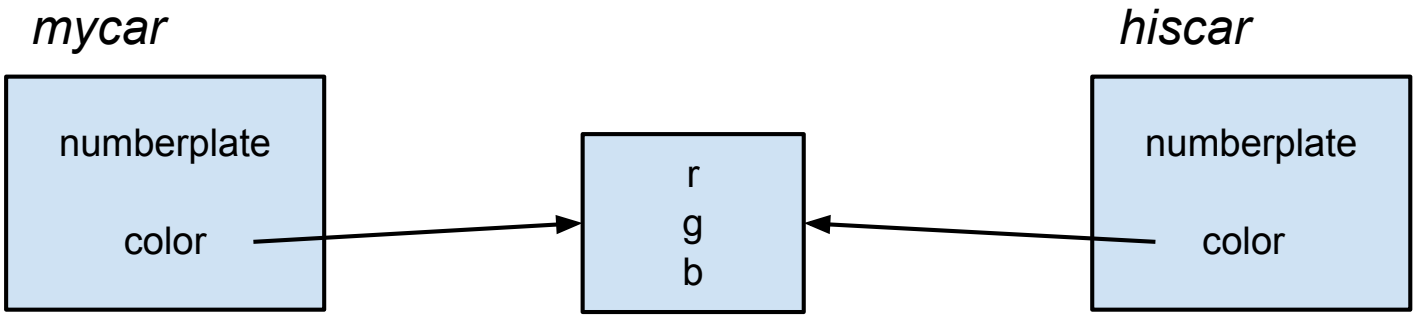
- One of the attributes is also a user-defined type.
- When defining `==` for `Car` we use the equality of `Color`.

A more complex example

```
>>> mycar = Car(np=123)
>>> hiscar = copy.copy(mycar)
>>> mycar is hiscar
False
```

A closer look to the attributes reveals something strange

```
>>> mycar.color is hiscar.color
True
```



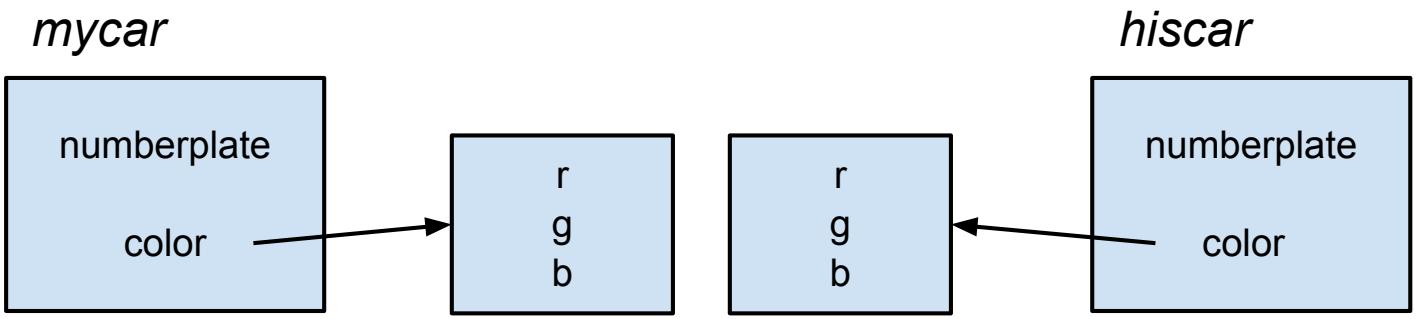
The `copy` function does a shallow copy.

A more complex example

The `deepcopy` function recursively copies the object.

```
>>> hiscar = copy.deepcopy(mycar)
>>> mycar is hiscar
False
```

```
>>> mycar.color is hiscar.color
False
```



Class attributes

- Classes are *also* objects

```
>>> print Car
<class '__main__.Car'>
```

- They can have attributes associated with no particular instance.

```
class Color(object):
    """Represents a color"""
    blackRGB = (0,0,0)
    redRGB = (1,0,0)
```

```
>>> print Color.redRGB
(1,0,0)
```

These are called *class attributes*.

Static methods

- Most of the methods we defined received 'self' as a parameter.
- Sometimes we don't want (or need) that.

```
# inside the Color class definition
    @staticmethod
    def in_range(component):
        return component >= 0 and component <= 1
```

- *Static methods* do not receive an instance.
- They are defined with the `@staticmethod` decorator.
- They belong to the same class because they concern the same concept.

```
# inside the class definition
    def setRGB(self, r, g, b):
        assert Color.in_range(r) and
               Color.in_range(g) and Color.in_range(b)
        ...
```

References

- Chapters 15–17 of the book
  http://greenteapress.com/thinkpython/thinkpython.html
- RGB
  http://en.wikipedia.org/wiki/RGB_color_model
- YIQ
  http://en.wikipedia.org/wiki/YIQ
- Shallow and deep copy
  http://docs.python.org/library/copy.html
- Python decorators
  http://docs.python.org/glossary.html#term-decorator
- Python static method decorator
  http://docs.python.org/library/functions.html#staticmethod