Classes and methods

Object-oriented features

1. Programs include class and method definitions.
2. Most of the computation is expressed in terms of operations on objects.
3. Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact.

Introduction to methods

A method is a function that is associated with a particular class. We have seen methods for strings, lists, dictionaries and tuples. In this chapter, we will define methods for programmer-defined types.

Differences between methods and functions

Methods are semantically the same as functions, but there are two syntactic differences:

• Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.

• The syntax for invoking a method is different from the syntax for calling a function.

Printing objects

Normal Function

```
class Time:
"""Represents the time of day."""
def print_time(time):
print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

To call this function, you have to pass a Time object as an argument:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

To make print_time a method, all we have to do is move the function definition inside the class definition.

Notice the change in indentation.

```
class Time:
        def print_time(time):
                print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Two ways of calling methods

Function syntax

```
>>>Time.print_time(start)
```

09:45:00

Method syntax

>>> start.print_time()
09:45:00

Explanation of method syntax

In this use of dot notation, print_time is the name of the method (again), and start is the object the method is invoked on, which is called the **subject**. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about. Inside the method, the subject is assigned to the first parameter, so in this case start is assigned to time.
By convention, the first parameter of a method is called self, so it would be more common to write print_time like this:

```
class Time:
        def print_time(self):
                print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

Example to print time and print the time after incrementing with output

```
class Time:
   def print_time(time):#Method
      print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))

   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

start = Time()
start.hour = 9
start.minute = 45
start.second = 00
start.print_time()#Method syntax


end=Time()
end = start.increment(1337)
print(end)#end.print_time()
#10:07:17
```

The subject, start, gets assigned to the first parameter, self. The argument, 1337, gets assigned to the second parameter, seconds.

More complicated example(Taking 2 time objects as parameters) with output
In this case it is conventional to name the first parameter self and the second parameter other.

```
class Time:
        def is_after(self, other):
                return self.time_to_int() > other.time_to_int()
```

To use this method, you have to invoke it on one object and pass the other as an argument:

```
>>> end.is_after(start)
True
```

The __init__ method
The init method (short for "initialization") is a special method that gets invoked when an object is instantiated. Its full name is **__init__**

```
class Time:
        def __init__(self, hour=0, minute=0, second=0):
                self.hour = hour
                self.minute = minute
                self.second = second
```

It is common for the parameters of __init__ to have the same names as the attributes. The statement

self.hour = hour stores the value of the parameter **hour** as an attribute of self.
The parameters are optional, so if you call Time with no arguments, you get the default values.

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

The __str__ method
__str__ is a special method, like __init__, that is supposed to return a string representation of an object. here is a str method for Time objects:

```
class Time:
        def __str__(self):
                return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

When you print an object, Python invokes the __str__ method:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

Program demonstrating working of __init__ operation

```
class Time:
```

```
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))

start = Time()
start.hour = 9
start.minute = 45
start.second = 00

#The init method
time = Time()
time.print_time()
#00:00:00
```

If we provide one argument, it overrides hour:

```
time = Time(9)
time.print_time()
#09:00:00
```

If we provide one argument, it overrides hour and minute :

```
time = Time(9,45)
time.print_time()
#09:45:00
```

Operator overloading

If you define a method named __add__ for the Time class, you can use the + operator on Time objects.

When you apply the + operator to Time objects, Python invokes __add__. When you print the result, Python invokes __str__.

Changing the behavior of an operator so that it works with programmer-defined types iscalled **operator overloading**.

Example

```
class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
```

```
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

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

    def __add__(self, other):
        print("Time is: ")
        seconds = self.time_to_int() + other.time_to_int()
        return self.int_to_time(seconds)

#Operator overloading
start = Time(1, 50)
duration = Time(2, 50)
print(start + duration)
#11:20:00
```

Type-based dispatch

In the previous section we added two Time objects, but you also might want to add an integer to a Time object. The following is a version of __add__ that checks the type of other and invokes either add_time or increment.

Example

```
class Time:
    #Type-based dispatch

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)

    def __add__(self, other):
        print("Time is...: ")

     if isinstance(other, Time):
```

```
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return self.int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return self.int_to_time(seconds)

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

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

    #“right-side add”
    def __radd__(self, other):
        return self.__add__(other)


#Type-based dispatch
start = Time(9, 45)
duration = Time(1, 35)
print(start + duration)
#11:20:00
print(start + 1337)
#10:07:17

#“right-side add”
print(1337 + start)
#10:07:17
```

Explanation:

The built-in function isinstance takes a value and a class object, and returns True if the value is an instance of the class. If other is a Time object, __add__ invokes add_time. Otherwise it assumes that the parameter is a number and invokes increment. This operation is called a **type-based dispatch** because it dispatches the computation to different methods based on the type of the arguments.

This implementation of addition is not commutative. If the integer is the first operand, we get error

Solution to this problem is to use the the special method **__radd__,** which stands for "right-side add". This method is invoked when a Time object appears on the right side of the + operator.

Polymorphism

```
class Time:
    #polymorphism
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return self.int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return self.int_to_time(seconds)

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

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

    #"right-side add"
    def __radd__(self, other):
        return self.__add__(other)
```

 # Demonstrating Polymorphism by adding 3 time objects and calculating time elapsed

```
t1 = Time(7, 43)
t2 = Time(7, 41)
t3 = Time(7, 37)
total = sum([t1, t2, t3])
print("Time Objects: ",t1,t2,t3)
print("Total Time is: ",total)
```