_C11_POO-Ch16&17-FunctionsMethods_90

May 15, 2020

0.1 Pure functions vs. Modifiers

```
In [2]: class Time(object):
            """Represents the time of day.
               attributes: hour, minute, second
            """


        time = Time()
        time.hour = 11
        time.minute = 59
        time.second = 30

        def add_time(t1, t2):
            sum = Time()
            sum.hour = t1.hour + t2.hour
            sum.minute = t1.minute + t2.minute
            sum.second = t1.second + t2.second
            return sum
```

The function creates a new Time object, initializes its attributes, and *returns a reference to the new object*. This is called a **pure function** because *it does not modify any of the objects passed to it as arguments and it has no effect*, like displaying a value or getting user input, *other than returning a value*.

Sometimes it is useful for a function *to modify the objects it gets as parameters*. In that case, the *changes are visible to the caller*. Functions that work this way are called **modifiers**.

```
In [ ]: def increment(time, seconds):
            time.second += seconds

            if time.second >= 60:
                time.second -= 60
                time.minute += 1

            if time.minute >= 60:
                time.minute -= 60
                time.hour += 1
```

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. But modifiers are convenient at times, and functional programs tend to be less efficient.

In general, I recommend that you write pure functions whenever it is reasonable and resort to modifiers only if there is a compelling advantage. This approach might be called a **functional programming style**.

0.2 Prototyping / Prototype and patch / Incremental development

0.3 vs.

0.4 Planning / Planned development

For each function, we wrote **a prototype** that performed the basic calculation and then tested it, **patching** errors along the way. This approach can be effective, especially *if we don't yet have a deep understanding of the problem*. **Incremental corrections** can generate code that is *unnecessarily complicated* — since it deals with many special cases — and *unreliable* — since it is hard to know if you have found all the errors.

An alternative is **planned development**, in which *high-level insight into the problem* can make the programming much easier.

```
In [ ]: #For our Time model, the insight is that
        #a Time object is really a three-digit number in base 60.
```

This observation suggests another approach to the whole problem—we can *convert Time objects to integers* and take advantage of the fact that the computer knows how to do integer arithmetic: - converts Times to integers - converts integers to Times - rewrite add_time function accordingly - subtracting two Times to find the duration between them - …

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (time_to_int and int_to_time), *we get a program that is shorter, easier to read and debug, and more reliable*.

… Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

An invariant represents a condition that should **always** be true during the execution of a program; if they are not true, then something has gone wrong.

0.5 Function vs. Method

It is not easy to define **object-oriented programming**, but we have already seen some of its **characteristics**: - Programs are made up of **object** definitions and **function** definitions, and most of the computation is expressed in terms of **operations on objects**. - Each object definition corresponds to some object or concept in *the real world*, and the functions that operate on that object correspond to *the ways real-world objects* **interact**.

A method is a function that is associated with a particular class. In this chapter, we will define methods for user-defined types.

Methods are *semantically the same as functions*, but there are **two syntactic differences**: - Methods are defined *inside a class definition* in order to make the relationship between the class and the method explicit. - The syntax for **invoking a method** is different from the syntax for **calling a function**.

```
In [4]: class Time(object):
            """Represents the time of day."""

        def print_time(time):
            print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)

        start = Time()
        start.hour = 9
        start.minute = 45
        start.second = 00
        print_time(start)
```

```
09:45:00
```

To make print_time a method, all we have to do is **move the function definition inside the class definition**. Notice the change in indentation.

```
In [9]: #PRINTING OBJECTS
        class Time(object):
            def print_time(time):
                print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)

        start = Time()
        start.hour = 9
        start.minute = 45
        start.second = 00

        #Now there are two ways to call print_time.
        Time.print_time(start)
        start.print_time()
```

```
09:45:00
09:45:00
```

In this use of dot notation, print_time is the name of the method (again), and **start** is the object the method is invoked on, which is called **the subject**. Inside the method, **the subject is assigned to the first parameter**, so in this case *start* is assigned to *time*.

```
In [ ]: #By convention, the first parameter of a method is called self,
        #so it would be more common to write print_time like this:

        class Time(object):
            def print_time(self):
                print '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

The reason for this convention is an implicit metaphor: - The syntax for **a function call**, print_time(start), suggests that **the function is the active agent**. It says something like, “Hey print_time! Here’s an object for you to print.” - In OOP, **the objects are the active agents**. A method invocation like start.print_time() says “Hey start! Please print yourself.”

0.6 Special methods: __init__, __str__, etc

The **__init__** method (short for "initialization") is a special method that **gets invoked when** an object is **instantiated**.

```
In [11]: class Time(object):
             def __init__(self, hour=0, minute=0, second=0):
                 self.hour = hour
                 self.minute = minute
                 self.second = second
         #!!! It is common for the parameters of __init__ to have the same names as the attribut
             def print_time(self):
                 print '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)

         #The parameters are optional, in the sense that:
         #if we call Time with no arguments, we get the default values.
         #if we provide one argument, it overrides hour
         #if we provide two arguments, they override hour and minute
         #if we provide three arguments, they override all three default values.
```

__str__ is a special method, like **__init__**, that is supposed to return a string representation of an object. When we **print** an object, Python invokes the **__str__** method **BY DEFAULT**(!!!)

```
In [12]: class Time(object):
             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)

         time = Time(10,30)
         print time
```

```
10:30:00
```

When you write a new class, it is recommended to start by writing **__init__**, which makes it easier to instantiate objects, and **__str__**, which is useful for debugging.

0.7 Operator overloading

By defining **other special methods**, we can *specify the behavior of operators on user-defined types*. For example, if we define a method named **__add__** for the Time class, we can **use the + operator on Time objects**.

```
In [17]: class Time(object):
             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(time):
            minutes = time.hour * 60 + time.minute
            seconds = minutes * 60 + time.second
            return seconds

    def int_to_time(seconds):
        time = Time()
        minutes, time.second = divmod(seconds, 60)
        time.hour, time.minute = divmod(minutes, 60)
        return time

    start = Time(9, 45)
    duration = Time(1, 35)
    print start + duration

    #??? WHY time_to_int is defined as method and
    #int-_to_time is defined as function?...
```

```
11:20:00
```

... The rest of Chapter 17 –> NEXT COURSE / WEEK: Type-based dispatch, Polymorphism, Interface & implementation