Lecture 8.1 : Object-oriented programming: More instance methods

Adding yet another instance method

- So far none of our methods have modified any of the instances passed to them (apart from `__init__()`). Let's change that now by writing an instance method that modifies the `Time` instance it is invoked on. The method increments a time by adding to it another time (it does not return anything). If we print the `Time` object before and after invoking the method on it we should find that it differs by the amount of time specified in the second parameter. Here is our first attempt at writing such a method:

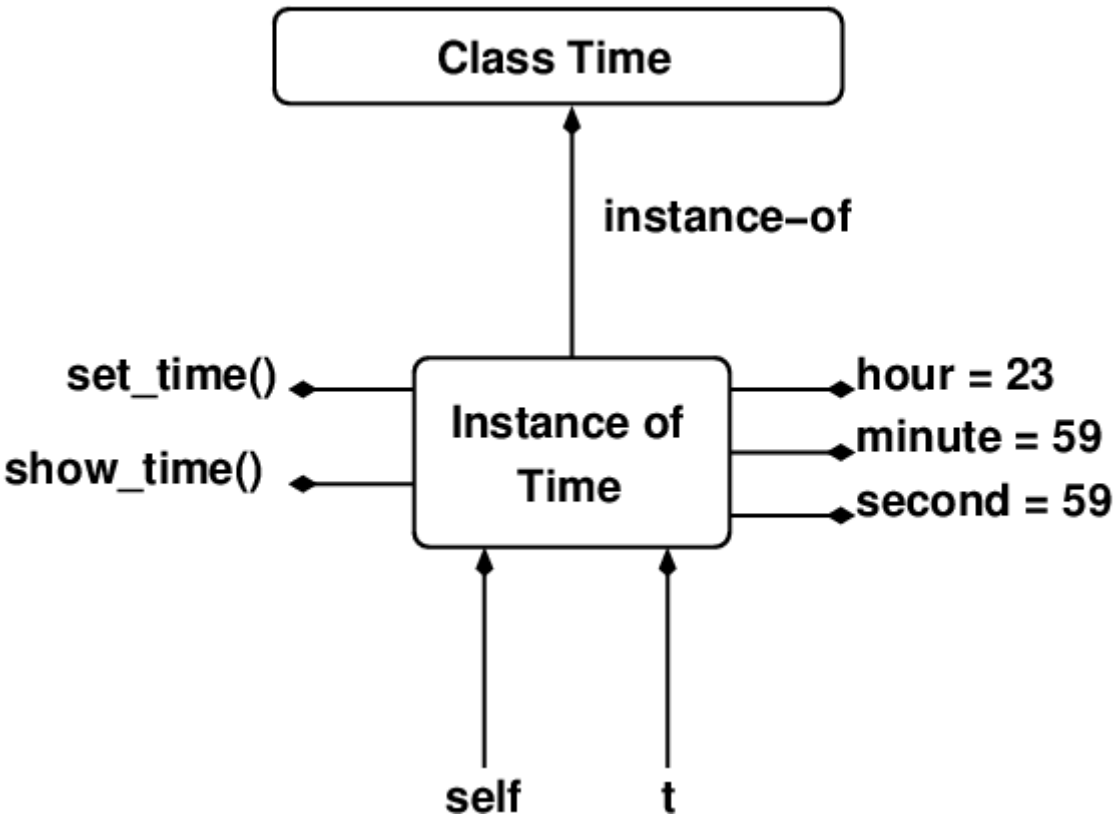
```
# time_v08.py
class Time(object):

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

    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)
```

- We can see what this new highlighted method is trying to do. We pass to it a `Time` to be incremented in `self`. In `other` we pass by how much we want `self` to be incremented. The method adds the two times together (by calling the instance method `plus()` which handles any wrap-around issues) to produce a new `Time` object `t`. Finally we *overwrite* `self` with a reference to this new `Time` object `t`. Will this new method work? Well let's try it and see:

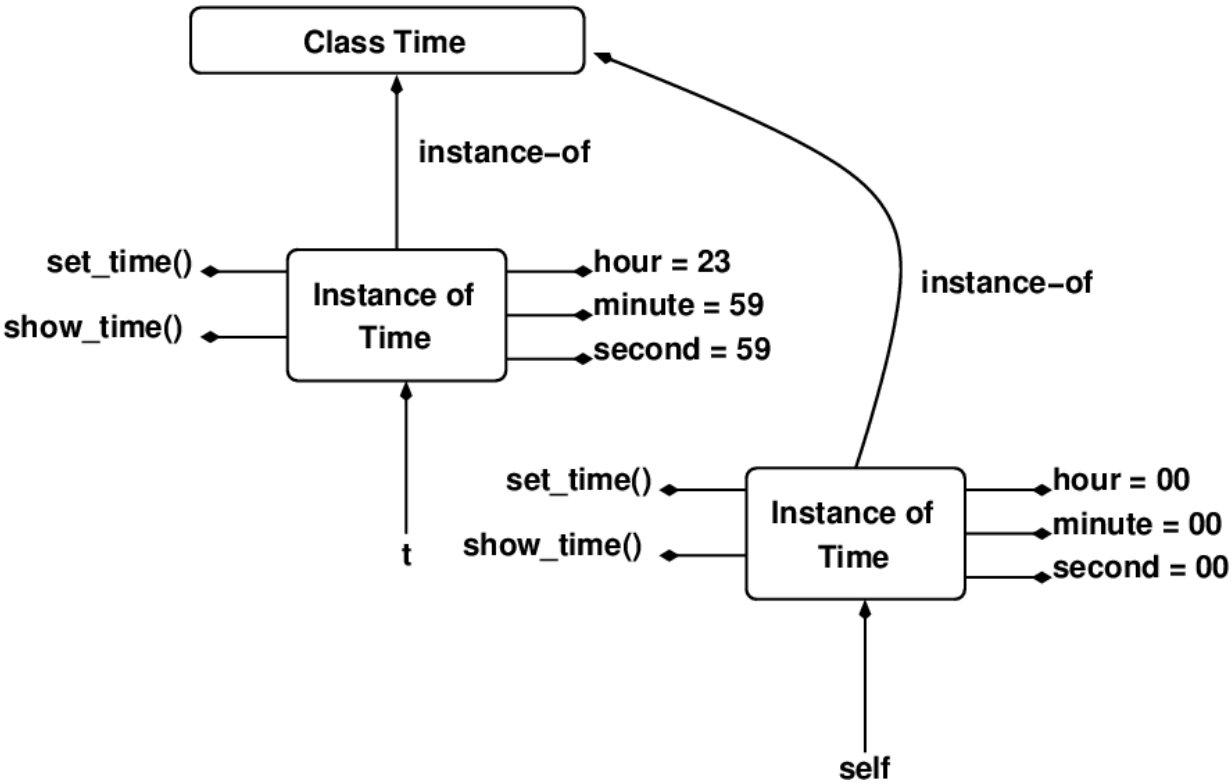
```
>>> from time_v08 import Time
>>> t = Time(23, 59, 59)
>>> i = Time(0, 0, 1)
>>> t.increment(i)
```

```
>>> print(t)
The time is 23:59:59
```

- Well that's disappointing! What is going on? Why is `t` unchanged after invoking the `increment()` method? `t` should now be `00:00:00` but our method has had no effect on it. The following diagram represents the situation on entering the `increment()` method:



- This diagram represented the situation on leaving the `increment()` method:



- When `increment()` is invoked, `self` becomes a copy of `t`. Thus `self` and `t` both reference the same object. When the method executes `self = z`, however, `self` is overwritten to point to a

new `Time` object `z`. Note however that `t` *still points to the original object* and this object remains unchanged. Thus when we print it we get back the original time.

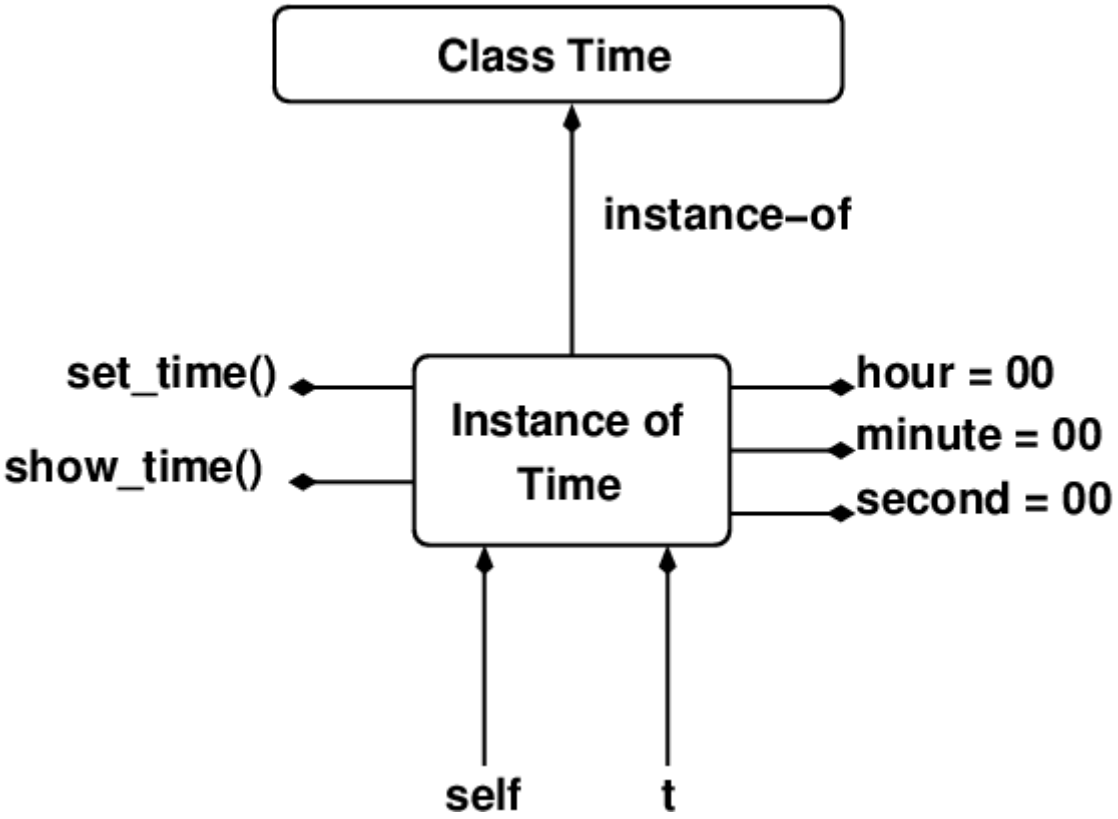
- To update the `t` object via the `increment()` method we must write *through* `self` in order to update the object that both `t` and `self` point to. What we cannot do is *overwrite* `self` because doing so will cause a new object to be created (one that is unrelated to `t`). Below we write *through* `self` to update its attributes and in so doing we update the attributes of `t` (since `self` and `t` are aliases for the same object):

```
# time_v09.py
class Time(object):

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

- Let's verify this version works as intended:

```
>>> from time_v09 import Time
>>> t = Time(23, 59, 59)
>>> i = Time(0, 0, 1)
>>> t.increment(i)
>>> print(t)
The time is 00:00:00
```

- That's more like it! We can represent this version with the following diagram:



A simple clock

- Let's use our class to implement a simple clock. We initialise a `Time` object to the current time and then proceed to increment it every second. Upon each increment we will print out the current time. To have our program sleep for one second between updates we make use of the `sleep()` function in Python's `time` module. To get the current time we use of the `now()` method in the `datetime` class of the `datetime` module. This is what the program looks like:

```
# clock.py
from time_v09 import Time
from time import sleep
from datetime import datetime

# Get the current time
now = datetime.now()

# Instantiate a Time object with the current time
t = Time(now.hour, now.minute, now.second)

# Instantiate a Time object with the increment
i = Time(0, 0, 1)

# Loop forever
while True:

    # Display the current time
    print(t)

    # Sleep for one second
    sleep(1)

    # Increment the time by one second
    t.increment(i)
```

Public and private attributes

- Consider the following bank account implementation where it is the job of the highlighted code to prevent balances going negative as the following example demonstrates:

```
class BankAccount(object):

    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if self.balance - amount >= 0:
            self.balance -= amount
        else:
            print('Insufficient funds available')

    def __str__(self):
        return 'Your current balance is: {:.2f} euro'.format(self.balance)
```

```
>>> b = BankAccount(100)
>>> b.balance
100
>>> b.withdraw(50)
>>> b.balance
50
>>> b.withdraw(100)
Insufficient funds available
```

- Unfortunately there is nothing to stop us manually reaching inside a `BankAccount` instance and setting the balance to a negative number:

```
>>> b.balance
50
>>> b.balance -= 100
>>> b.balance
-50
>>> b.balance -= 1000000
>>> b.balance
-1000050
```

- Clearly this is not ideal. We would like to protect the integrity of an object's data attributes. Can we stop users of an object going behind the class's back and changing an object's attributes without its knowing? Ideally we would like all attribute updates to go through methods.
- Our problem is that our object's attributes are currently all *public* meaning they can be accessed outside of methods. We want them to be *private* i.e. we want to indicate to users of the class that certain attributes should only be accessed by methods defined in the class.
- In Python, to mark an attribute as being *private* we simply prepend an underscore to its name:

```
class BankAccount(object):

    def __init__(self, balance=0):
        self._balance = balance
```

```
    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        if self._balance - amount >= 0:
            self._balance -= amount
        else:
            print('Insufficient funds available')

    def __str__(self):
        return 'Your current balance is: {:.2f} euro'.format(self._balance)
```

- Methods are also attributes. Thus they can also be made private by prepending an underscore to their name:

```
class BankAccount(object):

    def __init__(self, balance=0):
        self._balance = balance

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        if self._balance - amount >= 0:
            self._balance -= amount
        else:
            print('Insufficient funds available')

    def __str__(self):
        return 'Your current balance is: {:.2f} euro'.format(self._balance)

    def _bonus(self):
        self._balance += 1000000
```