Object Oriented Programming

General information

Corey Schafer

# General Information

I already know the basics so I’m just gonna show an example.

class Employee:  
  
 num\_of\_employees = 0  
 raise\_amount = 1.04  
  
 def \_\_init\_\_**(**self, first, last, pay**)**:  
 self.first = first  
 self.last = last  
 self.pay = int**(**pay**)** self.email = first + last + **'@company.com'** Employee.num\_of\_employees += 1  
  
 def fullname**(**self**)**:  
 return**(f'{**self.first.title**()} {**self.last.title**()}')** def apply\_raise**(**self**)**:  
 self.pay = int**(**self.pay \* self.raise\_amount**)**daniel\_g = Employee**('daniel'**, **'ghirasim'**, **'23400')**print**(**daniel\_g.fullname**())**print**(**daniel\_g.pay**)**daniel\_g.apply\_raise**()**print**(**daniel\_g.pay**)**print**(**Employee.num\_of\_employees**)**

In the example above we created a class Employee.

We have 2 methods for these (methods are functions within a class) : fullname() and apply\_raise().

We also have 2 global variables which apply to all instances of the class. We can manually override these variables by typing.

Employee.num\_of\_employees = 123

Employee.raise\_amount = 1.05

We can also see that everytime we create a new instance of the class we are going to increment employee number by 1. It doesn’t make sense to initialize num\_of\_employees within the init method because we want the total number of employees to be the same for all of our instances.

# Regular methods

Regular methods in a class automatically take the instance as the first argument.

def fullname(self):  
 return(f'{self.first.title()} {self.last.title()}')

self automatically refers to the instance of the class. By convention we call this self.

# Class Methods

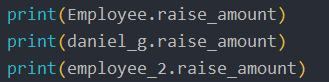
Class methods is the same as regular methods but it takes the class as it’s first argument so it’s going to refer to the class .

class Employee:  
  
 num\_of\_employees = 0  
 raise\_amount = 1.04

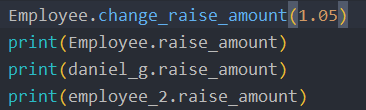
@classmethod  
def change\_raise\_amount(cls, amount):  
 cls.raise\_amount = amount

The class method is defined by the @classmethod decorator. Here by convetion we use cls since we cannot use class because that is a python keyword.

This is going to change the classes raise amount and all of our instances raise amount.

If we run the class method it is going to change all of the raise amounts

We can run class methods from instances themselves but that doesn’t really make sense so I’m not going to cover that here.

# Alternative Constructors

This means that we can use class methods in order to provide multiple ways for creating our objects.

Alternative constructors usually start with from\_ but this is just a convetion

Example:

Let’s say we receive our employee information in a weird string format and we want to make new employee instances from them.

emp\_str\_1 = 'John-Doe-70000'  
emp\_str\_2 = 'Steven-Smith-80000'

A way to parse this would be:

First, last, pay = emp\_str\_1.split(‘0’)

New\_emp\_1 = Employee(first, last, pay).

A better way to do this is to use an alternative constructor.

@classmethod  
def from\_string(cls, emp\_str):  
 first, last, pay = emp\_str.split('-')  
 return cls(first, last, pay)

Here cls is passed in automatically as the first argument and our weird string is the second argument.

After our first, last and pay variables are assigned and then we return cls(first, last, pay) what this means is that everytime we call this constructor method it is going to return us and instance of the class with the proper format.

new\_emp\_1 = Employee.from\_string(emp\_str\_1)  
print(new\_emp\_1.first, new\_emp\_1.last, new\_emp\_1.pay, new\_emp\_1.email)



## Static Methods

A lot of people get class methods and static methods confused. When working with classes, regular methods automatically pass the instance as the first argument which we call self , when working with class methods , the methods automatically pass in the class as the first argument which we call cls.

And static methods don’t pass anything automatically, so really they behave as regular functions because they have some logical connection with the class.

@staticmethod  
def is\_work\_day(day):  
 if day.weekday() == 5 or day.weekday() == 6:  
 return False  
 else:  
 return True

# In python , weekdays have these weekday methods where Monday is 0 and Sunday is 6 and everything in between are the rest of the days. (We have to import the datetime module for this)

We define a static method with the decorator @staticmethod.

We include this in our classes because they have some logical connection to our class like in the example above. Some people usually write regular methods or class methods that actually should be a static method, usually a giveaway that a method should be a static method is to check if the method doesn’t access any of the class or instance variables like in our example above. This goes for all the methods mentiond above.

import datetime  
my\_date = datetime.date(2020, 10, 14)  
print(Employee.is\_work\_day(my\_date))