

Classes

Class

- A class is an abstraction used as a template for creating objects.
- A class can also be used to describe all the objects of that type.
- Every object instantiated from a class will have the same set of attributes (with different values) and the same methods



Object

In Python, everything is an object - lists, dictionaries, strings, etc. Python is considered an **object-oriented** programming language. If we write classes, we can create our own custom objects, which will be similar to data types in Python.

You can think of a class as a blueprint, and an object as the actual house built from that blueprint. There can be multiple objects made from the same class, and they are all slightly different.

For example, a Python data type is `<list>`, this is the class. You can have two lists:

```
list1 = [1, 2, 3]
list2 = ["Python", "is", "fun"]
```

These lists are both objects derived from the list class, but they are different from each other.



Creating and Instantiating a Class

Similar to functions, loops, and conditionals, everything in a class is indented.

Classes have functions inside them. Functions that are part of a class are called **methods** (think of the built-in string/list/etc. methods we've seen so far).

Every class needs the **initializer** method (also called **constructor**), which initializes the class and sets its attributes. It must be called `__init__`

When you create an **object** from a class, you call the initializer and pass in values for the attributes.

```
Class <Classname>(object):
```

```
    def __init__(self, param1=default, param2=default):  
        self.variable1 = param1  
        self.variable2 = param2
```



```
obj = Classname(arg1, agr2)
```

Self

In Python, `self` is a keyword that refers to `this object`.

- In some other programming languages, the keyword is “this”

Most class methods take `self` as the first parameter.

However, when you call the method, you don't pass in `self` as an argument. So typically when you call class methods, you have one fewer argument than parameters.

Also, when you refer to attributes within a class, you must say `self.attribute` instead of just `attribute`.



Example: Point2D Class

Let's make a class that represents a point in two dimensions.


Here is the start of our class, with a class definition and initializer:

```
Class Point2D(object):
```

```
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y
```

```
my_point = Point2D(2,3)
```

Create a point object with
attributes x=2, y=3



Python Built-In Class Methods

Python has some built-in methods you can add to your classes. Every built-in method you add must be named specifically after a Python keyword with two underscores on each side.

The only required method is `__init__`. Another useful one is `__str__`.

The purpose of most other built-in methods is to **override operators**, so when you use an operator like `+` on objects from your class, it calls your method instead.

You can give operators like `+`, `==`, `<`, etc. any functionality you want.



Python Built-In Class Methods

Operator	Method	Description
str()	<code>__str__(self)</code>	Return a string representation of this object.
+	<code>__add__(self, other)</code>	Adds this object to another object from the same class, return a new object.
-	<code>__sub__(self, other)</code>	Subtracts another object from this object, return a new object.
==	<code>__eq__(self, other)</code>	Test equality between this object and another, return a boolean.

You can make a method for almost any Python operator you can think of.
Just make sure you get the name right.



Adding to Point2D

Let's add a `__str__` method to our `Point2D` class so we can print `Point2D` objects.

Our updated class:

```
class Point2D(object):  
  
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y  
  
    def __str__(self):  
        return f"({self.x}, {self.y})"
```

What other methods should we add to our `Point2D` class?



Equality

Another good method to add to any class is `__eq__`.

The default behavior for using `==` is reference equality, which checks if two things are the same object.

To implement value equality, we can write a `__eq__` method that returns `True` if all the attributes have the same value.

```
def __eq__(self, other):  
    if self.x == other.x and self.y == other.y:  
        return True  
    return False
```



Exercise

```
point1 = Point2D(3,4)
point2 = Point2D(3,4)

print(point1 == point2)
```

What is the output of this code if we **don't** override ==?

What is the output if we override == to use value equality?

Is it the same or different? Why?



```
class Point2D(object):

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

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

    def __eq__(self, other):
        if self.x == other.x and self.y == other.y:
            return True
        return False

    def __add__(self, other):
        return Point2D(self.x + other.x, self.y + other.y)
```



Exercise

Write the method that subtracts two Point2d objects.

```
def __sub__(self, other):
```



Exercise

Write the method that checks if one Point2D object is less than another Point2D object (<)

```
def __lt__(self, other):
```

How do you want to implement this method? What does it mean for one point to be "less than" another point?



Rules for Classes

- Don't create attributes outside the class.
- Don't directly access or change attributes except through class methods.
 - Languages like C++ and Java have constructions that enforce this.
 - In languages like Python it is not a hard-and-fast rule.
- Class design is often most effective by thinking about the required methods rather than the required attributes.
 - As an example, we rarely think about how the Python list and dict classes are implemented.



Accessor and Mutator Methods

These methods are created so the user can't directly access attributes.

An **accessor method** returns an attribute to the user, and a **mutator method** lets the user change an attribute.

They are more important in more complex classes, and in other languages like Java or C++.

- In other programming languages, you make class attributes "private" so they can't be accessed outside of the class, which makes accessor and mutator methods necessary.

They aren't usually necessary in Python, but they can be added in Python.



Accessor and Mutator Methods

With our Point2D class, the user can directly use the attributes like so:

```
var = point.x    # Directly accessing attribute  
point.x = 5      # Directly modifying attribute
```

But we can write methods in our Point2D to do this indirectly:

```
def change_x(self, new_x): # Mutator method  
    self.x = new_x  
def get_x(self): # Accessor method  
    return self.x
```

Now the user can do this instead:

```
var = point.get_x()      # Indirectly accessing attribute  
point.change_x(5)        # Indirectly modifying attribute
```



Exercise

This class represents a date, but it's incomplete!

Fill out the missing methods so it's a functional class.

```
class Date(object):  
  
    def __init__(self, year=1970, month=1, day=1):  
        self.year = year  
        self.month = month  
        self.day = day  
  
    def __str__(self):  
        pass  
  
    def __eq__(self, other):  
        pass  
  
    def __lt__(self, other):  
        pass  
  
    def is_leap_year(self):  
        pass
```



Printing Dates

```
def __str__(self):  
    # Return a string representing the date
```

The string method should return a string that represents the date, in this format:
mm/dd/yyyy

For example, if you run this code:

```
date1 = Date(2000,1,3)  
print(date1)
```

The output should be:

01/03/2000



Comparing Dates

```
def __eq__(self, other):  
    # Return True if these dates are the same; False otherwise  
  
def __lt__(self, other):  
    # Return True if this date was before the other date.  
    # This involves checking the year, month, and day.  
    # You may want to use nested if statements.
```



Check if Leap Year

```
def is_leap_year(self):  
    # Return True if this is a leap year; False otherwise
```

In a leap year, February is 29 days instead of 28 like it is normally.

Leap years happen every year that is divisible by 4.

However, there is an exception where if a year is divisible by 100 (ex: the year 1900) it is not a leap year.

However, there is an exception to that, where if a year is divisible by 400 (ex: the year 2000) it is a leap year.



Resources

https://www.w3schools.com/python/python_classes.asp

