Python Classes

Lesson 1 : Classes and instances

→ Why classes ?

↳ They allow us to logically group data & functions in a way that's easy to reuse and also build upon if needed.

↳ Data and function are called attributes & methods when talking about classes.

↳ Methods : a function that is associated with a class.

For e.g. while representing employees in a company, each employee is going to have specific attributes & methods such as name, email id, pay and actions they can perform. It would be nice if we have a class which can be used as a blueprint to create each employee.

↳ We make a class Employee.

```
# Python Object-Oriented Programming


class Employee:
    pass
```

← we'll get an error if create an empty class hence we fill it with pass

→ A class is just a blueprint for creating instances. Each unique employee that we create using Employee will be an instance of that class. for e.g.

```
class Employee:
    pass

emp_1 = Employee()
emp_2 = Employee()

print(emp_1)
print(emp_2)
```

} own unique instances of employee class.

Both Employee object but they are unique as they have different location in memory.

```
<__main__.Employee object at 0x1013775f8>
<__main__.Employee object at 0x101377668>
[Finished in 0.0s]
```

↳ Instance variables contain data that is unique to each instance.

```
emp_1 = Employee()
emp_2 = Employee()

print(emp_1)
print(emp_2)

emp_1.first = 'Corey'
emp_1.last = 'Schafer'
emp_1.email = 'Corey.Schafer@company.com'
emp_1.pay = 50000

emp_2.first = 'Test'
emp_2.last = 'User'
emp_2.email = 'Test.User@company.com'
emp_2.pay = 60000

print(emp_1.email)
print(emp_2.email)


<__main__.Employee object at 0x101a77a20>
<__main__.Employee object at 0x101a77a90>
Corey.Schafer@company.com
Test.User@company.com
[Finished in 0.0s]
```

manually creating instance variables for each employee

What if we wanted to set all of this info when they are created rather doing this manually like here.??

equivalent to

* when we create method within a class they receive the instance as the first argument automatically And by convention we call the instance self. We can call it whatever we want.

```
class Employee:

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first + '.' + last + '@company.com'


emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'User', 60000)

# print(emp_1)
# print(emp_2)

print(emp_1.email)
print(emp_2.email)


Corey.Schafer@company.com
Test.User@company.com
[Finished in 0.0s]
```

Like a constructor

* inside our __init__ method we're going to set all of these instance variables.

doesn't need to be same as arguments. e.g. it could also be self.fname = first

* After self we can specify what other arguments we want to accept.

* while creating employee, the instance is passed automatically, we only need to provide arguments.

* The __init__ method will run automatically and emp_1 will be passed in as self. And then it will set all of the attributes. such as emp1.first = 'corey' etc.

Now, let say we wanted some kind of action, say the ability to display the fullname of the employee.

```
print('{} {}'.format(emp_1.first, emp_1.last))
Corey.Schafer@company.com
Test.User@company.com
Corey Schafer
[Finished in 0.0s]
```

* this could be done like this, but this is not the efficient way. Instead we'll create a method within our class which allows us to put this functionality.

creating a method called fullname within our class

```
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first + '.' + last + '@company.com'

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


emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'User', 60000)

# print(emp_1)
# print(emp_2)

print(emp_1.email)
print(emp_2.email)

print(emp_1.fullname())
```

each method in a class will automatically takes the instance as the first argument.

be carefull here. Instead of emp1.first or emp1.last we have to use self.first and self.last so that the method work with all instances.

we need a () here since it's a method not an attribute.

```
Corey.Schafer@company.com
Test.User@company.com
Corey Schafer
[Finished in 0.0s]
```

```
emp_1.fullname()
print(Employee.fullname(emp_1))
# print(emp_2.fullname())
```

we can also call the method on a class but we have to manually pass the instance as argument. So unlike the previous line emp1.fullname() where we don't need to pass self as it happens automatically.

2] Lesson 2 : Class Variables

→ Class variables are variables that are shared by all instances of the class.

→ While instance variables can be unique for each instance like names, email, pay. Class variables should be the same for each instance. For example, The annual raises the company gives every year. so the amount may change for every year but whatever the amount is it's going to be same for all employees.

class variable

```
class Employee:

    raise_amount = 1.04

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first + '.' + last + '@company.com'

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

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)


emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'User', 60000)

print(emp_1.pay)
```

```
50000
52000
[Finished in 0.0s]
```

It's really important to keep in mind that the class variable must be accessed through either the class or the instance. i.e., either Employee.raise_amount or self.raise_amount. Accessing it without any i.e. raise_amount only in the method will cause "name error".

```
print(emp_1.__dict__)

# print(Employee.raise_amount)
# print(emp_1.raise_amount)
# print(emp_2.raise_amount)

{'first': 'Corey', 'pay': 50000, 'email': 'Corey.Schafer@company.com', 'last': 'Schafer'}
[Finished in 0.0s]
```

→ To view name space of emp_1

Observe that there's NO raise_amount here.

```
print(Employee.__dict__)

# print(Employee.raise_amount)
# print(emp_1.raise_amount)
# print(emp_2.raise_amount)

{'__init__': <function Employee.__init__ at 0x10197a6a8>, '__dict__': <attribute '__dict__' of 'Employee' objects>,
'apply_raise': <function Employee.apply_raise at 0x10197a7b8>, 'fullname': <function Employee.fullname at 0x10197a730>,
'raise_amount': 1.04, '__weakref__': <attribute '__weakref__' of 'Employee' objects>, '__doc__': None, '__module__':
'__main__'}
[Finished in 0.0s]
```

But here we can see that the class does contains raise_amount attribute

So, when we try to access the attribute from the instance, it will first check if the instance contains that attribute and if it doesn't then it will see if the class or any class that it's inherited from contains that attribute.

So, when we access raise_amount from our instance emp-1, it doesn't actually have raise_amount attribute rather it is accessing the class's raise_amount attribute.

→ Now let's try to change the raise_amount

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

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)


emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'User', 60000)

# print(Employee.__dict__)

Employee.raise_amount = 1.05

print(Employee.raise_amount)
print(emp_1.raise_amount)
print(emp_2.raise_amount)

1.05
1.05
1.05
[Finished in 0.0s]
```

You can see that, it changes the raise_amount for the class and all the instances.

What if we were to set the raise_amount using an instance instead of using the class ??

```
        self.pay = int(self.pay * self.raise_amount)


emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'User', 60000)

# print(Employee.__dict__)

emp_1.raise_amount = 1.05

print(Employee.raise_amount)
print(emp_1.raise_amount)
print(emp_2.raise_amount)

1.04
1.05
1.04
[Finished in 0.0s]
```

It only changed the raise_amount for emp-1. How come this is possible ??

It's possible because it created the attribute in the instance.

```
print(emp_1.__dict__)

{'raise_amount': 1.05, 'last': 'Schafer', 'first': 'Corey', 'email': 'Corey.Schafer@company.com', 'pay': 50000}
```

So `emp_1.raise_amount = 1.05` actually created the raise_amount attribute within emp_1 and then while `print(emp_1.__dict__)` or even while `print(emp_1.raise_amount)` it finds the attribute first within the instance and print it out rather going to the class in search of it. However this is not the case for emp_2 hence we get the value set for the class.

The Next question after this could be, if we have class variables then are there also class methods?

The answer is YES. There are two types of methods : Static method & Class methods.

Class methods v/s Static methods

→ As discussed previously, regular methods in a class automatically take the instance as the first argument and by convention we call it self.

→ we need to change it so that it takes the first argument as class now. To do that, we're gonna use class methods. To turn a regular method into a class method is as easy as adding a decorator to the top called @classmethod.

```
# Python OOP
class Employee:

    num_of_emps = 0
    raise_amt = 1.04

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.email = first + '.' + last + '@email.com'
        self.pay = pay

        Employee.num_of_emps += 1

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

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)

    @classmethod
    def set_raise_amt(cls, amount):
        pass
```

declaring a method as class method by adding a decorator

just like self is a convention for an instance variable name, the common convention for a class variable is cls bcz we cannot use "class" as variable name

```
    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)

    @classmethod
    def set_raise_amt(cls, amount):
        cls.raise_amt = amount

emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'Employee', 60000)

print(Employee.raise_amt)
print(emp_1.raise_amt)
print(emp_2.raise_amt)
```

→ we're setting the value of our class variable raise_amount equal to amount argument that we are accepting from this method

```
1.04
1.04
1.04
[Finished in 0.2s]
```

All are equal to 4% because we have our class variable raise_amount set to 4%

What if we wanted to change this to 5% ??

```
        self.pay = int(self.pay * self.raise_amt)

    @classmethod
    def set_raise_amt(cls, amount):
        cls.raise_amt = amount

emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'Employee', 60000)

Employee.set_raise_amt(1.05)

print(Employee.raise_amt)
print(emp_1.raise_amt)
print(emp_2.raise_amt)
```

By using the set_raise_amt method, which we created, we can change the raise_amount to 5%. Remember that, it automatically accepts the class i.e., cls so we need to pass directly the amount argument which is 1.05.

```
1.05
1.05
1.05
[Finished in 0.0s]
```

The reason all these are 5% is because we ran the set_raise_amt method, which is a class method i.e., we are working on class instead of instance and we're setting class variable raise_amount equal to the amount that we passed as our argument.

Let's consider an interesting case where we are getting the employee information in the form of a string separated by hyphens.

```
emp_str_1 = 'John-Doe-70000'
emp_str_2 = 'Steve-Smith-30000'
emp_str_3 = 'Jane-Doe-90000'

first, last, pay = emp_str_1.split('-')

new_emp_1 = Employee(first, last, pay)

# new_emp_1 = Employee.from_string(emp_str_1)

print(new_emp_1.email)
print(new_emp_1.pay)

John.Doe@email.com
70000
[Finished in 0.0s]
```

In such a case, we need to constantly parse the string before we create new employee.

Let's create an alternative constructor that allows us to pass in the string and we can create the employee.

usually by convention, an alternative constructor name start with from

we will be using cls instead of Employee

```
    def set_raise_amt(cls, amount):
        cls.raise_amt = amount

    @classmethod
    def from_string(cls, emp_str):
        first, last, pay = emp_str.split('-')
        return cls(first, last, pay)

emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'Employee', 60000)

emp_str_1 = 'John-Doe-70000'
emp_str_2 = 'Steve-Smith-30000'
emp_str_3 = 'Jane-Doe-90000'

new_emp_1 = Employee.from_string(emp_str_1)

print(new_emp_1.email)
print(new_emp_1.pay)

John.Doe@email.com
70000
[Finished in 0.0s]
```

So, instead of parsing the string manually, we've provided them from_string method. Then we just pass the strings as argument, after which it comes in alternative constructor, where the string is splitted on hyphen and then it creates a new employee object and returns it.

* Static methods : Regular methods automatically pass the instance self as the first argument and class methods automatically pass the class cls as the first argument. but static methods don't pass anything automatically. They simply behaves like regular function except we include them in our classes due to some logical connection with the class.

For example,
Let's say we wanted a simple function, that would return whether it's workday or not. So this has a logical connection to our Employee class but it doesn't actually depend on any specific instance or class variable. So we're gonna make it as static method. To create a static method we're also going to use a decorator @staticmethod.

```python
        return cls(first, last, pay)

    @staticmethod
    def is_workday(day):
        if day.weekday() == 5 or day.weekday() == 6:
            return False
        return True

emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'Employee', 60000)

import datetime
my_date = datetime.date(2016, 7, 10)

print(Employee.is_workday(my_date))
```

```
False
[Finished in 0.1s]
```

static methods don't take any instance or class as the first argument.

if day is saturday or sunday

Lesson 4: Inheritance - Creating Subclasses

↳ Inheritance allows us to inherit attributes and methods from a parent class.

↳ This is usefull because we can create subclasses and get all the functionality of our parent class and then we can overwrite or add completely new functionality without affecting the parent class in any way.

Let's consider that we want to create different types of employees such as developers and managers.

↳ These could be the best example of subclasses because both developers and managers are going to have names, email and a salary, and these are the things our Employee class already has.

```python
        self.email = first + '.' + last + '@email.com'
        self.pay = pay

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

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)


class Developer(Employee):
    pass


dev_1 = Developer('Corey', 'Schafer', 50000)
dev_2 = Developer('Test', 'Employee', 60000)

print(dev_1.email)
print(dev_2.email)
```

```
Corey.Schafer@email.com
Test.Employee@email.com
[Finished in 0.0s]
```

specifying what class that we want to inherit from

even by simply inheriting from the class, we inherited all of it's functionality.

instances of our Developer class.

we can access the attributes that were actually set in our parent Employee class.

↳ What happens here is that, when we instantiated our developers `dev_1 = Developer('Corey', 'Schafer', 50000)` it first look in our Developer class for `__init__` method and it doesn't find it within because it's empty. So python will walk up the chain of inheritance until it finds what it's looking for.. This chain is called Method Resolutioner.

→ By using `print(help(Developer))` we can easily undertand the above said concept.

```
Help on class Developer in module __main__:

class Developer(Employee)
 |  Method resolution order:
 |      Developer
 |      Employee
 |      builtins.object
 |
 |  Methods inherited from Employee:
 |
 |  __init__(self, first, last, pay)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  apply_raise(self)
 |
 |  fullname(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Employee:
```

→ So under Method resolution order, we can see the places Python searches for attributes and methods, and in what order i.e., when we created dev_1 & dev_2, it first looked at our Developer class for the `__init__()` method and when it didn't find it there then it went to the Employee class and it found it there and that's where it was executed. Now, if it hadn't found it there then the last place that it would've looked is the base object class.

→ Looking at the output further, it actually shows the methods and attributes that were inherited from Employee

- What if we wanted to customize our subclass a bit ??

↳ Let's say that we want our developers to have a raise amount of 10%.

```
class Developer(Employee):
    raise_amt = 1.10


dev_1 = Developer('Corey', 'Schafer', 50000)
dev_2 = Developer('Test', 'Employee', 60000)

# print(dev_1.email)
# print(dev_2.email)


print(dev_1.pay)
dev_1.apply_raise()
print(dev_1.pay)

50000
55000
[Finished in 0.0s]
```

By changing the raise_amount in our sub-class didn't affect on any of our Parent class i.e., Employee

```
class Developer(Employee):
    raise_amt = 1.10


dev_1 = Employee('Corey', 'Schafer', 50000)
dev_2 = Developer('Test', 'Employee', 60000)

# print(dev_1.email)
# print(dev_2.email)


print(dev_1.pay)
dev_1.apply_raise()
print(dev_1.pay)

50000
52000
[Finished in 0.0s]
```

- What if we want to initiate our subclasses with more information than our parent class can handle ??

↳ Let's say that we also wanted to pass the developer's main programming language as an attribute while we are creating them. But our Employee class only accepts first name, last name and pay.

↳ To do this, we are going to give our Developer class its own `__init__` method.

```python
class Developer(Employee):
    raise_amt = 1.10

    def __init__(self, first, last, pay, prog_lang):
        self.first = first
        self.last = last
        self.email = first + '.' + last + '@email.com'
        self.pay = pay
```

We can copy & paste the parent class's `__init__` method and then add `self.prog_lang`, but we don't need to do that, also it doesn't make sense to repeat same logic in multiple places.

So, instead of copying and pasting we will let our `__init__` method to handle first name, last name and pay and then get the new `prog_lang`. This can be done using `super().__init__()` and the passing the arguments of parent class's `__init__`

```python
def __init__(self, first, last, pay, prog_lang):
    super().__init__(first, last, pay)
```

the same can be achieved by →

```python
def __init__(self, first, last, pay, prog_lang):
    Employee.__init__(self, first, last, pay)
```

finally,

```python
    def __init__(self, first, last, pay, prog_lang):
        super().__init__(first, last, pay)
        self.prog_lang = prog_lang


dev_1 = Developer('Corey', 'Schafer', 50000, 'Python')
dev_2 = Developer('Test', 'Employee', 60000, 'Java')

print(dev_1.email)
print(dev_1.prog_lang)


# print(dev_1.pay)
# dev_1.apply_raise()
# print(dev_1.pay)
```

```
Corey.Schafer@email.com
Python
[Finished in 0.0s]
```

Let's try to create another class Manager with more complicated additions.

```python
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

    def add_emp(self, emp):
        if emp not in self.employees:
            self.employees.append(emp)

    def remove_emp(self, emp):
        if emp in self.employees:
            self.employees.remove(emp)

    def print_emps(self):
        for emp in self.employees:
            print('-->', emp.fullname())
```

↳ List of employees the manager supervises

} Add employees to the list of employees under supervision

} Remove employees from the list.

} Print out all of the employees, the manager supervises.

```python
dev_1 = Developer('Corey', 'Schafer', 50000, 'Python')
dev_2 = Developer('Test', 'Employee', 60000, 'Java')

mgr_1 = Manager('Sue', 'Smith', 90000, [dev_1])

print(mgr_1.email)

mgr_1.print_emps()

# print(dev_1.email)
# print(dev_1.prog_lang)


# print(dev_1.pay)
# dev_1.apply_raise()
```

```
Sue.Smith@email.com
--> Corey Schafer
[Finished in 0.0s]
```

```python
dev_1 = Developer('Corey', 'Schafer', 50000, 'Python')
dev_2 = Developer('Test', 'Employee', 60000, 'Java')

mgr_1 = Manager('Sue', 'Smith', 90000, [dev_1])

print(mgr_1.email)

mgr_1.add_emp(dev_2)

mgr_1.print_emps()

# print(dev_1.email)
# print(dev_1.prog_lang)


# print(dev_1.pay)
```

```
Sue.Smith@email.com
--> Corey Schafer
--> Test Employee
[Finished in 0.0s]
```

```python
dev_1 = Developer('Corey', 'Schafer', 50000, 'Python')
dev_2 = Developer('Test', 'Employee', 60000, 'Java')

mgr_1 = Manager('Sue', 'Smith', 90000, [dev_1])

print(mgr_1.email)

mgr_1.add_emp(dev_2)
mgr_1.remove_emp(dev_1)

mgr_1.print_emps()

# print(dev_1.email)
# print(dev_1.prog_lang)
```

```
Sue.Smith@email.com
--> Test Employee
[Finished in 0.0s]
```

There are two built-in python functions: `isinstance()` and `issubclass()` which we can use to check. wether an object is an instance or subclass of other.