## Python OOP Tutorial 1: Classes and Instances

Methods: A function associated with a class?

Create a class

```
class Employee:

    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)
```

Runs everytime we create a new employee

All attributes of the Class

Self:

Methods

Instance 1

Instance 2

```
emp_1 = Employee('Devin', 'Powers', 50000)
emp_2 = Employee('Lebron', 'James', 2000000)
```

own unique instance of the Class

When we run this line, the __init__ method will run automatically and emp_# will be passed in as self and then will set all the attribut

Difference Between

Do the exact same thing

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

→ Devin Powers

→ Devin Powers

emp_1 instance

fullname() call the method

Run it as a Class

how to pass the instance as an argument?

If you don't include the "()" if will print the method and not the return value:

```
print(emp_1.fullname)
```

prints the method

```
<bound method Employee.fullname of <__main__.Employee object at 0x113c99470>>
```

Vs.

```
print(emp_1.fullname())
```

printed the name

~~Kebron James~~

Devin Powers

## Class Variables (Video 2)

```
class Employee:

    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 * 1.04)


emp_1 = Employee('Devin', 'Powers', 50000)

emp_2 = Employee('Lebron', 'James', 2000000)

print(emp_1.pay)
emp_1.apply_raise()
print(emp_1.pay)
```

→ 50000

→ 52000

applying apply-raise method!

Now were going to pull that 4% out as a 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('Devin', 'Powers', 50000)

emp_2 = Employee('Lebron', 'James', 2000000)

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

→ 1.04

→ 1.04

→ 1.04

accessing the class variable through the class itself.

instances don't have the raise_amount attribute, their accessing the parent class attribute (Employee in the example)

Employee 1 namespace

```
print(emp_1.__dict__)
```

```
{'first': 'Devin', 'last': 'Powers', 'pay': 50000, 'email': 'Devin.Powers@company.com'}
```

~ no raise_amount attribute

```
print(Employee.__dict__)
```

~ Parent class Employee has the attribute?

```
{'__module__': '__main__', 'raise_amount': 1.04, '__init__': <function Employee.__init__ at 0x11ac4bae8>, 'fullname': <function
Employee.fullname at 0x11ac4b488>, 'apply_raise': <function Employee.apply_raise at 0x11ac0cb70>, '__dict__': <attribute '__dict__' of
'Employee' objects>, '__weakref__': <attribute '__weakref__' of 'Employee' objects>, '__doc__': None}
```

Using Class to change the attribute:

```
Employee.raise_amount = 1.05

print(Employee.raise_amount) → 1.05
print(emp_1.raise_amount)    → 1.05
print(emp_2.raise_amount)    → 1.05
```

Using the instance to change the attribute.

```
emp_1.raise_amount = 1.05

print(Employee.raise_amount) → 1.04
print(emp_1.raise_amount)    → 1.05  ← only changed for employee 1
print(emp_2.raise_amount)    → 1.04
```

Why?

```
emp_1.raise_amount = 1.05

print(emp_1.__dict__)
```

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

- finds it in its own namespace and returns the amount.

## Another example of a Class Variable

```
class Employee:

    raise_amount = 1.04
    num_of_emps = 0

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

        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_amount)

emp_1 = Employee('Devin', 'Powers', 50000)

emp_2 = Employee('Lebron', 'James', 2000000)

print(Employee.num_of_emps)
```

2

Keeping track of the # of Employees

] --init-- runs everytime We create a new employee!

```
class Employee:

    raise_amount = 1.04
    num_of_emps = 0

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

        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_amount)

print(Employee.num_of_emps)

emp_1 = Employee('Devin', 'Powers', 50000)

emp_2 = Employee('Lebron', 'James', 2000000)

print(Employee.num_of_emps)
```

0 ~ ran before we entered any employees.

2

## Python OOP Tutorial 3: Class Methods and Static Methods

```
class Employee:

    num_of_emps = 0

    raise_amt = 1.04


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

        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_amount)

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


emp_1 = Employee('Devin', 'Powers', 50000)

emp_2 = Employee('Lebron', 'James', 2000000)


print(Employee.raise_amt)   → 1.04
print(emp_1.raise_amt)      → 1.04
print(emp_2.raise_amt)      → 1.04
```

A regular method takes an instance or the first argument. How can we change it so it takes the class or the first argument. To do this we'll have to use class methods.

by adding decorator to the top!

More info? Go to Decorator notes!

* Decorator *

Pass cls as the first argument!

- We receive the class or the first argument instead of the instance

@ Classmethod is the same as saying / coding

set_raise_amt = classmethod(set_raise_amt)

```
class Employee:

    num_of_emps = 0

    raise_amt = 1.04


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

        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_amount)

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

★ Pass cls as the first argument ★

```
emp_1 = Employee('Devin', 'Powers', 50000)

emp_2 = Employee('Lebron', 'James', 2000000)

Employee.set_raise_amt(1.05)
```

← automatically accepts 'cls' so we just have to enter in an amount.

```
print(Employee.raise_amt) → 1.05
print(emp_1.raise_amt)  → 1.05
print(emp_2.raise_amt)  → 1.05
```

- Were working with the class instead of the instance.

```
class Employee:

    num_of_emps = 0

    raise_amt = 1.04


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

        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_amount)

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


emp_1 = Employee('Devin', 'Powers', 50000)
emp_2 = Employee('Lebron', 'James', 2000000)

emp_str_1 = 'John-Doe-7000'
emp_str_2 = 'Steve-Smith-3000'
emp_str_3 = 'Devin-Powers-6969'

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

new_emp_1 = Employee(first, last, pay)

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

Class method as alternative constructor

splitting strings

→ John.Doe@email.com

→ 7000

# [Alternative Constructor]

Instead of someone needing to parse the strings themselves, we've provided them with this strings method that they can call.

```
class Employee:

    num_of_emps = 0
    raise_amt = 1.04

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

        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_amount)

    @classmethod
    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)
```

Alternative Constructor

Use this method as an alternative constructor

line creates new employee then returns it

```
emp_1 = Employee('Devin', 'Powers', 50000)
emp_2 = Employee('Lebron', 'James', 2000000)

emp_str_1 = 'John-Doe-7000'
emp_str_2 = 'Steve-Smith-3000'
emp_str_3 = 'Devin-Powers-6969'

new_emp_1 = Employee.from_string(emp_str_1)

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

class → Employee
method → from_string

print(new_emp_1.email) → John.Doe@email.com
print(new_emp_1.pay) → 7000

Class methods as alternative constructors!

**Static Methods** ~ don't pass anything automatically

```
class Employee:

    num_of_emps = 0
    raise_amt = 1.04

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

        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_amount)

    @classmethod
    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)

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


emp_1 = Employee('Devin', 'Powers', 50000)
emp_2 = Employee('Lebron', 'James', 2000000)

import datetime
my_date = datetime.date(2020, 5, 17)

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

We can pass in argument we want to work with?

saturday

sunday

creating new date

→ False

since Weekend?

Video 4

## Python OOP Tutorial 4: Inheritance- Creating Subclasses

```
class Employee:

    raise_amt = 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_amt)


class Developer(Employee):
    pass


dev_1 = Developer('Devin', 'Powers', 50000)
dev_2 = Developer('Lebron', 'James', 2000000)

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

```
Devin.Powers@company.com
Lebron.James@company.com
```

## Using the Help

```
print(help(Developer))
```

```
Help on class Developer in module __main__:

class Developer(Employee)
 |  Developer(first, last, pay)
 |
 |  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:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from Employee:
 |
 |  raise_amt = 1.04

None
```

Let's customize the subclass

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

→ 50000
→ 52000

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

```
class Employee:

    raise_amt = 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_amt)

class Developer(Employee):
    raise_amt = 1.10

dev_1 = Developer('Devin', 'Powers', 50000)
dev_2 = Developer('Lebron', 'James', 2000000)

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

→ 50000
→ 55000

If we Changed it back to Employee Class

```
dev_1 = Employee('Devin', 'Powers', 50000)
dev_2 = Developer('Lebron', 'James', 2000000)

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

→ 50000
→ 52000

```
class Employee:

    raise_amt = 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_amt)

class Developer(Employee):
    raise_amt = 1.10

    def __init__(self, first, last, pay, prog_lang):
        super().__init__(first,last, pay)
       # Employee.__init(self, first, last, pay)  works the same as above
        self.prog_lang = prog_lang


dev_1 = Developer('Devin', 'Powers', 50000,'Python')
dev_2 = Developer('Lebron', 'James', 2000000, 'Java')

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

```
Devin.Powers@company.com
Python
```

## Creating a Manager Class

```
class Employee:

    raise_amt = 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_amt)

class Developer(Employee):
    raise_amt = 1.10

    def __init__(self, first, last, pay, prog_lang):
        super().__init__(first,last, pay)
        # Employee.__init(self, first, last, pay)  works the same as above
        self.prog_lang = prog_lang

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

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

dev_1 = Developer('Devin', 'Powers', 50000,'Python')
dev_2 = Developer('Lebron', 'James', 2000000, 'Java')

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

print(mgr_1.email)

mgr_1.print_emp()
```

**Add Another employee**

```
mgr_1.add_emp(dev_2)
mgr_1.print_emp()
```

**Remove an Employee**

```
mgr_1.add_emp(dev_2)
mgr_1.remove_emp(dev_2)

mgr_1.print_emp()
```

**Python has built in functions**

**isinstance()**

- Will tell us if an object is an instance of a class

```
print(isinstance(mgr_1, Manager))
print(isinstance(mgr_1, Employee))
print(isinstance(mgr_1, Developer))
```

**issubclass()**

- Will tell us if a class is a subclass of another

```
print(issubclass(Developer, Employee))
print(issubclass(Manager, Employee))
print(issubclass(Manager, Developer))
```

## Python OOP Tutorial 5: Special (Magic/Dunder) Methods

```
class Employee:

    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

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

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

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

    def __str__(self):
        return '{} - {}'.format(self.fullname(), self.email)

    def __add__(self, other):
        return self.pay + other.pay

    def __len__(self):
        return len(self.fullname())
```

__repr__ : called when run repr() on our objects
- Used for debugging or logging

__str__: called when run str()
- Used to display for users

```
class Employee:

    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

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

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

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

    #def __str__(self):
    #    return '{} - {}'.format(self.fullname(), self.email)

    def __add__(self, other):
        return self.pay + other.pay

    def __len__(self):
        return len(self.fullname())


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

print(emp_1)
```

```
Employee('Corey', 'Schafer', 50000)
```

```
class Employee:

    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

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

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

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

    def __str__(self):
        return '{} - {}'.format(self.fullname(), self.email)

    def __add__(self, other):
        return self.pay + other.pay

    def __len__(self):
        return len(self.fullname())


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

print(emp_1)
```

```
Corey Schafer - Corey.Schafer@email.com
```

-uses __str__ method

```
print(repr(emp_1))
print(str(emp_1))
```

```
Employee('Corey', 'Schafer', 50000)
Corey Schafer - Corey.Schafer@email.com
```

```
print(emp_1.__repr__())
print(emp_1.__str__())
```

```
Employee('Corey', 'Schafer', 50000)
Corey Schafer - Corey.Schafer@email.com
```

```
print(1+2)
print(int.__add__(1,2))
```

**__add__ Method**

```
def __add__(self, other):
    return self.pay + other.pay


print(emp_1 + emp_2)
```

**__len__ Method**

```
def __len__(self):
    return len(self.fullname())


print(len(emp_1))
```

## Python OOP Tutorial 6: Property Decorators – Getters, Setters, and Deleters

```
class Employee:

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

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


emp_1 = Employee('John', 'Smith')


print(emp_1.first)
print(emp_1.email)
print(emp_1.fullname())
```

```
John
John.Smith@email.com
John Smith
```

```
emp_1 = Employee('John', 'Smith')

emp_1.first = 'Jim'

print(emp_1.first)
print(emp_1.email)
print(emp_1.fullname())
```

```
Jim
John.Smith@email.com
Jim Smith
```

```
class Employee:

    def __init__(self, first, last):
        self.first = first
        self.last = last

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


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

emp_1 = Employee('John', 'Smith')

emp_1.first = 'Jim'

print(emp_1.first)
print(emp_1.email())
print(emp_1.fullname())
```

Without ()

```
emp_1 = Employee('John', 'Smith')

emp_1.first = 'Jim'

print(emp_1.first)
print(emp_1.email)
print(emp_1.fullname)
```

```
Jim
<bound method Employee.email of <__main__.Employee object at 0x113d245f8>>
<bound method Employee.fullname of <__main__.Employee object at 0x113d245f8>>
```

Adding the Property decorator

```
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)


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


emp_1 = Employee('John', 'Smith')

emp_1.first = 'Jim'

print(emp_1.first)
print(emp_1.email)
print(emp_1.fullname())
```

```
Jim
Jim.Smith@email.com
Jim Smith
```

```
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)


emp_1 = Employee('John', 'Smith')

emp_1.first = 'Jim'

print(emp_1.first)
print(emp_1.email)
print(emp_1.fullname)
```

```
Jim
Jim.Smith@email.com
Jim Smith
```

```
emp_1.fullname = 'Devin Powers'

print(emp_1.first)
print(emp_1.email)
print(emp_1.fullname)
```

```
Traceback (most recent call last):

  File "/Users/devinpowers/Desktop/MSU CSE 231 Class/Week 11 (Class Introduction)/
Corey Schafer Videos/Video5(special methods).py", line 20, in <module>
    emp_1.fullname = 'Devin Powers'

AttributeError: can't set attribute
```

**Use a Setter**

```
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


emp_1 = Employee('John', 'Smith')

emp_1.fullname = 'Devin Powers'

print(emp_1.first)
print(emp_1.email)
print(emp_1.fullname)
```

```
Devin
Devin.Powers@email.com
Devin Powers
```

**Deleter**

- Clean up code

```
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 = 'Devin Powers'

print(emp_1.first)
print(emp_1.email)
print(emp_1.fullname)

del emp_1.fullname
```

```
Devin
Devin.Powers@email.com
Devin Powers
Delete Name!
```