Lecture 8.2 : Object-oriented programming: More special methods

Testing objects for equality with == ¶

- Look at the following demonstration of some rather surprising behaviour:

```
>>> from time_v09 import Time
>>> t1 = Time(13,30,00)
>>> t2 = Time(13,30,00)
>>> t1 == t2
False
>>> t1 is t2
False
```

- When dealing with *user-defined classes*, such as the `Time` class above, the `==` operator tests whether two references are equal i.e. whether two references point to the same object. This means that for user-defined types the *default* behaviour of the `==` operator is identical to that of the `is` operator.

- Can we fix it so that when we write `t1 == t2` as above to compare two objects of the `Time` class, that instead of the default behaviour which compares two references for equality, we compare the *contents* of the two objects for equality? In other words, can we *override* the default behaviour of the `==` operator such that when we compare two `Time` objects with `==` it is a *user-defined method* of the `Time` class that is invoked? The answer (you probably guessed it already) is yes!

- After overriding the behaviour of the `==` operator its behaviour depends on the objects it compares: if we compare two `Time` objects one method runs but if we compare two objects of a different type e.g. `Dates` then a different method runs. This is *operator overloading* where an operator has numerous semantics depending on its operands. Operator overloading is a form of *polymorphism* since the behaviour of an operator depends on its operands.

Operator overloading

- If a class defines an `__eq__()` method then that method is invoked when we use the `==` operator to compare two instances of that class. The `__eq__()` method is a *special method* (like `__init__()`) in that it is not normally called directly (a fact hinted at by the double underscore prefix and suffix). If we add such a method to our `Time` class we get:

```
# time_v10.py
class Time(object):

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

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

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

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

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

    def increment(self, other):
        z = self.plus(other)
        self.hour, self.minute, self.second = z.hour, z.minute, z.second

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

- Now when we compare two `Time` objects with the == operator we observe the desired behaviour:

```
>>> from time_v10 import Time
>>> t1 = Time(13,30,00)
>>> t2 = Time(13,30,00)
>>> t1 == t2 # Invokes Time.__eq__(t1, t2)
True
>>> t1 is t2
False
```

- Hmm. This is interesting. We have just seen how operator overloading can be used to overload the == operator. Can we implement other special methods so that when we use operators like +, -, +=, -=, >, >=, <, <=, etc. with our objects that it is these methods that are invoked? If it were possible then we could replace our `plus()`, `is_later_than()` and `increment()` methods with the more intuitive +, > and += operators. It turns out that, as usual, the answer is yes! There is a large collection of special methods which when implemented will overload (i.e. add special meaning to) every operator you can think of. For example
    - Method `__add__()` overloads + (handles `t1 + t2`)
    - Method `__iadd__()` overloads += (handles `t1 += t2`)
    - Method `__sub__()` overloads - (handles `t1 - t2`)
    - Method `__isub__()` overloads -= (handles `t1 -= t2`)
    - Method `__mul__()` overloads * (handles `t * 2`)
    - Method `__imul__()` overloads *= (handles `t *= 2`)
    - Method `__rmul__()` overloads * (handles `2 * t`)
    - Method `__gt__()` overloads > (handles `t1 > t2`)
    - Method `__ge__()` overloads >= (handles `t1 >= t2`)
    - Method `__lt__()` overloads < (handles `t1 < t2`)
    - Method `__le__()` overloads <= (handles `t1 <= t2`)
- What is the difference between `__add__()` and `__iadd__()`? Well, they each specify two parameters, `self` and `other`. `__add__()` adds `self` and `other` to produce a *new object* and returns a reference to that object to the caller. Methods that overload *in-place* operators however, like

`__iadd__()`, should avoid returning a new object. They instead modify `self` in-place (in the `__iadd__()` case this involves reaching *inside* `self` to update its contents) and return a reference to it. (See the implementations of `__add__()` and `__iadd__()` below.)

- Also of interest are methods such as `__rmul__()`. When Python sees an expression such as `t * 2` (where `t` is an instance of `Time`) it checks the left hand object for a `__mul__()` method. Provided we have implemented one it is invoked where `self` is a reference to `t` and `other` is a reference to `2`. What if Python sees an expression such as `2 * t`? Again it invokes the left hand object's `__mul__()` method. But the `__mul__()` method of `2` (an integer) does not know how to work with `Time` objects so we are in trouble. But Python does not give up. It checks whether the right hand object implements an `__rmul__()` method. If it does it is invoked where, again, `self` is a reference to `t` and `other` is a reference to `2`.

- Special methods are documented here: https://docs.python.org/3/reference/datamodel.html#special-method-names.

```
# time_v11.py
class Time(object):

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

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

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

- Here is what we can now do with our `Time` objects thanks to operator overloading:

```
>>> from time_v11 import Time
>>> t1 = Time(12,0,0)
>>> t2 = Time(0,0,1)
>>> t1 == t2 # Invokes Time.__eq__(t1, t2)
False
```

```
>>> t1 != t2
True
>>> t1 > t2 # Invokes Time.__gt__(t1, t2)
True
>>> t1 < t2 # Invokes Time.__gt__(t2, t1)
False
>>> t2 > t1 # Invokes Time.__gt__(t2, t1)
False
>>> t2 < t1 # Invokes Time.__gt__(t1, t2)
True
>>> t3 = t1 + t2 # Invokes Time.__add__(t1, t2)
>>> print(t3)
The time is 12:00:01
>>> print(t2)
The time is 00:00:01
>>> print(t1)
The time is 12:00:00
>>> t1 += t2 # Invokes Time.__iadd__(t1, t2)
>>> print(t1)
The time is 12:00:01
>>> print(t2)
The time is 00:00:01
```

Everything in Python is an object

- Everything in Python is an object. When we ask Python to evaluate `3 + 4` it is easy to forget we are working with objects. The following illustrates that even in this simple example we are invoking methods on integer objects:

```
>>> 3 + 4
7
>>> (3).__add__(4)
7
```