# Unit 8

●●●

Lesson 1 - Classes, Objects, and Methods

## Learning Targets

- I can define the terms class and object as well as create them inside their programs.
- I can create an init method to give attributes to objects.
- I can create methods inside class definitions and call them on objects.

## From MIT Professor Dr. Ana Bell

“Next, I’d like to thank my parents and sister. My dad taught me programming when I was 12, and I’ll never forget how many times he had to explain object-oriented programming to me before it finally clicked.”

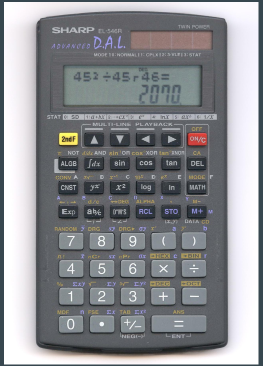
## Objects

You interact with objects every day

Every object is made up of other objects

Calculator can be decomposed into circuit board, screen, buttons, etc.

Each object has behaviors - calculator can add, subtract, multiply, divide, etc. but cannot graph functions



## Objects

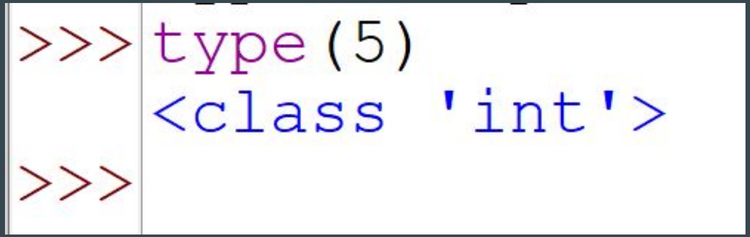
As you build more complex systems, you can reuse objects

For example, a more advanced calculator can reuse the logic chip that does basic arithmetic

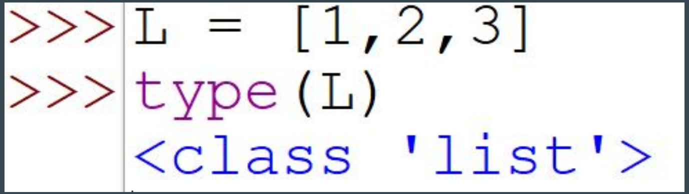
You have actually been using objects since you started writing Python. Python is a multi-paradigm language that can be object oriented, but it is object based. It uses objects even though you do not have to have a deep understanding of object oriented programming (OOP) to do powerful things with it

## Objects

- Integers, floats, strings, Booleans, tuples, lists, and dictionaries are all types of objects.
- They can be thought of as the atomic (scalar) objects since they are the fundamental objects of the language
- Strings (sort of), tuples, lists, sets, and dictionaries are nonatomic (container or collection) objects because they can be decomposed into other objects. Sets and dictionaries are unordered.



```
>>> type(5)
<class 'int'>
>>>
```



```
>>> L = [1,2,3]
>>> type(L)
<class 'list'>
```

## Objects

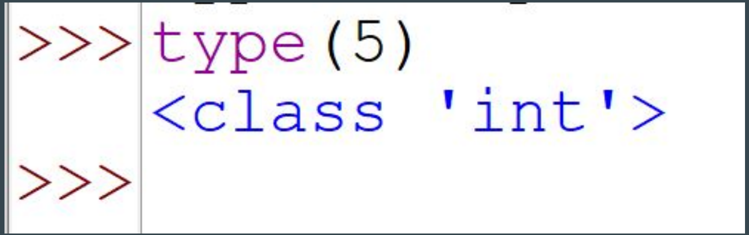
In Python, an object is an instance of a class or type

- 5 is an instance of int
- [1,2,3] is an instance of list

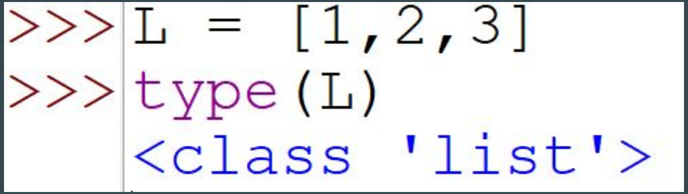
Objects are a data abstraction that have

1) an internal data representation

2) an interface for interactive with the object through methods - behaviors are defined by implementations are hidden



```
>>> type(5)
<class 'int'>
>>>
```



```
>>> L = [1,2,3]
>>> type(L)
<class 'list'>
```

## Objects

- **EVERYTHING IN PYTHON IS AN OBJECT**
- You can create new objects of some type
- You can manipulate objects
- You can destroy objects
  - Explicitly use del or just forget about them - Python has garbage collection

## Classes

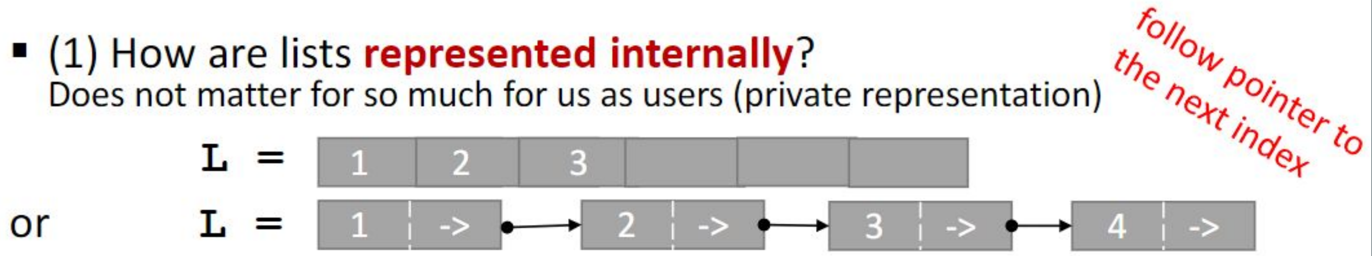
- The first part of creating your own object type is to define the class. You use the class keyword to do this.

class Point:                                    note: sometimes you may see class Point(object)

- Classes are the template, or blueprint, for making objects

# EXAMPLE:
# [1,2,3,4] has type list

- (1) How are lists **represented internally**?
  Does not matter for so much for us as users (private representation)



- (2) How to **interface with, and manipulate,** lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- Internal representation should be private
- Correct behavior may be compromised if you manipulate internal representation directly

## Advantages of OOP

- Group data, functions, and constants into modules
- Divide and conquer development
  - Increased modularity reduces complexity
  - Easy to test code as well
- Classes make it easy to reuse code

## BIG IDEA

You write the class so you make the design decisions.

You decide what data represents the class.

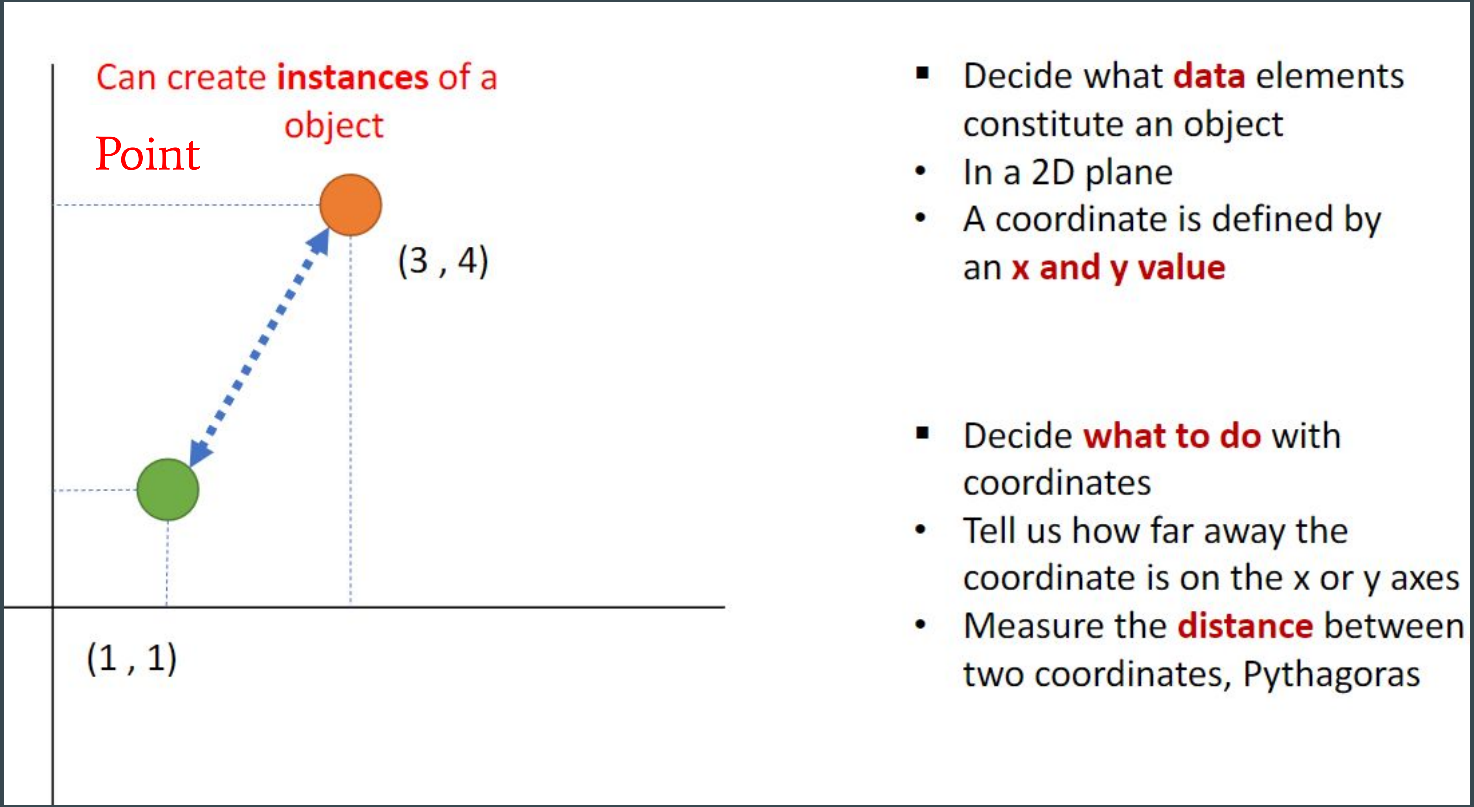
You decide what operations a user can do with the class

We are going to make the Point class

## CREATING AND USING YOUR  OWN TYPES WITH CLASSES

- Distinction between creating a class and using an instance of the class
- Creating the class involves
  - Defining the class name
  - Defining class attributes
- Using the class involves
  - Creating new instances of the class
  - Doing operations on the instances
  - for example, L=[1,2] and len(L)

## Point design decisions

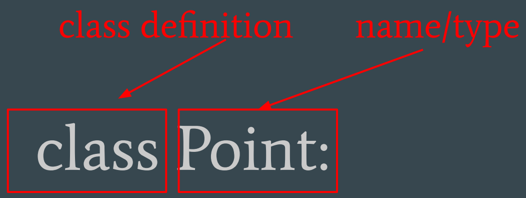


- Decide what **data** elements constitute an object
  - In a 2D plane
  - A coordinate is defined by an **x and y value**

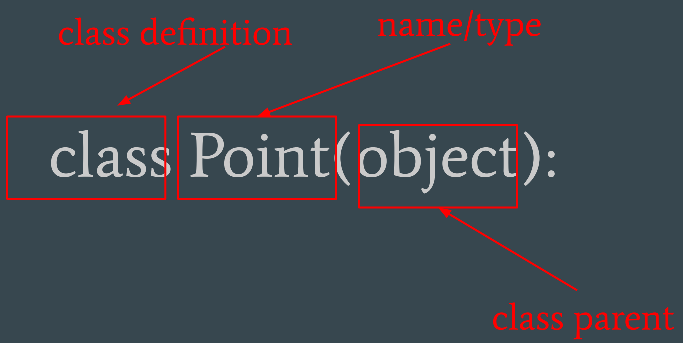
- Decide **what to do** with coordinates
  - Tell us how far away the coordinate is on the x or y axes
  - Measure the **distance** between two coordinates, Pythagoras

# Define your own types

- use the class keyword to define a new type



```python
class Point:
```



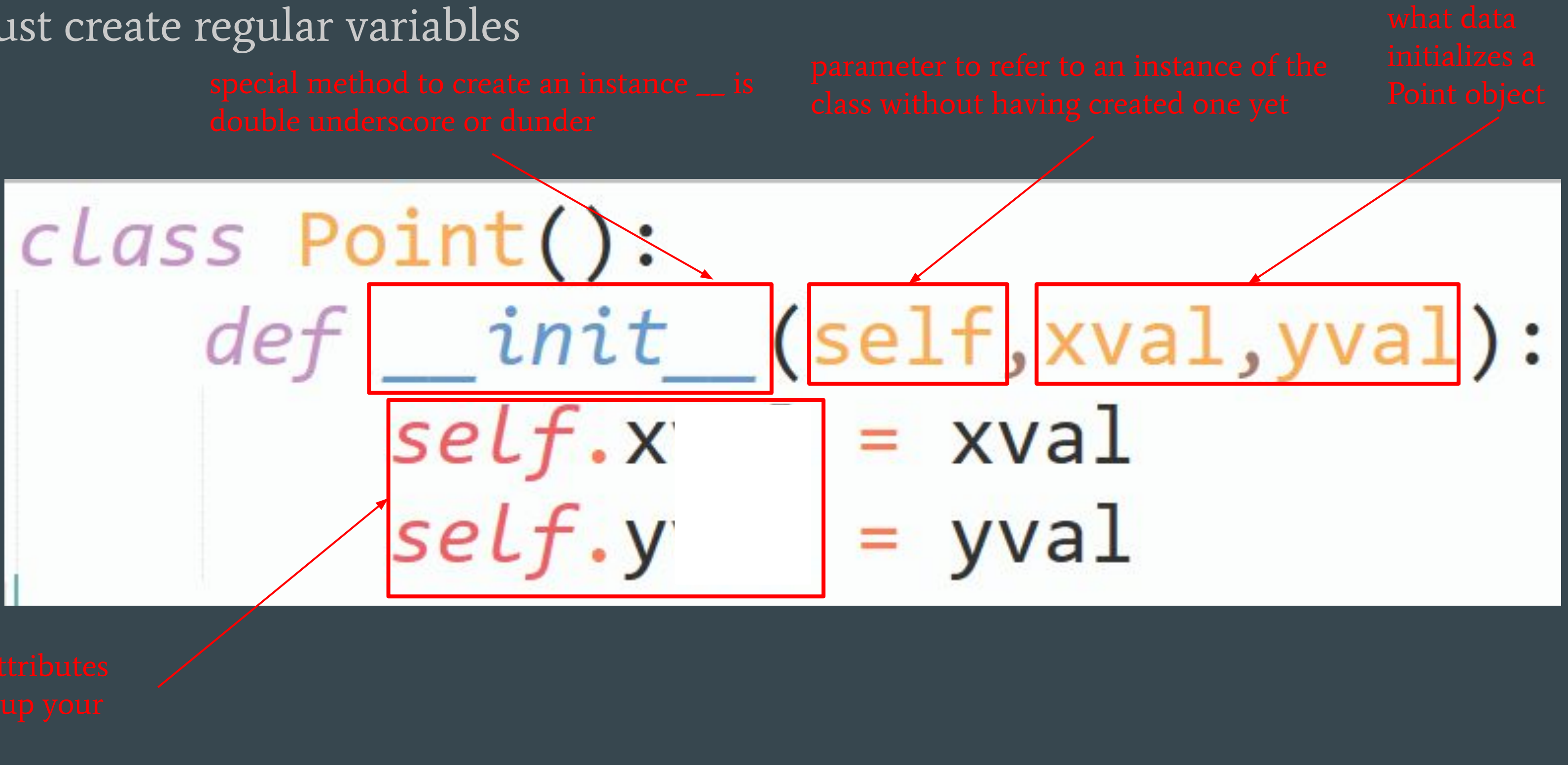
```python
class Point(object):
```

## Attributes

- Data and procedures that belong to the class and work with the class
- Data attributes
  - e.g. a point is made up of two numbers
- Methods
  - functions that only work with the class
  - e.g. distance between two points but it doesn’t make sense to talk about the distance between two booleans

## Constructor - creating an instance

Special method called __init__ to initialize some data attributes or perform initialization. Self allows you to create variables that belong to this object. Without self you just create regular variables

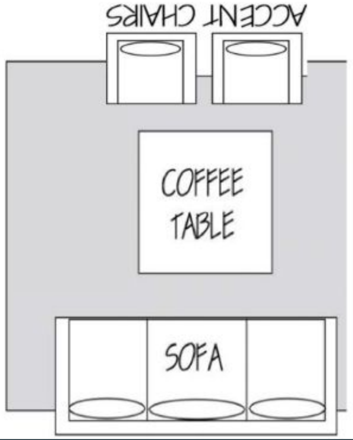

special method to create an instance __ is double underscore or dunder
parameter to refer to an instance of the class without having created one yet
what data initializes a Point object
two data attributes that make up your type

```
class Point():
    def __init__(self,xval,yval):
        self.x = xval
        self.y = yval
```

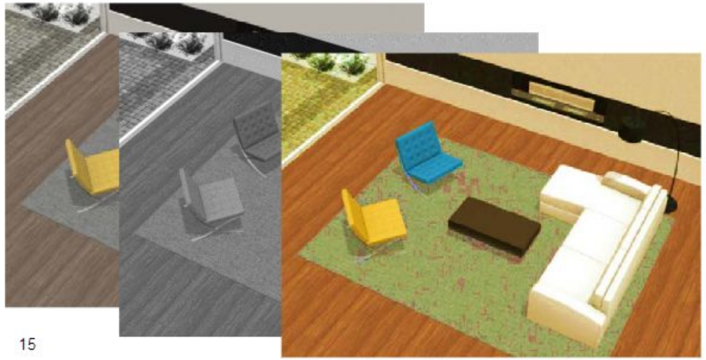
## WHAT is self? ROOM EXAMPLE

self is the blueprint's way of accessing attributes (data and methods)

- Think of the class definition as a **blueprint** with placeholders for actual items
  - self has a chair
  - self has a coffee table
  - self has a sofa



- Now when you create **ONE** instance (name it living_room), self becomes this actual object
  - living_room has a blue chair
  - living_room has a black table
  - living_room has a white sofa
- Can make **many instances** using the same blueprint



## Big Idea

When *defining* a class, we don’t have an actual object - just a definition, a blueprint

## Implementing vs. Using Classes

**Implementing**

```
class Point():

    def __init__(self,xval,yval):

        self.x = xval

        self.y = yval
```

**Using**

Create a new object of type Point and pass in 3 and 4 to the __init__

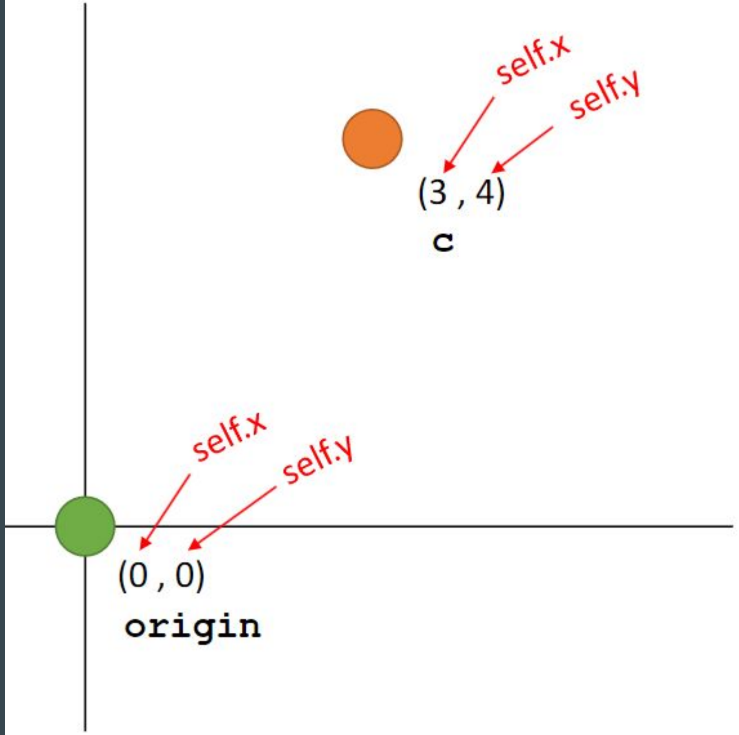
```
c = Point(3,4)

origin = Point(0,0)

print(c.x)
```

Data attributes of an instance are called instance variables. You can access them with the dot notation. p1.x gives you the x value of the p1 object which is an instance of the Point class.

## VISUALIZING INSTANCES: draw it



The template for a Coordinate type

```
class Coordinate(object):
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

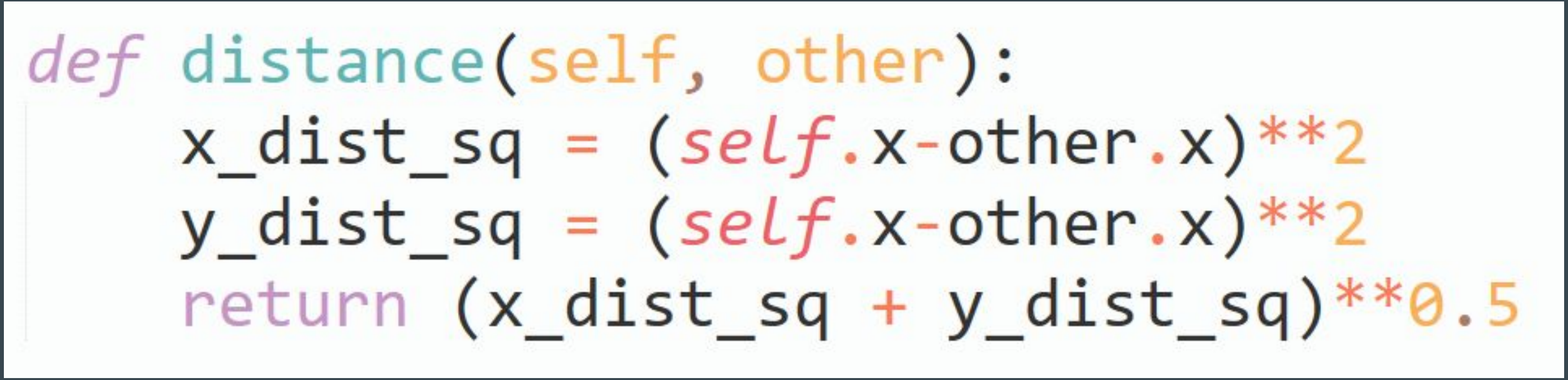
```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

Code to make actual tangible Coordinate objects (aka instances)

Pretend Coordinate is Point

## Methods

- A methods is a function that only works with a class
- Python always passes the object as the first argument, although you will not write it when you call the method
- Other than self and dot notation, methods behave just like functions (take params, do operations, return)



```
def distance(self, other):
    x_dist_sq = (self.x-other.x)**2
    y_dist_sq = (self.x-other.x)**2
    return (x_dist_sq + y_dist_sq)**0.5
```

- The “.” **operator** is used to access any attribute
  - A data attribute of an object (we saw `c.x`)
  - A method of an object
- Dot notation

```
<object_variable>.<method>(<parameters>)
```

Object to call method on, becomes self in the class def

Name of method

Not including self. `self` is the obj before the dot!

- Familiar?

```
my_list.append(4)
my_list.sort()
```

# HOW TO USE A METHOD

**Recall the definition of distance method:**

```
def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

## Using the class:

```
c = Coordinate(3,4)
orig = Coordinate(0,0)
print(c.distance(orig))
```

object to call method on

name of method

parameters not including self (self is implied to be c)

- Notice that `self` becomes the object you call the method on (the thing before the dot!)