Lecture 7.1: Introducing object-oriented programming

Classes and objects we have already met

- We have so far been programming with various built-in types: booleans, integers, floats, strings, lists, tuples, dictionaries, sets, etc. These are all *class types* meaning any particular example of a boolean, integer, float, etc. is an *object* of the *class* boolean, integer, float, etc. In other words every integer object, e.g. 5, is an *instance* of the class integer and every string object, e.g. 'daisy', is an *instance* of the class string, etc. Let's use Python to verify this is indeed the case:

```
>>> a = False
>>> b = 5
>>> c = 3.3
>>> d = 'daisy'
>>> e = [1, 2, 3]
>>> f = (1, 2, 3)
>>> g = {'name':'Joe'}
>>> h = {1, 2, 3}
>>> type(a)                # a is an object of the 'bool' class
<class 'bool'>
>>> isinstance(a, bool)    # a is an instance of the 'bool' class
True
>>> type(b)                # b is an object of the 'int' class
<class 'int'>
>>> isinstance(b, int)     # b is an instance of the 'int' class
True
>>> type(c)                # c is an object of the 'float' class
<class 'float'>
>>> isinstance(c, float)   # c is an instance of the 'float class
True
>>> type(d)                # d is an object of the 'str' class
<class 'str'>
>>> isinstance(d, str)     # d is an instance of the 'str' class
True
>>> type(e)                # e is an object of the 'list' class
<class 'list'>
>>> isinstance(e, list)    # e is an instance of the 'list' class
True
>>> type(f)                # f is an object of the 'tuple' class
<class 'tuple'>
>>> isinstance(f, tuple)   # f is an instance of the 'tuple' class
True
>>> type(g)                # g is an object of the 'dict' class
<class 'dict'>
>>> isinstance(g, dict)    # g is an instance of the 'dict' class
True
>>> type(h)                # h is an object of the 'set' class
<class 'set'>
>>> isinstance(h, set)     # h is an instance of the 'set' class
True
```

- *Everything in Python is an object*, or equivalently, *everything in Python is an instance of a particular class*.

Interacting with objects through methods

- To implement some operation on a particular object we invoke one of its *methods* (in other words *we invoke a method on the object*). This involves writing `object_name.method_name(method_arguments)`. We have seen this many times already but, for review purposes, let's invoke some string methods on a particular string object:

```
>>> s = 'This is a string object, an instance of the string class'
>>> s.count('a') # Invoke the count method on object s
4
>>> s.endswith('class') # Invoke the endswith method on object s
True
>>> s.find('string') # Invoke the find method on object s
10
>>> s.isnumeric() # Invoke the isnumeric method on object s
False
>>> s.split() # Invoke the split method on object s
['This', 'is', 'a', 'string', 'object,', 'an', 'instance', 'of', 'the', 'string
>>> s.swapcase() # Invoke the swapcase method on object s
'tHIS IS A STRING OBJECT, AN INSTANCE OF THE STRING CLASS'
```

- Which methods can be invoked on any particular object? Well that depends on the class of the object. It is the class that defines the behaviours, i.e. methods, that instances of that class support. The list of methods defined by a class tells us the things we can do with an instance of that class. To see a list of all the methods a particular object supports use `help(class_name)`, e.g. `help(str)` or `help(list)`.

Adding a new type to a program

- Knowledge of Python's built-in types and their capabilities is essential to being a good Python programmer. However, sometimes, in order to solve a particular problem, Python's built-in types may not entirely suit and we would like to *add our own types* to programs in order to elegantly solve a particular problem. Since everything in Python is an object, i.e. an instance of some class type, so too our new types will also be class types. How do we define a new class in Python? With the `class` keyword:

```
# myfirstclass.py
class MyFirstClass(object):
    pass
```

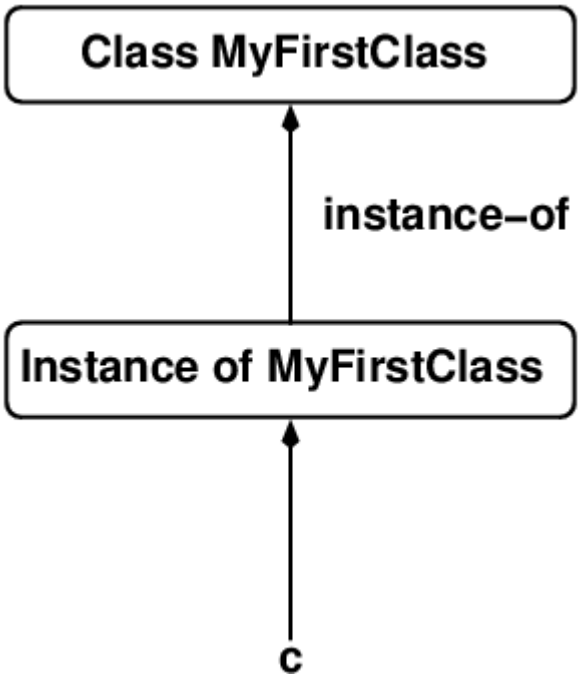
- The code above defines a new class called `MyFirstClass`. The class is empty (it contains only a placeholder `pass` statement so supports only trivial functionality) but it is still a valid class definition. (We ignore the object reference in brackets and simply accept that, for now, any of our new class definitions will include it.)

- Note that in Python `class` statements are executable. Any `class` statements in a module are executed when that module is imported. Once a `class` statement has been executed a new type is defined and objects of that class type can be *instantiated* (i.e. created). For example, given the above class definition we can do this:

```
>>> from myfirstclass import MyFirstClass
>>> c = MyFirstClass()           # Invoke the class as a function to make an c
>>> type(c)                      # c is an object of the 'MyFirstClass' class
<class 'myfirstclass.MyFirstClass'>
```

```
>>> isinstance(c, MyFirstClass)   # c is an instance of the 'MyFirstClass' clas
True
>>> print(c)                      # c is an object of the 'MyFirstClass' class
<myfirstclass.MyFirstClass object at 0x7ffa97c58a50>
```

- To create an object of type `MyFirstClass` we call the class as if it were a function (see the highlighted code above). Calling the class as a function will instantiate an instance of that class and return a reference to it. Here we assign the reference to a variable *c*. Inspecting the type of the object *c* shows it is indeed an instance of the `MyFirstClass` class. We can represent the situation as follows:



Adding another new type to a program

- Let's define a (slightly more useful) class to represent the time in 24-hour format. This class defines a method (*functions* inside class definitions are called *methods*) that allows us to set the time on `Time` objects:

```
# time_v01.py
class Time(object):

    def set_time(time_object, hour, minute, second):
        time_object.hour = hour
        time_object.minute = minute
        time_object.second = second
```

- The `set_time` method requires four arguments be passed to it in order for it to do its job:

    1. The `time_object` whose time we are setting,
    2. the `hour` we want to set on the `time_object`,
    3. the `minute` we want to set on the `time_object`,
    4. the `second` we want to set on the `time_object`.
- As illustrated by the highlighted code above, we access the hour, minute and second *attributes* of the `time_object` using the period operator, e.g. `time_object.minute = minute`. Do not confuse the `hour`, `minute` and `second` in the parameter list with `time_object.hour`,

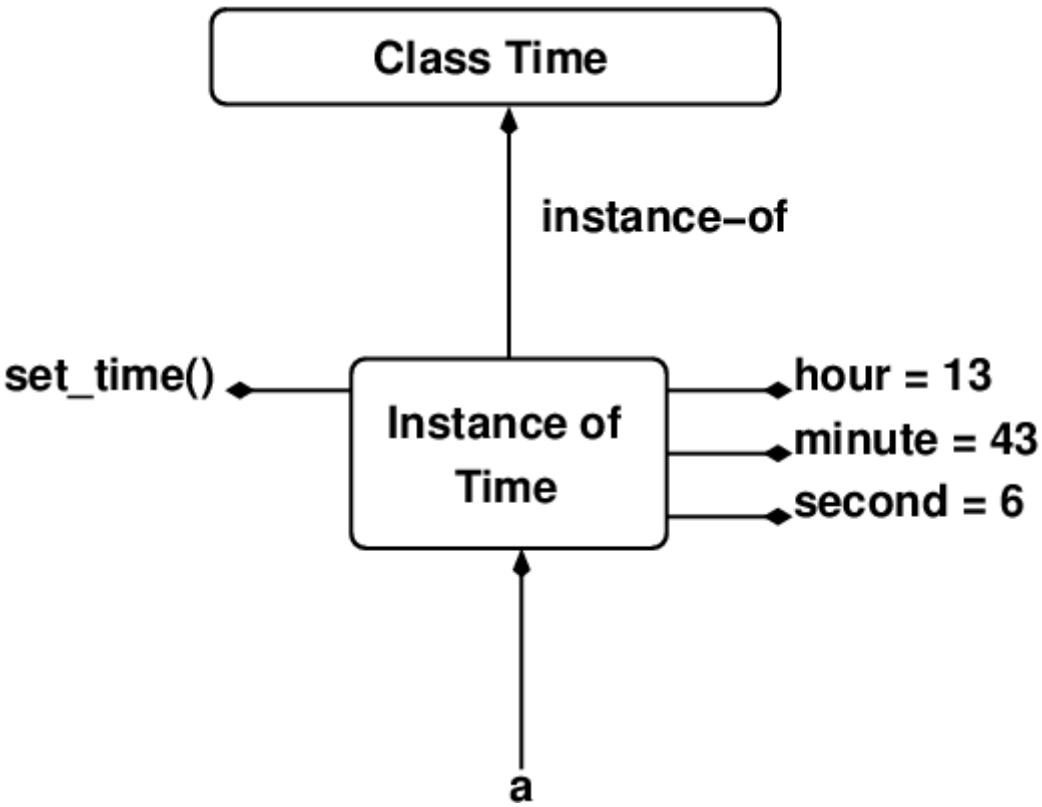
`time_object.minute` and `time_object.second`. The former are local variables while the latter are attributes attached to the `time_object`. There is no name clash.

- How can we use this new method? We can do so by writing `class_name.method_name(method_arguments)`. Here is an example:

```
>>> from time_v01 import Time
>>> a = Time()
>>> Time.set_time(a, 13, 43, 6)
>>> print(a.hour)
13
>>> print(a.minute)
43
>>> print(a.second)
6
```

- The highlighted line invokes the `set_time` method of the `Time` class on the `a` object. The `a` argument to `set_time` becomes `time_object` parameter in the method definition (`13` becomes the `hour`, `43` the `minute` and `6` the `second`). When the method executes it updates a single object, in this case the object referenced by `a`. It updates the object by assigning values to its `hour`, `minute` and `second` attributes.

- Once the code highlighted above has executed we can represent the resulting situation with the following diagram where attributes and (instance) methods are attached to objects:



- Some terminology: the `set_time()` method is called an *instance method* because it acts upon an instance of the class. The instance it acts on will always be the first parameter of the method.

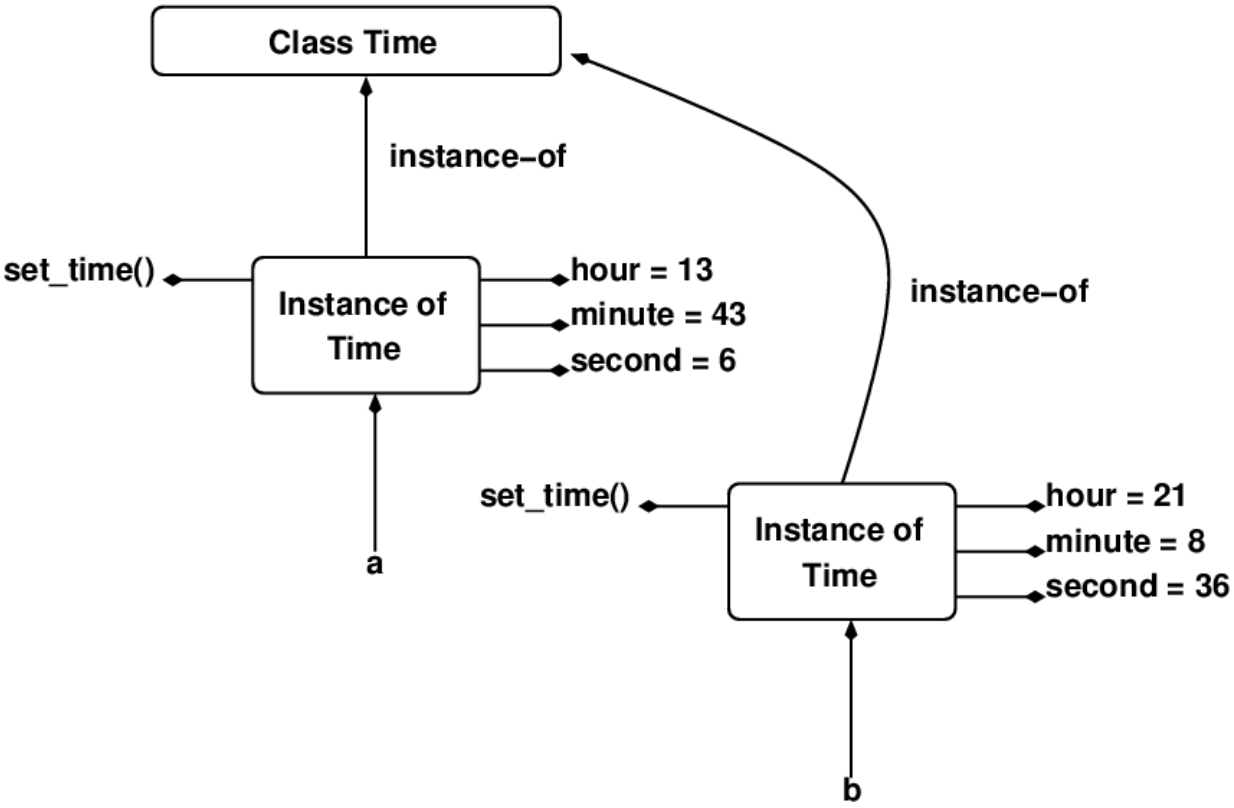
- Some more terminology: `time_object.hour`, `time_object.minute` and `time_object.second` are called *instance variables* because they are variables attached to a particular instance of a `Time` object. They are also referred to as an object's *data attributes*.

Multiple objects of the same type

- There is nothing to stop us making multiple objects of type `Time`. *Crucially, attached to each object is its own distinct set of instance variables (data attributes)*. For example:

```
>>> from time_v01 import Time
>>> a = Time()
>>> Time.set_time(a, 13, 43, 6)
>>> print(a.hour)
13
>>> print(a.minute)
43
>>> print(a.second)
6
>>> b = Time()
>>> Time.set_time(b, 21, 8, 36)
>>> print(b.hour)
21
>>> print(b.minute)
8
>>> print(b.second)
36
>>> c = Time()
>>> Time.set_time(c, 4, 44, 51)
>>> print(c.hour)
4
>>> print(c.minute)
44
>>> print(c.second)
51
```

- Above we instantiate three objects, `a`, `b` and `c`, of the class `Time`. We can create as many instances of a class as we wish. For this reason a class is sometimes referred to as an *object factory*. A class definition serves as a *blueprint* for generating objects. As we have seen, a new object is generated whenever we call the class as a function.
- Note again how the object being updated serves as an argument to the `set_time` method. Thus each time the latter method is invoked above it is setting the data attributes of a *different object*.
- We can represent the situation (for `a` and `b`) as follows:



Adding a method to print the time

- Let's add a function that prints the time. Where should we add this function? Since the `Time` class encapsulates everything related to processing `Time` objects it makes sense to add it there. This is what *object oriented programming* is about. Our `Time` class is where we bundle together all `Time`-related data and functions. Since functions in a class are called methods, our new function is really a method. Here is the resulting expanded class, now boasting two methods:

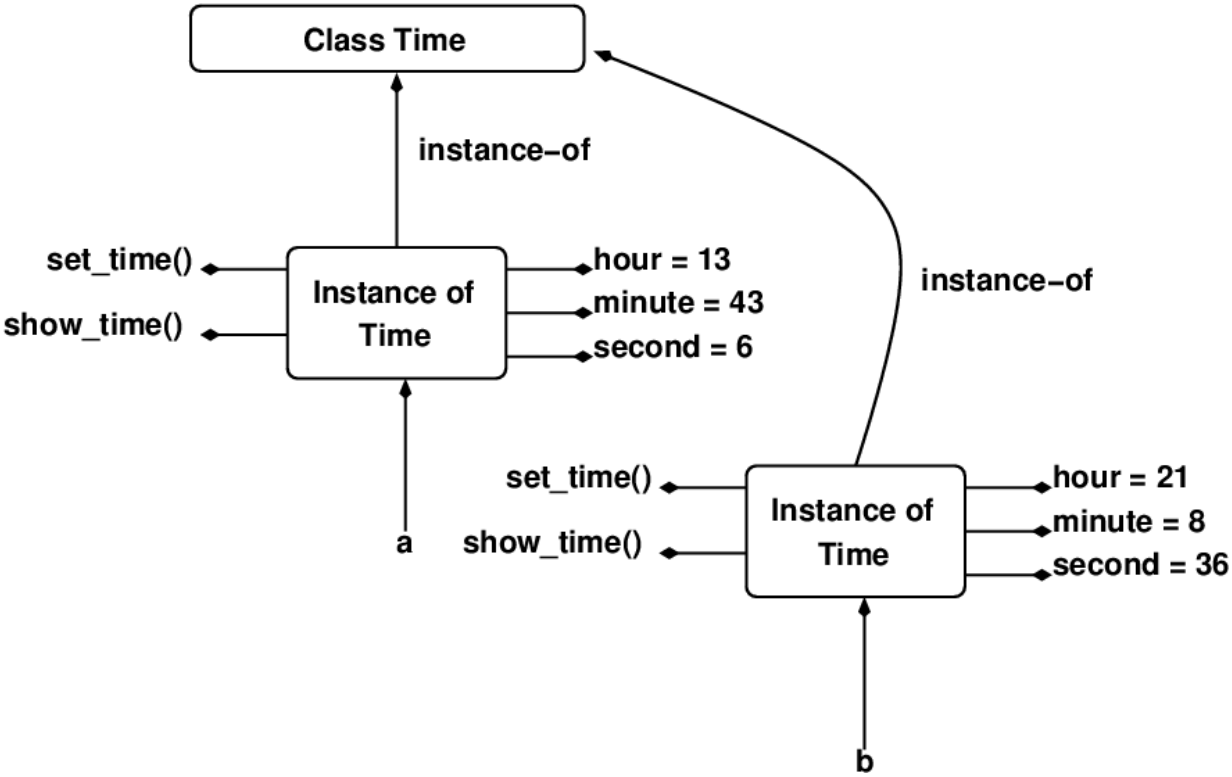
```
# time_v01.py
class Time(object):

    def set_time(time_object, hour, minute, second):
        time_object.hour = hour
        time_object.minute = minute
        time_object.second = second

    def show_time(time_object):
        print('The time is {:02d}:{:02d}:{:02d}'.format(time_object.hour,
                                                        time_object.minute,
                                                        time_object.second))
```

- This method specifies a single parameter i.e. the object's whose time we want to print. Now we can print an object's time as follows:

```
>>> from time_v01 import Time
>>> a = Time()
>>> Time.set_time(a, 13, 43, 6)
>>> Time.show_time(a)
The time is 13:43:06
>>> b = Time()
>>> Time.set_time(b, 21, 8, 36)
>>> Time.show_time(b)
The time is 21:08:36
>>> c = Time()
```

```
>>> Time.set_time(c, 4, 44, 51)
>>> Time.show_time(c)
The time is 04:44:51
```

- We can depict the change to our objects as follows where a new instance method called `show_time()` is attached to each:



Is there a handier way to call methods on an object?

- The way we are invoking methods on our `Time` objects requires we name both the class (e.g. `Time`) and the object (e.g. `a`) as shown below:

```
>>> Time.show_time(a)
The time is 13:43:06
```

- Remember how we earlier invoked methods on string (and other) objects? It seemed handier because it did not require us naming the `str` class and, interestingly, the string object `s` did not *appear* to be an argument for the method:

```
>>> s = 'This is a string object, an instance of the string class'
>>> s.count('a') # Invoke the count method on object s
4
```

- It turns out, however, that `s.count('a')` is really just shorthand for `str.count(s, 'a')`. We can adopt the same approach with objects of our `Time` class i.e. rather than calling the `Time` class's methods by explicitly referencing the class name we can instead invoke our methods directly on an object of the `Time` class:

```
>>> from time_v01 import Time
>>> a = Time()
```

```
>>> a.set_time(13, 43, 6) # Equivalent to: Time.set_time(a, 13, 43, 6)
>>> a.show_time()         # Equivalent to: Time.show_time(a)
The time is 13:43:06
>>> b = Time()
>>> b.set_time(21, 8, 36) # Equivalent to: Time.set_time(b, 21, 8, 36)
>>> b.show_time()         # Equivalent to: Time.show_time(b)
The time is 21:08:36
>>> c = Time()
>>> c.set_time(4, 44, 51) # Equivalent to: Time.set_time(c, 4, 44, 51)
>>> c.show_time()         # Equivalent to: Time.show_time(c)
The time is 04:44:51
```

- To understand how to write and call methods it is vital you appreciate that:

    0. `a.set_time(13, 43, 6)` is equivalent to `Time.set_time(a, 13, 43, 6)`,
    1. `a.show_time()` is equivalent to `Time.show_time(a)`,
    2. `b.set_time(21, 8, 36)` is equivalent to `Time.set_time(b, 21, 8, 36)`,
    3. `b.show_time()` is equivalent to `Time.show_time(b)`,
    4. `c.set_time(4, 44, 51)` is equivalent to `Time.set_time(c, 4, 44, 51)`,
    5. `c.show_time()` is equivalent to `Time.show_time(c)`,
    6. etc.
- Note when we call `a.show_time()` or `a.set_time(13, 43, 6)` the `show_time` and `set_time` methods still require the object `a` be passed as an argument. **Thus when we invoke an instance method on a Python object that object is automatically supplied as the first argument to the method**. This means that whenever we invoke an instance method on an object **we supply one fewer arguments than the number of parameters listed in the method definition** inside the class. We do so because Python automatically supplies the missing object argument on our behalf.
- What is *an instance method*? An instance method is one that acts upon a particular instance of an object. It is a method whose first parameter is the object operated upon. In our `Time` class both methods `set_time` and `show_time` are instance methods and the first parameter of each is a `time_object`. A class's methods are by default instance methods whose first parameter will always be the instance on which the method was invoked.
- By convention the first parameter of instance methods is named `self`. Thus our `Time` class should really be implemented as follows:

```
# time_v02.py
class Time(object):

    def set_time(self, hour, minute, second):
        self.hour = hour
        self.minute = minute
        self.second = second

    def show_time(self):
        print('The time is {:02d}:{:02d}:{:02d}'.format(self.hour,
                                                       self.minute,
                                                       self.second))
```

- We will from now on write and invoke our instance methods as outlined in this section.