Lecture 7-1

Introducing Classes and Object Oriented Programming

Week 7 Monday and 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."""
```

In [1]:

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

It is customary to use a docstring header to explain what the class is for. There is currently nothing else inside the class definition at this point in time.

In [1]:

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

It is customary to use a docstring header to explain what the class is for. There is currently nothing else inside the class definition at this point in time.

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

In [1]:

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

It is customary to use a docstring header to explain what the class is for. There is currently nothing else inside the class definition at this point in time.

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

In [2]:

```python
print(Point)
```

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

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. There is currently nothing else inside the class definition at this point in time.

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.

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()
```

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 0x251432f9248>
```

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 0x251432f9248>
```

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.

Attributes

You can assign values to an instance using dot notation.

In [5]:

```
blank.x = 3.0
blank.y = 4.0
```

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**.

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
```

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
```

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
```

In [8]:

```
x
```

Out[8]:

```
3.0
```

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

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 [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. Recall: you can insert numeric values into strings with the `{}` and `.format` notation.

You can use the dot notation as part of any expression. Recall: you can insert numeric values into strings with the `{}` and `.format` notation.

In [11]:
```
"({:g}, {:g})".format(blank.x, blank.y)
```

Out[11]:
```
'(5, 4)'
```

You can use the dot notation as part of any expression. Recall: you can insert numeric values into strings with the `{}` and `.format` notation.

In [11]:
```
"({:g}, {:g})".format(blank.x, blank.y)
```

Out[11]:
```
'(5, 4)'
```

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

You can use the dot notation as part of any expression. Recall: you can insert numeric values into strings with the `{}` and `.format` notation.

In [11]:
```
"({:g}, {:g})".format(blank.x, blank.y)
```

Out[11]:
```
'(5, 4)'
```

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

In [12]:
```
def print_point(p):
    print ("({:g}, {:g})".format(p.x, p.y))
```

You can use the dot notation as part of any expression. Recall: you can insert numeric values into strings with the `{}` and `.format` notation.

In [11]:

```
"({:g}, {:g})".format(blank.x, blank.y)
```

Out[11]:

```
'(5, 4)'
```

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

In [12]:

```
def print_point(p):
    print ("({:g}, {:g})".format(p.x, p.y))
```

In [13]:

```
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.

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 [14]:

```
class Rectangle:
    """Represent a rectangle

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

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 [14]:

```
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 [15]:

```python
# 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

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 [16]:

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

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 [16]:

```
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 [17]:

```
center = find_center(box)
```

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 [16]:

```
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 [17]:

```
center = find_center(box)
```

In [18]:

```
center
```

Out[18]:

```
<__main__.Point at 0x25143346e08>
```

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 [16]:

```
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 [17]:

```
center = find_center(box)
```

In [18]:

```
center
```

Out[18]:

```
<__main__.Point at 0x25143346e08>
```

In [19]:

```
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.

Objects are mutable

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

In [20]:

```
box.width
```

Out[20]:

```
100.0
```

Objects are mutable

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

In [20]:
```
box.width
```

Out[20]:
```
100.0
```

In [21]:
```
box.width = box.width + 50
box.height = box.height + 100
```

Objects are mutable

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

In [20]:
```
box.width
```

Out[20]:
```
100.0
```

In [21]:
```
box.width = box.width + 50
box.height = box.height + 100
```

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

Out[22]:
```
150.0
```

You can also write functions that modify objects.

You can also write functions that modify objects.

In [23]:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

You can also write functions that modify objects.

In [23]:

```python
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

In [24]:

```python
box.width, box.height
```

Out[24]:

```
(150.0, 300.0)
```

You can also write functions that modify objects.

In [23]:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

In [24]:

```
box.width, box.height
```

Out[24]:

```
(150.0, 300.0)
```

In [25]:

```
grow_rectangle(box, 50, 100)
```

You can also write functions that modify objects.

In [23]:
```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

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

Out[24]:
```
(150.0, 300.0)
```

In [25]:
```
grow_rectangle(box, 50, 100)
```

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

Out[26]:
```
(200.0, 400.0)
```

You can also write functions that modify objects.

In [23]:
```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

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

Out[24]:
```
(150.0, 300.0)
```

In [25]:
```
grow_rectangle(box, 50, 100)
```

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

Out[26]:
```
(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.

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 [27]:

```
p1 = Point()
```

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 [27]:

```
p1 = Point()
```

In [28]:

```
p1.x = 3.0
p1.y = 4.0
```

In [29]:

```
import copy
```

In [29]:

```
import copy
```

In [30]:

```
p2 = copy.copy(p1)
```

In [29]:

```
import copy
```

In [30]:

```
p2 = copy.copy(p1)
```

In [31]:

```
print_point(p1)
```

```
(3, 4)
```

In [29]:
```
import copy
```

In [30]:
```
p2 = copy.copy(p1)
```

In [31]:
```
print_point(p1)
```

```
(3, 4)
```

In [32]:
```
print_point(p2)
```

```
(3, 4)
```

In [33]:
```
p1 == p2
```

Out[33]:
```
False
```

In [33]:
```
p1 == p2
```
Out[33]:
```
False
```
In [34]:
```
p1 is p2
```
Out[34]:
```
False
```

In [33]:

```
p1 == p2
```

Out[33]:

```
False
```

In [34]:

```
p1 is p2
```

Out[34]:

```
False
```

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

In [33]:

```
p1 == p2
```

Out[33]:

```
False
```

In [34]:

```
p1 is p2
```

Out[34]:

```
False
```

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

In [35]:

```
p1
```

Out[35]:

```
<__main__.Point at 0x25144343388>
```

In [33]:
```
p1 == p2
```
Out[33]:
```
False
```
In [34]:
```
p1 is p2
```
Out[34]:
```
False
```
Although p1 and p2 have the same data, they are not the same instance of a point object.

In [35]:
```
p1
```
Out[35]:
```
<__main__.Point at 0x25144343388>
```
In [36]:
```
p2
```
Out[36]:
```
<__main__.Point at 0x251433565c8>
```

Shallow copies and deep copies

Shallow copies and deep copies

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

Shallow copies and deep copies

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

In [37]:
```
box.width, box.height
```

Out[37]:
```
(200.0, 400.0)
```

Shallow copies and deep copies

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

In [37]:

```
box.width, box.height
```

Out[37]:

```
(200.0, 400.0)
```

In [38]:

```
box.corner
```

Out[38]:

```
<__main__.Point at 0x25143346f88>
```

Shallow copies and deep copies

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

In [37]:
```
box.width, box.height
```

Out[37]:
```
(200.0, 400.0)
```

In [38]:
```
box.corner
```

Out[38]:
```
<__main__.Point at 0x25143346f88>
```

In [39]:
```
box.corner.x, box.corner.y
```

Out[39]:
```
(0.0, 0.0)
```

Let's make a copy of `box`

Let's make a copy of `box`

In [40]:

```
box2 = copy.copy(box)
```

Let's make a copy of `box`

In [40]:
```
box2 = copy.copy(box)
```

In [41]:
```
box2 is box
```

Out[41]:
```
False
```

Let's make a copy of `box`

In [40]:
```
box2 = copy.copy(box)
```

In [41]:
```
box2 is box
```

Out[41]:
```
False
```

box2 is a different instance of the Rectangle object than box

In [42]:

```
box2.corner is box.corner
```

Out[42]:

```
True
```

In [42]:
```
box2.corner is box.corner
```

Out[42]:
```
True
```

In [43]:
```
box2.corner
```

Out[43]:
```
<__main__.Point at 0x25143346f88>
```

In [42]:
```python
box2.corner is box.corner
```
Out[42]:
```
True
```
In [43]:
```python
box2.corner
```
Out[43]:
```
<__main__.Point at 0x25143346f88>
```
In [44]:
```python
box.corner
```
Out[44]:
```
<__main__.Point at 0x25143346f88>
```

In [42]:

```
box2.corner is box.corner
```

Out[42]:

```
True
```

In [43]:

```
box2.corner
```

Out[43]:

```
<__main__.Point at 0x25143346f88>
```

In [44]:

```
box.corner
```

Out[44]:

```
<__main__.Point at 0x25143346f88>
```

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 [45]:

```
box.corner.x = 1
```

In [45]:
```
box.corner.x = 1
```

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

Out[46]:
```
1
```

In [45]:
```
box.corner.x = 1
```

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

Out[46]:
```
1
```

In [47]:
```
box2.corner.x
```

Out[47]:
```
1
```

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

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

In [48]:

```
box.height
```

Out[48]:

```
400.0
```

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

In [48]:
```
box.height
```

Out[48]:
```
400.0
```

In [49]:
```
box2.height
```

Out[49]:
```
400.0
```

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

In [48]:
```
box.height
```

Out[48]:
```
400.0
```

In [49]:
```
box2.height
```

Out[49]:
```
400.0
```

In [50]:
```
box.height = 200
```

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

In [48]:
```
box.height
```

Out[48]:
```
400.0
```

In [49]:
```
box2.height
```

Out[49]:
```
400.0
```

In [50]:
```
box.height = 200
```

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

Out[51]:
```
200
```

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

In [48]:
```
box.height
```

Out[48]:
```
400.0
```

In [49]:
```
box2.height
```

Out[49]:
```
400.0
```

In [50]:
```
box.height = 200
```

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

Out[51]:
```
200
```

In [52]:
```
box2.height
```

Out[52]:
```
400.0
```

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

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

In [53]:

```
box3 = copy.deepcopy(box)
```

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

In [53]:
```
box3 = copy.deepcopy(box)
```

In [54]:
```
box3 is box
```

Out[54]:
```
False
```

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

In [53]:
```
box3 = copy.deepcopy(box)
```

In [54]:
```
box3 is box
```

Out[54]:
```
False
```

In [55]:
```
box3.corner is box.corner
```

Out[55]:
```
False
```

Classes and Functions

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

Classes and Functions

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

Let's create a class called `Time`

Classes and Functions

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

Let's create a class called `Time`

In [56]:

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

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

Classes and Functions

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

Let's create a class called `Time`

In [56]:

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

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

In [57]:

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

Classes and Functions

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

Let's create a class called `Time`

In [56]:

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

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

In [57]:

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

In [58]:

```
def print_time(t):
    print('{:0>2d}:{:0>2d}:{:0>2d}'.format(t.hour, t.minute, t.second))
```

Classes and Functions

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

Let's create a class called `Time`

In [56]:

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

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

In [57]:

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

In [58]:

```
def print_time(t):
    print('{:0>2d}:{:0>2d}:{:0>2d}'.format(t.hour, t.minute, t.second))
```

In [59]:

```
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.

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.

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 [60]:

```
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 [61]:

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

In [61]:

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

In [62]:

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

In [61]:

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

In [62]:

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

In [63]:

```
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.

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 [64]:

```
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
```

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 [64]:

```
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 [65]:

```
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.

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 [66]:

```
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 [67]:

```
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [67]:

```
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [68]:

```
increment(test_time, 90)
print_time(test_time)
```

```
09:46:30
```

In [67]:

```
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [68]:

```
increment(test_time, 90)
print_time(test_time)
```

```
09:46:30
```

In [69]:

```
increment(test_time, 185)
print_time(test_time)
```

```
09:47:155
```

In [67]:

```
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [68]:

```
increment(test_time, 90)
print_time(test_time)
```

```
09:46:30
```

In [69]:

```
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 [70]:

```python
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 [70]:

```python
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 [71]:

```python
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [70]:

```
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 [71]:

```
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [72]:

```
increment(test_time, 185)
print_time(test_time)
```

```
09:48:05
```

In [70]:

```python
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 [71]:

```python
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [72]:

```python
increment(test_time, 185)
print_time(test_time)
```

```
09:48:05
```

In [73]:

```python
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.

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 [74]:

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

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 [74]:
```
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:

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 [74]:
```
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 [75]:
```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

In [76]:

```
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [76]:

```
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [77]:

```
time_to_int(test_time)
```

Out[77]:

```
35100
```

In [76]:
```
test_time = Time()
test_time.hour = 9
test_time.minute = 45
test_time.second = 0
print_time(test_time)
```

```
09:45:00
```

In [77]:
```
time_to_int(test_time)
```

Out[77]:
```
35100
```

In [78]:
```
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.

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 [79]:

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

Methods

Methods are functions that are associated with a particular class.

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

- Methods are defined inside a class definition.
- The syntax for invoking a method is different from the syntax for calling a function.

The Time Class

We have created a class `Time` that can be used to store time data.

We wrote a function called `print_time()`

The Time Class

We have created a class `Time` that can be used to store time data.

We wrote a function called `print_time()`

In [80]:

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

The Time Class

We have created a class `Time` that can be used to store time data.

We wrote a function called `print_time()`

In [80]:

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

In [81]:

```
def print_time(time):
    print('{:0>2d}:{:0>2d}:{:0>2d}'.format(time.hour, time.minute, time.second))
```

The Time Class

We have created a class `Time` that can be used to store time data.

We wrote a function called `print_time()`

In [80]:

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

In [81]:

```
def print_time(time):
    print('{:0>2d}:{:0>2d}:{:0>2d}'.format(time.hour, time.minute, time.second))
```

To call the function, we had to create a `Time` object and pass it as an argument.

The Time Class

We have created a class `Time` that can be used to store time data.

We wrote a function called `print_time()`

In [80]:

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

In [81]:

```
def print_time(time):
    print('{:0>2d}:{:0>2d}:{:0>2d}'.format(time.hour, time.minute, time.second))
```

To call the function, we had to create a `Time` object and pass it as an argument.

In [82]:

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

```
09:45:00
```

Making a method

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

Note the indentation.

Making a method

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

Note the indentation.

In [83]:

```
class Time:
    def print_time(time):
        print('{:0>2d}:{:0>2d}:{:0>2d}'.format(time.hour, time.minute, time.second))
```

Making a method

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

Note the indentation.

In [83]:

```
class Time:
    def print_time(time):
        print('{:0>2d}:{:0>2d}:{:0>2d}'.format(time.hour, time.minute, time.second))
```

Now that we have redefined the class with a method defined inside, we can call the method.

Note, I have to re-create start as a `Time` class object because the old start object was created under the old definition of class `Time`. The changes don't apply retroactively.

Making a method

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

Note the indentation.

In [83]:

```
class Time:
    def print_time(time):
        print('{:0>2d}:{:0>2d}:{:0>2d}'.format(time.hour, time.minute, time.second))
```

Now that we have redefined the class with a method defined inside, we can call the method.

Note, I have to re-create start as a `Time` class object because the old start object was created under the old definition of class `Time`. The changes don't apply retroactively.

In [84]:

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

Calling the Method

There are two ways to call a method, but in practice, most people use one.

The less common way is to use the Class name and pass an object of that class to the method.

Calling the Method

There are two ways to call a method, but in practice, most people use one.

The less common way is to use the Class name and pass an object of that class to the method.

In [85]:

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

```
09:45:00
```

Calling the Method

There are two ways to call a method, but in practice, most people use one.

The less common way is to use the Class name and pass an object of that class to the method.

In [85]:
```
Time.print_time(start)
```

```
09:45:00
```

The more common way is to call the method directly from the object itself using dot notation.

Calling the Method

There are two ways to call a method, but in practice, most people use one.

The less common way is to use the Class name and pass an object of that class to the method.

In [85]:
```
Time.print_time(start)
```

```
09:45:00
```

The more common way is to call the method directly from the object itself using dot notation.

In [86]:
```
start.print_time()
```

```
09:45:00
```

Calling the Method

There are two ways to call a method, but in practice, most people use one.

The less common way is to use the Class name and pass an object of that class to the method.

In [85]:
```
Time.print_time(start)
```

```
09:45:00
```

The more common way is to call the method directly from the object itself using dot notation.

In [86]:
```
start.print_time()
```

```
09:45:00
```

When you call a method from the object itself, the object is known as the **subject** of the method.

The subject gets passed to the method as the first argument.

So in our example above, the object `start` gets passed as the first argument in thge method `print_time()`.

`self`

By convention, the first argument of a method is called `self`.

The idea is that when you call a method from an object with dot notation, you are applying to function to itself.

With this in mind, we'ld write our `Time` class as follows.

I've also included another method that converts the time to number of seconds after midnight (time_to_int) which will be useful for adding times.

`self`

By convention, the first argument of a method is called `self`.

The idea is that when you call a method from an object with dot notation, you are applying to function to itself.

With this in mind, we'ld write our `Time` class as follows.

I've also included another method that converts the time to number of seconds after midnight (time_to_int) which will be useful for adding times.

In [87]:

```
class Time:
    def print_time(self):
        print('{:0>2d}:{:0>2d}:{:0>2d}'.format(self.hour, self.minute, self.second))

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

In [88]:

```
# again, we redefine the object of class Time()
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
```

In [88]:

```
# again, we redefine the object of class Time()
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
```

In [89]:

```
start.print_time()
```

```
09:45:00
```

In [88]:
```
# again, we redefine the object of class Time()
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
```

In [89]:
```
start.print_time()
```

```
09:45:00
```

In [90]:
```
start.time_to_int()
```

Out[90]:
```
35100
```

We'll use the following function in the next class definition.

It converts the number of seconds into a time.

It can't be made as a method of a time object because the argument of this function is an integer value of seconds, and there is no object to invoke the method on.

We'll use the following function in the next class definition.

It converts the number of seconds into a time.

It can't be made as a method of a time object because the argument of this function is an integer value of seconds, and there is no object to invoke the method on.

In [91]:

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

Adding more methods

Adding more methods

In [92]:

```
class Time:
    def print_time(self):
        print('{:0>2d}:{:0>2d}:{:0>2d}'.format(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 increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

Adding more methods

In [92]:

```
class Time:
    def print_time(self):
        print('{:0>2d}:{:0>2d}:{:0>2d}'.format(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 increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

In [93]:

```
# again, we redefine the object of class Time()
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
```

Adding more methods

In [92]:

```
class Time:
    def print_time(self):
        print('{:0>2d}:{:0>2d}:{:0>2d}'.format(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 increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

In [93]:

```
# again, we redefine the object of class Time()
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
```

In [94]:

```
start.print_time()
```

```
09:45:00
```

Adding more methods

In [92]:

```
class Time:
    def print_time(self):
        print('{:0>2d}:{:0>2d}:{:0>2d}'.format(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 increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

In [93]:

```
# again, we redefine the object of class Time()
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
```

In [94]:

```
start.print_time()
```

```
09:45:00
```

In [95]:

```
end = start.increment(1337)
end.print_time()
```

```
10:07:17
```

Errors with method calls

Keep in mind that when you call a method from an object, the object itself is always passed as the first argument of the method.

Errors with method calls

Keep in mind that when you call a method from an object, the object itself is always passed as the first argument of the method.

In [96]:

```
end = start.increment(1337, 460)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_17864/297969520.py in <module>
----> 1 end = start.increment(1337, 460)

TypeError: increment() takes 2 positional arguments but 3 were given
```

Errors with method calls

Keep in mind that when you call a method from an object, the object itself is always passed as the first argument of the method.

In [96]:

```
end = start.increment(1337, 460)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_17864/297969520.py in <module>
----> 1 end = start.increment(1337, 460)

TypeError: increment() takes 2 positional arguments but 3 were given
```

The above call returns an error. "increment() takes 2 positional arguments but 3 were given"

It can be confusing because we see only two arguments (1337 and 460) in parentheses. We must remember that we have also passed `self` (the subject) as the first argument, so there really are three arguments.

Methods with other class objects

We can create a method inside `Time` called `is_after()` which will check to see if one time takes place after another time.

This function takes in two `Time` class objects and compares them. I add this method to the end of our class definition.

Because we expect the argument passed to `is_after()` is another `Time` class object, we can invoke the methods of the object. By convention, the first parameter of the method is `self`, and the parameter for the other class object being passed is named `other`

Methods with other class objects

We can create a method inside `Time` called `is_after()` which will check to see if one time takes place after another time.

This function takes in two `Time` class objects and compares them. I add this method to the end of our class definition.

Because we expect the argument passed to `is_after()` is another `Time` class object, we can invoke the methods of the object. By convention, the first parameter of the method is `self`, and the parameter for the other class object being passed is named `other`

In [97]:

```
class Time:
    def print_time(self):
        print('{:0>2d}:{:0>2d}:{:0>2d}'.format(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 increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)

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

Calling the new method

Calling the new method

In [98]:
```
# redefine the objects
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
start.print_time()
```

```
09:45:00
```

Calling the new method

In [98]:

```
# redefine the objects
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
start.print_time()
```

```
09:45:00
```

In [99]:

```
end = start.increment(1337)
end.print_time()
```

```
10:07:17
```

Calling the new method

In [98]:
```
# redefine the objects
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
start.print_time()
```

```
09:45:00
```

In [99]:
```
end = start.increment(1337)
end.print_time()
```

```
10:07:17
```

In [100]:
```
end.is_after(start)
```

Out[100]:
```
True
```

Calling the new method

In [98]:
```
# redefine the objects
start = Time()
start.hour = 9
start.minute = 45
start.second = 0
start.print_time()
```

```
09:45:00
```

In [99]:
```
end = start.increment(1337)
end.print_time()
```

```
10:07:17
```

In [100]:
```
end.is_after(start)
```

Out[100]:
```
True
```

In [101]:
```
start.is_after(end)
```

Out[101]:
```
False
```

The `init` method

It's been annoying that every time I redefine the `start` object, I have to assign the `hour`, `minute`, and `second` attributes.

There is a special initialization method called `__init__` (also called double-under init) that gets invoked whenever a new object of the class is created.

It is useful to use this method to assign values to attributes that would be used in the class. By convention, the parameters of init have the same names as the attributes.

In [102]:

```python
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('{:0>2d}:{:0>2d}:{:0>2d}'.format(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 increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)

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

Creating new `Time` class objects is much easier now.

Creating new `Time` class objects is much easier now.

In [103]:

```
midnight = Time()
midnight.print_time()
```

```
00:00:00
```

Creating new `Time` class objects is much easier now.

In [103]:

```
midnight = Time()
midnight.print_time()
```

```
00:00:00
```

In [104]:

```
new_time = Time(9)
new_time.print_time()
```

```
09:00:00
```

Creating new `Time` class objects is much easier now.

In [103]:

```
midnight = Time()
midnight.print_time()
```

```
00:00:00
```

In [104]:

```
new_time = Time(9)
new_time.print_time()
```

```
09:00:00
```

The argument 9 gets passed in as the value for `hour` because `self` is always the first argument, even in the `__init__` method.

Creating new `Time` class objects is much easier now.

In [103]:

```
midnight = Time()
midnight.print_time()
```

```
00:00:00
```

In [104]:

```
new_time = Time(9)
new_time.print_time()
```

```
09:00:00
```

The argument 9 gets passed in as the value for `hour` because `self` is always the first argument, even in the `__init__` method.

In [105]:

```
new_time = Time(9, 45)
new_time.print_time()
```

```
09:45:00
```

The `__str__` method

`__str__` is another special method that should return a string representation of an object.

When you call `print()` on an object, Python invokes the `__str__` method.

So far we have been using the method `print_time()` which we defined inside the class.

Instead, we will modify this method to work with the `__str__` method. To view the time, we will call `print()` on the object.

Note that in the conversion, we no longer call `print` but use `return` to return a string object.

In [106]:

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

    def __str__(self):
        return '{:0>2d}:{:0>2d}:{:0>2d}'.format(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 increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)

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

In [107]:

```
new_time = Time(9, 45)
```

In [107]:

```
new_time = Time(9, 45)
```

In [108]:

```
print(new_time)
```

```
09:45:00
```

Operator overloading

There are even more special "double-under" methods that have special uses.

One is the `__add__` method which will be invoked with the `+` operator.

I'll add the following method inside the class definition

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

In [109]:

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

    def __str__(self):
        return '{:0>2d}:{:0>2d}:{:0>2d}'.format(self.hour, self.minute, self.second)

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

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

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

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

In [110]:

```
start = Time(9, 45)
duration = Time(1, 35)
```

In [110]:

```
start = Time(9, 45)
duration = Time(1, 35)
```

In [111]:

```
print(start + duration)
```

```
11:20:00
```

Type-based dispatch

The previous defintion of `__add__` allowed us to add two `Time` class objects together. But we might also want the option to add integers as well.

We can use type-based dispatch to call different methods depending on the type of input. To perform this, we use the `isinstance()` to see if the object belongs to a particular class or not.

Inside the class definition of Time, I'll add the following:

```
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)
```

In [112]:

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

    def __str__(self):
        return '{:0>2d}:{:0>2d}:{:0>2d}'.format(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 time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

    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)

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

In [113]:

```
start = Time(9, 45)
```

In [113]:

```
start = Time(9, 45)
```

In [114]:

```
duration = Time(1, 35)
```

In [113]:

```
start = Time(9, 45)
```

In [114]:

```
duration = Time(1, 35)
```

In [115]:

```
print(start + duration)
```

```
11:20:00
```

In [113]:
```
start = Time(9, 45)
```

In [114]:
```
duration = Time(1, 35)
```

In [115]:
```
print(start + duration)
```

```
11:20:00
```

In [116]:
```
print(start + 1337)
```

```
10:07:17
```

Commutative Addition

The add method as we've implemented it is not commutative.

Commutative Addition

The add method as we've implemented it is not commutative.

In [117]:

```
print(1337 + start)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_17864/241212550.py in <module>
----> 1 print(1337 + start)

TypeError: unsupported operand type(s) for +: 'int' and 'Time'
```

Commutative Addition

The add method as we've implemented it is not commutative.

In [117]:
```
print(1337 + start)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_17864/241212550.py in <module>
----> 1 print(1337 + start)

TypeError: unsupported operand type(s) for +: 'int' and 'Time'
```

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.

If we want this to work, we have to use another special method: `__radd__` which stands for "right-side add"

We can pull this off quite easily:

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

In [118]:

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

    def __str__(self):
        return '{:0>2d}:{:0>2d}:{:0>2d}'.format(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 __radd__(self, other):
        return self.__add__(other)

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

    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)

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

In [119]:

```
start = Time(9, 45)
```

In [119]:

```
start = Time(9, 45)
```

In [120]:

```
duration = Time(1, 35)
```

In [119]:

```
start = Time(9, 45)
```

In [120]:

```
duration = Time(1, 35)
```

In [121]:

```
print(start + duration)
```

```
11:20:00
```

In [119]:

```
start = Time(9, 45)
```

In [120]:

```
duration = Time(1, 35)
```

In [121]:

```
print(start + duration)
```

```
11:20:00
```

In [122]:

```
print(start + 1337)
```

```
10:07:17
```

In [119]:

```
start = Time(9, 45)
```

In [120]:

```
duration = Time(1, 35)
```

In [121]:

```
print(start + duration)
```

```
11:20:00
```

In [122]:

```
print(start + 1337)
```

```
10:07:17
```

In [123]:

```
print(1337 + start)
```

```
10:07:17
```

Polymorphism

Many functions will work on different types. These are known as **polymorphic** functions.

For example, the function `sum` can work on any object that support addition.

Polymorphism

Many functions will work on different types. These are known as **polymorphic** functions.

For example, the function `sum` can work on any object that support addition.

In [124]:

```
t1 = Time(1, 20)
t2 = Time(1, 40)
t3 = Time(1, 30)
total = sum([t1, t2, t3])
print(total)
```

```
04:30:00
```

Important tips

It is legal to add attributes to objects at any time. But if you have objects of the same type that don't have the same attributes, it can cause problems.

It is recommended to initialize all of the objects attributes inside the `__init__` method.

A useful function for debugging is teh `vars()` function which will print all of the attributes an object has as a dictionary.

Important tips

It is legal to add attributes to objects at any time. But if you have objects of the same type that don't have the same attributes, it can cause problems.

It is recommended to initialize all of the objects attributes inside the `__init__` method.

A useful function for debugging is teh `vars()` function which will print all of the attributes an object has as a dictionary.

In [125]:
```
vars(start)
```

Out[125]:
```
{'hour': 9, 'minute': 45, 'second': 0}
```