Classes and functions

14.1. Time

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

```
class Time:
    pass
```

We can create a new Time object and assign attributes for hours, minutes, and seconds:

```
time = Time()
time.hours = 11
time.minutes = 59
time.seconds = 30
```

The state diagram for the Time object looks like this:

14.2. Pure functions

In the next few sections, we'll write two versions of a function called add_time, which calculates the sum of two Times. They will demonstrate two kinds of functions: pure functions and modifiers.

The following is a rough version of add_time:

```
def add_time(t1, t2):
    sum = Time()
    sum.hours = t1.hours + t2.hours
    sum.minutes = t1.minutes + t2.minutes
    sum.seconds = t1.seconds + t2.seconds
    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 parameters and it has no side effects, such as displaying a value or getting user input.

Here is an example of how to use this function. We'll create two Time objects: current_time, which contains the current time; and bread_time, which contains the amount of time it takes for a breadmaker to make bread. Then we'll use add_time to figure out when the bread will be done. If you haven't finished writing print_time yet, take a look ahead to Section before you try this:

```
>>> current_time = Time()
>>> current_time.hours = 9
>>> current_time.minutes = 14
>>> current_time.seconds =  30
>>> bread_time = Time()
>>> bread_time.hours = 3
>>> bread_time.minutes = 35
>>> bread_time.seconds = 0
>>> done_time = add_time(current_time, bread_time)
>>> print_time(done_time)
12:49:30
```

The output of this program is 12:49:30, which is correct. On the other hand, there are cases where the result is not correct. Can you think of one?

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 minutes column or the extra minutes into the hours column.

Here's a second corrected version of the function:

```
def add_time(t1, t2):
    sum = Time()
    sum.hours = t1.hours + t2.hours
    sum.minutes = t1.minutes + t2.minutes
    sum.seconds = t1.seconds + t2.seconds

    if sum.seconds >= 60:
        sum.seconds = sum.seconds - 60
        sum.minutes = sum.minutes + 1

    if sum.minutes >= 60:
        sum.minutes = sum.minutes - 60
```

```
    sum.hours = sum.hours + 1

  return sum
```

Although this function is correct, it is starting to get big. Later we will suggest an alternative approach that yields shorter code.

14.3. Modifiers

There are times when it is useful for a function to modify one or more of the objects it gets as parameters. Usually, the caller keeps a reference to the objects it passes, so any changes the function makes are visible to the caller. Functions that work this way are called **modifiers**.

increment, which adds a given number of seconds to a Time object, would be written most naturally as a modifier. A rough draft of the function looks like this:

```
def increment(time, seconds):
  time.seconds = time.seconds + seconds

  if time.seconds >= 60:
    time.seconds = time.seconds - 60
    time.minutes = time.minutes + 1

  if time.minutes >= 60:
    time.minutes = time.minutes - 60
    time.hours = time.hours + 1
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.

Is this function correct? What happens if the parameter seconds is much greater than sixty? In that case, it is not enough to carry once; we have to keep doing it untilseconds is less than sixty. One solution is to replace the if statements with while statements:

```
def increment(time, seconds):
  time.seconds = time.seconds + seconds
```

```
    while time.seconds >= 60:
        time.seconds = time.seconds - 60
        time.minutes = time.minutes + 1

    while time.minutes >= 60:
        time.minutes = time.minutes - 60
        time.hours = time.hours + 1
```

This function is now correct, but it is not the most efficient solution.

14.4. Prototype development versus planning

In this chapter, we demonstrated an approach to program development that we call **prototype development**. In each case, we wrote a rough draft (or prototype) that performed the basic calculation and then tested it on a few cases, correcting flaws as we found them.

Although this approach can be effective, it can lead to 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 this case, the insight is that a Timeobject is really a three-digit number in base 60! The second component is the ones column, the minute component is the sixties column, and the hour component 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 a Time object into a single number and take advantage of the fact that the computer knows how to do arithmetic with numbers. The following function converts a Time object into an integer:

```
def convert_to_seconds(t):
    minutes = t.hours * 60 + t.minutes
    seconds = minutes * 60 + t.seconds
    return seconds
```

Now, all we need is a way to convert from an integer to a Time object:

```
def make_time(seconds):
    time = Time()
    time.hours = seconds/3600
    seconds = seconds - time.hours * 3600
    time.minutes = seconds/60
    seconds = seconds - time.minutes * 60
    time.seconds = seconds
    return time
```

You might have to think a bit to convince yourself that this technique to convert from one base to another is correct. Assuming you are convinced, you can use these functions to rewrite add_time:

```
def add_time(t1, t2):
    seconds = convert_to_seconds(t1) + convert_to_seconds(t2)
    return make_time(seconds)
```

This version is much shorter than the original, and it is much easier to demonstrate that it is correct (assuming, as usual, that the functions it calls are correct).

14.5. Generalization

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 times is better.

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

It is also easier to add features later. For example, imagine subtracting two Times to find the duration between them. The naive approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

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

14.6. Algorithms

When you write a general solution for a class of problems, as opposed to a specific solution to a single problem, you have written an **algorithm**. We mentioned this word before but did not define it carefully. It is not easy to define, so we will try a couple of approaches.

First, consider something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

But if you were lazy, you probably cheated by learning a few tricks. For example, to find the product of *n* and 9, you can write n-1 as the first digit and 10-n as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

In our opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literally, require no intelligence.

On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

14.7. Glossary

pure function

A function that does not modify any of the objects it receives as parameters. Most pure functions are fruitful.

modifier

A function that changes one or more of the objects it receives as parameters. Most modifiers are void.

functional programming style

A style of program design in which the majority of functions are pure.

prototype development

A way of developing programs starting with a prototype and gradually testing and improving it.

planned development

A way of developing programs that involves high-level insight into the problem and more planning than incremental development or prototype development.

algorithm

A set of instructions for solving a class of problems by a mechanical, unintelligent process.

14.8. Exercises

Write a function print_time that takes a Time object as an argument and prints it in the form hours:minutes:seconds.

Write a boolean function after that takes two Time objects, t1 and t2, as arguments, and returns True if t1 follows t2 chronologically and False otherwise.

Rewrite the increment function so that it doesn't contain any loops.

Now rewrite increment as a pure function, and write function calls to both versions.

Source: http://openbookproject.net/thinkcs/python/english2e/ch14.html