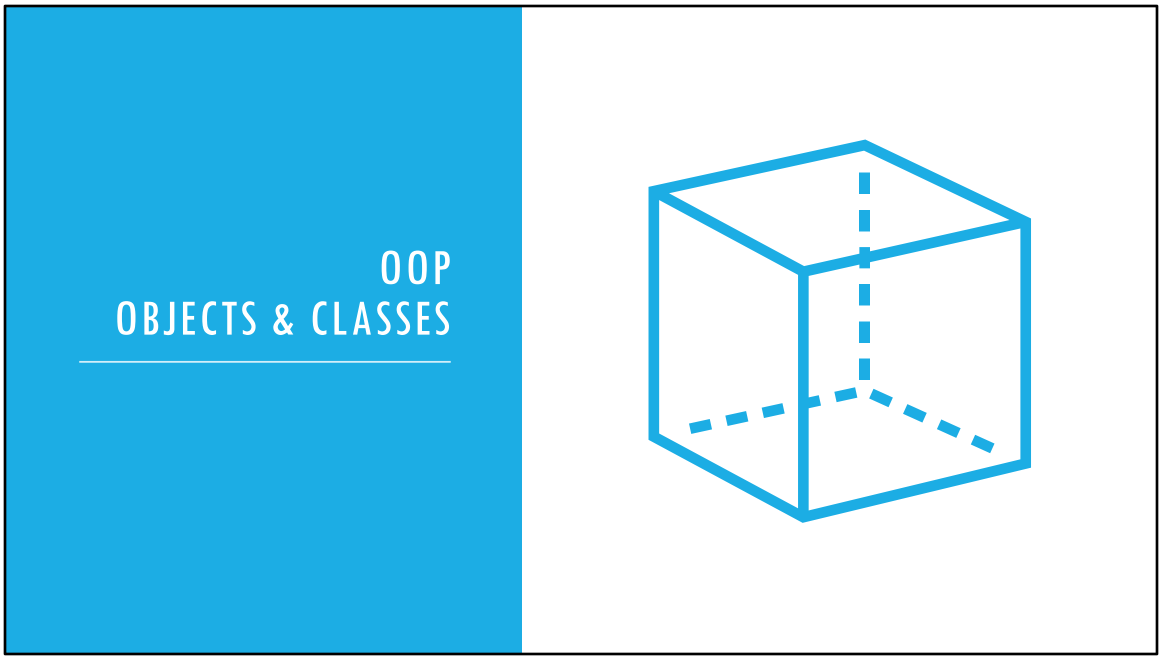


OOP
OBJECTS & CLASSES

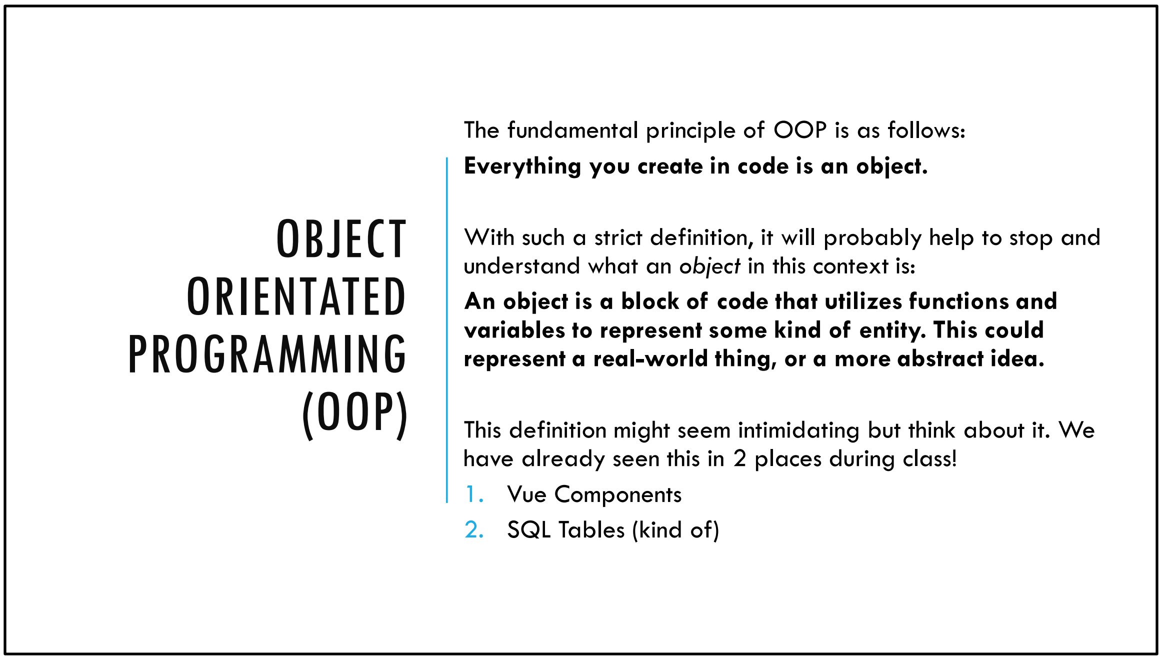
INTRODUCTION

So far in the course, we have learnt a couple different **programming paradigms.** Remember, a paradigm is a way of thinking about something. So, a programming paradigm is a way to think about programming.

We started learning the procedural paradigm where code runs line by line in order. From there we learnt about the event driven paradigm where we wrote functions and assigned those functions to run when certain actions were taken by the users.

OOP (Object Orientated Programming) is a paradigm that specializes in modeling. The idea behind OOP is that you take an idea and represent it with an object (think key-value pairs).

Believe it or not, we have a decent amount of experience with OOP in a very specific context. Vue is a component-based architecture and components are a branch of OOP.



OBJECT ORIENTATED PROGRAMMING (OOP)

The fundamental principle of OOP is as follows:
Everything you create in code is an object.

With such a strict definition, it will probably help to stop and understand what an *object* in this context is:
An object is a block of code that utilizes functions and variables to represent some kind of entity. This could represent a real-world thing, or a more abstract idea.

This definition might seem intimidating but think about it. We have already seen this in 2 places during class!

1. Vue Components
2. SQL Tables (kind of)

SQL tables might be a bit of a stretch, but they do a really good job of highlighting the difference between the structure of something (schema / columns of a table) and actual instances of that thing (rows). They also do a really good job of teaching people that entities can have different kinds of relationships between them in code (one-to-one, one-to-many, many-to-many).

OOP CONT.

Let’s focus on that “everything is an object” statement. Just like in Vue where everything was a component (object) the goal of turning everything you create in code into an object is to create reusable independent blocks of code that can be dropped in anywhere you need.

Think about Foodie. A good design for the components broke our pages up into small reusable components. If done correctly, you can throw those components anywhere you want on your page and expect them to act the same.

Although we don’t have a “page” in python to display on, that doesn’t mean we can’t use the idea of breaking up our program into smaller objects and stich them together as needed!

OOP CAUTION

Just because everything can be an object does not mean everything should be an object. In Vue you had an entire framework built around the idea that everything you create was a component (object).

In python we are not as constricted into this idea. We can use OOP where we think we need it and then use procedural or event-driven where that makes more sense.

We need to use OOP like a tool on our toolbelt. We use it when we think it will help, but we don’t force it into situations it doesn’t belong.

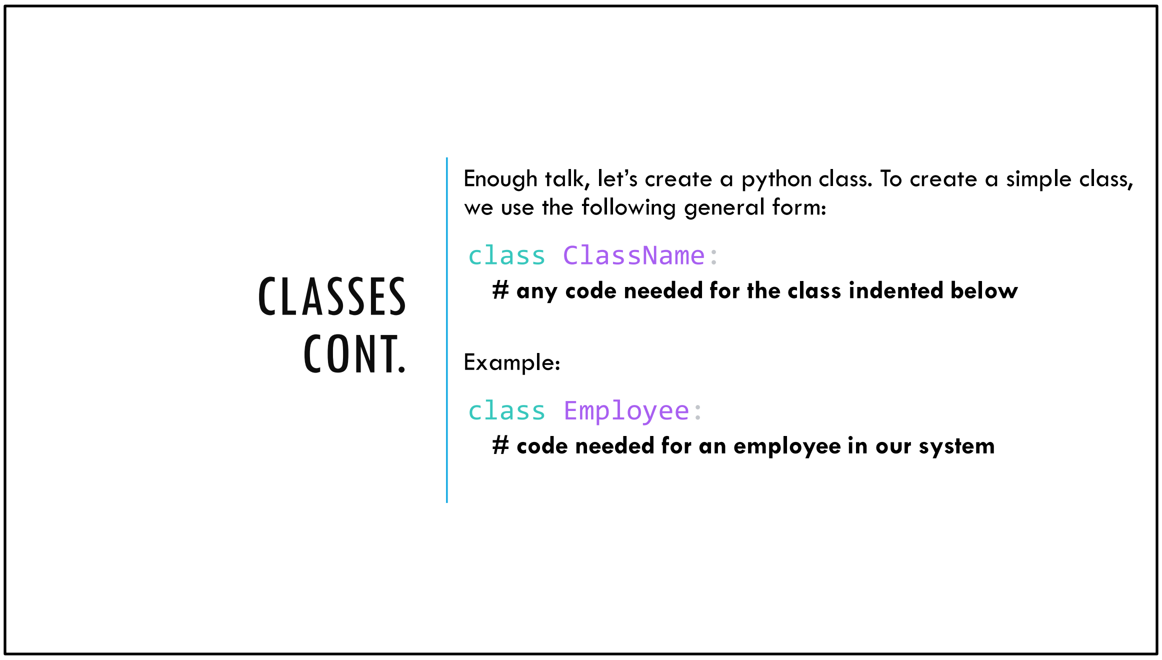
CLASSES

In OOP, we have the notion of a **class**. A class is simply the code that defines an **object**.

A class is supposed to represent the structure of either a real-world thing (car, house, tree, animal, person, etc.) or an abstract idea (user, bank account, menu_item, etc.).

In SQL we represented the definition of something using a table. We gave a table columns that defined that "thing". For example, an employee has a first_name, last_name, salary, position and start_date. These columns represented the idea of an employee.

A class is supposed to achieve the same goal, but with more power! We can use variables to define attributes of the thing we are trying to represent but we can also create functions to represent actions that thing can do!



CLASSES CONT.

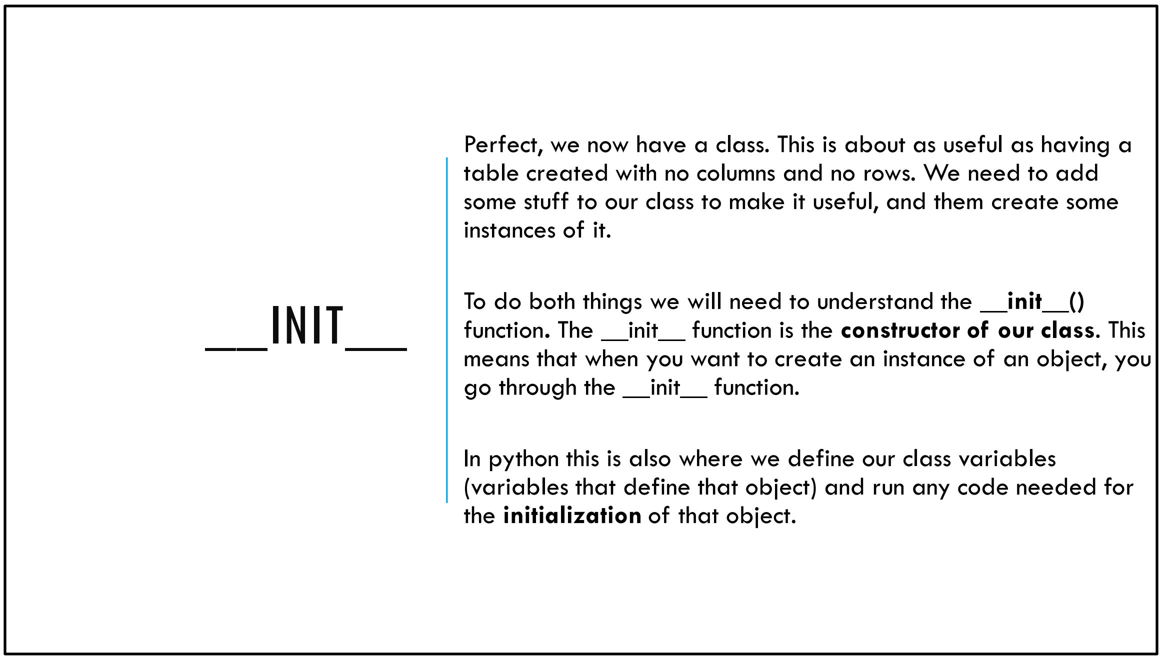
Enough talk, let's create a python class. To create a simple class, we use the following general form:

```
class ClassName:
    # any code needed for the class indented below
```

Example:

```
class Employee:
    # code needed for an employee in our system
```

Class names are given PascalCase in python (and in almost any language that supports OOP)!



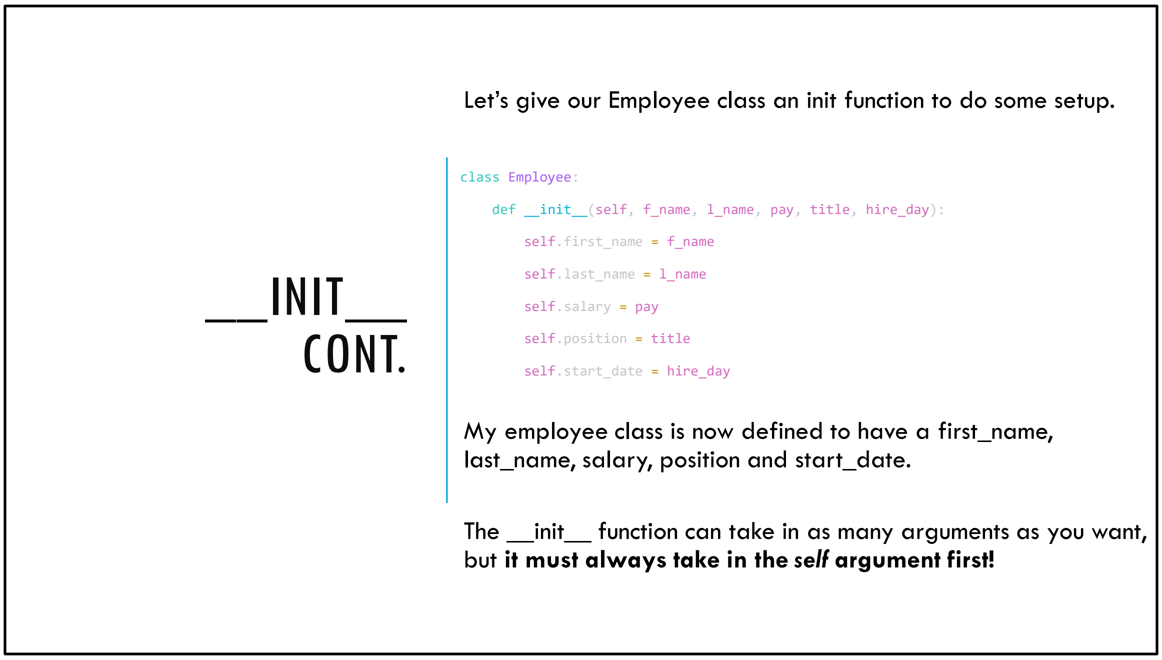
__INIT__

Perfect, we now have a class. This is about as useful as having a table created with no columns and no rows. We need to add some stuff to our class to make it useful, and them create some instances of it.

To do both things we will need to understand the __**init**__() function. The __init__ function is the **constructor of our class**. This means that when you want to create an instance of an object, you go through the __init__ function.

In python this is also where we define our class variables (variables that define that object) and run any code needed for the **initialization** of that object.

__init__ is two underscores on both sides.



__INIT__ CONT.

Let's give our Employee class an init function to do some setup.

```
class Employee:
    def __init__(self, f_name, l_name, pay, title, hire_day):
        self.first_name = f_name
        self.last_name = l_name
        self.salary = pay
        self.position = title
        self.start_date = hire_day
```

My employee class is now defined to have a first_name, last_name, salary, position and start_date.

The __init__ function can take in as many arguments as you want, but **it must always take in the *self* argument first!**

Ignore the meaning of the **self** argument for now. It is something python needs in every function that lives inside a class to work behind the scenes and we will talk about it later.

What this __init__ function is saying is that if you want to create an employee you must pass 5 arguments. These arguments make up the class variables of an employee. You can think of this as saying "an employee has a first_name, last_name, salary, position and start_date".

Remember, the variables defined in the arguments of __init__ are just local variables to the init function (f_name, l_name, pay, title and hire_day). It is the variables attached to the **self** keyword that define your object!

This is very much like the **props** section of a component. Variables that were local to a component that needed to be passed in.

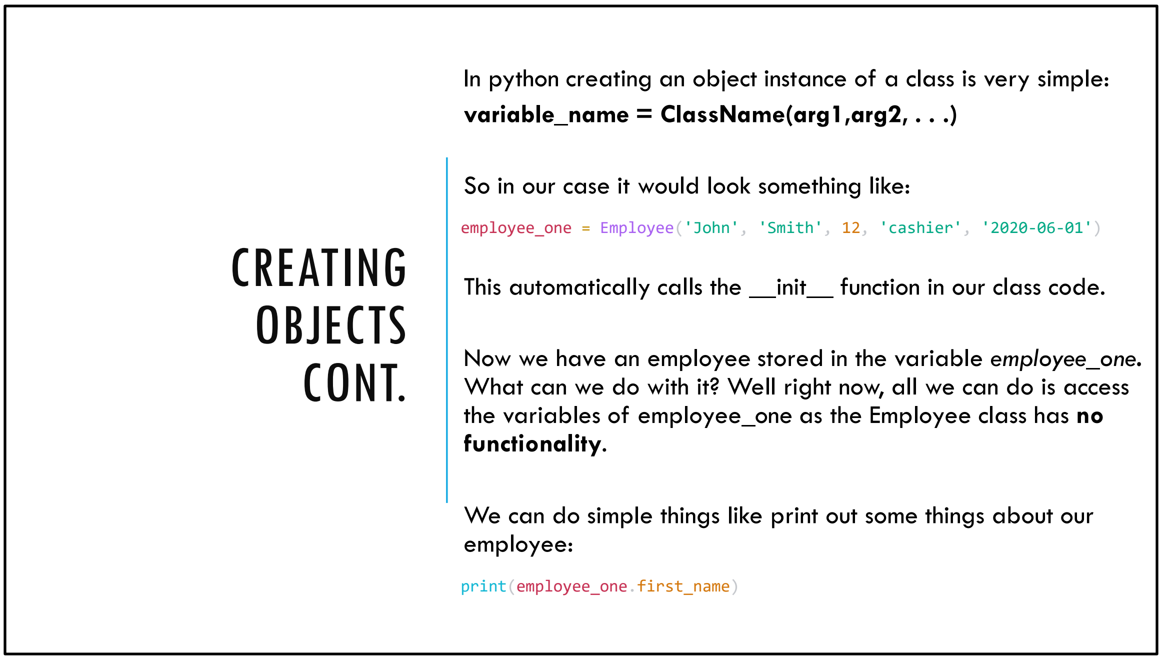
Not every class variables needs to be passed in through arguments to __init__. You can set variables to some hardcoded value if needed.

CREATING OBJECTS

So far in our class code we have defined how to create an employee. This is a lot like defining the columns of a table. You have set the rules of how to create a specific thing.

All we need now is to create some actual employees! This is like creating rows in SQL where each row is an actual real thing.

In OOP, creating a “thing” is usually called **instantiation (creating an object instance)**. We create an actual instance of that class and store it in a variable. Much like creating a row (minus the variable part).



CREATING OBJECTS CONT.

In python creating an object instance of a class is very simple:
variable_name = ClassName(arg1,arg2, . . .)

So in our case it would look something like:

```
employee_one = Employee('John', 'Smith', 12, 'cashier', '2020-06-01')
```

This automatically calls the __init__ function in our class code.

Now we have an employee stored in the variable *employee_one*. What can we do with it? Well right now, all we can do is access the variables of employee_one as the Employee class has **no functionality**.

We can do simple things like print out some things about our employee:

```
print(employee_one.first_name)
```

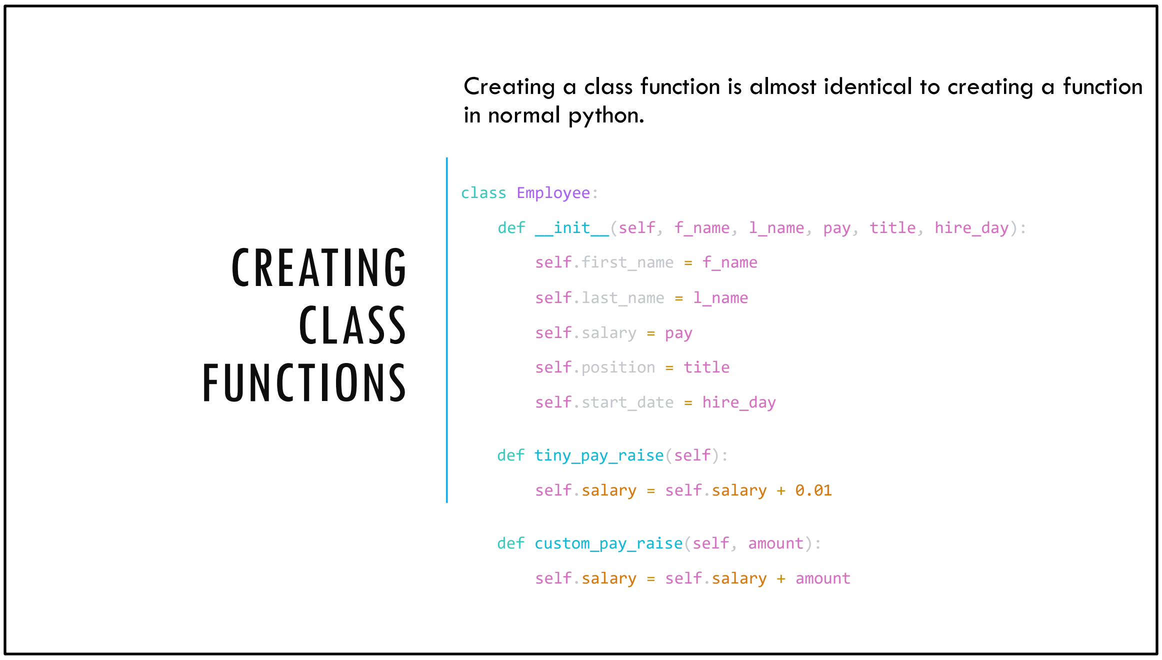
Notice:

1. When creating our employee we ignore the **self** argument that the __init__ function is expecting! That is just something python uses behind the scenes.
2. To access our class variables we use the **dot syntax** (object_variable_name.class_variable_name)

CLASS FUNCTIONS

If all we need is the ability to store simple data in our objects, then we are good to go! Chances are you will want your objects to do something though.

We can give our classes functions that can give our objects the ability to do things. This is like methods in Vue components.



CREATING CLASS FUNCTIONS

Creating a class function is almost identical to creating a function in normal python.

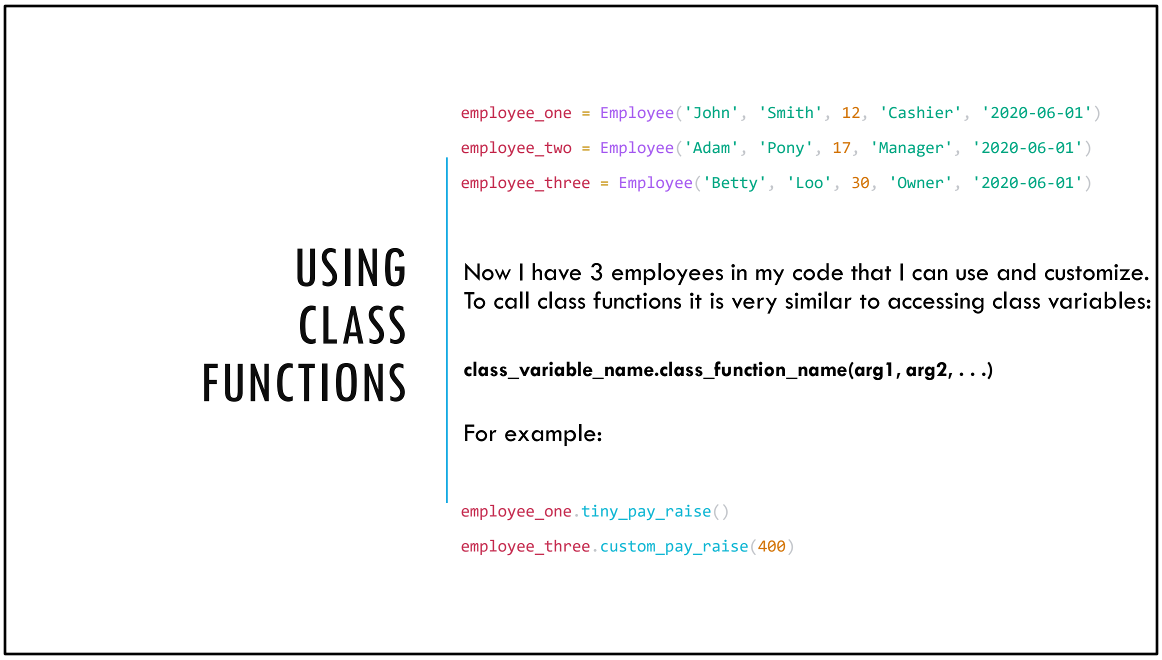
```
class Employee:
    def __init__(self, f_name, l_name, pay, title, hire_day):
        self.first_name = f_name
        self.last_name = l_name
        self.salary = pay
        self.position = title
        self.start_date = hire_day

    def tiny_pay_raise(self):
        self.salary = self.salary + 0.01

    def custom_pay_raise(self, amount):
        self.salary = self.salary + amount
```

Now every employee can be given a tinyPayRaise.

Again notice the **self** argument. **Every class function needs to have the self argument as the first argument always.** Normal arguments can simply follow the self argument.



```python
employee_one = Employee('John', 'Smith', 12, 'Cashier', '2020-06-01')
employee_two = Employee('Adam', 'Pony', 17, 'Manager', '2020-06-01')
employee_three = Employee('Betty', 'Loo', 30, 'Owner', '2020-06-01')
```

USING CLASS FUNCTIONS

Now I have 3 employees in my code that I can use and customize. To call class functions it is very similar to accessing class variables:

class_variable_name.class_function_name(arg1, arg2, . . .)

For example:

```python
employee_one.tiny_pay_raise()
employee_three.custom_pay_raise(400)
```

Again notice that when calling the functions we simply ignored the **self** argument declared in the function definition!

SELF

We have been seeing this **self** argument referenced a few times in our code. This is very similar to the **this** keyword in JS. The **self** variable is known as a **context variable.**

What this really means is python needs it to understand what object is being referred to under the hood inside a class definition.

So the general rule of thumb:

Every function inside a class definition needs to have the self argument as the first argument

This argument is used to *reference* class variables and functions when working inside the class itself. When calling a class function, we can safely ignore this argument and python will handle it!

OBJECT RELATIONSHIPS

In OOP, our objects can relate to each other based on how they are supposed to interact. We have two very basic relationships in OOP:

1. Inheritance
   1. This is often referred to as a "is-a" relationship
2. Composition
   1. This is often referred to as a "has-a" relationship

We will talk about each one of these relationships in details later as they each deserve a longer talk!

INHERITANCE

One of the most common way that people like to structure their objects in an OOP code base is through **inheritance.** Inheritance is modeled after the idea that Parents pass down traits to their children.

In code, we have seen this parent-child relationship many times. In OOP we make this relationship very explicit. The idea is to have a **child** object inherit the functionality of the **parent** object.

This idea might seem strange, but if you are trying to model or represent real world ideas in code this comes in handy a lot.

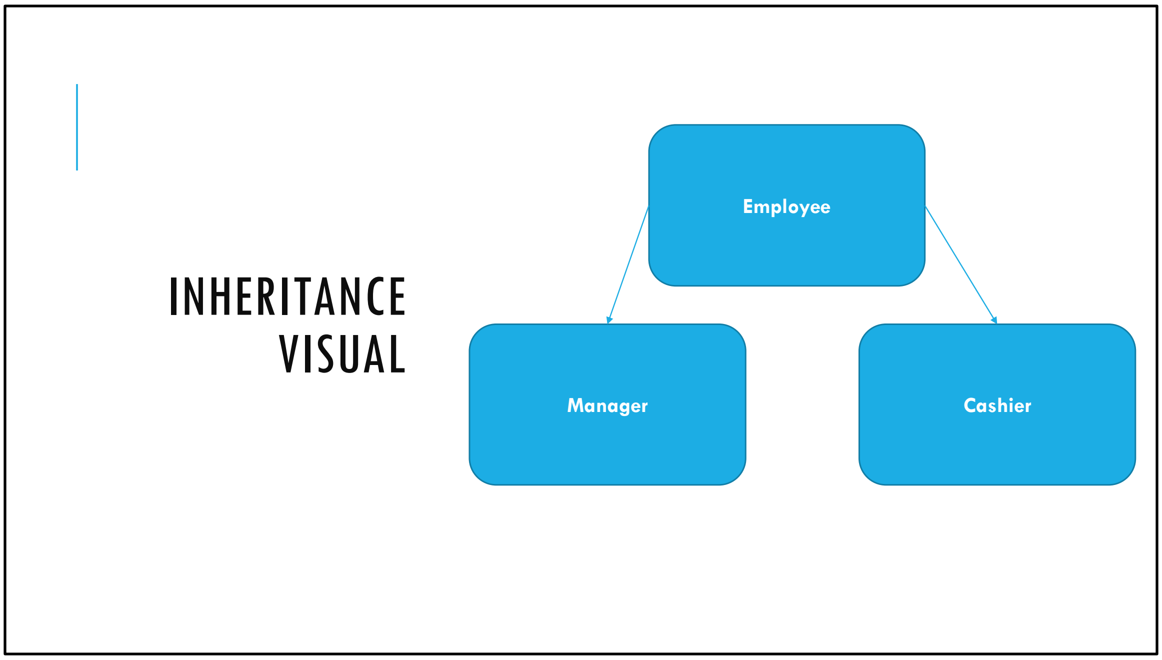
INHERITANCE CONT.

Let’s think about a real world example:

Imagine you wanted to create a code base that had all kinds of “employee” interactions. Perhaps some software for time or payroll management.

In your software, **every** employee has some basic functionality (maybe things like log hours, take vacation, etc.). There are also certain things that employees can do. Perhaps a manager can terminate or promote non-management employees. Maybe a cashier can request a pay raise.

We can start to think of the entities we are trying to represent as objects and organize them into a hierarchy.


INHERITANCE VISUAL

Employee
Manager
Cashier

This is the visual idea of inheritance. Both a manager and a cashier are employees. Both the cashier and the manager will **inherit** the attributes and functionality from the Employee class.

This means you should put things that **all** employees have in the Employee class, and things for specifically manager and cashiers in their respective classes.

For example, chances are every employee has a name, start_date, or SIN. Perhaps only a Manager has a bonus_target or rrsp_match. These more specific attributes would go in the manager class.

Splitting code like this allows us to limit the amount of copy paste we need to do in different objects.

PURPOSE

All this information is vital to know if you are going to be building OOP based projects with a focus on inheritance. We are not doing this.

We are however going to leverage inheritance to our advantage by using **built-in classes.** Just like built-in functions in python, they also have some built-in classes that we can **inherit** from to gain functionality.

One of the easiest examples of this in python is creating your own **exceptions**.

CUSTOM EXCEPTIONS

In python, there is a **base class** included called **Exception** that we can take advantage of to create our own exceptions.

To start, let's make our own class:

```
class ValueTooSmall:
```

Good start. Now, how do we **inherit** all the code from the parent class **Exception?**

It is as simple as:

```
class ValueTooSmall(Exception):
```

CUSTOM EXCEPTIONS CONT.

```
class ValueTooSmall(Exception):
```

So what have we done here? All we have added in our class definition is round brackets and the name of a class. If this class exists, python will **inherit** all the code (functions, variables) from the Exception class and your ValueTooSmall will have access to it.

We have just created a relationship where the Exception class is the parent and the ValueTooSmall class is the child. Any specific behaviour we want can go into the ValueTooSmall.

You can follow this syntax for any class, built in or one that you have built yourself:

class ClassName(ParentClass):

ACCESSING THE PARENT CLASS

So now we can inherit functionality from a parent class, but how do we make use of it? We have a couple of options.

1. Using the **super** keyword
   1. You may see things on the internet that refer to the **super** class. This is a term in OOP means the parent class. If you need to access the **super** class, python gives you the built in function super() to access anything you need from the parent
2. Using the parent class name
   1. You can also access a parent class just by using the class name. Either one gets you access to the same content in the parent class in simple cases.

We are not digging into the weeds very much when it comes to OOP, so for our use cases these methods are pretty much the same. In real OOP practice they are different, but they are different in ways most other languages don't even consider to be OOP anyways.

ACCESSING THE PARENT CLASS CONT.

In the child class we can now access the parent. This is useful for many cases, one of the main cases is accessing the parent **constructor** or __init__ function.

```
class ValueTooSmall(Exception):
    def __init__(self):
        super().__init__('Oh no! That value was too small')
```

Here our constructor is simply calling the parent (Exception) constructor. Notice we are passing a string to the constructor; this is a common pattern in custom exceptions. This string will be the error message printed to the terminal!

ACCESSING THE PARENT CLASS CONT.

We can make use of this in our Employee class from before, not allowing you to make an employee with too low of pay:

```
class Employee:
    def __init__(self, f_name, l_name, pay, title, hire_day):
        if(pay <= 5.0):
            raise ValueTooSmall()
        self.first_name = f_name
        self.last_name = l_name
        self.salary = pay
        self.position = title
        self.start_date = hire_day
```

The **raise** keyword in python is just a way to raise an exception on purpose in your code.

WRAPPING UP

One of the reasons students struggle with OOP is they forget that it is supposed to **change the way the think and understand programming.** It is a totally different way to problem solve, and it is a tool that solves some problems well and other problems very poorly.

Many languages allow for OOP syntax, the most famous is **Java.** If something about OOP speaks to you and you feel like learning more, remember that just because you think about coding differently does not mean that OOP isn't just built on the fundamentals that you learnt.

The same way that in your event-driven programming you still needed functions, conditionals, loops and variables you need the same thing in OOP.

There are a few more topics when it comes to OOP (encapsulation, polymorphism, and other scary words) but those can be covered on your own if you really dig OOP.

Remember, OOP is just a **way** to think about coding. It is not one of the fundamentals. Having a strong understanding of the fundamentals is much more important than having a strong understanding of OOP. I would say that having a strong understand of OOP isn't even possible without the fundamentals.