Module 4
Classes and objects, Classes and functions, Classes and methods

---

Classes and Objects

Programmer-defined types

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.

A programmer-defined type is also called a **class**. A class definition looks like this:

class Point:
"""Represents a point in 2-D space."""

The header indicates that the new class is called Point. The body is a doc string that explains what the class is for. You can define variables and methods inside a class definition. Defining a class named Point creates a **class object**.
>>> **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.
>>> **blank = Point()**
>>> **blank**
<__main__.Point object at 0xb7e9d3ac>
The return value is a reference to a Point object, which we assign to blank.Creating a new object is called **instantiation**, and the object is an **instance** of the class.When you print an instance, Python tells you what class it belongs to and where it is stored in memory. Every object is an instance of some class, so "object" and "instance" are interchangeable.

Attributes

You can assign values to an instance using dot notation:
>>> **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** or **string.whitespace**. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**.

read the value of an attribute
>>> **blank.y**
4.0
>>> **x = blank.x**
>>> x

3.0
The expression blank.x means, "Go to the object blank refers to and get the value of x."

Representation using dot(.) notation
>>> '**(%g, %g)' % (blank.x, blank.y)**
'(3.0, 4.0)'

Usage of functions for representation
def print_point(p):
print('(%g, %g)' % (p.x, p.y))
print_point takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass blank as an argument:
>>> **print_point(blank)**
(3.0, 4.0)
Inside the function, p is an alias for blank, so if the function modifies p, blank changes.

Rectangles

class definition:

class Rectangle:
"""Represents a rectangle.
attributes: width, height, corner.
"""
The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.
To **represent a rectangle**, you have to instantiate a Rectangle object and assign values to the attributes:
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
The expression box.corner.x means, "Go to the object box refers to and select the attribute named corner; then go to that object and select the attribute named x."

Instances as return values

Functions can return instances. For example, find_center takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle:
def find_center(rect):
p = Point()
p.x = rect.corner.x + rect.width/2
p.y = rect.corner.y + rect.height/2
return p
Here is an example that passes box as an argument and assigns the resulting Point to center:

>>> **center = find_center(box)**
>>> **print_point(center)**
(50, 100)

Objects are mutable

We can change the state of an object by making an assignment to one of its attributes.
Example
We can change the size of a rectangle without changing its position, you can modify the values of width and height:
box.width = box.width + 50
box.height = box.height + 100
we can also write functions that modify objects.
Example: grow_rectangle takes a Rectangle object and two numbers, dwidth and dheight, and adds the numbers to the width and height of the rectangle:
def grow_rectangle(rect, dwidth, dheight):
rect.width += dwidth
rect.height += dheight
Here is an example that demonstrates the effect:
>>> **box.width, box.height**
(150.0, 300.0)
>>> **grow_rectangle(box, 50, 100)**
>>> **box.width, box.height**
(200.0, 400.0)

Copying

Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object
>>> **p1 = Point()**
>>> **p1.x = 3.0**
>>> **p1.y = 4.0**
>>> **import copy**
>>> **p2 = copy.copy(p1)**
p1 and p2 contain the same data, but they are not the same Point.
>>> **print_point(p1)**
(3, 4)
>>> **print_point(p2**)
(3, 4)

Note:
>>> **p1 is p2**
False
>>> **p1 == p2**
False
The **is** operator indicates that p1 and p2 are not the same object, which is what we expected.But you might have expected == to yield True because these points contain the same data. In that

case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence.

Note:simply using copy and shallow copy are same

Shallow copy concept

If we use copy.copy to duplicate a Rectangle, we will find that it copies the Rectangle object but not the embedded Point.

>>> **box2 = copy.copy(box)**
>>> **box2 is box**
False
>>> **box2.corner is box.corner #HERE CORNER IS NOT THE SEPARETE OBJECT, SO RESULT ISTRUE**
True

This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.

Deep copy concept

copy module provides a method named deepcopy that copies not only the object but also the objects it refers to, and the objects they refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy**.

>>> **box3 = copy.deepcopy(box)**
>>> **box3 is box**
False
>>> **box3.corner is box.corner #HERE CORNER IS THE SEPARETE OBJECT ,SO TRUE**
False

box3 and box are completely separate objects.

Classes and Functions

Time

class Time:
"""Represents the time of day.
attributes: hour, minute, second
"""

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

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

Pure functions

The two functions that add time values, demonstrate two kinds of functions: ***pure functions and modifiers.*** They also demonstrate a development plan called **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**: where **t1** and **t2** are time objects

```
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.

Testing of function
Lets create two **Time** objects:
start contains the start time of a movie, 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.

```
>>> 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
```

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.
Note: print_time function takes a Time object and prints it in the form hour:minute:second

Improved version

```
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
```

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 is called **modifiers**. **increment**, which adds a given number of seconds to a **Time** object, can be written naturally as a modifier.

Here is a rough draft:

```
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
```

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

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. This approach is not efficient**

Prototyping versus planning

The development plan demonstrated here is called **"prototype and patch".** For each function, 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 have a deep understanding of the problem. But 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.

Designed development

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 .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:

```
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).

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

Classes and Methods

(notes in part 2)…..