Think Python 2e, Chapter 16 Notes

Classes and Functions

October 21, 2022

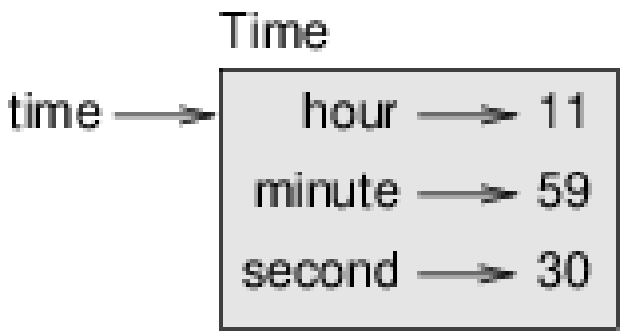
A new data type: Time

```
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

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

Object diagram:



A function of time

```
def print_time(t):
    print('%d:%.2d:%.2d'
              % (t.hour,t.minute,t.second))
```

```
>>> print_time(time)
11:59:30
```

Prototype of add_time

```
1 def add_time(t1, t2):
2     sum = Time()
3     sum.hour = t1.hour + t2.hour
4     sum.minute = t1.minute + t2.minute
5     sum.second = t1.second + t2.second
6     return sum
```

This is called a **pure function** because:

- it does not modify its arguments
- it has no effect, like printing or reading
- it **only** returns a value

It is like a mathematical function.

Figure out when the movie ends

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second =  0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

Disappointing.

Figure out when the movie ends, improved

```
1  def add_time(t1, t2):
2      sum = Time()
3      sum.hour = t1.hour + t2.hour
4      sum.minute = t1.minute + t2.minute
5      sum.second = t1.second + t2.second
6
7      if sum.second >= 60:
8          sum.second -= 60
9          sum.minute += 1
10
11     if sum.minute >= 60:
12         sum.minute -= 60
13         sum.hour += 1
14
15     return sum
```

Still not perfect. We'll improve it later.

Modifiers

- Functions that have side effects are not pure functions.
- They are called **modifiers**.

```
1  def increment(time, seconds):
2      time.second += seconds
3
4      if time.second >= 60:
5          time.second -= 60
6          time.minute += 1
7
8      if time.minute >= 60:
9          time.minute -= 60
10         time.hour += 1
```

Does this really work? What's wrong? How would you fix it?

Pure Functions *vs.* Modifiers

- Anything that can be done with modifiers can be done with pure functions.
- Programming with pure functions is called **functional programming style**.
- Some programming languages do not allow modifiers.
- There is evidence that pure functions are faster to develop and less error-prone.
- Functional programs are occasionally less efficient due to copying.

Prototyping *vs.* Planning

- The development we've seen in this chapter is called **prototype and patch**.
- Begin with a simple prototype, test it, patch it, repeat.
- Incremental correction can generate code that is unnecessarily complex and unreliable.
- It's hard to know when you've found all the errors.

Planned development

- An alternative to prototype and patch is **designed development**.
- Here we use a high-level understanding of the problem to restructure the code from the beginning.
- As an example, the `Time` class is really just a single integer in base 60.
- The seconds are the ones, the minutes are the sixties, and the hours are the sixty-squareds, or thirty-six hundreds.
- Since time is just an integer, we can just use integer addition to add times.
- We just have to convert the times to integers, add them, and convert them back.

Convert times to and from integers

```
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
```

New approach makes things MUCH easier

```
1 def add_time(t1, t2):
2     seconds = time_to_int(t1) + time_to_int(t2)
3     return int_to_time(seconds)
```

New approach makes things MUCH easier

- This version is much shorter than the original.
- This version is correct.
- It is easy to see it must be correct.
- We had to see the problem abstractly.
- We had to do more work at the start:
  - write the to and from conversion functions
- It makes it easier to add more features, *e.g.* subtraction.
- Sometimes making a problem harder makes it easier.

Debugging

- An **invariant** is something that should always be true.

```
1 def valid_time(time):
2     if time.hour<0 or time.minute<0 or time.second<0:
3         return False
4     if time.minute >= 60 or time.second >= 60:
5         return False
6     return True
```

Debugging

We can use this in every function of time:

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

`assert` raises an exception when its condition is false:

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Using `assert` helps distinguish normal code from error-checking.

Vocabulary

prototype and patch: A development plan that involves writing a rough draft of a program, testing, and correcting errors as they are found.

designed development: A development plan that involves high-level insight into the problem and more planning than incremental development or prototype development.

Vocabulary

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

modifier: A function that changes one or more of the objects it receives as arguments. Most modifiers are void; that is, they return None.

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

Vocabulary

invariant: A condition that should always be true during the execution of a program.

assert statement: A statement that checks a condition and raises an exception if it fails.