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

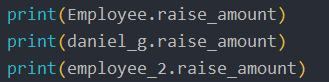
Class methods is the same as regular methods but it takes the class as it’s first argument so it’s going to refer to the class .

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

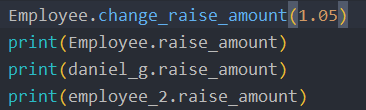
@classmethod  
def change\_raise\_amount(cls, amount):  
 cls.raise\_amount = amount

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 and they are used 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))

# Class Inheritance

Inheritance allows us to inherit attributes and methods from a parent class. This is useful because we can create subclasses and get all of the functionality of our parent class + add new functionality to our subclass without affecting our parent class in any way.

We are going to use the next example to explain class inheritance.

import datetime  
  
class Employee():  
 employee\_no = 0  
 raise\_amount = 1.04  
  
 def \_\_init\_\_(self, first, last, pay):  
 self.first = first  
 self.last = last  
 self.email = self.first + self.last + '@company.com'  
 self.pay = pay  
 Employee.employee\_no += 1  
  
 def employee\_info(self):  
 print(self.first.title() + ' ' + self.last.title())  
  
 def apply\_raise(self):  
 self.pay \*= self.raise\_amount  
  
 @classmethod  
 def from\_string(cls, string):  
 first, last, pay = string.split('-')  
 return cls(first, last, pay)  
  
 @classmethod  
 def change\_raise\_amount(cls, amount):  
 cls.raise\_amount = amount  
  
 @staticmethod  
 def is\_workday(day):  
 if datetime.datetime.weekday(day) == 5 or datetime.datetime.weekday(day) == 6:  
 return False  
 return True  
  
class Developer(Employee):  
 pass  
  
dev\_1 = Developer('Daniel','Ghirasim', 50000)  
dev\_2 = Developer('Corey','Schafer', 90000)

1. We create a new class called Developer that inherits from our Employee class. In layman terms this means that every attribute or method we have in Employee class we are going to have in our Developer class.

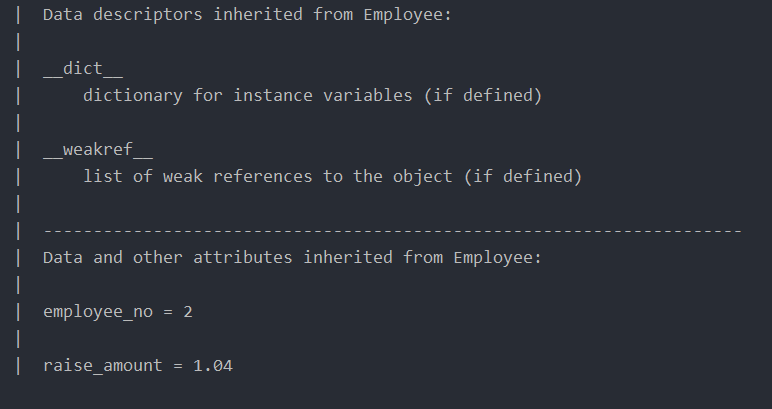
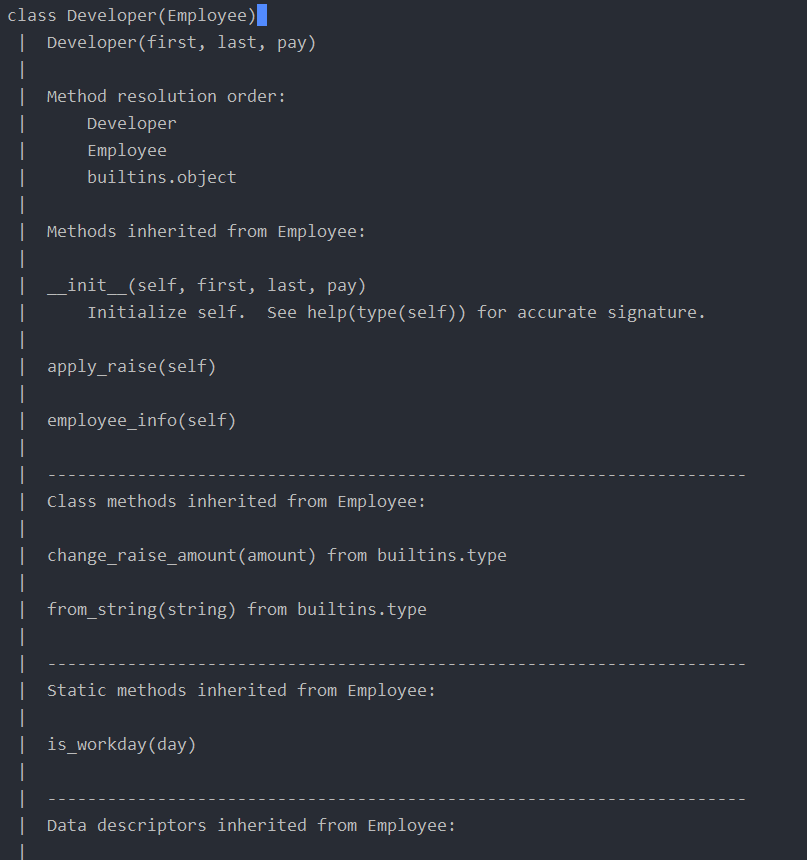
class Developer(Employee):  
 pass

dev\_1 = Developer('Daniel','Ghirasim', 50000)  
dev\_2 = Developer('Corey','Schafer', 90000)

Even tough we entered pass to our Developer class we can create instances of our Developer class because. What happened here, when we instanciated our developers it first looked in our developer class for the \_\_init\_\_ method , but it’s not going to find that there because it’s currently empty . What python is going to do next is walk up this chain of inheritance until it finds what it’s looking for, this chain is called **Method Resolution Order** (it’s going to look for our class attributes up the inheritance chain until it finds them in other words Method Resolution Order is the place where python will look for our attributes or methods)

I really useful function to visualize the Method Resolution order is the help() function.

If we call help(Developer) the following appears:



Here we can see our Method Resultion Order !

Changing the raise\_amount for our developer class.

After we made two new instances of our Developer class and call the apply\_raise() function, this will apply the default raise we set in our Employee Class (1.04). Let’s say we want our developers to have a higher raise than normal employees, to do this we just have to add a new raise\_amount variable into our Developer class.

class Developer(Employee):  
 raise\_amount = 1.10

Now when we call :

dev\_1.apply\_raise()

This will raise the pay by 10% instead of 4% like for our normal employees

Developer Class \_\_init\_\_ methods.

So let’s say we want to add a new attribute to our Developer class called ‘prog\_lang’ and keep the other attributes from our employee class.

In order to do this we have to do the following.

class Developer(Employee):  
 raise\_amount = 1.10  
  
 def \_\_init\_\_(self, first, last, pay, prog\_lang):  
 super().\_\_init\_\_(first, last, pay)  
 self.prog\_lang = prog\_lang

print(dev\_1.prog\_lang)

>>> Python

We initialize first, last , pay and prog\_lang and then with super().\_\_init\_\_(first, last, pay) we basically tell python take that code from our Employee class and after that we make a new attribute for our class called self.prog\_lang that is going to be the parameter that we pass in.

More examples.

Here we are going to create a manager class that inherits from our developer class.

class Manager(Employee):  
  
 def \_\_init\_\_(self, first, last, pay, employees=None):  
 super().\_\_init\_\_(first, last, pay)  
 if employees is None:  
 self.employees = []  
 else:  
 self.employees = employees

We use Employees = None to avoid the Default mutable trap argument (we never want to pass a list or a dictionary as default arguments). If Employees is None we are going to make an empty list else we are going to use the employees passed into parameter.

def add\_employee(self, emp):  
 if emp not in self.employees:  
 self.employees.append(emp)  
 print(emp.fullname(), 'added to list.')  
  
def remove\_employee(self, emp):  
 if emp in self.employees:  
 self.employees.remove(emp)  
 print(f'{emp.fullname()} removed from list.')  
  
def print\_employees(self):  
 return [print('-->', x.fullname()) for x in self.employees]

manager = Manager('Big and Bad', 'Manager', 10000)  
manager.add\_employee(dev\_1)  
manager.remove\_employee(dev\_2)  
manager.print\_employees()

>>> Daniel Ghirasim added to list.

>>> --> Daniel Ghirasim

## Isinstance() and issubclass()

Isinstance() and is subclass() are two very useful functions in python. (This further demonstrates that everything in python is an object).

print(isinstance(dev\_1, Employee))

>>> True

With this line of code we check if dev\_1 is instance of Employee which is true.

print(isinstance(dev\_1, Manager))

>>> False

With issubclass() we have to make sure that our first parameter is a class

print(issubclass(dev\_1, Manager))

TypeError: issubclass() arg 1 must be a class

print(issubclass(Developer, Employee))

>>> True

print(issubclass(Manager, Employee))

>>> True

# Special Methods

Special methods are also called magic methods or dunder methods.

Special methods are always surrounded by dubble underscores.

Ex:

\_\_init\_\_

\_\_str\_\_

\_\_repr\_\_

Two common dunder methods that we should always implement are the following : \_\_repr\_\_ , \_\_str\_\_

## \_\_repr\_\_

So let’s say you want to print out an instance of a class:

class Employee():  
 employee\_no = 0  
 raise\_amount = 1.04  
  
 def \_\_init\_\_(self, first, last, pay):  
 self.first = first  
 self.last = last  
 self.email = self.first + self.last + '@company.com'  
 self.pay = round(pay, 2)  
 Employee.employee\_no += 1

emp\_1 = Employee('Daniel','Ghirasim', 50000)

print(emp\_1)

You will get something like this:

>>> <\_\_main\_\_.Employee object at 0x000001B2B2F81788>

In order to print out proper information about our class we can make a \_\_repr\_\_ method

def \_\_repr\_\_(self):  
 return f"Employee('{self.first}', '{self.last}', '{self.pay}')"

So now when we call print(emp\_1) we get the following:

>>>Employee('Daniel','Ghirasim','50000')

Repr is meant to be an unambiguous representation of the object and should be used for debugging, logging and stuff like that (It’s kinda meant for other developers). And \_\_str\_\_ is meant to be more of a readable representation of the object and is meant to be used by the end-user.

If we have atleast an \_\_repr\_\_ method when we call print on the instance it will default to repr as fallback. A good rule of thumb is when you have an \_\_repr\_\_ method is to display something that you can copy and paste back to python code so you can recreate that same object . Like we did above.

## \_\_str\_\_

This is a bit more arbitrary (however you want it to be).

def \_\_str\_\_(self):  
 return f'{self.fullname()} -- {self.email}'

returns:

>>>Daniel Ghirasim -- [danielghirasim@company.com](mailto:danielghirasim@company.com)

print(emp\_1.\_\_str\_\_())  
print(emp\_1.\_\_repr\_\_())

returns:

Daniel Ghirasim -- danielghirasim@company.com

Employee('Daniel', 'Ghirasim', '50000')

## Example of \_\_add\_\_

For context, when we type:

print(1+2)  
print("a"+"b")

This calls the \_\_add\_\_ dunder method of the int() class and the str() class and prints out our result.

An example where we can use to demonstrate this is the following (don’t use this in real life apps, this is just an example).Let’s say we want to add instances of employees together and the result we want to get is the sum of salaries then this is how we would go about doing it:

dev\_2 = Developer('Corey','Schafer', 90000, 'Python, Java')  
emp\_1 = Employee('Daniel','Ghirasim', 50000)

print(emp\_1 + dev\_2)

result:

>>>100000

## Other examples:

When working with classes you can add a lot of special methods in order to do magical stuff with your classes add , substract , len , and etc.

### \_\_len\_\_

For example even len() is a special method.

Let’s say you want to call len() on your class (on the instance of the class to be more specific) then you would go about it this way.

def \_\_len\_\_(self):  
 return len(self.fullname())

Now when we call len() our instance it will return the length of what fullname() returns >> self.first + self.last

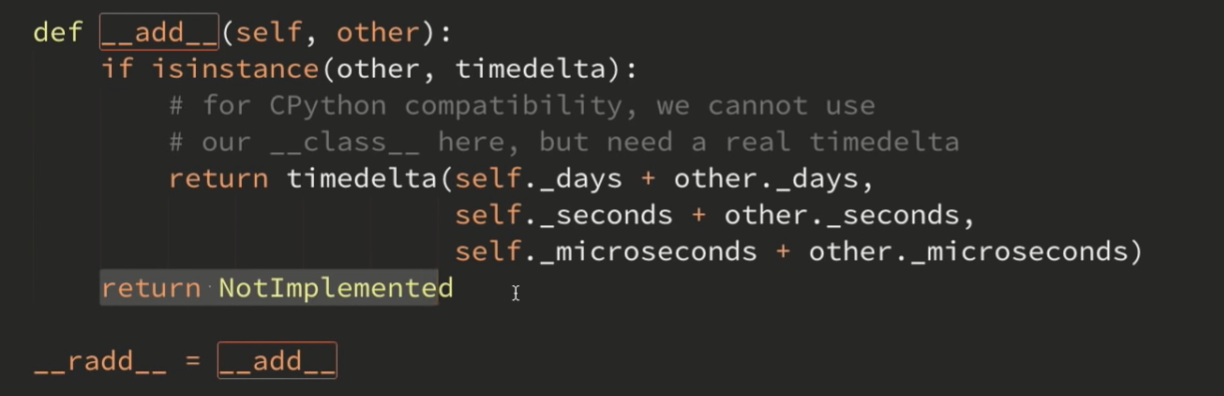
print(len(emp\_1)) # Daniel Ghirasim

returns:

>>>15

Example from datetime module:

### \_\_add\_\_ example from datetime module

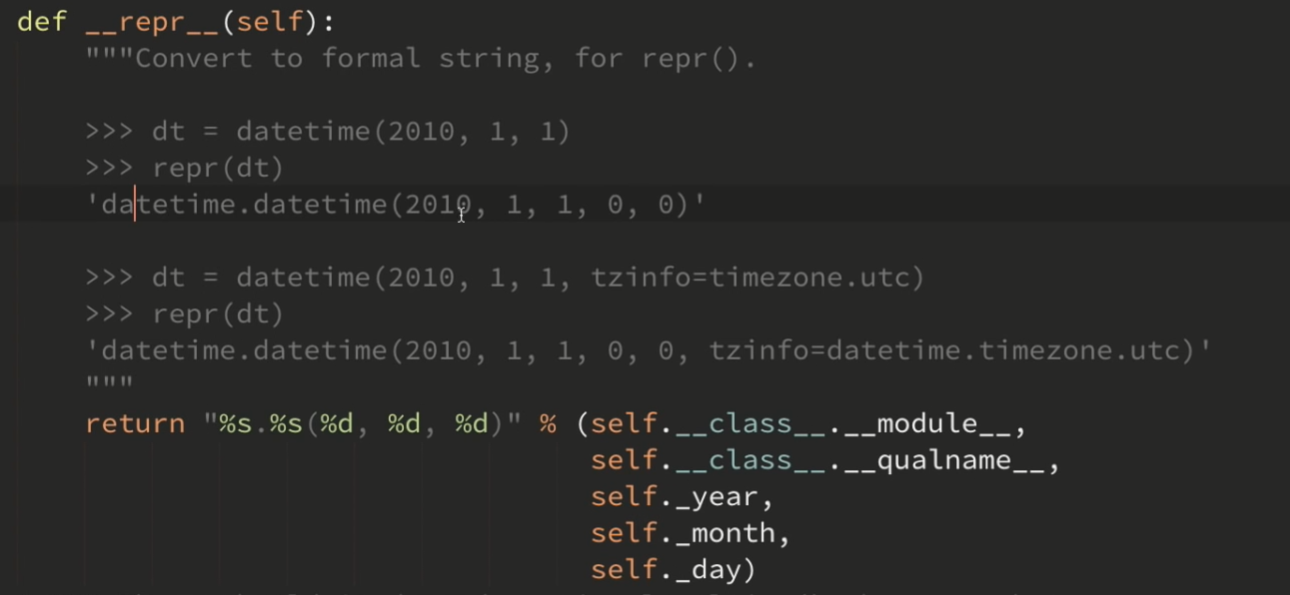


Here we can see when we call add between instances of datetime it returns the timedelta sum of each day, second and microsecond.

### return NotImplemented

We use this when we don’t want to throw an error , basically what this means if our custom \_\_add\_\_ function doesn’t work it will default back to the original \_\_add\_\_ function and if that doesn’t know how to handle it, then eventually it will throw an error

### \_\_repr\_\_ example from datetime module



# Property Decorators – Getters, Setters and Deleters

## @property decorator – Getter in python

We are going to start out with an example of our slimmed down Employee class.

class Employee:  
  
 def \_\_init\_\_(self, first, last, pay):  
 self.first = first  
 self.last = last  
 self.email = self.first.lower() + self.last.lower() + '@company.com'  
 self.pay = pay  
  
 def full\_name(self):  
 return self.first.title() + ' ' + self.last.title()  
  
emp\_1 = Employee('John', 'Smith', 60000)  
print(emp\_1.first)  
print(emp\_1.last)  
print(emp\_1.full\_name())  
print(emp\_1.email)

returns:

John

Smith

John Smith

johnsmith@company.com

Now let’s say we want to modify our emp\_1 first name to Jim

emp\_1 = Employee('John', 'Smith', 60000)  
emp\_1.first = 'Jim'  
print(emp\_1.first)  
print(emp\_1.last)  
print(emp\_1.full\_name())  
print(emp\_1.email)

returns:

Jim

Smith

Jim Smith

johnsmith@company.com

Notice how this didn’t change our email, this is because everytime we run our fullname() method it automatically grabs the latest version of self.first which in our case is ‘Jim’. We could easily make another method for returning the email address but let’s say we have thousands of employees in our class then this will break the code for anyone currently using this class.

Here come the benefits of getters and setters. In python we use the property decorator, this allows us to define a method but we can access it as an attribute.

Below we are going to show the example that will break the code for anyone who was previously using our class.

class Employee:  
  
 def \_\_init\_\_(self, first, last, pay):  
 self.first = first  
 self.last = last  
 self.pay = pay  
 **#Bad example do not use future Daniel**  
 def email(self):  
 return f'{self.first.lower()}{self.last.lower()}@company.com'

def full\_name(self):  
 return self.first.title() + ' ' + self.last.title()  
  
emp\_1 = Employee('John', 'Smith', 60000)  
emp\_1.first = 'Jim'  
# print(emp\_1.first)  
# print(emp\_1.last)  
# print(emp\_1.full\_name())  
print(emp\_1.email())

returns:

JimSmith@company.com

Now the fix is very easy. We just have to add @property decorator before the email:

@property  
def email(self):  
 return f'{self.first.lower()}{self.last.lower()}@company.com'

And now instead of calling email as a emp1.email() we can just type:

print(emp\_1.email)

returns:

jimsmith@company.com

And it will give us our properly formatted email without breaking any code.

## Setters in python

An example within our class.

Let’s say we want to set our name to the following:

emp\_1.full\_name = 'Daniel Ghirasim'

AttributeError: can't set attribute

We can’t do this because we will get an attribute error.

And from this fullname we want to update our self.first, self.last and our email. Right now we can’t do this. In order to do what we want we are going to need a setter.

The name we are going to use for our setter is going to be the name of our property.setter (@fullname.setter)

@full\_name.setter  
def full\_name(self, name):  
 first, last = name.split(' ')  
 self.first = first  
 self.last = last

Now we can use:

emp\_1.full\_name = 'Daniel Ghirasim'

And everything is updated and works like before but with our new set name.

emp\_1.full\_name = 'Daniel Ghirasim'  
print(emp\_1.first)  
print(emp\_1.last)  
print(emp\_1.email)

returns:

Daniel

Ghirasim

danielghirasim@company.com

## Deleters in python

Deleters work the same way setters do but they delete obviously. Example below

@full\_name.deleter  
def full\_name(self):  
 print('Deleted.')  
 self.first = None  
 self.last = None

emp\_1.full\_name = 'Daniel Ghirasim'  
del(emp\_1.full\_name)  
print(emp\_1.first)  
print(emp\_1.last)  
print(emp\_1.email)

returns:

Deleted.

None

None