Classes

University of Wyoming COSC 1010

Adapted from: *Python Crash Course 3rd Ed* By Eric Matthes

Classes

- Objected-oriented programming (OOP) is an effective approach to writing software
- In OOP you write *classes* that represent real-world things and situations
- Then you create *objects* based on those classes
- When you write a class you define the general behavior that a whole category of objects can have

Classes

- You can create individual objects from the class
- Each object is automatically equipped with the general behavior of the class
  - Then each object can have unique behavior
- Real world situations can be modeled very well with OOP

Classes

- Making an object from a class is called *instantiation*
  - You work with *instances* of a class
- Here we will demonstrate how to define classes and create instances
- In addition to defining a class to hold information, we can add functionality to the class
- With classes you can do more with less code

Classes

- Learning about OOP will help you see the world as a programmer does
- It will help you understand your code
  - Not just line by line but also the overall concepts
- Knowing the logic behind classes helps to allow you to think more logically

- Classes also make life easier on you as a programmer when you work with increasingly complex challenges

Creating and Using a Class

- You can model almost anything with the use of classes
- For example you can start with a `Dog` class, that represents a dog
- But not just a specific dog **any** dog

Creating and Using a Class

- What do we know about dogs in general?
  - They have a name
  - They have (a) breed(s)
  - They have an age
  - Most also know commands like sit and stay

Creating and Using a Class

- All of that information can be put into a class as they are common attributes
- This class will tell Python how to make an object representing a dog
- After the class is written it can be used to make individual instances
- Each of these instances can represent a specific dog

In [ ]:
```
class Dog:
    """A simple Dog class"""
    def __init__(self, name, age, breed):
        """Initialize the name, age, and breed"""
        self.name = name
        self.age = age
        self.breed = breed
    def sit(self):
        """Simulate a dog sitting"""
        print(f"{self.name.title()} is now sitting")
    def stay(self):
        """Simulate the dog rolling over"""
        print(f"{self.name.title()} is now staying")
```

Creating the Dog Class

- Each instance created from the `Dog` class will store:

    - The name
    - The age
    - The breed
- In addition each instance will have the ability to
    - Sit
    - Stay

Creating the Dog Class

- There is a lot to going, but the structure will be broken down in the remaining portions of the lecture
- You'll notice that the definition is `class Dog:`
    - We begin with the `class` keyword
- By convention class names begin with capitol letters
- There are no parenthesis for class declarations in python
- It is followed by a docstring

The `__init__()` Method

- A function that is part of a class is a *method*
- Everything we have previously learned about functions applies to methods
- The primary difference is how methods and functions are called

The `__init__()` Method

- The `__init__()` method is a special method that Python runs automatically when a new instance is made
- This method has two leading underscores and two trailing underscores
- This is a convention that helps default Python method names from conflicting with ones you make
- The two underscores on each side are needed
- If you use just one on each side the method won't be called automagically

The `__init__()` Method

- The `__init()__` method is defined to have four parameters:
    - self

  - name
  - age
  - breed

The `__init__()` Method

- The `self` parameter is required in the method definition and must come first
- It must be included in the definition because when Python calls this method later, `self` is automatically passed
- Every method call associated with an instance will automatically pass `self` as an argument
- `self` is a reference to the instance itself
  - It gives an individual instance access to the attributes and the methods in the class

The `__init__()` Method

- When an instance of `Dog` is made Python calls the `__init__()` method
- `Dog()` will be passed a name, age, and breed as arguments
  - Self is passed automatically so we don't need to
- Whenever you want to make an instance of `Dog` you will need to pass the appropriate arguments

The `__init__()` Method

- The variables within the `__init__()` method each have the `self` prefix
- Any variable with `self` prefixed to it will be available to every class method
- In addition they can be accessed through the class instance
- The line `self.name = name` takes the value associated with the `name` parameter and assigns it to the variable `name`
- The variable `name` is attached to the instance created

The `__init__()` Method

- Variables that are accessible through instances are called *attributes*
- The `Dog` class also has two methods defined `sit()` and `stay()`
- These methods don't need any additional information to run, so they are only defined

with the `self` parameter
- Any instances made will have access to these methods

The `__init__()` Method

- That means any instances of `Dog` will be able to sit and stay
- These functions don't do a lot, but this concept can extend as needed
- If these were part of a computer game they could make an animated dog sit and stay

Making an Instance from a Class

- Think of a class like a set of instructions
- These are instructions on how an instance should be created
- So how can we represent a specific class instance?

In [ ]:
```
my_dog = Dog("Apollo", 3, 'GSD/Husky')

print(f"My dog's name is {my_dog.name} and is a {my_dog.breed} that is {my_dog.age}
```

```
My dog's name is Apollo and is a GSD/Husky that is 3 years old
```

Making an Instance from a Class

- That code utilizes the `Dog` class from earlier
- We create an instance of `Dog` with:
  - `name` set to be Apollo
  - `age` set as 3
  - `breed` set as GSD/Husky

Making an Instance from a Class

- Note we didn't need to explicitly call `__init__()`
- Instead when python sees `my_dog = Dog("Apollo", 3, 'GSD/Husky')` it calls it for us
- The `__init__()` method then creates an instance represented by this particular dog
- It sets the `name`, `age`, and `breed` attributes
- Python then returns an instance representing the dog

Making an Instance from a Class

- The returned instance is then assigned to the variable `my_dog`
- Naming conventions help, the capitalized `Dog` refers to the class
- And a lowercase like `my_dog` refers to a single instance created from a class

Accessing Attributes

- To access the attributes of an instance the dot notation is used
  - `my_dog.name`
- Dot notation is often used in python
- That syntax demonstrates how to access an attributes value
- Note, there are no parenthesis, those are only for function/method calls

Accessing Attributes

- `my_dog.name` refers to the same attribute referred to by `self.name`
- This same approach can be used with any attribute
- Or, methods!

Calling Methods

- After an instance is created the dot notation can also be used to call methods
- This is done by using the method name followed by parenthesis
- As with other function calls if any arguments are needed they are placed in the parenthesis
- But, even if no arguments are needed the parenthesis still are needed

In [ ]:
```
my_dog.sit()
my_dog.stay()
```

```
Apollo is now sitting
Apollo is now staying
```

Calling Methods

- To call a method give:
  - The name of the instance
  - The dot operator
  - The method you wish to call

- `my_dog.sit()`

Calling Methods

- When Python sees `My_dog.sit()` it looks for the method `.sit()` in the class `Dog`
- If the corresponding method is found, then the code is run
- This syntax is helpful
- When attributes and methods have been given appropriately descriptive names they can be easily used
- It also makes it easier to understand what a block of code is doing

Creating Multiple Instances

- You can create as many instances of a class as you need
- You could create a whole sled team's worth of dogs for example,
- Or really whatever you may want or need

In [ ]:
```
your_dog = Dog("Lassie","Collie", "6")

print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()
print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
```

```
My dog's name is Apollo.
My dog is 3 years old.
Apollo is now sitting

Your dog's name is Lassie.
Your dog is Collie years old.
Lassie is now sitting
```

Creating Multiple Instances

- That example instantiates another dog
- We still have access to the original dog as well
- Both instances function similarly, but they each hold their unique attributes
- Even if you pass in the same attributes to a dog, Python would make a new instance

Working with Classes and Instances

- You can use classes to represent many real-word situations
- Once a class is written, most of what you will work with are the instances
- One of the first tasks being modifying the associated attributes
- Attributes can be modified directly, or through the use of methods

The Car Class

- We can write a new class, this time representing a car
- This class will store information relating to a car
- As well as a method to summarize information

In [ ]:
```
class Car:
    """A simple car representation"""
    def __init__(self, make, model, year):
        """Initialize attributes to describe a car"""
        self.make = make
        self.model = model
        self.year = year
    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name"""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name

new_car = Car ('audi', 'R8', '2023')
print(new_car.get_descriptive_name())
```

```
2023 audi R8
```

The Car Class

- In the car class we define the `__init__()` method with:
  - The `self` parameter
  - The additional parameters `make, model, year`
- `__init__()` once again takes in the parameters and assigns them to corresponding attributes
- These attributes will be associated with instances made from this class
- When a new car is made `make, model, year` all need to be specified

The Car Class

- In addition a method `get_descriptive_name()` is defined
- It puts a cars year, make, and model in a nice format
  - To work with the attributes the `self.` prefix is needed within the method

- Outside of the class a new car can be instantiated and assigned to a variable
- As the class is right now it is relatively static, it can be expanded to include changing attributes

Setting a Default Value for an Attribute

- When an instance is created attributes can be defined without being passed as parameters
- These attributes can be defined in the `__init__()` method
  - They would have a default value assigned within
- Suppose we want to expand car to also have an `odometer_reading`
  - That always starts with `0`

In [ ]:
```
class CarMK2:
    """A simple car representation"""
    def __init__(self, make, model, year):
        """Initialize attributes to describe a car"""
        self.make = make
        self.model = model
        self.year = year
        self.odometer = 0
    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name"""
        long_name = f"{self.year} {self.make} {self.model}".title()
        return long_name
    def read_odometer(self):
        """A function to read the odometer of a car"""
        print(f"This car has {self.odometer} miles on it")

new_car_two = CarMK2 ('audi', 'R8', '2023')
print(new_car_two.get_descriptive_name())
new_car_two.read_odometer()
```

```
2023 Audi R8
This car has 0 miles on it
```

Setting a Default Value for an Attribute

- That time when Python calls the `__init__()` method to create a new instance, it stores the make, model, and year
- But, Python also creates a new attribute `odometer` that it sets to 0
- We also added a new method `read_odometer()`
- But, the odometer will not always be `0` so we need a way to change it

Modifying Attribute Values

- An attributes value can be changed in three different ways:
  - Directly through the instance
  - Through a method
  - Increment through a method

Modifying an Attribute's Value Directly

- The simplest way to modify the value of an attribute is to access the attribute directly
- This can be done using the dot operator and doing an assignment

In [ ]:
```
new_car_two.read_odometer()
new_car_two.odometer = 13
new_car_two.read_odometer()
```

```
This car has 0 miles on it
This car has 13 miles on it
```

Modifying an Attribute's Value Directly

- Dot notation is used to access the car's `odometer_reading` attribute
- it then is able to directly set the value
- This line `new_car_two.odometer = 13` tells python to:
  - Take the instance `new_car_two`
  - Find the attribute `odometer_reading` associated with it
  - Set the value of that attribute to be 13

Modifying an Attribute's Value Through a Method

- It can be helpful to have methods update certain attributes
- Instead of access the attribute directly, you can pass the new value to a method
- The method then directly handles the changing of the variable

In [ ]:
```
class CarMK3:
    """A simple car representation"""
    def __init__(self, make, model, year):
        """Initialize attributes to describe a car"""
        self.make = make
        self.model = model
        self.year = year
        self.odometer = 0
    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name"""
```

```
        long_name = f"{self.year} {self.make} {self.model}".title()
        return long_name
    def read_odometer(self):
        """A function to read the odometer of a car"""
        print(f"This car has {self.odometer} miles on it")
    def update_odometer(self, mileage):
        if mileage >= 0:
            self.odometer += mileage
```

In [ ]:
```
new_car_thr = CarMK3 ('audi', 'R8', '2023')
print(new_car_thr.get_descriptive_name())
new_car_thr.read_odometer()
new_car_thr.update_odometer(13)
new_car_thr.read_odometer()
```

```
2023 Audi R8
This car has 0 miles on it
This car has 13 miles on it
```

Modifying an Attribute's Value Through a Method

- the only modification to `CarMK3` is the addition of `Update_odometer()`
- this method takes in a mileage value and increments the odometer by it
- Using the `new_car_thr` instance we can call `update_odometer` with `13` as an argument
- This sets the odometer reading to 13
- It also ensures we are only incrementing the odometer by positive values

Inheritance

- You don't always have to start from scratch when writing a class
- If the class you're writing is a specialized version of another class you wrote, you can use *inheritance*
- When one class *inherits* from another it takes on the attributes and methods from the first class
- The original class is the *parent* class and the new class is the *child*
- The *child* can inherent any and all the attributes and methods from its parent class, or define new ones

The `__init__()` Method for a Child Class

- When writing a new class based on an existing one, often the parent `__init__()` needs to be called

- This will initialize any attributes that were defined in the parent
- It also makes these attributes available to the child class
- An example could be modeling an electric car, which really is just a specific type of car

The `__init__()` Method for a Child Class

- This means we could have a child class that only has attributes specifically for an electric car
- It would be based on the Car class from earlier
  - Specifically mk3
- We shall name this new class `ElectricCar`

In [ ]:
```
class ElectricCar(CarMK3):
    """Represents aspects of an electric car"""

    def __init__(self,make,model,year):
        super().__init__(make,model,year)

leaf = ElectricCar('nissan','leaf',2023)
print(leaf.get_descriptive_name())
```

```
2023 Nissan Leaf
```

The `__init__()` Method for a Child Class

- This class builds on the `CarMK3` class
- When using inheritance the parent class must be in the current file
- Then a child class is defined, `ElectricCar`
- The name of the parent must be included in parenthesis in the definition of the child class
- The `__init__()` method takes the information required to make a Car

The `__init__()` Method for a Child Class

- The `super()` function is a special function that lets Python call a method from the parent class
- Specifically `super().__init__(make,model,year)` tells it to call the `__init__()` from the parent
- This gives an `ElectricCar` instance all the attributes defined in the parent `__init__()`
- The *super* name comes from a convention of calling the parent class a *superclass* and the

child class a *subclass*

The `__init__()` Method for a Child Class

- To show that inheritance is working correctly by creating an `ElectricCar` instance
  - Then we call the parent's `.get_descriptive_name()` method
  - Which, is not defined in the `ElectricCar` class
- `leaf = ELectricCar('nissan','leaf',2023)` calls the `__init__()` of `ElectricCar` which in turn calls the `__init__()` of the parent class `CarMK3`

Defining Attributes and Methods for the Child Class

- Once you have a child class defined that inherits from a parent class, new attributes and methods can be added
- So, for the electric cars concept we can add something unique to electric cars
- How about battery size?

In [ ]:
```
class ElectricCarMK2(CarMK3):
    """Represents aspects of an electric car"""

    def __init__(self,make,model,year):
        super().__init__(make, model, year)
        self.battery_size = 40

    def describe_battery(self):
        """Describes the battery associated with this car"""
        print(f"This car has a {self.battery_size}-KWh battery")

leaf = ElectricCarMK2('nissan','leaf',2023)
print(leaf.describe_battery())
```

```
This car has a 40-KWh battery
None
```

Defining Attributes and Methods for the Child Class

- A new attribute `batter_size` is added, with an initial value of 40
- This attribute will only be associated with instances created from `ElectricCarMK2`
- It won't be associated with any instances of `CarMK3`
- We also added a method just for the subclass

Defining Attributes and Methods for the Child Class

- There is no limit to how much you can specialize the `ElectricCar` class
- You can add as many attributes and methods as are needed
- An attribute that could belong to any kind of car, not just an electric one, should go in the parent class
- Then, anyone using `Car` will have that functionality available
- And the `ElectricCar` will only contain things for electric cars

Overriding Methods from then Parent Class

- You can override any method from the parent class that doesn't fit the child class
- To do this, define a method in the child class with the same name as one in the parent class
- Python will then disregard the parent class method and only pay attention to the child's method

In [ ]:
```
class Adult:
    def __init__(self):
        pass
    def vote(self):
        print("the adult voted")
    def sayHi():
        print("Hi")
class Child(Adult):
    def __init__(self):
        pass
    def vote():
        print("Children can't vote")

person = Child

person.sayHi()
person.vote()
```

```
Hi
Children can't vote
```

Overriding Methods from then Parent Class

- If someone tries to call `.vote()` on the child they will see a different message than if the parent hadn't be inherited
- Python will find the vote method in the `Child` class and run it instead
- When using inheritance make child classes retain what they need
- Overwrite what they don't

Instances as Attributes

- When modeling something from the *real-world* in code you may end up adding more and more detail to a class
- You'll find that classes may have a growing list of attributes and methods
- In these situations you might recognize that part of one class can be written as a separate class
- Large classes can be broken into smaller classes that work together
- This is called *composition*

Instances as Attributes

- For example we could add more details to the `ElectricCar` class outlining the battery
- Or, we could instead make a separate `Battery` class that deals only with the battery
- Then, an instance of battery can be used within the `ElectricCar` class

In [ ]:
```
class Battery:
    def __init__(self,battery_size = 40):
        self.battery_size = battery_size

    def describe_battery(self):
        """Describes the battery associated with this car"""
        print(f"This car has a {self.battery_size}-KWh battery")

class ElectricCarMK3(CarMK3):
    """Represents aspects of an electric car"""

    def __init__(self,make,model,year):
        super().__init__(make, model, year)
        self.battery = Battery()

leaf = ElectricCarMK3('nissan','leaf',2023)
leaf.battery.describe_battery()
```

```
This car has a 40-KWh battery
```

Instances as Attributes

- There a new class `Battery` was defined, which doesn't inherit from any other class
- The `__init__()` method has one parameter, `battery_size` , in addition to `self`
- This is an optional parameter, as it has a default value associated with it
- `describe_battery` has moved to this class as well

Instances as Attributes

- `leaf.battery.describe_battery()` looks at the instance leaf, then finds teh attribute `battery` within, which is another class instance
- While this may look like extra work, it ultimately allows `ElectricCar` to remain (relatively) uncluttered

Modeling Real-World Objects

- As you begin to model more complicated things, like electric cars, there is an interesting question?
  - Is the range of a car a property of the car or the battery?
- And the answer is, it depends and is a bit more nuanced
- Describing a single kind of car, maybe range belongs with the battery
- Describing a whole line-up of cars? Maybe range belongs in `ElectricCar`, as different cars' weight will change the range
- Or, you could keep it with battery and pass the parameter from `ElectricCar` to `Battery`

Modeling Real-World Objects

- This hopefully brings you to an interesting point in your growth as a programmer
- When you wrestle with questions like these you're thinking at a high logical level
  - Rather than purely syntax focused
- You begin to think about not Python itself, but how to use code to represent the "real world"
- When your each this point you may quickly realize there aren't necessarily right and wrong answers

Python Standard Library

- The *Python Standard Library* is a set of modules included with every Python installation
- These modules are things other programmers have written,a dn you can use them to expand your code's functionality
- You can use any function or class in the standard library by including an `import` statement

Python Standard Library

- To begin we will look at the `random` module
- It includes a helpful function `randint()` among others
- This function takes two integers and returns a randomly selected integer between those numbers, inclusive
- `random` also allows you to randomly pick an element out of a list or tuple with `choice()`
- `random` shouldn't be used for security related projects

In [ ]:
```
from random import randint
from random import choice
print(randint(1,5))
tree_species = [
    "Oak",
    "Maple",
    "Pine",
    "Birch",
    "Willow",
    "Cedar",
    "Fir",
    "Redwood",
    "Ash",
    "Elm"
]

print(choice(tree_species))
```

```
4
Fir
```

Styling Classes

- There are a few styling issues related to classes
- Class names should be written in *CamelCase*
  - The first letter of every word should be capitalized
  - Underscores aren't used
- Instance and module names should be written in lowercase, with underscores between words

Styling Classes

- Every class should have a docstring immediately following the class definition
- The docstring should be a brief description of what the class does

- The same formatting conventions should be followed as writing function docstrings
- Blank lines can be sued to organize code, but not excessively
- When importing things the imports should occur at the top of the file