

Classes and Objects

Python, like C++ and Java, is **object oriented**. The basic data model in Python is that everything is an object of some sort. An **object** combines data and methods. Everything in Python is an object, including "simple" data like integers and floats.

A **class** in python defines a set of **attributes**: these can be variables or methods. This defines a set of properties that you want all objects of a certain type to have. An object in Python is an **instance** of a class: it shares attributes with all other classes, but can also have attributes that are different from other instances. This lets you have objects with their own "local" data.

Methods for a class take an extra `self` argument. When you invoke a method on an object (think `myList.append(x)`), this `self` argument refers to the object you invoked the method on (in the example, `myList`).

Let's walk through an example of defining a class for a counter, and instantiating objects from this class to keep their own counts:

```
In [1]: class Counter () :
        totalCount = 0 #shared number across all instances

        def __init__(self) : #constructor for the class
            self.count = 0 #local count for each instance

        def incr(self) : #method for the class
            Counter.totalCount += 1
            self.count += 1

        def __str__(self) : #special function which overwrites "print" for t
his class
            return "Total count: {}, Local count: {}".format(Counter.totalCo
unt, self.count)
```

```
In [2]: c1 = Counter()
        c2 = Counter()
        print(c1)
        print(c2)
```

```
Total count: 0, Local count: 0
Total count: 0, Local count: 0
```

```
In [3]: for i in range(0,5):
        c1.incr()
        c2.incr()

        print(c1)
        print(c2)
```

Total count: 10, Local count: 5

Total count: 10, Local count: 5

Classes themselves, like functions, are just objects, as are the methods inside them:

```
In [4]: print(type(Counter))
        print(type(Counter.incr))
```

<class 'type'>

<class 'function'>

Here is one of the examples we did in the lecture slides:

```
In [5]: class Foo :
        x = 7 #this will be accessible to all Foos

        #called when a new Foo is created
        def __init__(self, i) :
            self.y = i #this is specific to each Foo

        def bar(self) :
            return self.x + self.y
```

```
In [6]: a = Foo(1) #a.x = 7, a.y = 1
        b = Foo(2) #b.x = 7, b.y = 2

        print(a.bar()) #prints 8
        print(b.bar()) #prints 9
```

8

9

Here is an example of a class for "cars" which tracks variables such as make/model and includes a method to calculate sale price:

```
In [7]: class Car() :
        """A car for sale by Jeffco Car Dealership.

        Attributes:
            wheels: An integer representing the number of wheels the car has.

            miles: The integral number of miles driven on the car.
            make: The make of the car as a string.
            model: The model of the car as a string.
            year: The integral year the car was built.
            sold_on: The date the vehicle was sold.
        """

        def __init__(self, wheels, miles, make, model, year, sold_on):
            """Return a new Car object."""
            self.wheels = wheels
            self.miles = miles
            self.make = make
            self.model = model
            self.year = year
            self.sold_on = sold_on

        def sale_price(self):
            """Return the sale price for this car as a float amount."""
            if self.sold_on is not None:
                return 0.0 # Already sold
            return 5000.0 * self.wheels

        def purchase_price(self):
            """Return the price for which we would pay to purchase the car.
            r."""
            if self.sold_on is None:
                return 0.0 # Not yet sold
            return 8000 - (.10 * self.miles)
```

```
In [8]: v = Car(4, 0, 'Honda', 'Accord', 2014, None)
```

```
In [9]: v.sale_price()
```

```
Out[9]: 20000.0
```

```
In [10]: v.purchase_price()
```

```
Out[10]: 0.0
```

```
In [11]: v.sold_on = '10-31-2019'
```

```
In [12]: v.purchase_price()
```

```
Out[12]: 8000.0
```

For more information and examples, please refer to documentation on the Python data model (<https://docs.python.org/2/reference/datamodel.html> (<https://docs.python.org/2/reference/datamodel.html>)) and Python classes (<https://docs.python.org/2/tutorial/classes.html> (<https://docs.python.org/2/tutorial/classes.html>)).

All examples from lecture notes

```
In [13]: # Integers, lists, functions and objects
# (and even classes) are objects in Python
my_integer = 5
my_list = [1.0, 2, 3]
def my_function(): return 0
class MyClass: pass
my_object = MyClass()

# Show id and type of each object
for o in [my_integer, my_list,
          my_function, my_object, MyClass]:
    print(f'id={id(o)}, type={type(o)}')
```

id=4429469216, type=<class 'int'>
id=4483350856, type=<class 'list'>
id=4484656120, type=<class 'function'>
id=4484987984, type=<class '__main__.MyClass'>
id=140526985691224, type=<class 'type'>

```
In [14]: class Foo :
    x = 7 #this will be accessible to all Foos
    #it is a class variable

    #this is called when a new Foo is created
    def __init__(self, i) :
        self.y = i #this is specific to each Foo
        #it is an instance variable

    #this will be available to all Foos
    #it is a class method
    def bar(self) :
        return self.x + self.y

#defining objects as instances of class Foo
a = Foo(1) #a.x = 7, a.y = 1
b = Foo(2) #b.x = 7, b.y = 2

#invoking the bar method on the objects
print(a.bar()) #prints 8
print(b.bar()) #prints 9
```

8

9

```
In [15]: class SimpleClass():
    def __init__(self, x):
        # internal created
        self.myx = x
    def add(self, y):
        # internal access and update
        self.myx = self.myx + y
my_object = SimpleClass(10)
# external access
print(my_object.myx) # 10
# internal update
my_object.add(15)
print(my_object.myx) # 25
# external update
my_object.myx = 200
print(my_object.myx) # 200
# external variable creation
my_object.myz = 18
print(my_object.myz) # 18
# external variable deletion
del my_object.myz
try:
    print(my_object.myz) # Error
except:
    print('Error accessing myz since deleted')
```

10

25

200

18

Error accessing myz since deleted

```
In [16]: class MultipleLists():
    def __init__(self):
        self.lists = []
    def __add__(self, a):
        newlists = MultipleLists()
        newlists.lists = self.lists.copy()
        newlists.lists.append(a)
        return newlists
    def __len__(self):
        return sum([len(a) for a in self.lists])
    def __str__(self):
        return ', '.join([
            f'L{i+1}={a}'
            for i, a in enumerate(self.lists)
        ])
many_lists = MultipleLists()
print(many_lists)      # ''
print(len(many_lists)) # 0
many_lists = many_lists + [3,5,1]
print(many_lists)     # L1=[3, 5, 1]
print(len(many_lists)) # 3
many_lists += [8, 4]
print(many_lists)     # L1=[3, 5, 1], L2=[8, 4]
print(len(many_lists)) # 5
```

```
0
L1=[3, 5, 1]
3
L1=[3, 5, 1], L2=[8, 4]
5
```

```
In [17]: class Employee:
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print("Total employees: %d" % Employee.empCount)

    def displayEmployee(self):
        print("Name: ", self.name, ", Salary: ", self.salary)

emp1 = Employee("Alice", 100000)
emp1.displayEmployee() # Name: Alice , Salary: 100000
emp1.displayCount()   # Total Employees: 1

emp2 = Employee("Bob", 50000)
emp2.displayEmployee() # Name: Bob , Salary: 50000
emp1.displayCount()   # Total Employees: 2
```

```
Name: Alice , Salary: 100000
Total employees: 1
Name: Bob , Salary: 50000
Total employees: 2
```