Lecture 8.3 : Object-Oriented Programming: Class variables and methods

Class variables

- Object-oriented programming introduces new variable scopes: *instance variables* and *class variables*. Instance variables we have met already, they are the *data attributes attached to objects* e.g. `hour`, `minute` and `second` in `time.hour`, `time.minute` and `time.second` are instance variables.

- *Class variables are attached to a class* rather than any particular object. The classic class variable example is an object counter: each time an object is created we increment a `count` variable. This variable is a running count of objects created. Let's add one to our `Time` class:

```
# time_v13.py
class Time(object):

    count = 0 # class variable shared by all instances

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
        Time.count += 1

    def __eq__(self, other):
        return ((self.hour, self.minute, self.second) ==
                (other.hour, other.minute, other.second))

    def __add__(self, other):
        return (seconds_to_time(self.time_to_seconds() +
                                other.time_to_seconds()))

    def __gt__(self, other):
        return self.time_to_seconds() > other.time_to_seconds()

    def __iadd__(self, other):
        z = self + other
        self.hour, self.minute, self.second = z.hour, z.minute, z.second
        return self

    def __str__(self):
        return ('The time is {:02d}:{:02d}:{:02d}'.format(self.hour,
                                                          self.minute,
                                                          self.second))


    def time_to_seconds(self):
        return self.hour*60*60 + self.minute*60 + self.second

def seconds_to_time(s):
    minute, second = divmod(s, 60)
    hour, minute = divmod(minute, 60)
    overflow, hour = divmod(hour, 24)
    return Time(hour, minute, second)
```

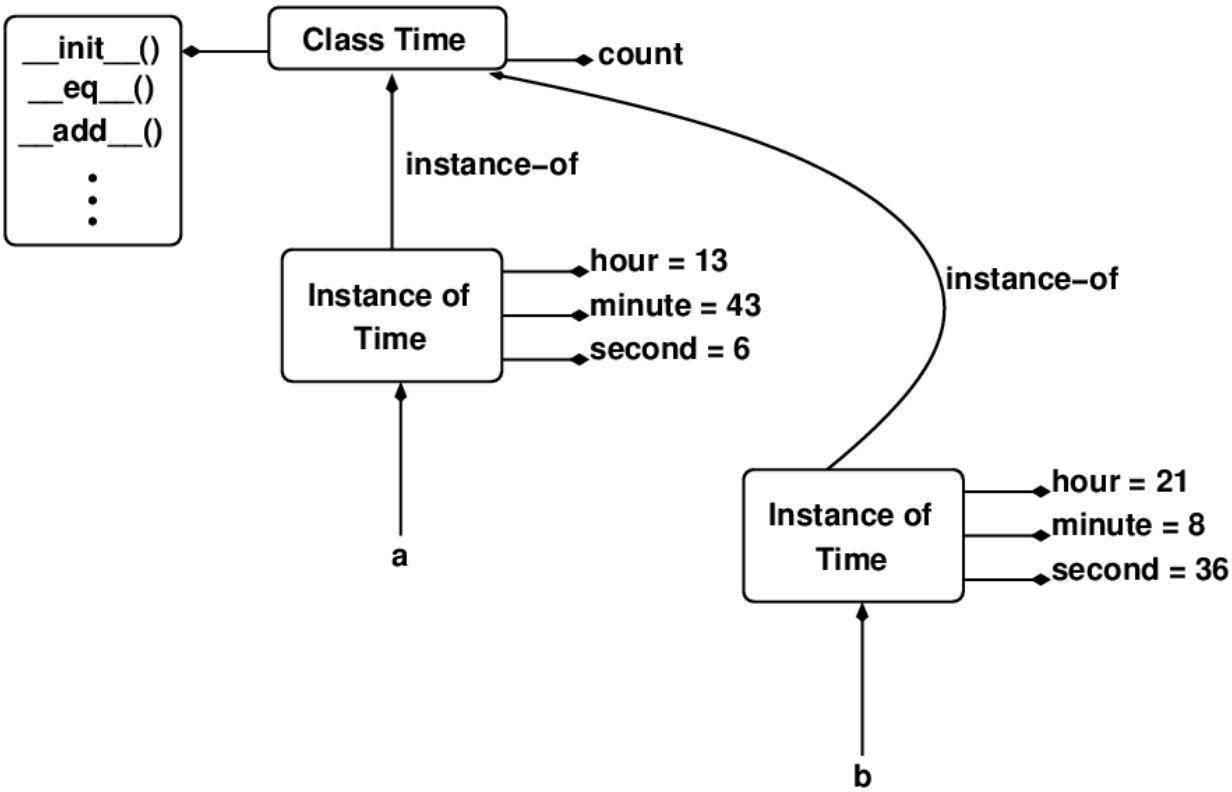
- Two lines are highlighted above. The first declares and initialises a *class variable* (or *class attribute*) called `count`. The second line increments this counter every time a `Time` object is created

(every time a `Time` object is created the `__init__()` method is executed).

- Below is a demonstration of our new class variable in action. Note how we can access a class variable even *before* any instances of the class have been instantiated. Also note how we can access the class variable both through the class name and through the instance:

```
>>> from time_v13 import Time
>>> Time.count # We can access class variables before any instances created
0
>>> t1 = Time()
>>> t2 = Time()
>>> Time.count # How many Time objects have we created?
2
>>> t3 = Time()
>>> t4 = Time()
>>> t5 = Time()
>>> t5.count # How many Time objects have we created?
5
```

- Note that since there is only a single copy of any class variable, each is shared by all instances of the class. Any change to a class variable is visible to and affects all instances. The following diagram illustrates the difference between a shared class variable vs. instance-specific instance variable:



- We can think of both class variables (and methods) as being attached to classes rather than to objects. Note even *instance methods* should be thought of as being attached to a class (even though they act only on the instance of the class passed to them in the `self` parameter).

- Class variables may also serve as class-wide constants:

```
# circle_v01.py
class Circle(object):

    Pi = 3.14159

    def __init__(self, radius):
```

```
        self.radius = radius

    def area(self):
        return Circle.Pi * self.radius**2

    def circumference(self):
        return 2 * Circle.Pi * self.radius
```

```
>>> from circle_v01 import Circle
>>> Circle.Pi # Access class variable directly
3.14159
>>> c = Circle(10)
>>> c.area()
314.159
>>> c.circumference()
62.8318
```

Class methods

- Class variables can be accessed before we have an instance of the class. So too *class methods* can be invoked before we have an instance of the class.
- Remember our `seconds_to_time()` function? We had considered including it in the `Time` class as an instance method but that did not make sense because its code was independent of any instance: we should be able to call `t2 = seconds_to_time(1000)`. Making it an instance method would require we invoke it on an instance as follows: `t2 = t1.seconds_to_time(1000)`. That would require us creating a dummy instance `t1` in order to call the method. That would be silly!
- If class methods can be invoked in the absence of an instance and we want to invoke `seconds_to_time()` in the absence of an instance then it seems clear `seconds_to_time()` is a candidate for a class method.
- By default all methods defined in a class are *instance methods* whose first parameter is `self`. A class method's first parameter, however, is not an object but the class itself. This first parameter is called `cls` by default. We use a *decorator* to mark a method as a class method. Here is the updated `Time` class where we have turned `seconds_to_time()` into a class method:

```
# time_v14.py
class Time(object):

    # class variable shared by all instances
    count = 0

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
        Time.count += 1

    def __eq__(self, other):
        return ((self.hour, self.minute, self.second) ==
                (other.hour, other.minute, other.second))

    def __add__(self, other):
        return (self.seconds_to_time(self.time_to_seconds() +
                                     other.time_to_seconds()))

    def __gt__(self, other):
```

```
        return self.time_to_seconds() > other.time_to_seconds()

    def __iadd__(self, other):
        z = self + other
        self.hour, self.minute, self.second = z.hour, z.minute, z.second
        return self

    def __str__(self):
        return ('The time is {:02d}:{:02d}:{:02d}'.format(self.hour,
                                                          self.minute,
                                                          self.second))

    def time_to_seconds(self):
        return self.hour * 60 * 60 + self.minute * 60 + self.second

    @classmethod
    def seconds_to_time(cls, s):
        minute, second = divmod(s, 60)
        hour, minute = divmod(minute, 60)
        overflow, hour = divmod(hour, 24)
        return cls(hour, minute, second)
```

- Because a class method is permanently *bound* to its class we can invoke such a method through an instance or through a class. In the `__add__()` method the `seconds_to_time()` class method is invoked through the `self` instance.
- Interestingly, the `cls` parameter is used by the `seconds_to_time()` method as a function to build an instance of the corresponding class (here `Time`) a reference to which is returned to the caller.
- The following demonstration shows our new class method in action:

```
>>> from time_v14 import Time
>>> t1 = Time.seconds_to_time(1000)
>>> print(t1)
The time is 00:16:40
>>> t2 = Time(0,3,20)
>>> t3 = t1 + t2
>>> print(t3)
The time is 00:20:00
```

Static methods

- In our class method above we made use of the `cls` parameter. If a method is associated with a class but needs neither access to the class nor an instance we can make it a *static method*.
- Below we have added a static method to our `Time` class that validates its `hour`, `minute` and `second` parameters are fit to form a valid `Time`. You can think of static methods just like ordinary functions: Python will not auto-magically supply any extra arguments when a static method is invoked. What you see is what you get (unlike with class methods and instance methods).

```
# time_v14.py
class Time(object):

    # class variable shared by all instances
    count = 0

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
```

```
        self.second = second
        Time.count += 1

    def __eq__(self, other):
        return ((self.hour, self.minute, self.second) ==
                (other.hour, other.minute, other.second))

    def __add__(self, other):
        return (self.seconds_to_time(self.time_to_seconds() +
                                     other.time_to_seconds()))

    def __gt__(self, other):
        return self.time_to_seconds() > other.time_to_seconds()

    def __iadd__(self, other):
        z = self + other
        self.hour, self.minute, self.second = z.hour, z.minute, z.second
        return self

    def __str__(self):
        return ('The time is {:02d}:{:02d}:{:02d}'.format(self.hour,
                                                          self.minute,
                                                          self.second))


    def time_to_seconds(self):
        return self.hour * 60 * 60 + self.minute * 60 + self.second

    @classmethod
    def seconds_to_time(cls, s):
        minute, second = divmod(s, 60)
        hour, minute = divmod(minute, 60)
        overflow, hour = divmod(hour, 24)
        return cls(hour, minute, second)

    @staticmethod
    def validate_time(hour, minute, second):
        return 0 <= hour <= 23 and 0 <= minute <= 59 and 0 <= second <= 59
```

- We can invoke a static method either through the class or through an instance of the class as the following demonstrates:

```
>>> from time_v14 import Time
>>> Time.validate_time(10,10,20)
True
>>> Time.validate_time(10,10,50)
True
>>> Time.validate_time(10,10,70)
False
>>> Time.validate_time(25,10,10)
False
>>> t1 = Time.seconds_to_time(32654)
>>> t1.hour
9
>>> t1.minute
4
>>> t1.second
14
>>> t1.validate_time(t1.hour, t1.minute, t1.second)
True
```

Be careful when modifying class variables

- You might be tempted to write the `__init__()` method as follows:

```
# time_v15.py
class Time(object):

    count = 0 # class variable shared by all instances

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
        self.count += 1
```

- Let's try it and see if it works:

```
>>> from time_v15 import Time
>>> t1 = Time()
>>> t2 = Time()
>>> t3 = Time()
>>> Time.count # Huh?!
0
>>> t1.count # Huh?!
1
>>> t2.count # Huh?!
1
>>> t2.count # Huh?!
1
```

- Hmm. There is something strange going on. It seems that `self.count += 1` has *a very different effect* to `Time.count += 1`. Why is that? Well `self.count += 1` is really `self.count = self.count + 1`. The class variable count, being an integer, is an *immutable* type. The `self.count` on the right hand side is the class variable `count` which is initialised to zero. When we *assign* to `self.count` on the left hand side however we *create a new data attribute* attached to the object `self`. The class variable `count` is never modified and remains zero indefinitely. This is not the behaviour we want.

- Below we again attempt to update a class variable through an instance. Ensure you understand the resulting behaviour:

```
>>> from time_v14 import Time
>>> t1 = Time()
>>> t2 = Time()
>>> t3 = Time()
>>> Time.count
3
>>> t1.count
3
>>> t2.count
3
>>> t3.count
3
>>> t1.count += 1 # attaches a new data attribute to the t1 instance
>>> Time.count
3
>>> t1.count
4
>>> t2.count
3
>>> t3.count
3
```

Class variables and local variables

- You might be tempted to write the `__init__()` method as follows:

```
# time_v16.py
class Time(object):

    count = 0 # class variable shared by all instances

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
        count += 1
```

- Will it work? Let's see:

```
>>> from time_v16 import Time
>>> t1 = Time()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./time_v16.py", line 10, in __init__
    count += 1
UnboundLocalError: local variable 'count' referenced before assignment
```

- As you can see this approach produces an error when we try to create a `Time` instance. Why? Well `count += 1` is really `count = count + 1`. The `count` here is a *local variable* and is totally unrelated to the class variable of the same name. Thus when Python tries to read `count` on the right hand side it is reading from an uninitialised local variable which produces the error message shown above.