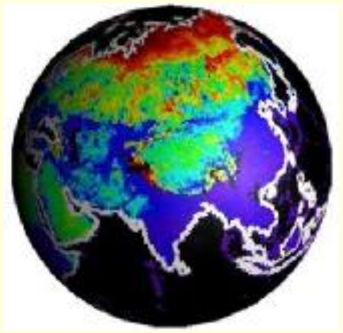


Chapter 17 Classes and methods

Jihn-Fa Jan, Ph.D.
Associate Professor
Department of Land Economics
National Chengchi University

Object-oriented features

- **Python is an object-oriented programming language, which means that it provides features that support object-oriented programming.**
- **It is not easy to define object-oriented programming, but we have already seen some of its characteristics:**
  - Programs are made up of object definitions and function definitions, and most of the computation is expressed in terms of operations on objects.
  - Each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.
- **For example, the Time class defined in Chapter 16 corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the Point and Rectangle classes correspond to the mathematical concepts of a point and a rectangle.**

Object-oriented features (cont'd)

- So far, we have not taken advantage of the features Python provides to support object-oriented programming. These features are not strictly necessary; most of them provide alternative syntax for things we have already done. But in many cases, the alternative is more concise and more accurately conveys the structure of the program.
- For example, in the Time program, there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one Time object as an argument.
- This observation is the motivation for 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 user-defined types.

Object-oriented features (cont'd)

- **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

- In Chapter 16, we defined a class named Time and in Exercise 16.1, you wrote a function named print_time:

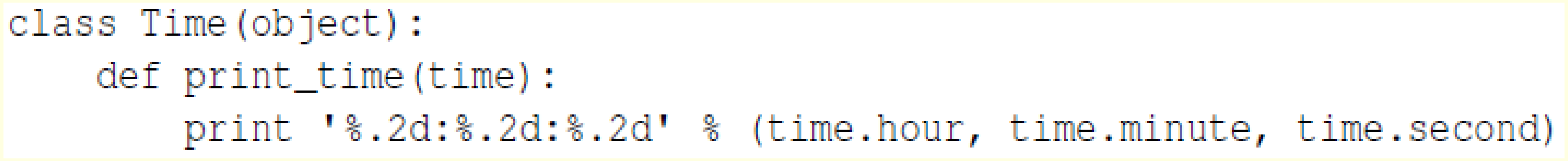
```
class Time(object):
    """represents the time of day.
       attributes: hour, minute, second"""

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
```

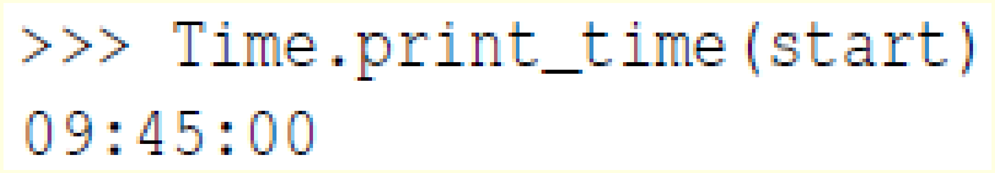
Printing objects (cont'd)

- **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(object):
    def print_time(time):
        print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

- **Now there are two ways to call print_time. The first (and less common) way is to use function syntax:**

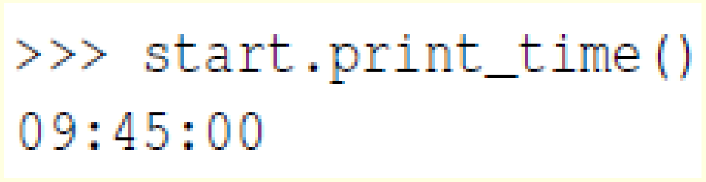


```
>>> Time.print_time(start)
09:45:00
```

  - In this use of dot notation, Time is the name of the class, and print_time is the name of the method. start is passed as a parameter.

Printing objects (cont'd)

- The second (and more concise) way is to use method syntax:

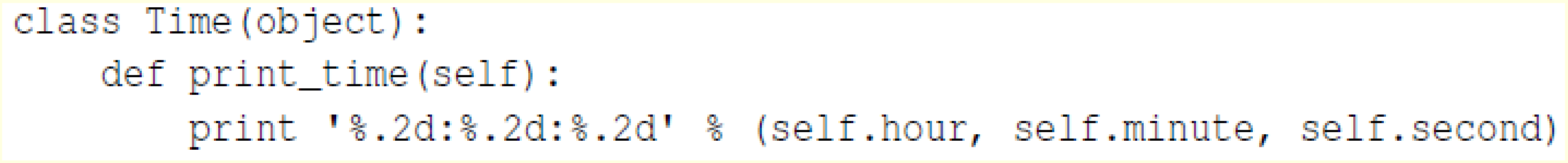


```
>>> start.print_time()
09:45:00
```

- 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.

Printing objects (cont'd)

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

- **The reason for this convention is an implicit metaphor:**
  - The syntax for a function call, print_time(start), suggests that the function is the active agent. It says something like, "Hey print_time! Here's an object for you to print."
  - In object-oriented programming, the objects are the active agents. A method invocation like start.print_time() says "Hey start! Please print yourself."

Printing objects (cont'd)

- This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions, and makes it easier to maintain and reuse code.

Another example

- **Here’s a version of increment (from Section 16.3) rewritten as a method:**

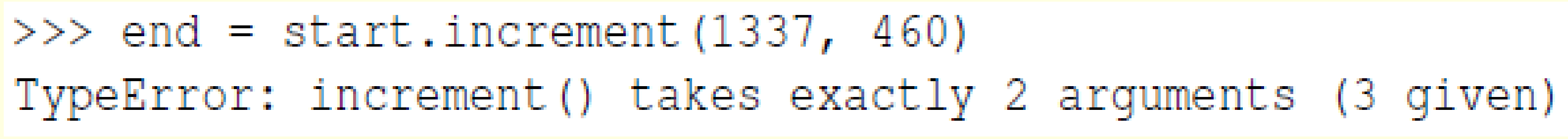
```
# inside class Time:

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

- **This version assumes that time_to_int is written as a method, as in Exercise 17.1. Also, note that it is a pure function, not a modifier. Here’s how you would invoke increment:**

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

Another example (cont'd)

- **The subject, start, gets assigned to the first parameter, self. The argument, 1337, gets assigned to the second parameter, seconds.**
- **This mechanism can be confusing, especially if you make an error. For example, if you invoke increment with two arguments, you get:**



```
>>> end = start.increment(1337, 460)
TypeError: increment() takes exactly 2 arguments (3 given)
```

  - The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that's three.

A more complicated example

- **is_after (from Exercise 16.2) is slightly more complicated because it takes two Time objects as parameters. In this case it is conventional to name the first parameter self and the second parameter other:**

```
# inside 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
```

- **One nice thing about this syntax is that it almost reads like English: "end is after start?"**

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__ (two underscore characters, followed by init, and then two more underscores). An init method for the Time class might look like this:**

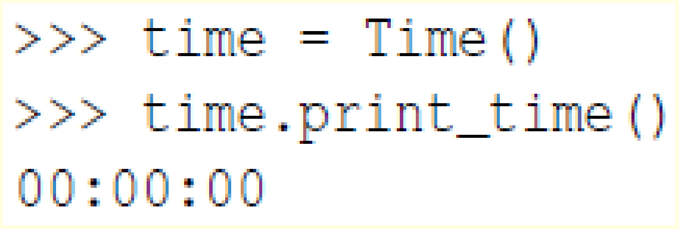
```
# inside class Time:

    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 following statement stores the value of the parameter hour as an attribute of self.**

```
self.hour = hour
```

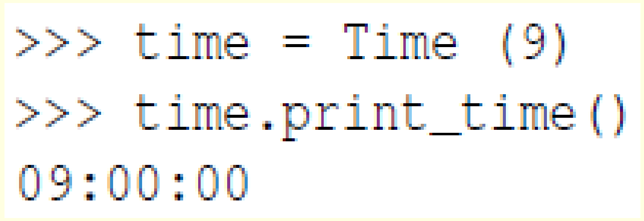
The init method (cont'd)

- The parameters are optional, so if you call Time with no arguments, you get the default values.



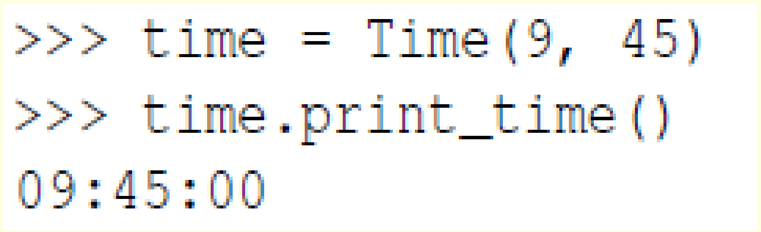
```
>>> time = Time()
>>> time.print_time()
00:00:00
```

- If you provide one argument, it overrides hour:



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

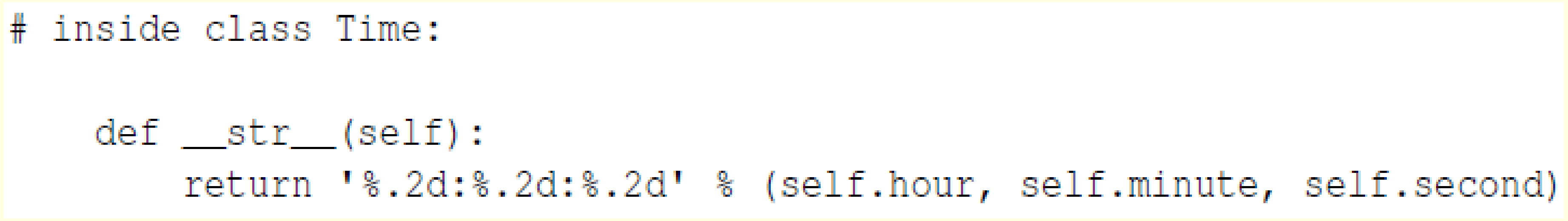
- If you provide two arguments, they override hour and minute. And if you provide three arguments, they override all three default values.



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

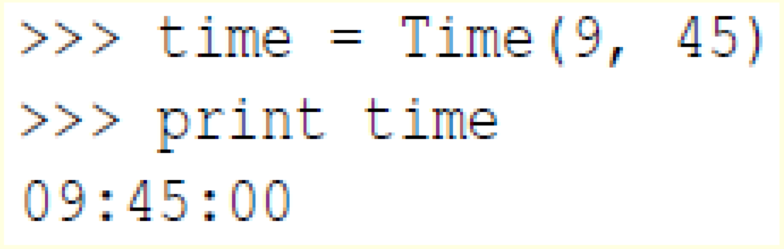
The str method

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



```
# inside 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
```

- When I write a new class, I almost always start by writing __init__, which makes it easier to instantiate objects, and __str__, which is useful for debugging.

Operator overloading

- By defining other special methods, you can specify the behavior of operators on user-defined types. For example, if you define a method named __add__ for the Time class, you can use the + operator on Time objects.

```
# inside class Time:

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

- And here is how you could use it:

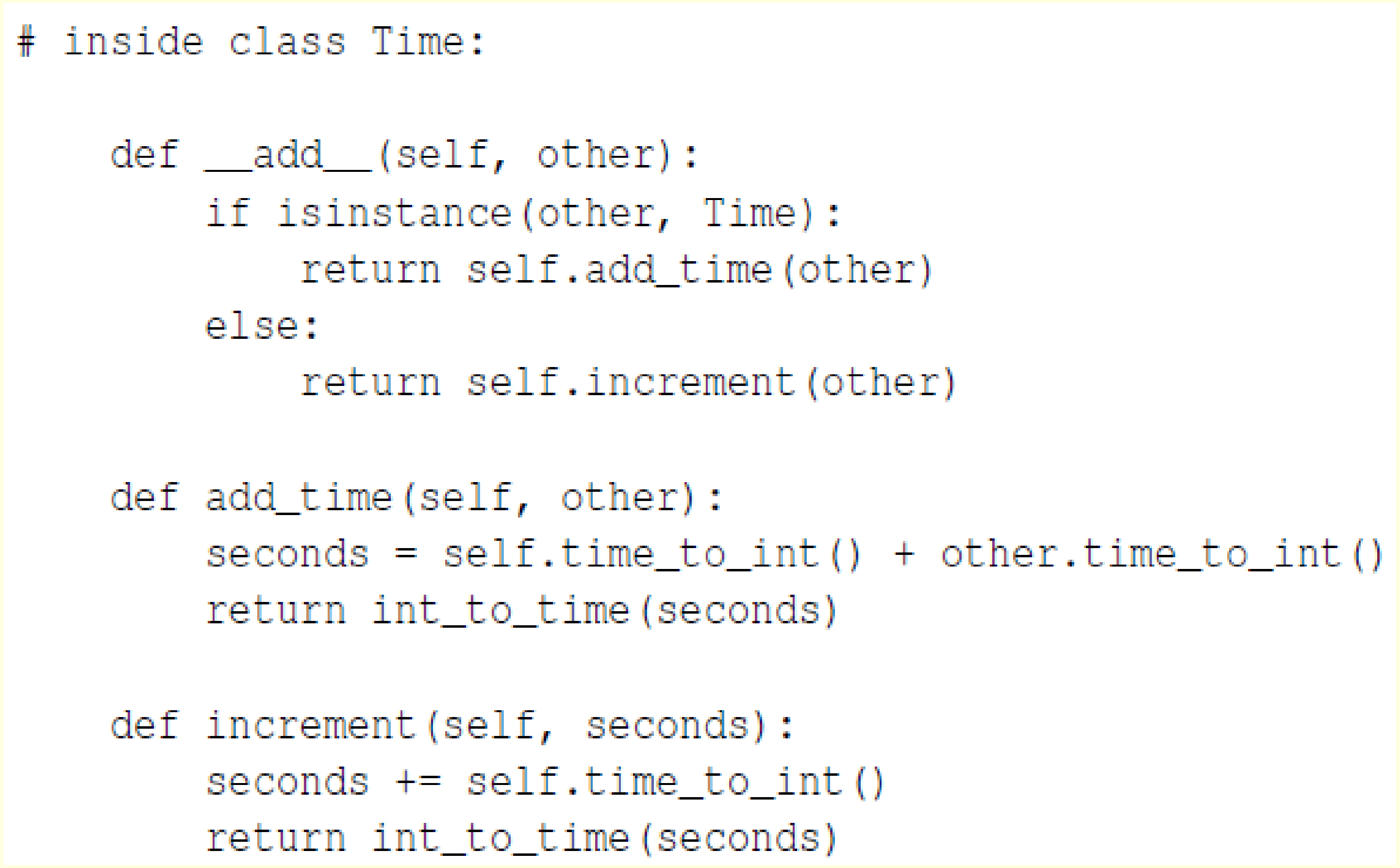
```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
```

Operator overloading (cont'd)

- When you apply the + operator to Time objects, Python invokes __add__. When you print the result, Python invokes __str__. So there is quite a lot happening behind the scenes!
- Changing the behavior of an operator so that it works with user-defined types is called operator overloading. For every operator in Python there is a corresponding special method, like __add__. For more details, see docs.python.org/ref/specialnames.html.

Type-based dispatch

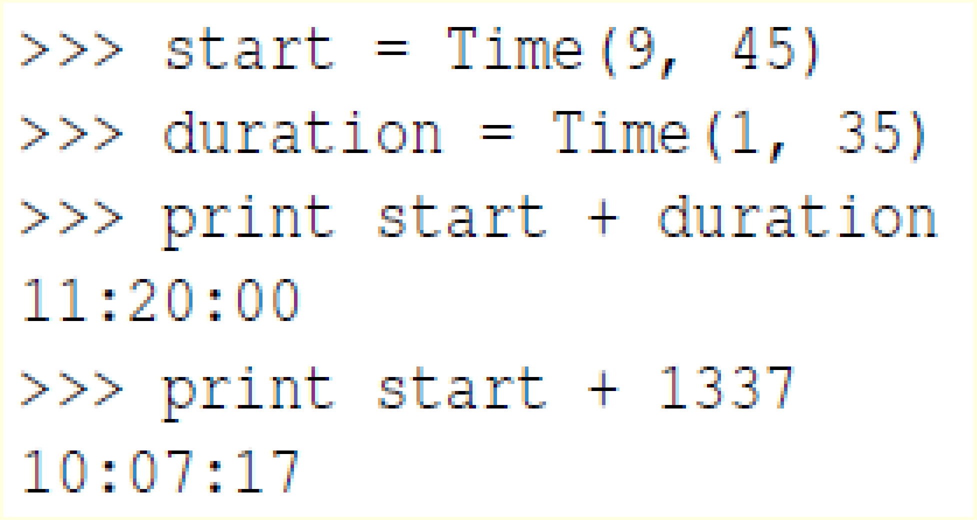
- In the previous section we added two Time objects, but you also might want to add an integer to a Time object. The following is a version of __add__ that checks the type of other and invokes either add_time or increment:



```
# inside class Time:

    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 int_to_time(seconds)

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

Type-based dispatch (cont'd)

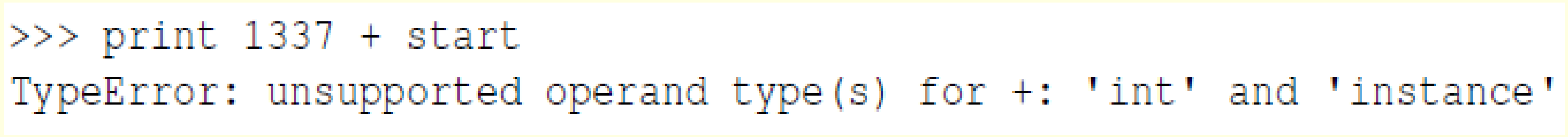
- The built-in function isinstance takes a value and a class object, and returns True if the value is an instance of the class.
- If other is a Time object, __add__ invokes add_time. Otherwise it assumes that the parameter is a number and invokes increment. This operation is called a type-based dispatch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



```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
>>> print start + 1337
10:07:17
```

Type-based dispatch (cont'd)

- Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get



```
>>> print 1337 + start
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

- The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how to do that.

Type-based dispatch (cont'd)

- **But there is a clever solution for this problem: 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. Here's the definition:**

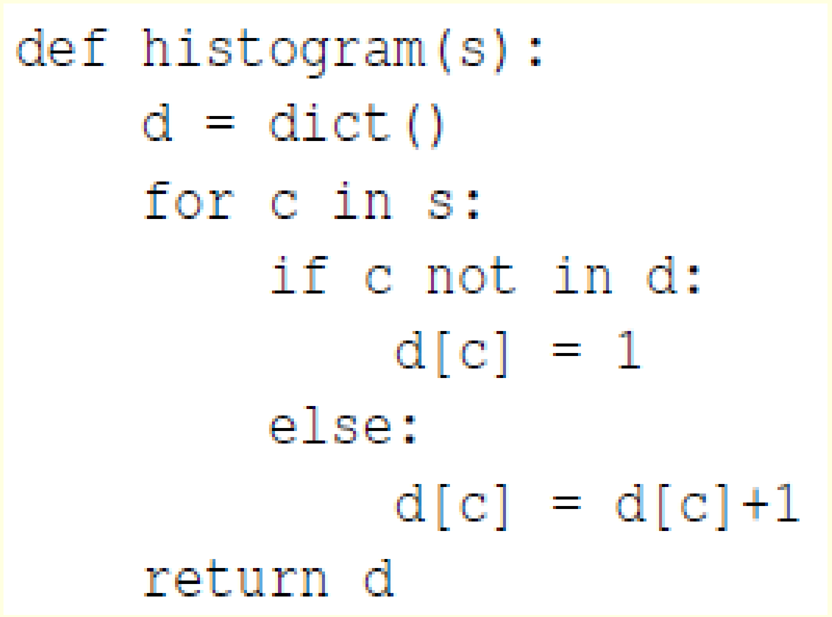
```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

- **And here's how it's used:**

```
>>> print 1337 + start
10:07:17
```

Polymorphism

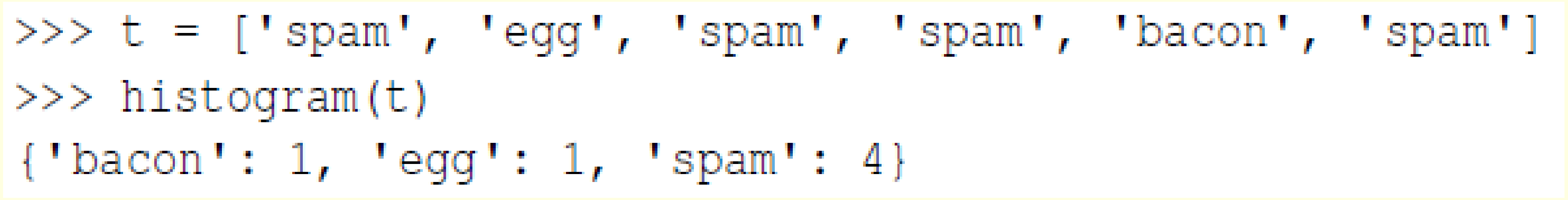
- Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types.
- Many of the functions we wrote for strings will actually work for any kind of sequence. For example, in Section 11.1 we used histogram to count the number of times each letter appears in a word.



```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

Polymorphism (cont'd)

- This function also works for lists, tuples, and even dictionaries, as long as the elements of s are hashable, so they can be used as keys in d.

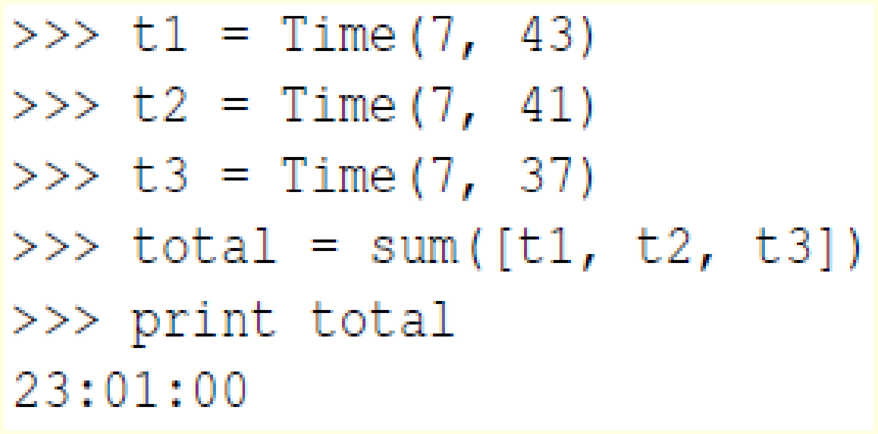


```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

- Functions that can work with several types are called polymorphic. Polymorphism can facilitate code reuse. For example, the built-in function sum, which adds the elements of a sequence, works as long as the elements of the sequence support addition.

Polymorphism (cont'd)

- Since Time objects provide an add method, they work with sum:

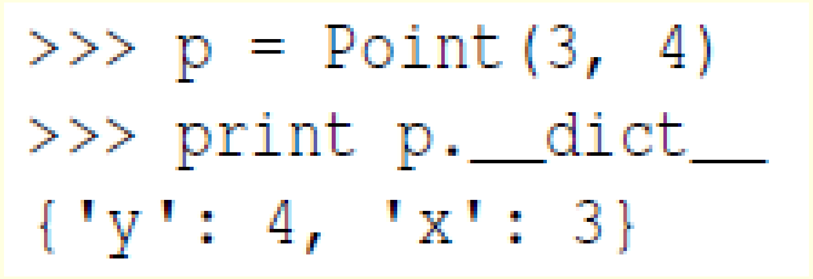


```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print total
23:01:00
```

- In general, if all of the operations inside a function work with a given type, then the function works with that type.
- The best kind of polymorphism is the unintentional kind, where you discover that a function you already wrote can be applied to a type you never planned for.

Debugging

- It is legal to add attributes to objects at any point in the execution of a program, but if you are a stickler for type theory, it is a dubious practice to have objects of the same type with different attribute sets. It is usually a good idea to initialize all of an objects attributes in the init method.
- If you are not sure whether an object has a particular attribute, you can use the built-in function hasattr (see Section 15.7).
- Another way to access the attributes of an object is through the special attribute __dict__, which is a dictionary that maps attribute names (as strings) and values:



```
>>> p = Point(3, 4)
>>> print p.__dict__
{'y': 4, 'x': 3}
```

Debugging (cont'd)

- For purposes of debugging, you might find it useful to keep this function handy:

```
def print_attributes(obj):
    for attr in obj.__dict__:
        print attr, getattr(obj, attr)
```

- print_attributes traverses the items in the object's dictionary and prints each attribute name and its corresponding value.
- The built-in function getattr takes an object and an attribute name (as a string) and returns the attribute's value.