Chapter 16
Classes and Objects

A. Downey, *Think Python: How to Think Like a Computer Science,* 3rd ed., O'Reilly, 2024.

https://allendowney.github.io/ThinkPython/

Contents

1. Creating a Point

- In computer graphics, a location on the screen is often represented using a pair of coordinates in an *x*-*y* plane.
  - By convention, the point (0, 0) usually represents the upper-left corner of the screen, and (*x*, *y*) represents the point *x* units to the right and *y* units down from the origin.
  - Compared to the Cartesian coordinate system you might have seen the y-axis is upside down.
- There are several ways we might represent a point in Python:
  - We can store the coordinates separately in two variables, **`x`** and **`y`**.
  - We can store the coordinates as elements in a list or tuple.
  - **We can create a new type to represent points as objects.**

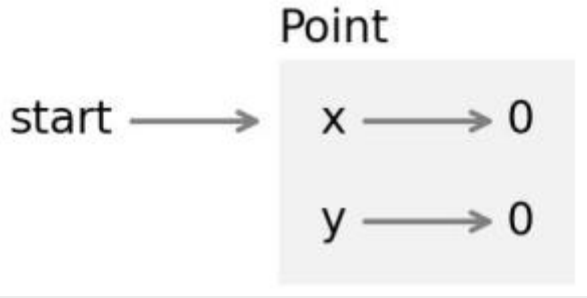
1. Creating a Point

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

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f'Point({self.x}, {self.y})'
```

```
start = Point(0, 0)
print(start)
```

```
Point(0, 0)
```

- The **__init__** method takes the coordinates as parameters and assigns them to attributes **x** and **y**.
- The **__str__** method returns a string representation of the **Point**.
- The following diagram shows the state of the new object:
  - As usual, a programmer-defined type is represented by a box with the name of the type outside and the attributes inside.



1. Creating a Point

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

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f'Point({self.x}, {self.y})'

    def translate(self, dx, dy):
        self.x += dx
        self.y += dy
```

```
start = Point(0, 0)

start.translate(10, 20)
print(start)
```

```
Point(10, 20)
```

- In general, programmer-defined types are mutable, so we can write a method like **translate** that takes two numbers, **dx** and **dy**, and adds them to the attributes **x** and **y**.
  - This function translates the **Point** from one location in the plane to another.

1. Creating a Point

- In the same way that the built-in function **sort** modifies a list, and the **sorted** function creates a new list, now we have a **translate** method that modifies a **Point**, and a **translated** method that creates a new one.

```
1  from copy import copy
2
3  class Point:
4      """Represents a point in 2-D space."""
5
6      def __init__(self, x, y):
7          self.x = x
8          self.y = y
9
10     def __str__(self):
11         return f'Point({self.x}, {self.y})'
12
13     def translate(self, dx, dy):
14         self.x += dx
15         self.y += dy
16
17     def translated(self, dx=0, dy=0):
18         new_point = copy(self)
19         new_point.translate(dx, dy)
20         return new_point
```

```
1  start = Point(0, 0)
2
3  end = start.translated(10, 20)
4  print(end)
```

```
Point(10, 20)
```

2. Creating a Line

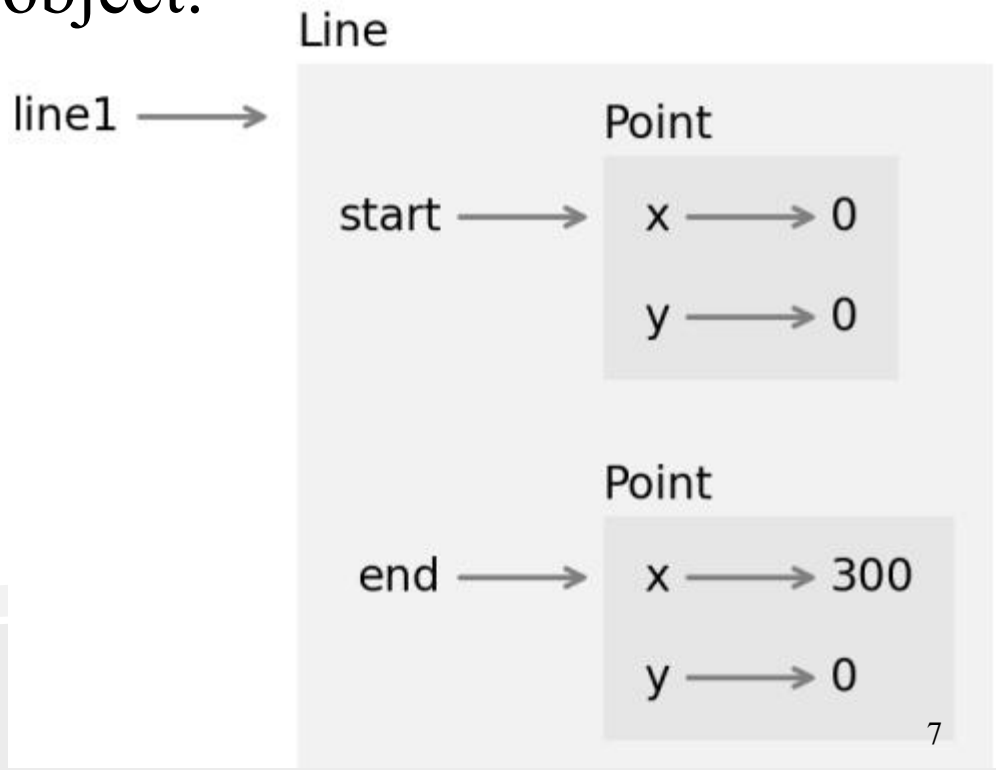
- Define a class that represents the line segment between two points.

```
class Line:
    def __init__(self, p1, p2):
        self.p1 = p1
        self.p2 = p2

    def __str__(self):
        return f'Line({self.p1}, {self.p2})'
```

```
start = Point(10, 10)
end = Point(20, 80)

line1 = Line(start, end)
print(line1)
```

```
Line(Point(10, 10), Point(20, 80))
```

- The following object diagram shows the state of this **Line** object:



2. Creating a Line

- Define a method **len** in the class **Line** to calculate and return the length of the line segment

```python
import math

class Line:
    def __init__(self, p1, p2):
        self.p1 = p1
        self.p2 = p2

    def __str__(self):
        return f'Line({self.p1}, {self.p2})'

    def len(self):
        len_2 = (self.p1.x - self.p2.x) * (self.p1.x - self.p2.x) + \
            (self.p1.y - self.p2.y) * (self.p1.y - self.p2.y)
        return math.sqrt(len_2)
```

```python
p1 = Point(3, 0)
p2 = Point(0, 4)

line = Line(p1, p2)
line.len()
```

```
5.0
```

2. Creating a Line

- Write a method called **translate** that takes two numbers, **dx** and **dy.** It translates the Line object from one location in the plane to another.

```
import math

class Line:
    def __init__(self, p1, p2):
        self.p1 = p1
        self.p2 = p2

    def __str__(self):
        return f'Line({self.p1}, {self.p2})'

    def len(self):
        len_2 = (self.p1.x - self.p2.x) * (self.p1.x - self.p2.x) + \
            (self.p1.y - self.p2.y) * (self.p1.y - self.p2.y)
        return math.sqrt(len_2)

    def translate(self, dx, dy):
        self.p1.translate(dx, dy)
        self.p2.translate(dx, dy)
```

```
line = Line(Point(10, 20), Point(100, 200))

line.translate(50, 50)
print(line)
```

```
Line(Point(60, 70), Point(150, 250))
```

3. Equivalence and Identity

- Suppose we create two points with the same coordinates:

```
1 p1 = Point(200, 100)
2 p2 = Point(200, 100)
```

- If we use the **==** operator to compare them, we get the default behavior for programmer-defined types—the result is **True** only if they are the **same object**.

```
1 p1 = Point(200, 100)
2 p2 = Point(200, 100)
3
4 p1 == p2
```

```
False
```

- If we want to change that behavior, we can provide a special method called **__eq__** that defines what it means for two **Point** objects to be equal.

3. Equivalence and Identity

```
1  from copy import copy
2
3  class Point:
4      """Represents a point in 2-D space."""
5
6      def __init__(self, x, y):
7          self.x = x
8          self.y = y
9
10     def __str__(self):
11         return f'Point({self.x}, {self.y})'
12
13     def __eq__(self, other):
14         return (self.x == other.x) and (self.y == other.y)
15
16     def translate(self, dx, dy):
17         self.x += dx
18         self.y += dy
19
20     def translated(self, dx=0, dy=0):
21         new_point = copy(self)
22         new_point.translate(dx, dy)
23         return new_point
```

- This definition considers two **Points** to be equal if their attributes are equal.
  - Now when we use the **==** operator, it invokes the **__eq__** method, which indicates that **p1** and **p2** are considered equal:

```
1  p1 = Point(200, 100)
2  p2 = Point(200, 100)
3
4  p1 == p2
```

```
True
```

  - But the **is** operator still indicates that they are diffe

```
1  p1 = Point(200, 100)
2  p2 = Point(200, 100)
3
4  p1 is p2
```

```
False
```

3. Equivalence and Identity

- It's not possible to override the **is** operator—it always checks whether the objects are **identical**.
- But for programmer-defined types, you can override the **==** operator so it checks whether the objects are **equivalent**. You can define what equivalent means.

4. Creating a Rectangle

- Define a class that represents and draws rectangles. To keep things simple, we'll assume that the rectangles are either vertical or horizontal, not at an angle.
- What attributes do you think we should use to specify the location and size of a rectangle? There are at least two possibilities:
  - You could specify the width and height of the rectangle and the location of one corner.
  - You could specify two opposing corners.

```
1  class Rectangle:
2      """Represents a rectangle.
3      attributes: width, height, corner.
4      """
5      def __init__(self, width, height, corner):
6          self.width = width
7          self.height = height
8          self.corner = corner
9
10     def __str__(self):
11         return f'Rectangle({self.width}, {self.height}, {self.corner})'
12
13
```

- The **__init__** method assigns the parameters to attributes and the **__str__** returns a string representation of the object.

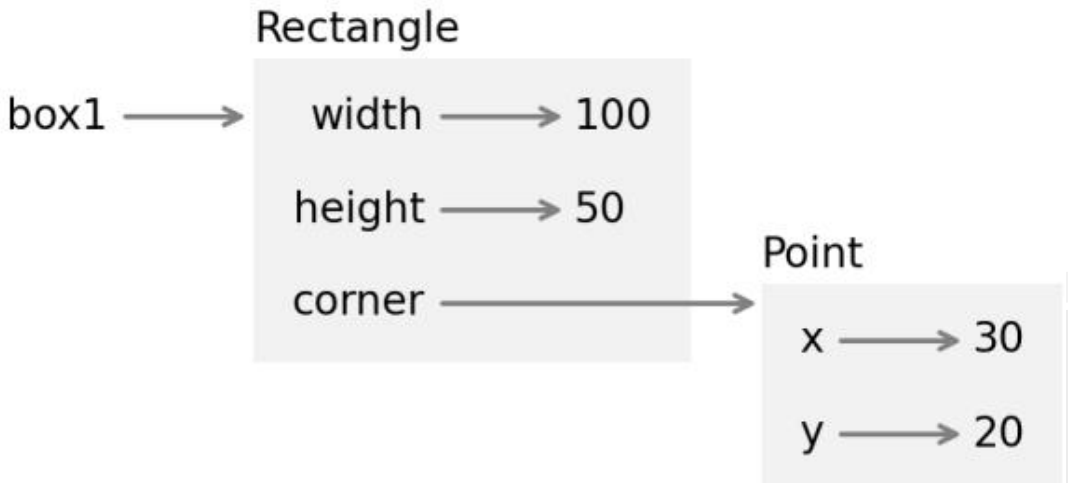
4. Creating a Rectangle

- Instantiate a **Rectangle** object, using a **Point** as the location of the upper-left corner:

```
1 corner = Point(30, 20)
2 box1 = Rectangle(100, 50, corner)
3 print(box1)
```

```
Rectangle(100, 50, Point(30, 20))
```

- The following diagram shows the state of this object:



4. Creating a Rectangle

- Define a method **make_points** to return four **Point** object to represent the corners of a rectangle:

```
1  class Rectangle:
2      """Represents a rectangle.
3      attributes: width, height, corner.
4      """
5      def __init__(self, width, height, corner):
6          self.width = width
7          self.height = height
8          self.corner = corner
9
10     def __str__(self):
11         return f'Rectangle({self.width}, {self.height}, {self.corner})'
12
13     def make_points(self):
14         p1 = self.corner
15         p2 = p1.translated(self.width, 0)
16         p3 = p2.translated(0, self.height)
17         p4 = p3.translated(-self.width, 0)
18         return p1, p2, p3, p4
```

4. Creating a Rectangle

- Define a method **make_lines** to return four **Line** objects to represent the sides of a rectangle:

```
1  class Rectangle:
2      """Represents a rectangle.
3      attributes: width, height, corner.
4      """
5      def __init__(self, width, height, corner):
6          self.width = width
7          self.height = height
8          self.corner = corner
9
10     def __str__(self):
11         return f'Rectangle({self.width}, {self.height}, {self.corner})'
12
13     def make_points(self):
14         p1 = self.corner
15         p2 = p1.translated(self.width, 0)
16         p3 = p2.translated(0, self.height)
17         p4 = p3.translated(-self.width, 0)
18         return p1, p2, p3, p4
19
20     def make_lines(self):
21         p1, p2, p3, p4 = self.make_points()
22         return Line(p1, p2), Line(p2, p3), Line(p3, p4), Line(p4, p1)
```

5. Changing Rectangles

- Consider two methods that modify rectangles, **grow** and **translate**.
  - **grow** : Enlarge a rectangle
  - **translate** : Translate a rectangle from a position to another position in the plane

```
corner = Point(30, 20)
box1 = Rectangle(100, 50, corner)

box2 = copy(box1)
box2.grow(10, 10)

print(box1)
print(box2)
```

```
Rectangle(100, 50, Point(30, 20))
Rectangle(110, 60, Point(30, 20))
```

```
class Rectangle:
    """Represents a rectangle.
    attributes: width, height, corner.
    """
    def __init__(self, width, height, corner):
        self.width = width
        self.height = height
        self.corner = corner

    def __str__(self):
        return f'Rectangle({self.width}, {self.height}, {self.corner})'

    def make_points(self):
        p1 = self.corner
        p2 = p1.translated(self.width, 0)
        p3 = p2.translated(0, self.height)
        p4 = p3.translated(-self.width, 0)
        return p1, p2, p3, p4

    def make_lines(self):
        p1, p2, p3, p4 = self.make_points()
        return Line(p1, p2), Line(p2, p3), Line(p3, p4), Line(p4, p1)
```

```
    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight
```

5. Changing Rectangles

```
corner = Point(30, 20)
box1 = Rectangle(100, 50, corner)

box2 = copy(box1)
box2.grow(10, 10)
box2.translate(20, 20)

print(box1)
print(box2)
```

```
Rectangle(100, 50, Point(50, 40))
Rectangle(110, 60, Point(50, 40))
```

- It looks like both rectangles moved, which is not what we intended!

```
class Rectangle:
    """Represents a rectangle.
    attributes: width, height, corner.
    """
    def __init__(self, width, height, corner):
        self.width = width
        self.height = height
        self.corner = corner

    def __str__(self):
        return f'Rectangle({self.width}, {self.height}, {self.corner})'

    def make_points(self):
        p1 = self.corner
        p2 = p1.translated(self.width, 0)
        p3 = p2.translated(0, self.height)
        p4 = p3.translated(-self.width, 0)
        return p1, p2, p3, p4

    def make_lines(self):
        p1, p2, p3, p4 = self.make_points()
        return Line(p1, p2), Line(p2, p3), Line(p3, p4), Line(p4, p1)

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight

    def translate(self, dx, dy):
        self.corner.translate(dx, dy)
```

6. Deep Copy

```
corner = Point(30, 20)
box1 = Rectangle(100, 50, corner)

box2 = copy(box1)
box2.grow(60, 40)
box2.translate(30, 20)

print(box1)
print(box2)
```

```
Rectangle(100, 50, Point(60, 40))
Rectangle(160, 90, Point(60, 40))
```

```
box1 is box2
```

```
False
```

```
box1.corner is box2.corner
```

```
True
```

- When we use **copy** to duplicate **box1**, it creates a new **Rectangle** object and copies the **Rectangle** object but not the **Point** object it contains.
- So **box1** and **box2** are different objects, but their **corner** attributes refer to the **same object**.

```
box1 is box2
```

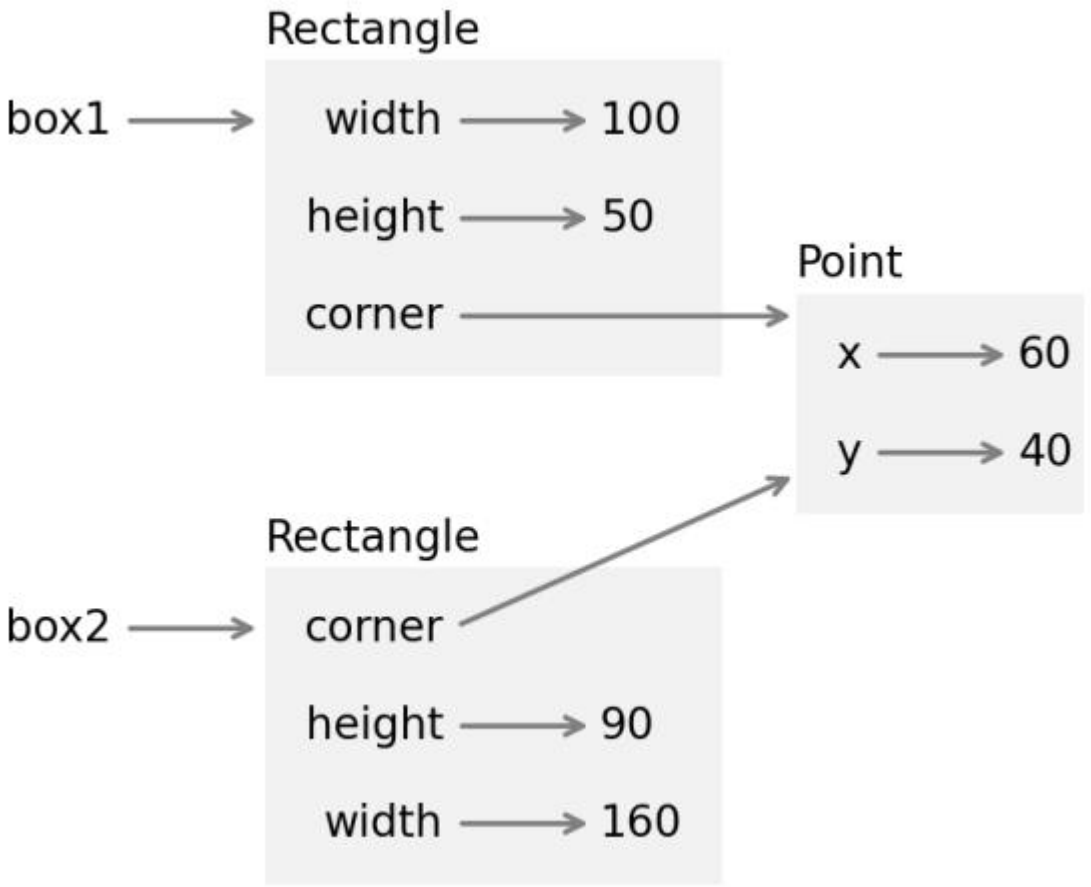
```
False
```

```
box1.corner is box2.corner
```

```
True
```

6. Deep Copy

- The following diagram shows the state of these objects:



- What copy does is create a **shallow copy** because it copies the object **but not the objects it contains**.
  - As a result, changing the **width** or **height** of one **Rectangle** does not affect the other, but changing the attributes of the shared **Point** affects both! This behavior is confusing and error prone.
- Fortunately, the copy module provides another function, called **deepcopy**, that copies not only the object but also the objects it refers to, and the objects they refer to, and so on. This operation is called a **deep copy**.

6. Deep Copy

```
from copy import deepcopy

corner = Point(30, 20)
box1 = Rectangle(100, 50, corner)

box2 = deepcopy(box1)
box2.grow(60, 40)
box2.translate(30, 20)

print(box1)
print(box2)
```

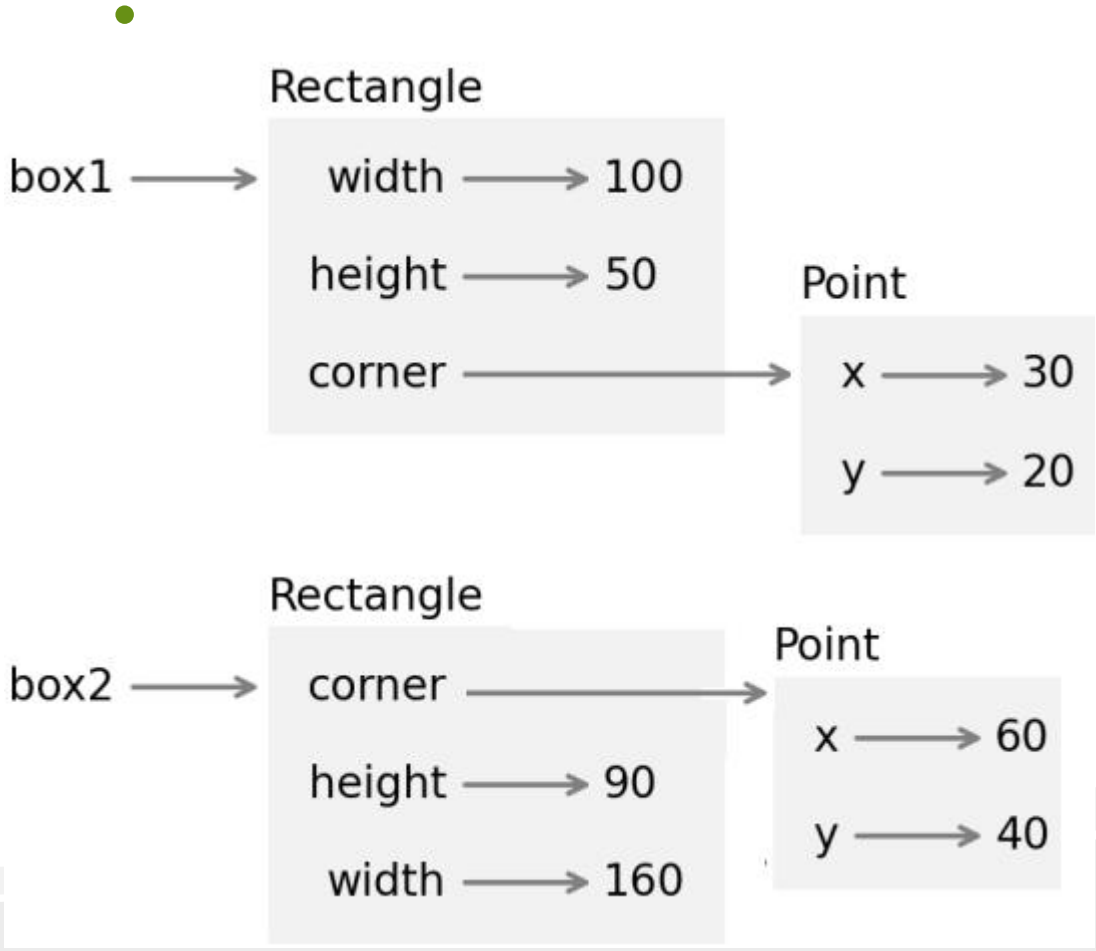
```
Rectangle(100, 50, Point(30, 20))
Rectangle(160, 90, Point(60, 40))
```

```
box1 is box2
```

```
False
```

```
box1.corner is box2.corner
```

```
False
```



7. Polymorphism

```
point = Point(10, 20)
line = Line(Point(0,0), Point(100, 200))
rectangle = Rectangle(50, 80, Point(100, 100))

shapes = [point, line, rectangle]
for shape in shapes:
    shape.translate(10, 20)
```

- In OOP, **polymorphism** is the ability of different types to provide the same methods, which makes it possible to perform many computations by invoking the **same method** on **different types of objects**.
  - The word comes from Greek roots that mean “many shaped.”