Contents

[Classes and Instances 2](#_Toc13583197)

[Class Variables 3](#_Toc13583198)

[@classmethod & @staticmethod 4](#_Toc13583199)

[Creating subclasses 5](#_Toc13583200)

[e.g.01 5](#_Toc13583201)

[e.g.02 6](#_Toc13583202)

[Special (Magic-Dunder) Methods 7](#_Toc13583203)

[Property Decorators - Getters, Setters, and Deleters 9](#_Toc13583204)

[\*args, \*\*kwargs 10](#_Toc13583205)

# Classes and Instances

[don’t watch the video if you understand this code]

# class is the blueprint of instances

class Employee:

def \_\_init\_\_(self, first, last, pay):

# these are called instance variables as well

self.first = first

self.last = last

self.email = first + '.' + last + '@email.com'

self.pay = pay

def fullname(self):

return '{} {}'.format(self.first, self.last)

emp\_1 = Employee('Corey', 'Schafer', 50000)

emp\_2 = Employee('Test', 'Employee', 60000)

# These two lines of code are identical

emp\_1.fullname()

print(Employee.fullname(emp\_1))

**# what if we accidentally left self as arg to fullname() method here**

class Employee:

    def \_\_init\_\_(self, first, last, pay):

        self.first = first

        ...

    def fullname(): # TypeError:fullname() takes 0 positional arguments but 1 was given

        return '{} {}'.format(self.first, self.last)

# and also using self Above allow any subclass to override it!

emp\_1 = Employee('Corey', 'Schafer', 50000)

emp\_2 = Employee('Test', 'Lasty', 60000)

print(emp\_1.fullname())

# TypeError: fullname() takes 0 positional arguments but 1 was given

# It doesn't looks like 1 positional argument is passed here,

# the thing is that 'emp\_1' will automatically pass in to 'fullname()' method

# Class Variables

Class variables are going to be the same for all isntances of our employees.

In order to access class variables, you can use either (ClassName.class\_var) or (self.class\_var).

In this example raise\_amount and num\_of\_instances are suitable candidates for Employee class since they are common among all employees.

class Employee:

num\_of\_instances = 0

class variables are common among all instances of the class

raise\_amount = 1.04

def \_\_init\_\_(self, first, last, pay):

self.first = first

Instance variables are unique to each instance of the class.

self.last = last

self.email = first + '.' + last + '@gmail.com'

self.pay = pay

Employee.num\_of\_instances += 1 # self.num\_of\_instances wont make sense here

def fullname(self):

return '{} {}'.format(self.first, self.last)

def apply\_raise(self):

        # 'self.raise\_amount' in contrast to 'Employee.raise\_amount'

        # will allow us to bring changes to each single instances!

        # Note: 'Employee.raise\_amount' won't make sense here, but it makes sense for

        #       'num\_of\_instances' class variable.

self.pay = int(self.pay \* self.raise\_amount)

# and also using self Above allows any subclass to override it!

emp\_1 = Employee('Corey', 'Schafer', 50000)

emp\_2 = Employee('Test', 'Lasty', 60000)

emp\_2 = Employee('Lucy', 'Ignard', 70000)

Employee.raise\_amount = 1.09

emp\_1.raise\_amount = 1.08

print(emp\_1.raise\_amount)  # 1.08 # only this one is affected

print(emp\_2.raise\_amount)  # 1.09

print(emp\_3.raise\_amount)  # 1.09

print(Employee.num\_of\_instances)  # 3

print(emp\_1.\_\_dict\_\_)

# {'first': 'Corey', 'last': 'Schafer', 'email': 'Corey.Schafer@gmail.com',

# 'pay': 50000, 'raise\_amount': 1.08}

# @classmethod & @staticmethod

**@classmethod** - with class methods we work on classes, manuplate variables of the class and do more. Class methods take **'cls'** as an argument, **'cls'** is a convention here.

**@staticmethod** - with static methods we work neither on class nor on its instances because they don’t depend on either of those.

import datetime

class Employee:

    num\_of\_instances = 0

    raise\_amount = 1.04

    def \_\_init\_\_(self, first, last, pay):

        self.first = first

        self.last = last

        self.email = first + '.' + last + '@gmail.com'

        self.pay = pay

        Employee.num\_of\_instances += 1

    def fullname(self):

        return '{} {}'.format(self.first, self.last)

    def apply\_raise(self):

        self.pay = int(self.pay \* self.raise\_amount)

    # class methods takes 'cls' as an argument, 'cls' is a convention here

    # regular methods automatically pass 'self' as an argument

    # with class methods we work on class rather than instances

    # with regular method we operate or work on instances rather than class

    @classmethod

    def set\_raise\_amount(cls, amount):

        cls.raise\_amount = amount

    # This class method is used as an alternative class constructor!

    @classmethod

    def from\_emp\_constructor(cls, emp\_str):

        first, last, pay = emp\_str.split("-")

        return cls(first, last, pay)

    # static methods pass neither 'self' nor 'cls' as argument because

# they don’t depend on class or instances.

    # with static methods we work neither on class nor on its instances

    # we include them in a class for they have a logical connection with class

    @staticmethod

    def is\_workday(date):

        if date.weekday() == 5 or date.weekday == 6:

            return False

        return True

emp\_1 = Employee('Basir', 'Payenda', 50000)

print(emp\_1.raise\_amount)  # 1.04

# now let's use classmethod and set the raise amount to 10 percent (1.1)

Employee.set\_raise\_amount(1.1)  # 10

print(emp\_1.raise\_amount)

# using alternative class constructor made with @classmethod

emp\_2 = Employee.from\_emp\_constructor('Noor-Payenda-200000')

print(emp\_2.first)  # Noor

# static method

d = datetime.date(2016, 12, 15)

print(Employee.is\_workday(d))  # True

# Creating subclasses

With inheritance you can add all functionalities of parent class to the child class and you can even overwrite certain methods and attributes of it.

## e.g.01

import datetime

class Employee:

num\_of\_instances = 0

raise\_amount = 1.04

def \_\_init\_\_(self, first, last, pay):

self.first = first

self.last = last

self.email = first + '.' + last + '@gmail.com'

self.pay = pay

Employee.num\_of\_instances += 1

def fullname(self):

return '{} {}'.format(self.first, self.last)

def apply\_raise(self):

self.pay = int(self.pay \* self.raise\_amount)

class Developer(Employee):

raise\_amount = 1.08

def \_\_init\_\_(self, first, last, pay, programming\_language):

super().\_\_init\_\_(first, last, pay)

# Employee.\_\_init\_\_(first, last, pay) # or inherit like this

self.programming\_language = programming\_language

dev\_1 = Developer('Basir', 'Payenda', 100000, 'Python')

print(dev\_1.fullname()) # Basir Payenda

print(dev\_1.programming\_language) # Python

## e.g.02

class Manager(Employee):

def \_\_init\_\_(self, first, last, pay, employees=None):

super().\_\_init\_\_(first, last, pay)

if employees is None:

employees = []

else:

self.employees = employees

def add\_employee(self, emp):

if emp not in self.employees:

self.employees.append(emp)

def remove\_employee(self, emp):

if emp in self.employees:

self.employees.remove(emp)

def list\_employee(self):

for emp in self.employees:

print('-->', emp)

man\_1 = Manager('Jawid', 'Rahimi', 90000, ['Mustafia', 'Muskan', 'Murwarid', 'Mariam'])

man\_1.remove\_employee('Murwarid')

man\_1.add\_employee('Muzhgan')

print(man\_1.list\_employee())

# --> Mustafia

# --> Muskan

# --> Mariam

# --> Muzhgan

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

mgr\_1 = Manager('Jawid', 'Rahimi', 90000, [

'Mustafia', 'Muskan', 'Murwarid', 'Mariam'])

print(isinstance(mgr\_1, Employee)) # True

print(isinstance(mgr\_1, Manager)) # True

print(issubclass(Manager, Employee)) # True

print(issubclass(mgr\_1, Employee)) # this throws an error

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

class Person:

    def \_\_init\_\_(self, name, age):

        self.fullname = name

        self.age = age

    def introduce(self):

        print(f'My name is {self.fullname}')

class English(Person):

    def \_\_init\_\_(self, name, age, position):

        super(English, self).\_\_init\_\_(name, age)

        self.position = position

    def introduce(self):

        super(English, self).introduce()

        print(f'... I am {self.age} years old and I am {self.position}.')

p1 = English('John Doe', 23, 'CEO')

p1.introduce()

# My name is John Doe

# ... I am 23 years old and I am CEO.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Special (Magic-Dunder) Methods

class Employee:

num\_of\_instances = 0

raise\_amount = 1.04

def \_\_init\_\_(self, first, last, pay):

self.first = first

self.last = last

self.email = first + '.' + last + '@gmail.com'

self.pay = pay

Employee.num\_of\_instances += 1

def fullname(self):

return '{} {}'.format(self.first, self.last)

def apply\_raise(self):

self.pay = int(self.pay \* self.raise\_amount)

# \_\_repr\_\_ is for developers, \_\_str\_\_ is for customers.

def \_\_repr\_\_(self):

return 'Employee("{}", "{}", {})'.format(self.first, self.last,

self.pay)

def \_\_str\_\_(self):

return "{} - {}".format(self.fullname(), self.email)

emp\_1 = Employee('Basir', 'Payenda', 900000)

print(emp\_1) # Basir Payenda - Basir.Payenda@gmail.com

print(emp\_1.\_\_repr\_\_()) # Employee("Basir", "Payenda", 900000)

print(emp\_1.\_\_str\_\_()) # Basir Payenda - Basir.Payenda@gmail.com

print(repr(emp\_1)) # Employee("Basir", "Payenda", 900000)

print(str(emp\_1)) # Basir Payenda - Basir.Payenda@gmail.com

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

print(1+2)

print('a'+'b')

# Above example in background using dunder add method

print(int.\_\_add\_\_(1, 2)) # 3

print(str.\_\_add\_\_('a', 'b')) # ab

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

class Employee:

num\_of\_instances = 0

raise\_amount = 1.04

def \_\_init\_\_(self, first, last, pay):

self.first = first

self.last = last

self.email = first + '.' + last + '@gmail.com'

self.pay = pay

Employee.num\_of\_instances += 1

def fullname(self):

return self.first + ' ' + self.last

def \_\_repr\_\_(self):

return 'Employee("{}", "{}", {})'.format(self.first, self.last, self.pay)

def \_\_str\_\_(self):

return "{} - {}".format(self.fullname(), self.email)

# adding functionality to show combined salary of two employees

# eg. print(emp\_1 + emp\_2) # 1.900.000

def \_\_add\_\_(self, next\_arg):

return self.pay + next\_arg.pay

def \_\_len\_\_(self):

return len(self.fullname())

emp\_1 = Employee('Basir', 'Payenda', 900000)

emp\_2 = Employee('Bashir', 'Payenda', 1000000)

print(len(emp\_1)) # 15

# Property Decorators - Getters, Setters, and Deleters

class Employee:

def \_\_init\_\_(self, first, last):

self.first = first

self.last = last

@property

def email(self):

return '{}.{}@email.com'.format(self.first, self.last)

@property

def fullname(self):

return '{} {}'.format(self.first, self.last)

@fullname.setter

def fullname(self, name):

first, last = name.split(' ')

self.first = first

self.last = last

@fullname.deleter

def fullname(self):

print('Delete Name!')

self.first = None

self.last = None

emp\_1 = Employee('John', 'Smith')

emp\_1.fullname = "Corey Schafer" # not emp\_1.fullname("Corey Schafer")

print(emp\_1.first)

print(emp\_1.email)

print(emp\_1.fullname)

del emp\_1.fullname # Delete Name!

# \*args, \*\*kwargs (Python Crash Course Book )

class Restaurant:

resto\_nums = 0 # Class Variable

def \_\_init\_\_(self, name, location, \*foods, \*\*pizza\_toppings):

# Instance Variables

self.name = name

self.location = location

self.foods = list(foods)

self.pizza\_toppings = dict(pizza\_toppings)

Restaurant.resto\_nums += 1

def open\_restaurant(self):

print(f'{self.name} is open.')

def get\_location(self):

print(f'The location of "{self.name}" is {self.location}.')

def get\_foods(self):

for food in self.foods:

print(food)

def get\_pizza\_toppings(self):

res = f'The ordered toppings for your pizza is '

#Iterate over the dictionary via enumerate

for index, (key, value) in enumerate(self.pizza\_toppings.items()):

# For the last element, change the string slightly

if index == len(self.pizza\_toppings)-1:

res += f'and {self.pizza\_toppings[key]} of {key}. '

# Get rid of extra comma at second last item of dictionary

elif index == (len(self.pizza\_toppings)-2):

res += f" {self.pizza\_toppings[key]} of {key}"

# Else append string the normal way

else:

res += f'{self.pizza\_toppings[key]} of {key}, '

print(res)

karoki = Restaurant('Karoki Restaurant', 'NYC 2rd rue', 'Pizza', 'Potato Chips', 'Chicken Fried Steak', sauce= '2 ounce', fromage='4 ounce', chips='3 ounce')

karoki.get\_pizza\_toppings()

# The ordered toppings for your pizza is 2 ounce of sauce, 4 ounce of fromageand 3 ounce of chips.

# If \_\_name\_\_ == ‘\_\_main\_\_’:

if \_\_name\_\_ == '\_\_main\_\_':

print('Run Directly')

else:

print('Run from import')

# When python runs a module it sets some special variables and one of those

# variables is \_\_name\_\_.

# when you run a python file directly it sets **\_\_name\_\_** variable to **\_\_main\_\_.** for

# instance:

print(f'This Module\'s Name is: {\_\_name\_\_}')

# This Module's Name is: \_\_main\_\_

# Assuming that the name of current python file is 'name\_and\_main\_vars.py' & if we

# import the very same module in another file. the 'print' code from above will be

# executed and this is the result:

# This Module's Name is: name\_and\_main\_vars