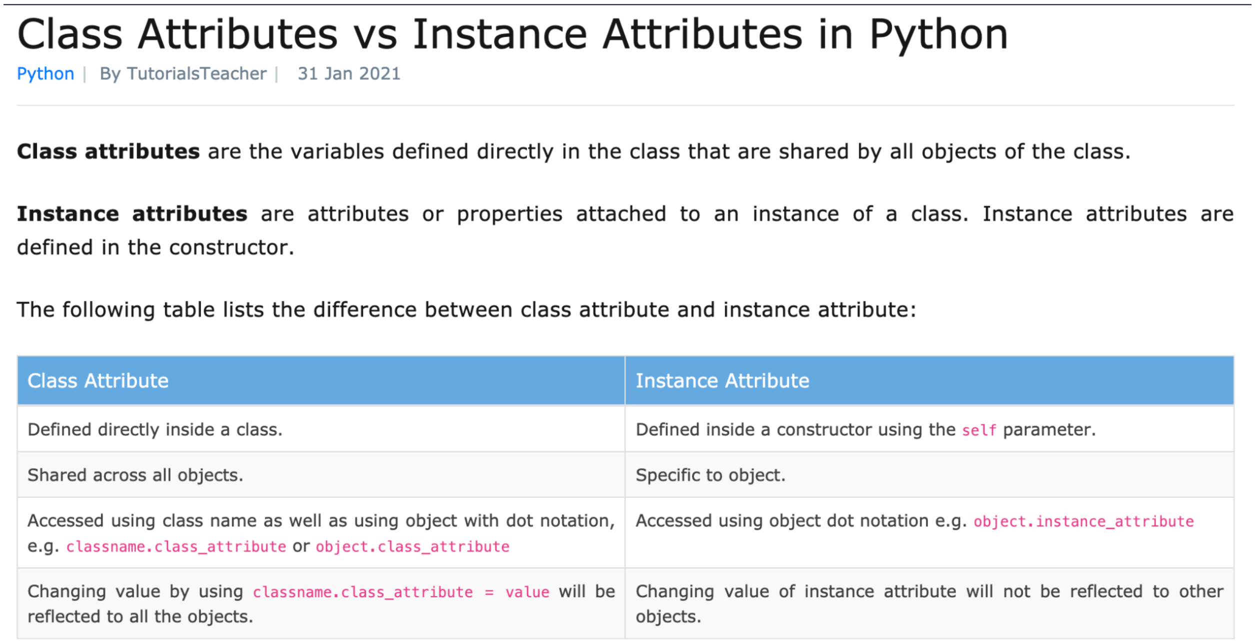
Object Oriented Programming (cont’d)

Class Attributes:

- A class attribute is defined once and lives on the class itself.
- They are defined directly on a class which the attributes are then shared by all instances of a class and the class itself.



Class Attributes vs Instance Attributes in Python

Python | By TutorialsTeacher | 31 Jan 2021

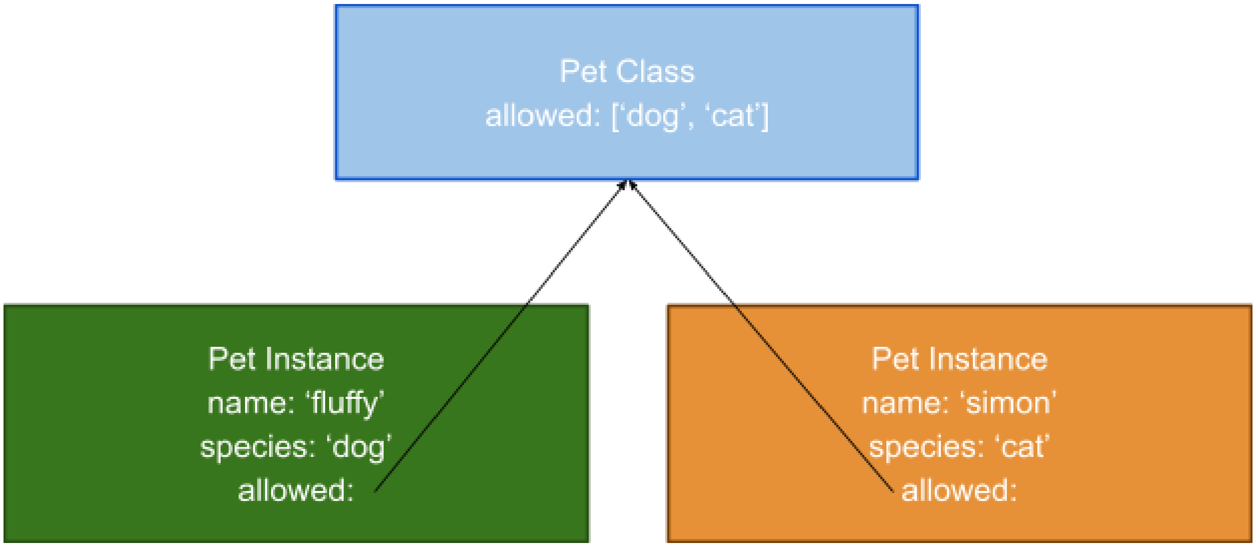
Class attributes are the variables defined directly in the class that are shared by all objects of the class.

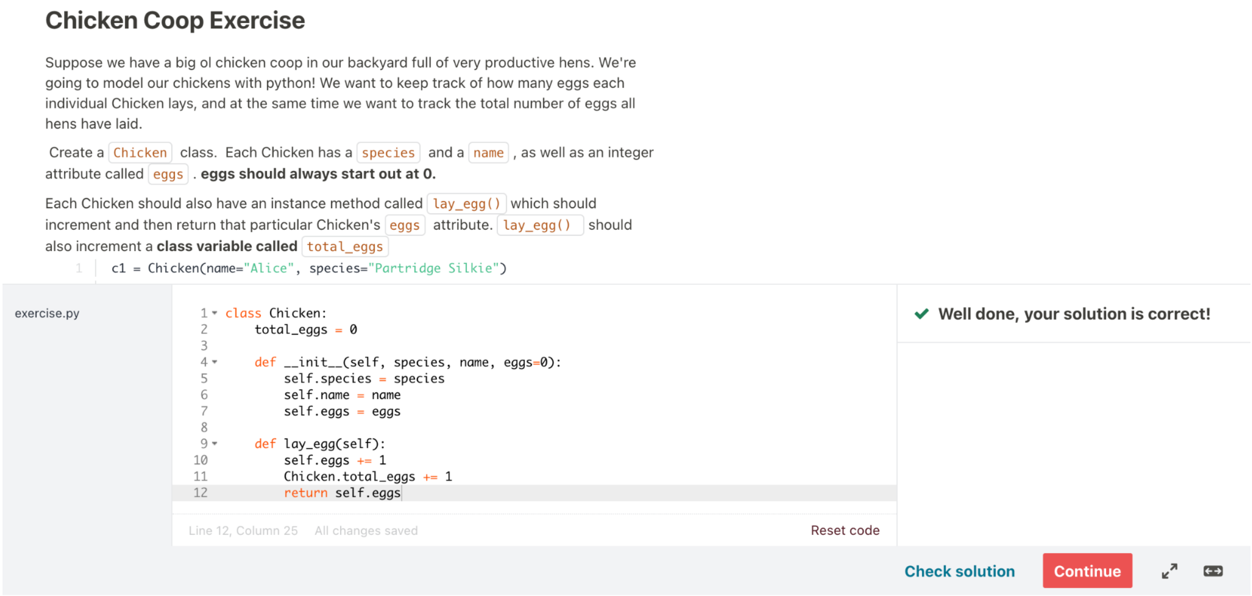
Instance attributes are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor.

The following table lists the difference between class attribute and instance attribute:

Class Attribute	Instance Attribute
Defined directly inside a class.	Defined inside a constructor using the `self` parameter.
Shared across all objects.	Specific to object.
Accessed using class name as well as using object with dot notation, e.g. `classname.class_attribute` or `object.class_attribute`	Accessed using object dot notation e.g. `object.instance_attribute`
Changing value by using `classname.class_attribute = value` will be reflected to all the objects.	Changing value of instance attribute will not be reflected to other objects.

We can also define attributes on a class that are shared by all instances of a class and the class itself.





Chicken Coop Exercise

Suppose we have a big ol chicken coop in our backyard full of very productive hens. We're going to model our chickens with python! We want to keep track of how many eggs each individual Chicken lays, and at the same time we want to track the total number of eggs all hens have laid.

Create a `Chicken` class. Each Chicken has a `species` and a `name`, as well as an integer attribute called `eggs`. **eggs should always start out at 0.**

Each Chicken should also have an instance method called `lay_egg()` which should increment and then return that particular Chicken's `eggs` attribute. `lay_egg()` should also increment a **class variable called** `total_eggs`

```
c1 = Chicken(name="Alice", species="Partridge Silkie")
```

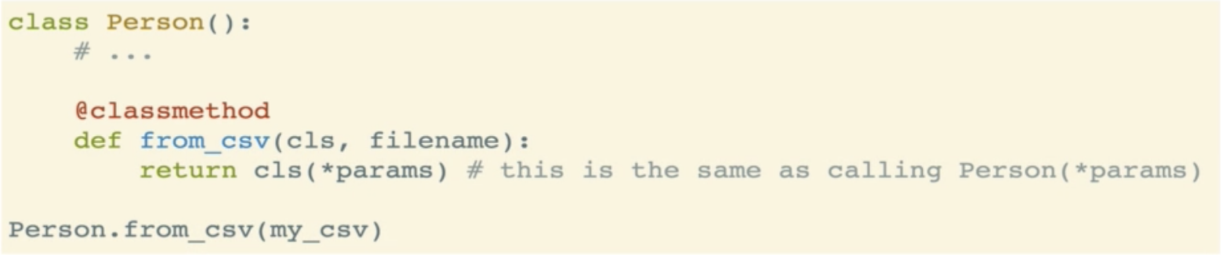
exercise.py

```
class Chicken:
    total_eggs = 0

    def __init__(self, species, name, eggs=0):
        self.species = species
        self.name = name
        self.eggs = eggs

    def lay_egg(self):
        self.eggs += 1
        Chicken.total_eggs += 1
        return self.eggs
```

Line 12, Column 25   All changes saved

✓ Well done, your solution is correct!

<u>Class Methods:</u>
Class methods are methods (with the @classmethod decorator) that are not concerned with instances, but the class itself.



```
class Person():
    # ...

    @classmethod
    def from_csv(cls, filename):
        return cls(*params) # this is the same as calling Person(*params)

Person.from_csv(my_csv)
```

The function defined underneath @classmethod (the decorator—which will be discussed further later) is a class method.

By creating class methods, the class is automatically going to be passed to the method. Therefore, instead of putting ‘self’ within the parameters (not that it actually matters), the standard parameter to put is ‘cls’. This helps signify to the developers that what is within the class method is not an instance, but the actual class.

Class methods are used when the method does not need to know about the specific instance; instance methods are the opposite.

The __repr__ method:
String representation is one of the several ways to provide a nicer string representation.

i.e.

```
def __repr__(self):
        return f"{self.first} is {self.age}"    # returns Tom is 89
```

__Remember: A class is a blueprint for constructing objects; an instance is an object constructed from the class definition.__**

__Encapsulation— is the princess of designing a programmatic class using public and private methods and attributes to implement abstraction.__

__Abstraction— the idea of exposing only “relevant” data in a class interface, hiding private attribute and methods (aka the “inner workings”) from users.__

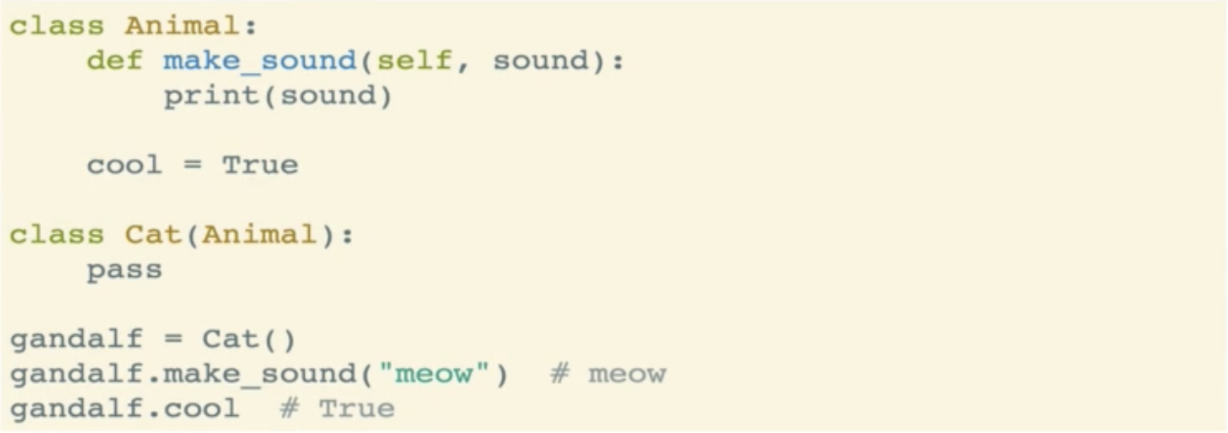
__Inheritance and Objectives:__

Objectives:

- Implement inheritance, including multiple
- Understand Method Resolution Order
- Understand polymorphism
- Add special methods to classes

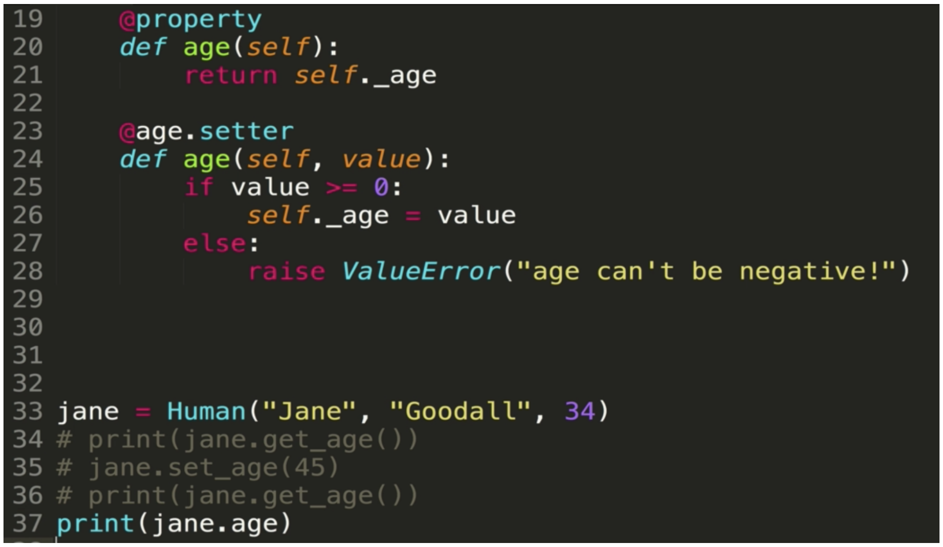
A key feature of OOP is the ability to define a class which inherits from another class (a “base” or “parent” class).

In Python, inheritance works by passing the parent class as an argument to the definition of a child class:



```
class Animal:
    def make_sound(self, sound):
        print(sound)

    cool = True

class Cat(Animal):
    pass

gandalf = Cat()
gandalf.make_sound("meow")  # meow
gandalf.cool  # True
```

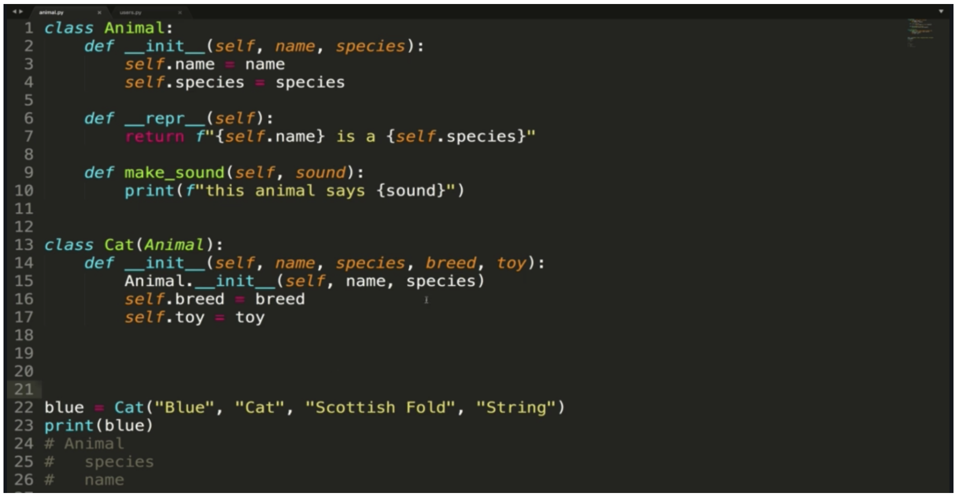
@property:
It is a pythonic way for a setter. It is also useful as a getter, but even more so as a setter.



```
    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value >= 0:
            self._age = value
        else:
            raise ValueError("age can't be negative!")




jane = Human("Jane", "Goodall", 34)
# print(jane.get_age())
# jane.set_age(45)
# print(jane.get_age())
print(jane.age)
```

Read more here: https://www.programiz.com/python-programming/property

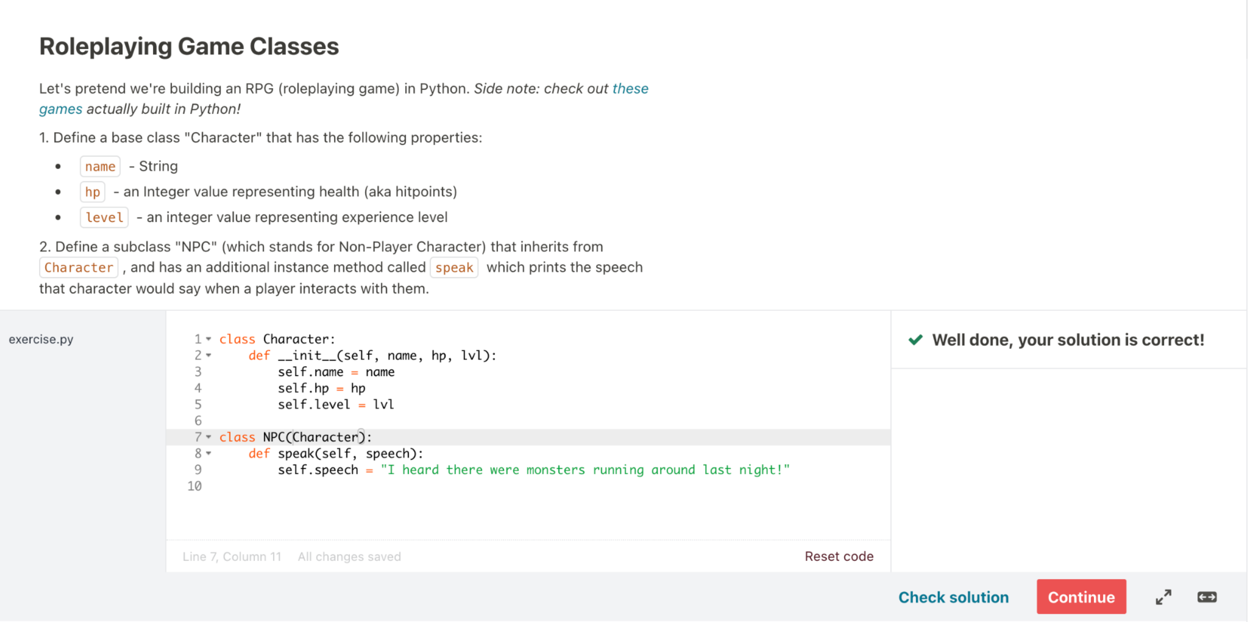
Super():



```
 1 class Animal:
 2     def __init__(self, name, species):
 3         self.name = name
 4         self.species = species
 5
 6     def __repr__(self):
 7         return f"{self.name} is a {self.species}"
 8
 9     def make_sound(self, sound):
10         print(f"this animal says {sound}")
11
12
13 class Cat(Animal):
14     def __init__(self, name, species, breed, toy):
15         Animal.__init__(self, name, species)
16         self.breed = breed
17         self.toy = toy
18
19
20
21
22 blue = Cat("Blue", "Cat", "Scottish Fold", "String")
23 print(blue)
24 # Animal
25 #     species
26 #     name
```

In the example shown above, we can use super() in line 15, to replace the entire line. Therefore, line 15 should look like this:

super().__init__(name, species)   # there is no need to put self in param



Roleplaying Game Classes

Let's pretend we're building an RPG (roleplaying game) in Python. *Side note: check out these games actually built in Python!*

1. Define a base class "Character" that has the following properties:

- `name` - String
- `hp` - an Integer value representing health (aka hitpoints)
- `level` - an integer value representing experience level

2. Define a subclass "NPC" (which stands for Non-Player Character) that inherits from `Character` , and has an additional instance method called `speak` which prints the speech that character would say when a player interacts with them.

```python
class Character:
    def __init__(self, name, hp, lvl):
        self.name = name
        self.hp = hp
        self.level = lvl

class NPC(Character):
    def speak(self, speech):
        self.speech = "I heard there were monsters running around last night!"

```

✔ Well done, your solution is correct!

<u>Multiple Inheritance:</u>

When it comes to multiple inheritance, it’s not used that often, but it is still listed in the course as additional information to learn and have thorough knowledge of Python.

i.e.

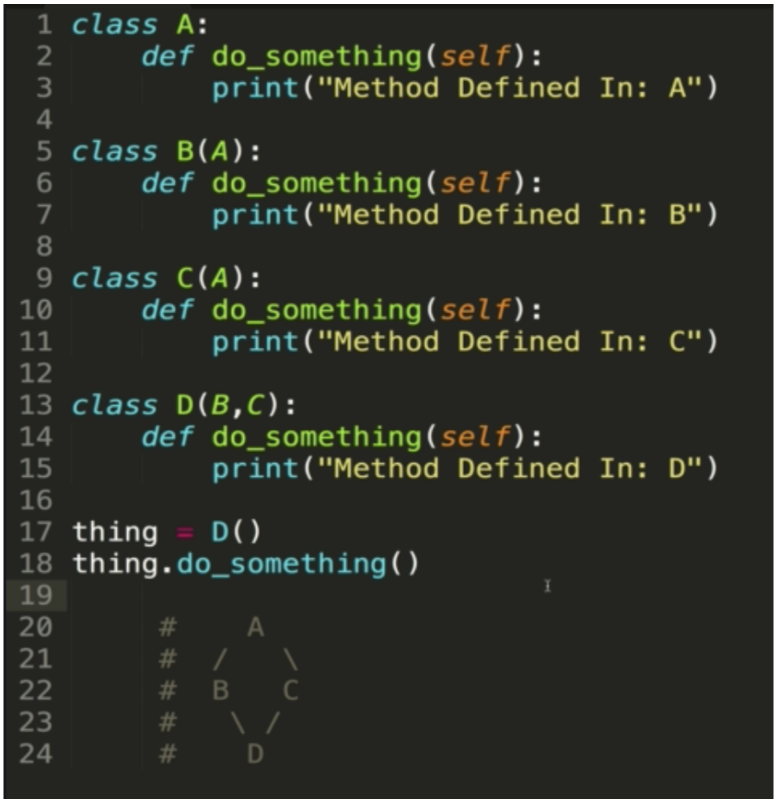
```
class Aquatic:
    def __init__(self, name):
        self.name = name

    def swim(self):
        return f”{self.name} is swimming”

    def greet(self):
        return f”I am {self.name} of the sea!”

class Ambulatory:
    def __init__(self, name):
        self.name = name

    def walk(self):
        return f”{self.name} is walking”

    def greet(self):
        return f”I am {self.name} of the land!”

class Penguin(Ambulatory, Aquatic):
    def __init__(self, name):
        super().__init__(name=name)
```

With multiple inheritance, Python allows classes to inherit from more than one parent class. The example shown above shows us how multiple inheritance works.

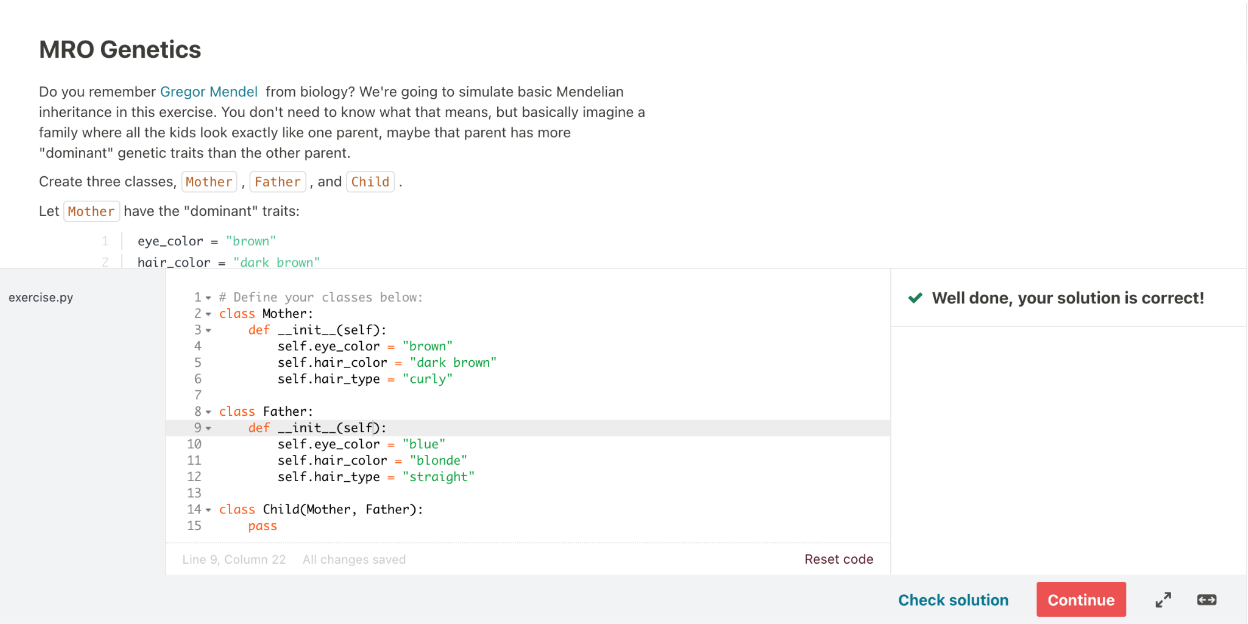
Method Resolution Order:
Whenever you create a class, Python sets a Method Resolution Order, or MRO, for that class, which is the order in which Python will look for methods on instances of that class—essentially, it is a hierarchy.

It is possible to programmatically reference the MRO in three ways:

- __mro__ attribute on the class
- Use the mro() method on the class
- Use the built-in help(cls) method   # best for HUMAN readability



```
class A:
    def do_something(self):
        print("Method Defined In: A")

class B(A):
    def do_something(self):
        print("Method Defined In: B")

class C(A):
    def do_something(self):
        print("Method Defined In: C")

class D(B,C):
    def do_something(self):
        print("Method Defined In: D")

thing = D()
thing.do_something()

     #    A
     #  /   \
     #  B   C
     #   \ /
     #    D
```



MRO Genetics

Do you remember Gregor Mendel from biology? We're going to simulate basic Mendelian inheritance in this exercise. You don't need to know what that means, but basically imagine a family where all the kids look exactly like one parent, maybe that parent has more "dominant" genetic traits than the other parent.

Create three classes, `Mother`, `Father`, and `Child`.

Let `Mother` have the "dominant" traits:

```
eye_color = "brown"
hair_color = "dark brown"
```

exercise.py

```python
# Define your classes below:
class Mother:
    def __init__(self):
        self.eye_color = "brown"
        self.hair_color = "dark brown"
        self.hair_type = "curly"

class Father:
    def __init__(self):
        self.eye_color = "blue"
        self.hair_color = "blonde"
        self.hair_type = "straight"

class Child(Mother, Father):
    pass
```

Line 9, Column 22   All changes saved

✓ Well done, your solution is correct!

Polymorphism:

A key principle in OOP is the idea of polymorphism—an object can take on many (poly) forms (morph).

While a formal definition of polymorphism is more difficult, here are two important practical applications:

1. The same class method works in a similar way for different classes
2. The same operation works for different kinds of objects

Polymorphism & Inheritance:

1. The same class method works in a similar way for different classes

    A common implementation of this is to have a method in a base (or parent) class that is overridden by a subclass. This is called **method overriding**.

    Each subclass will have a different implementation of the method.
    If the method is not implemented in the subclass, the version in the parent class is called instead.

Special Methods:

2. (Polymorphism) The same operation works for different kinds of objects

    How does the following work in Python?

    ```
    8 + 2          # 10
    “8” + “2”      # 82
    ```

Special Methods Example:

What is happening in our example? i.e.

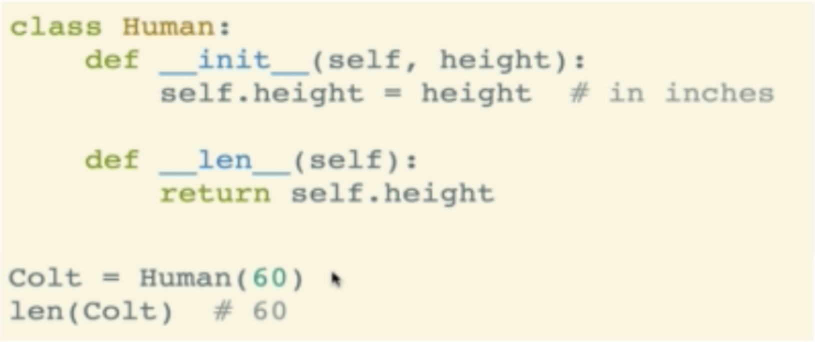
```
8 + 2          # 10
“8” + “2”      # 82
```

The + operator is shorthand for a special method called **__add__()** that gets called on the first operand.

If the first (left) operand is an instance of **int**, __add__() does mathematical **addition**. If it’s a **string**, it does string **concatenation**.

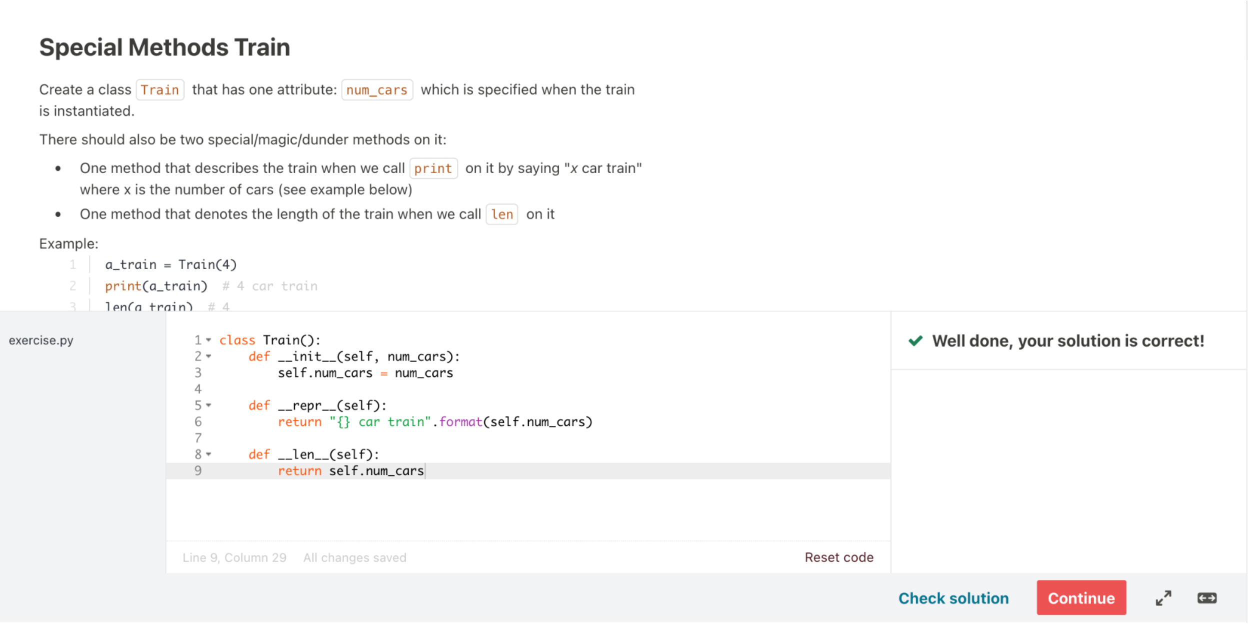
Special Method Applied:

Therefore, you can declare special methods on your own classes to mimic the behavior of bulletin objects, like so using **__len__**:



```python
class Human:
    def __init__(self, height):
        self.height = height  # in inches

    def __len__(self):
        return self.height

Colt = Human(60)
len(Colt)  # 60
```

More to read here: https://docs.python.org/3/reference/datamodel.html#special-method-names



Special Methods Train

Create a class `Train` that has one attribute: `num_cars` which is specified when the train is instantiated.

There should also be two special/magic/dunder methods on it:

- One method that describes the train when we call `print` on it by saying "*x* car train" where x is the number of cars (see example below)
- One method that denotes the length of the train when we call `len` on it

Example:

```python
a_train = Train(4)
print(a_train)  # 4 car train
len(a_train)  # 4
```

exercise.py

```python
class Train():
    def __init__(self, num_cars):
        self.num_cars = num_cars

    def __repr__(self):
        return "{} car train".format(self.num_cars)

    def __len__(self):
        return self.num_cars
```

✓ Well done, your solution is correct!