# Module 5
# Classes and Objects

**Q) Define class, object, attributes, object diagram.**

A programmer-defined type is also called a class. A class definition looks like this: class Point: """Represents a point in 2-D space."""

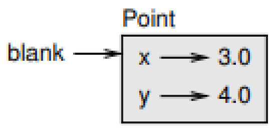


Figure : Object diagram.

The header indicates that the new class is called Point. The body is a docstring that explains what the class is for. We can define variables and methods inside a class definition, but we will get back to that later.
Defining a class named Point creates a class object.
>>> Point
<class '_ _ main _ _ Point'>
Because Point is defined at the top level, its "full name" is __main__.Point.
The class object is like a factory for creating objects. To create a Point, we call Point as if it were a function.
>>> blank = Point()
>>> blank
<_ _ main _ _ Point object at 0xb7e9d3ac>
The return value is a reference to a Point object, which we assign to blank.
Creating a new object is called instantiation, and the object is an instance of the class

We can assign values to an instance using dot notation:
>>> blank.x = 3.0
>>> blank.y = 4.0
This syntax is similar to the syntax for selecting a variable from a module, such as math.pi or string.whitespace. In this case, though, we are assigning values to named elements of an object. These elements are called attributes.

Figure above is a state diagram that shows the result of these assignments. A state diagram that shows an object and its attributes is called an object diagram.

The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.

>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0

The expression blank.x means, “Go to the object blank refers to and get the value of x.” In the example, we assign that value to a variable named x. There is no conflict between the variable x and the attribute x.

We can use dot notation as part of any expression.

**Example:**

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

We can pass an instance as an argument in the usual way.

**Example:**

```
def print_point(p):
        print('(%g, %g)' % (p.x, p.y))
```

print_point takes a point as an argument and displays it in mathematical notation. To invoke it, we can pass blank as an argument:

```
>>> print_point(blank)
(3.0, 4.0)
```

Inside the function, p is an alias for blank, so if the function modifies p, blank changes.

**Q) Explain the concept of copying using copy module with an example.**

Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

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

p1 and p2 contain the same data, but they are not the same Point.

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
```

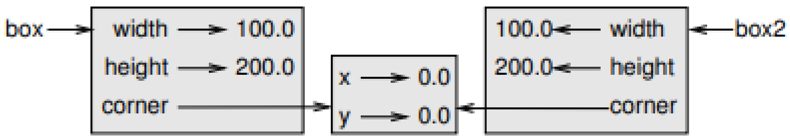


Figure : Object diagram.

```
>>> p1 == p2
False
```

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

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

Figure above shows what the object diagram looks like. This operation is called a shallow copy because it copies the object and any references it contains, but not the embedded objects.

**Q) Briefly explain about pure function and give example.**

The function creates a new Time object, initializes its attributes, and returns a reference to the new object. This is called a pure function because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

```
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

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

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

>>> done = add_time(start, duration)
>>> print_time(done)
```

10:80:00

**Q) Explain modifiers and give example.**

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 naturally as a modifier. Here is a rough draft:

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60 time.minute += 1
    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

**Q) Briefly explain about Object-oriented features**

Python is an object-oriented programming language, which means that it provides features that support object-oriented programming, which has these defining characteristics:

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

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.

**Q) Briefly explain the Printing of objects with an example.**

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

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

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

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

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

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

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

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.

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

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

**Q) Explain _ _init_ _() and _ _str_ _() method with an examples.**

**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 statement

self.hour = hour

stores the value of the parameter hour as an attribute of self.

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

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

If we provide one argument, it overrides hour:

```
>>> time = Time (9)
```

```
>>> time.print_time()
09:00:00
```

If we provide two arguments, they override hour and minute.

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

And if we provide three arguments, they override all three default values

**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 we 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.

**Q) Briefly explain about Operator overloading.**

By defining other special methods, we can specify the behavior of operators on programmer-defined types. For example, if we define a method named __add__ for the Time class, we 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)

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

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

Changing the behavior of an operator so that it works with programmer-defined types is called operator overloading. For every operator in Python there is a corresponding special method, like __add__.

**Q) Briefly explain about Type-based dispatch.**

We added two Time objects, but we 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)
```

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.

**Example:**

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

**Q) Briefly explain about Polymorphism.**

Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often we can avoid it by writing functions that work correctly for arguments with different types. Many of the functions we wrote for strings also work for other sequence types.

**Example: H**istogram 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
```

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 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. 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
```