Chapter_15_Classes_and_functions

March 7, 2024

0.1 16.1 Time

As another example of a programmer-defined type, we'll define a class called Time that records the time of day. The class definition looks like this:

[1]:
```
class Time:
    """Represents the time of day.
    attributes: hour, minute, second
    """
```

[2]:
```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

0.2 16.2 Pure functions

In the next few sections, we'll write two functions that add time values. They demonstrate two kinds of functions: pure functions and modifiers. They also demonstrate a development plan I'll call prototype and patch, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the complications.

Here is a simple prototype of add_time:

[3]:
```
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

To test this function, I'll create two Time objects: start contains the start time of a movie, like Monty Python and the Holy Grail, and duration contains the run time of the movie, which is one hour 35 minutes.

add_time figures out when the movie will be done.

[4]:
```
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
```

[5]:
```
duration = Time()
duration.hour = 1
duration.minute = 35
duration.second = 0
```

[9]:
```
done = add_time(start, duration)
print(done.hour,":",done.minute,":",done.second)
```

```
10 : 80 : 0
```

The result, 10:80:00 might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column. Here's an improved version:

[11]:
```
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
    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1
    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1
    return sum
```

Although this function is correct, it is starting to get big. We will see a shorter alternative later.

0.3 16.3 Modifiers

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.

[12]:
```
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
```

Is this function correct? What happens if seconds is much greater than sixty? In that case, it is not enough to carry once; we have to keep doing it until time.second is less than sixty. One solution is to replace the if statements with while statements. That would make the function correct, but not very efficient. As an exercise, write a correct version of increment that doesn't contain any loops.

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.

0.4 16.4 Prototyping versus planning

The development plan I am demonstrating is called "prototype and patch". For each function, I wrote a prototype that performed the basic calculation and then tested it, patching errors along the way.

An alternative is designed development, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is really a three-digit number in base 60 (see http://en.wikipedia.org/wiki/Sexagesimal). The second attribute is the "ones column", the minute attribute is the "sixties column", and the hour attribute is the "thirty-six hundreds column".

When we wrote add_time and increment, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

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.

Here is a function that converts Times to integers:

[13]:
```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

And here is a function that converts an integer to a Time (recall that divmod divides the first argument by the second and returns the quotient and remainder as a tuple).

[14]:
```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

[15]:
```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

This version is shorter than the original, and easier to verify

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with time values is better. 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.

[ ]: