# Lecture 14 - Classes and Methods

## Week 6 Friday


## Miles Chen, PhD


Taken directly from Chapters 17 of Think Python by Allen B Downey

# Methods

Methods are functions that are associated with a particular class.

Methods are the same as functions, but there are two key differences:

- Methods are defined inside a class definition.
- The syntax for invoking a method is different from the syntax for calling a function.

# Revisiting The Time Class

In the previous chapter/lecture we created a class `Time`.

We also wrote a function called `print_time()`

In [1]:
```
class Time:
    """Represents the time of day."""
```

In [2]:
```
def print_time(time):
    print("%.2d:%.2d:%.2d" % (time.hour, time.minute, time.second))
```

To call the function, we had to create a `Time` object and pass it as an argument.

In [3]:
```
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
print_time(start)
```

```
09:45:00
```

# Making a method

To make print_time a method, all we have to do is move the function definition inside the class definition.

Note the indentation.

In [4]:
```
class Time:
    def print_time(time):
        print("%.2d:%.2d:%.2d" % (time.hour, time.minute, time.second))
```

Now that we have redefined the class with a method defined inside, we can call the method.

Note, I have to re-create start as a `Time` class object because the old start object was created under the old definition of class `Time`. The changes don't apply retroactively.

In [5]:
```
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
```

# Calling the Method

There are two ways to call a method, but in practice, most people use one.

The less common way is to use the Class name and pass an object of that class to the method.

In [6]:
```
Time.print_time(start)
```

```
09:45:00
```

The more common way is to call the method directly from the object itself using dot notation.

In [7]:
```
start.print_time()
```

```
09:45:00
```

When you call a method from the object itself, the object is known as the **subject** of the method.

The subject gets passed to the method as the first argument.

So in our example above, the object `start` gets passed as the first argument in thge method `print_time()`.

# self

By convention, the first argument of a method is called `self`.

The idea is that when you call a method from an object with dot notation, you are applying to function to itself.

With this in mind, we'ld write our `Time` class as follows.

I've also included another method that converts the time to number of seconds after midnight (time_to_int) which will be useful for adding times.

In [8]:
```
class Time:
    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hour, self.minute, self.second))

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds
```

In [9]:
```
# again, we redefine the object of class Time()
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
```

In [10]:
```
start.print_time()
```

```
09:45:00
```

In [11]:
```
start.time_to_int()
```

Out[11]:
```
35100
```

We'll use the following function in the next class definition.

It converts the number of seconds into a time.

It can't be made as a method of a time object because the argument of this function is an integer value of seconds, and there is no object to invoke the method on.

In [12]:
```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

# Adding more methods

In [13]:
```
class Time:
    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hour, self.minute, self.second))

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

In [14]:
```
# again, we redefine the object of class Time()
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
```

In [15]:
```
start.print_time()
```

```
09:45:00
```

In [16]:
```
end = start.increment(1337)
end.print_time()
```

```
10:07:17
```

# Errors with method calls

Keep in mind that when you call a method from an object, the object itself is always passed as the first argument of the method.

In [17]:
```
end = start.increment(1337, 460)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-a8fd8b8bdfbc> in <module>
----> 1 end = start.increment(1337, 460)

TypeError: increment() takes 2 positional arguments but 3 were given
```

The above call returns an error. "increment() takes 2 positional arguments but 3 were given"

It can be confusing because we see only two arguments (1337 and 460) in parentheses. We must remember that we have also passed `self` (the subject) as the first argument, so there really are three arguments.

# Methods with other class objects

We can create a method inside `Time` called `is_after()` which will check to see if one time takes place after another time.

This function takes in two `Time` class objects and compares them. I add this method to the end of our class definition.

Because we expect the argument passed to `is_after()` is another `Time` class object, we can invoke the methods of the object. By convention, the first parameter of the method is `self`, and the parameter for the other class object being passed is named `other`

In [18]:
```
class Time:
    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hour, self.minute, self.second))

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

## Calling the new method

In [19]:
```
# redefine the objects
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
start.print_time()
```

```
09:45:00
```

In [20]:
```
end = start.increment(1337)
end.print_time()
```

```
10:07:17
```

In [21]:
```
end.is_after(start)
```

Out[21]:
```
True
```

In [22]:
```
start.is_after(end)
```

Out[22]:
```
False
```

# The `init` method

It's been annoying that every time I redefine the `start` object, I have to assign the `hour`, `minute`, and `second` attributes.

There is a special initialization method called `__init__` (also called double-under init) that gets invoked whenever a new object of the class is created.

It is useful to use this method to assign values to attributes that would be used in the class. By convention, the parameters of init have the same names as the attributes.

In [23]:

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

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hour, self.minute, self.second))

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

Creating new `Time` class objects is much easier now.

In [24]:
```
midnight = Time()
midnight.print_time()
```

```
00:00:00
```

In [25]:
```
new_time = Time(9)
new_time.print_time()
```

```
09:00:00
```

The argument 9 gets passed in as the value for `hour` because `self` is always the first argument, even in the `__init__` method.

In [26]:
```
new_time = Time(9, 45)
new_time.print_time()
```

```
09:45:00
```

# The `__str__` method

`__str__` is another special method that should return a string representation of an object.

When you call `print()` on an object, Python invokes the `__str__` method.

So far we have been using the method `print_time()` which we defined inside the class.

Instead, we will modify this method to work with the `__str__` method. To view the time, we will call `print()` on the object.

Note that in the conversion, we no longer call `print` but use `return` to return a string object.

In [27]:

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

    def __str__(self):
        return "%.2d:%.2d:%.2d" % (self.hour, self.minute, self.second)

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

In [28]:
```
new_time = Time(9, 45)
```

In [30]:
```
print(new_time)
```

```
09:45:00
```

# Operator overloading

There are even more special "double-under" methods that have special uses.

One is the `__add__` method which will be invoked with the `+` operator.

I'll add the following method inside the class definition

```
def __add__(self, other):
    seconds += self.time_to_int() + other.time_to_int()
    return int_to_time(seconds)
```

In [35]:

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

    def __str__(self):
        return "%.2d:%.2d:%.2d" % (self.hour, self.minute, self.second)

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

In [36]:
```
start = Time(9, 45)
duration = Time(1, 35)
```

In [38]:
```
print(start + duration)
```

```
11:20:00
```

# Type-based dispatch

The previous defintion of `__add__` allowed us to add two `Time` class objects together. But we might also want the option to add integers as well.

We can use type-based dispatch to call different methods depending on the type of input. To perform this, we use the `isinstance()` to see if the object belongs to a particular class or not.

Inside the class definition of Time, I'll add the following:

```
def __add__(self, other):
    if isinstance(other, Time):
        return self.add_time(other)
    else:
        return self.increment(other)

def add_time(self, other):
    seconds = self.time_to_int() + other.time_to_int()
    return int_to_time(seconds)

def increment(self, seconds):
    seconds += self.time_to_int()
    return int_to_time(seconds)
```

In [39]:

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

    def __str__(self):
        return "%.2d:%.2d:%.2d" % (self.hour, self.minute, self.second)

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

In [40]:
```
start = Time(9, 45)
```

In [41]:
```
duration = Time(1, 35)
```

In [42]:
```
print(start + duration)
```

```
11:20:00
```

In [43]:
```
print(start + 1337)
```

```
10:07:17
```

# Commutative Addition

The add method as we've implemented it is not commutative.

In [44]:
```
print(1337 + start)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-44-f808e24131be> in <module>
----> 1 print(1337 + start)

TypeError: unsupported operand type(s) for +: 'int' and 'Time'
```

The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object and it doesn't know how.

If we want this to work, we have to use another special method: `__radd__` which stands for "right-side add"

We can pull this off quite easily:

```
def __radd__(self, other):
    return self.__add__(other)
```

In [45]:

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

    def __str__(self):
        return "%.2d:%.2d:%.2d" % (self.hour, self.minute, self.second)

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def __radd__(self, other):
        return self.__add__(other)

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

In [46]:
```
start = Time(9, 45)
```

In [47]:
```
duration = Time(1, 35)
```

In [48]:
```
print(start + duration)
```

```
11:20:00
```

In [49]:
```
print(start + 1337)
```

```
10:07:17
```

In [51]:
```
print(1337 + start)
```

```
10:07:17
```

# Polymorphism

Many functions will work on different types. These are known as **polymorphic** functions.

For example, the function `sum` can work on any object that support addition.

In [53]:
```
t1 = Time(1, 20)
t2 = Time(1, 40)
t3 = Time(1, 30)
total = sum([t1, t2, t3])
print(total)
```

```
04:30:00
```

# Important tips

It is legal to add attributes to objects at any time. But if you have objects of the same type that don't have the same attributes, it can cause problems.

It is recommended to initialize all of the objects attributes inside the `__init__` method.

A useful function for debugging is teh `vars()` function which will print all of the attributes an object has as a dictionary.

In [54]:
```
vars(start)
```

Out[54]:
```
{'hour': 9, 'minute': 45, 'second': 0}
```