Lecture 13 - Introducing Classes and Object Oriented Programming

Week 6 Wednesday

Miles Chen, PhD

Taken directly from Chapters 15 and 16 of Think Python by Allen B Downey

Programmer defined types

We have used many of Python's built-in types; now we are going to define a new type.

We will create a type called `Point` that represents a point in two-dimensional space (x, y).

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, x and y.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

Creating a new type is more complicated, but it has advantages.

A programmer defined type is called a **class**. We define a class with the keyword `class`

Convention is to Capitalize class.

In [1]:
```
class Point:
    """Represents a point in 2-D space."""
```

It is customary to use a docstring header to explain what the class is for.

Defining a class named `Point` creates a **class object**.

In [2]:
```
print(Point)
```

```
<class '__main__.Point'>
```

Because Point is defined at the top level, its "full name" is `__main__.Point`

The class object is like a factory for creating objects. To create a Point, you call Point as if it were a function.

In [3]:
```
blank = Point()
```

In [4]:
```
blank
```

Out[4]:
```
<__main__.Point at 0x1def8385978>
```

The return value is a reference to a Point object, which we assign to blank.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory.

Attributes

You can assign values to an instance using dot notation.

In [5]:
```
blank.x = 3.0
blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as `math.pi`.

In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**.

In [6]:
```
blank.y
```

Out[6]:
```
4.0
```

In [7]:
```
x = blank.x
x
```

Out[7]:
```
3.0
```

There is no conflict between naming a variable x and having an attribute `x` inside the class. These are unrelated.

In [8]:
```
x
```

Out[8]:
```
3.0
```

Changing the value of blank.x will not affect the value of x.

In [9]:
```
blank.x = 5.0
```

In [10]:
```
x
```

Out[10]:
```
3.0
```

You can use the dot notation as part of any expression.

In [12]:
```
"(%g, %g)" % (blank.x, blank.y)
```

Out[12]:
```
'(5, 4)'
```

You can pass the object as an argument and access the attributes.

In [14]:
```
def print_point(p):
    print ("(%g, %g)" % (p.x, p.y))
```

In [15]:
```
print_point(blank)
```

```
(5, 4)
```

Example: A class to represent Rectangles

How can we design a class to represent a rectangle?

A couple options:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

Let's say we go with the first option.

In [16]:
```
class Rectangle:
    """Represent a rectangle

    attributes: width, height, corner"""
```

The width and height will be numbers.

To represent the corner, we will use a `Point` object.

In [17]:

```
# we create an instance of the Rectangle object and begin assigning attributes.
box = Rectangle()
box.width = 100.0
box.height = 200.0
# for the corner attribute, we create an instance of Point
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

Instances as return values

Functions can return instances. For example, we create a function `find_center` that takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle

In [18]:
```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

In [19]:
```
center = find_center(box)
```

In [20]:
```
center
```

Out[20]:
```
<__main__.Point at 0x1def83e0240>
```

In [21]:
```
print_point(center)
```

```
(50, 100)
```

Objects are mutable

You can change the state of an object by making an assignment to one of its attributes.

In [22]:
```
box.width
```

Out[22]:
```
100.0
```

In [23]:
```
box.width = box.width + 50
box.height = box.height + 100
```

In [24]:
```
box.width
```

Out[24]:
```
150.0
```

You can also write functions that modify objects.

In [25]:
```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

In [26]:
```
box.width, box.height
```

Out[26]:
```
(150.0, 300.0)
```

In [27]:
```
grow_rectangle(box, 50, 100)
```

In [28]:
```
box.width, box.height
```

Out[28]:
```
(200.0, 400.0)
```

Inside the function `grow_rectangle`, the argument `rect` becomes an alias for the object `box` when we call the function on `box`. When the function modifies the attributes of `rect`, it modifies `box`

Copying

The fact that objects are mutable can sometimes make the code difficult to read, especially when you have functions that modify the objects without necessarily reporting or printing anything to the screen.

We can use the `copy` module to make duplicates of an object.

In [29]:
```
p1 = Point()
```

In [30]:
```
p1.x = 3.0
p1.y = 4.0
```

In [31]:
```
import copy
```

In [32]:
```
p2 = copy.copy(p1)
```

In [33]:
```
print_point(p1)
```

```
(3, 4)
```

In [34]:
```
print_point(p2)
```

```
(3, 4)
```

In [35]:
```
p1 == p2
```

Out[35]:
```
False
```

In [36]:
```
p1 is p2
```

Out[36]:
```
False
```

Although p1 and p2 have the same data, they are not the same instance of a point object.

In [37]:
```
p1
```

Out[37]:
```
<__main__.Point at 0x1def83e00f0>
```

In [38]:
```
p2
```

Out[38]:
```
<__main__.Point at 0x1def83e0b38>
```

Shallow copies and deep copies

We have an instance of the `Rectangle` class called `box`

In [39]:
```
box.width, box.height
```

Out[39]:
```
(200.0, 400.0)
```

In [40]:
```
box.corner
```

Out[40]:
```
<__main__.Point at 0x1def83e0940>
```

In [41]:
```
box.corner.x, box.corner.y
```

Out[41]:
```
(0.0, 0.0)
```

Let's make a copy of `box`

In [42]:
```
box2 = copy.copy(box)
```

In [43]:
```
box2 is box
```

Out[43]:
```
False
```

box2 is a different instance of the Rectangle object than box

```
In [44]: box2.corner is box.corner
```

```
Out[44]: True
```

```
In [45]: box2.corner
```

```
Out[45]: <__main__.Point at 0x1def83e0940>
```

```
In [46]: box.corner
```

```
Out[46]: <__main__.Point at 0x1def83e0940>
```

However, the corner attribute in box is a Point object. Both `box` and `box2` 's corner attribute refer to the same Point object.

When we used `copy.copy()` , it create a copy of the object and the references inside, but did not make copies of the embedded objects.

In [47]:
```
box.corner.x = 1
```

In [48]:
```
box.corner.x
```

Out[48]: 1

In [49]:
```
box2.corner.x
```

Out[49]: 1

`box` and `box2` are separate, but they share the same Point object for their corner attribute.

In [50]:
```
box.height
```

Out[50]:
```
400.0
```

In [51]:
```
box2.height
```

Out[51]:
```
400.0
```

In [52]:
```
box.height = 200
```

In [53]:
```
box.height
```

Out[53]:
```
200
```

In [54]:
```
box2.height
```

Out[54]:
```
400.0
```

If we want to copy the embedded objects too, we have to make a deep copy.

In [55]:
```
box3 = copy.deepcopy(box)
```

In [56]:
```
box3 is box
```

Out[56]: False

In [57]:
```
box3.corner is box.corner
```

Out[57]: False

Classes and Functions

We often want to write functions that interact with objects and classes.

Let's create a class called `Time`

In [58]:
```
class Time:
    """Represents the time of day.

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

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

In [60]:
```
def print_time(t):
    print('%.2d:%.2d:%.2d' % (t.hour, t.minute, t.second))
```

In [61]:
```
print_time(time)
```

```
11:59:30
```

Pure functions vs modifiers

A pure function does not modify any of the objects passed to it as arguments.

It has no effect other than returning a value.

We use a development plan called **prototype and patch** to tackle complex problems.

We start with a prototype - a simple version of the program and incrementally add complications.

In [62]:
```
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
```

In [63]:
```
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
```

In [64]:
```
duration = Time()
duration.hour = 1
duration.minute = 35
duration.second = 0
```

In [65]:
```
done = add_time(start, duration)
print_time(done)
```

```
10:80:00
```

The result, `10:80:00` is not quite right. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty.

In [66]:
```
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
```

In [67]:
```
done = add_time(start, duration)
print_time(done)
```

```
11:20:00
```

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**.

`increment`, which adds a given number of seconds to a `Time` object, can be written as a modifier.

In [68]:
```
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
```

In [69]:
```
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [70]:
```
increment(test_time, 90)
print_time(test_time)
```

```
09:46:30
```

In [71]:
```
increment(test_time, 185)
print_time(test_time)
```

```
09:47:155
```

The function doesn't quite work 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.

Instead we can use modular division.

In [72]:
```
def increment(time, seconds):
    minutes, seconds = divmod(seconds, 60)
    hours, minutes = divmod(minutes, 60)
    time.second += seconds
    time.minute += minutes
    time.hour += hours
    if time.second >= 60:
        time.second -= 60
        time.minute += 1
    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

In [73]:
```
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [74]:
```
increment(test_time, 185)
print_time(test_time)
```

```
09:48:05
```

In [75]:
```
increment(test_time, 4800) # 4800 seconds is 1 hour 20 minutes
print_time(test_time)
```

```
11:08:05
```

Anything that can be done with a modifier can also be done with a pure function.

Modifiers are convenient, but can become difficult to debug.

In contrast to Python, most of R only allows pure functions (exception is R6 and reference classes).

Prototyping versus planning

"prototype and patch": 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 you don't yet have a deep understanding of the problem. But incremental corrections can generate code that is unnecessarily complicated—since it deals with many special cases.

An alternative is **designed development**

When applied to the time problem, we can convert all times into the integer number of seconds from midnight.

In [76]:
```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

We then create a function that is able to convert from seconds back to a time:

In [77]:
```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

In [78]:
```
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [79]:
```
time_to_int(test_time)
```

Out[79]:
```
35100
```

In [80]:
```
print_time(int_to_time(35100))
```

```
09:45:00
```

Now that we have the functions to conver time to integers and back, we can add times together easily. Convert the times both to integers, and then convert the sum back to a time.

In [81]:
```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```