Object Oriented Programming and Classes

1. Creating and Using a Class
2. Inheritance
3. Encapsulation
4. Polymorphism

In the first lecture, we mentioned that everything in `Python` is an object, so we've been using objects constantly. ***Object Oriented Programming (OOP)*** is a programming paradigm that allows you to group variables (data/attributes) and functions (methods) into new custom data types called ***classes***, from which you can create ***objects (instance)***.

In the first lecture, we mentioned that everything in `Python` is an object, so we've been using objects constantly. ***Object Oriented Programming (OOP)*** is a programming paradigm that allows you to group variables (data/attributes) and functions (methods) into new custom data types called ***classes***, from which you can create ***objects (instance)***.

When you write a class, you define the general behavior that a whole category of objects can have. When you create individual objects from the class, each object is automatically equipped with the general behavior; you can then give each object whatever unique traits you desire. Making an object from a class is called ***instantiation***, and you work with instances of a class.

In the first lecture, we mentioned that everything in `Python` is an object, so we've been using objects constantly. ***Object Oriented Programming (OOP)*** is a programming paradigm that allows you to group variables (data/attributes) and functions (methods) into new custom data types called ***classes***, from which you can create ***objects (instance)***.

When you write a class, you define the general behavior that a whole category of objects can have. When you create individual objects from the class, each object is automatically equipped with the general behavior; you can then give each object whatever unique traits you desire. Making an object from a class is called ***instantiation***, and you work with instances of a class.

You've already used lots of classes created by other people (`int`, `str`, `float`, `list`, `dict`, etc.); these are designed to represent simple pieces of information, such as the cost of an apple, the name of a student. What if you want to represent something more complex? **In this chapter, you'll learn how to create your custom classes.**

Creating and Using a Class

You can model almost anything using classes. Let's start by writing a simple class, `Dog`, that represents a dog — not one dog in particular, but any dog.

You can model almost anything using classes. Let's start by writing a simple class, `Dog`, that represents a dog — not one dog in particular, but any dog.

What do we know about most pet dogs? Well, they all have a `name` and an `age`. We also know that most dogs `sit` and `roll over`.

You can model almost anything using classes. Let's start by writing a simple class, `Dog`, that represents a dog — not one dog in particular, but any dog.

What do we know about most pet dogs? Well, they all have a `name` and an `age`. We also know that most dogs `sit` and `roll over`.

Those two pieces of information (`name` and `age`) and those two behaviors (`sit` and `roll over`) will go in our `Dog` class because they're common to most dogs.

Creating the `Dog` Class

Each instance created from the `Dog` class will store a `name` and an `age`, and we'll give each dog the ability to `sit()` and `roll_over()`:

Each instance created from the `Dog` class will store a `name` and an `age`, and we'll give each dog the ability to `sit()` and `roll_over()`:

In [2]:
```
class Dog:
    """A simple attempt to model a dog."""
    def __init__(self, name, age):
        """Initialize name and age attributes."""
        self.name = name
        self.age = age
    def sit(self):
        """Simulate a dog sitting in response to a command."""
        print(f"{self.name} is now sitting.")
    def roll_over(self):
        """Simulate rolling over in response to a command."""
        print(f"{self.name} rolled over!")
```

Each instance created from the `Dog` class will store a `name` and an `age`, and we'll give each dog the ability to `sit()` and `roll_over()`:

In [2]:
```
class Dog:
    """A simple attempt to model a dog."""
    def __init__(self, name, age):
        """Initialize name and age attributes."""
        self.name = name
        self.age = age
    def sit(self):
        """Simulate a dog sitting in response to a command."""
        print(f"{self.name} is now sitting.")
    def roll_over(self):
        """Simulate rolling over in response to a command."""
        print(f"{self.name} rolled over!")
```

We first define a class called `Dog` with the `class` keyword. By convention, capitalized names refer to classes in Python. We then write a docstring describing what this class does.

The `__init__()` Method

1. A function that's part of a class is a ***method***. The `__init__()` method is a special method that `Python` runs whenever we create a new instance based on the `Dog` class. This method has two leading underscores and two trailing underscores, a convention that helps prevent `Python`'s default method names from conflicting with your method names.

1. A function that's part of a class is a ***method***. The `__init__()` method is a special method that `Python` runs whenever we create a new instance based on the `Dog` class. This method has two leading underscores and two trailing underscores, a convention that helps prevent `Python`'s default method names from conflicting with your method names.

2. The `self` parameter is required in the method definition, and it must come first before any other parameters. It must be included in the definition because when `Python` calls this method later, the method call will automatically pass the object to the `self`. The two variables defined in the body of the `__init__()` method each have the prefix `self`. Any variable prefixed with `self` is available to every method in the class, and we'll also be able to access these variables through any instance created from the class, which can be different between instances.

3. The line `self.name = name` takes the value associated with the parameter `name` and assigns it to the instance variable `name`, which is then attached to the instance being created. The same process happens with `self.age = age`. Variables that are accessible through instances like this are called (instance) ***attributes***.

3. The line `self.name = name` takes the value associated with the parameter `name` and assigns it to the instance variable `name`, which is then attached to the instance being created. The same process happens with `self.age = age`. Variables that are accessible through instances like this are called (instance) ***attributes***.

4. The `Dog` class has two other methods defined: `sit()` and `roll_over()`. Because these methods don't need additional information to run, we define them to have one parameter, `self`, so that the instances we create later will have access to these methods. In other words, they'll be able to sit and roll over.

Making an Instance from a Class

When we make an instance of `Dog`, `Python` will call the `__init__()` method from the `Dog` class. We'll pass `Dog()` a `name` and an `age` as arguments; `self` is passed automatically, so we don't need to pass it.

When we make an instance of `Dog`, `Python` will call the `__init__()` method from the `Dog` class. We'll pass `Dog()` a `name` and an `age` as arguments; `self` is passed automatically, so we don't need to pass it.

In [3]:
```python
my_dog = Dog('Willie', 6) # This is known as constructor expression

# You can access their instance attributes using dot notation:
print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
```

```
My dog's name is Willie.
My dog is 6 years old.
```

When we make an instance of `Dog`, `Python` will call the `__init__()` method from the `Dog` class. We'll pass `Dog()` a `name` and an `age` as arguments; `self` is passed automatically, so we don't need to pass it.

In [3]:

```
my_dog = Dog('Willie', 6) # This is known as constructor expression

# You can access their instance attributes using dot notation:
print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
```

```
My dog's name is Willie.
My dog is 6 years old.
```

Here, we tell `Python` to create a dog whose name is 'Willie' and whose age is 6, which is known as ***constructor expression***. When `Python` reads this line, it calls the `__init__()` method in `Dog` with the arguments 'Willie' and 6.

The `__init__()` method creates an instance representing this particular dog and sets the `name` and `age` attributes using the values we provided. `Python` then returns an instance representing this dog.

The `__init__()` method creates an instance representing this particular dog and sets the `name` and `age` attributes using the values we provided. `Python` then returns an instance representing this dog.

We assign that instance to the variable `my_dog`. To access the attributes of an instance, you use **dot notation**. After we create an instance from the class `Dog`, we can use dot notation to call any method defined in `Dog`.

The `__init__()` method creates an instance representing this particular dog and sets the `name` and `age` attributes using the values we provided. `Python` then returns an instance representing this dog.

We assign that instance to the variable `my_dog`. To access the attributes of an instance, you use **dot notation**. After we create an instance from the class `Dog`, we can use dot notation to call any method defined in `Dog`.

In [6]:
```
print(my_dog.name)
print(my_dog.age)
my_dog.sit()
my_dog.roll_over()
```

```
Willie
6
Willie is now sitting.
Willie rolled over!
```

Creating Multiple Instances

Once you create a class, you can use it to create different objects.

Once you create a class, you can use it to create different objects.

In [7]:
```python
my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)
# Even though my_dog and your_dog are both instances of the Dog class, they r
print(my_dog == your_dog)

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()
```

```
False
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

Working with Classes and Instances

The `Car` Class

Here, we create another class that `Car` with four instance attributes:

Here, we create another class that `Car` with four instance attributes:

In [11]:
```python
class Car:
    """A simple attempt to represent a car."""
    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"Created in {self.year}, {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")
```

Here, we create another class that `Car` with four instance attributes:

In [11]:
```
class Car:
    """A simple attempt to represent a car."""
    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"Created in {self.year}, {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")
```

In the `Car` class, we define the `__init__()` method with the `self` parameter first, just like we did with the `Dog` class. We also give it other parameters: `make`, `model`, `year` and `odometer_reading`. The `__init__()` method takes in these parameters and assigns them to the attributes associated with instances made from this class.

Note that when an instance is created, attributes can be defined without being passed in as parameters. Since attributes can be defined in the `__init__()` method, which assigns a default value. In the above example, an attribute called `odometer_reading` always starts with a value of 0.

Note that when an instance is created, attributes can be defined without being passed in as parameters. Since attributes can be defined in the `__init__()` method, which assigns a default value. In the above example, an attribute called `odometer_reading` always starts with a value of 0.

Outside of the class, we make an instance from the `Car` class and assign it to the variable `my_new_car`. Then we call `get_descriptive_name()` to show what kind of car we have! Our car starts with a mileage of 0:

In [12]:
```
my_new_car = Car('audi', 'a4', 2023)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

```
Created In 2023, Audi A4
This car has 0 miles on it.
```

Note that when an instance is created, attributes can be defined without being passed in as parameters. Since attributes can be defined in the `__init__()` method, which assigns a default value. In the above example, an attribute called `odometer_reading` always starts with a value of 0.

Outside of the class, we make an instance from the `Car` class and assign it to the variable `my_new_car`. Then we call `get_descriptive_name()` to show what kind of car we have! Our car starts with a mileage of 0:

In [12]:
```
my_new_car = Car('audi', 'a4', 2023)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

```
Created In 2023, Audi A4
This car has 0 miles on it.
```

Not many cars are sold with exactly 0 miles on the odometer, so we need a way to change the value of this attribute.

In [4]:

```
display_quiz(path+"create_class.json", max_width=800)
```

What is printed by the following code?

```
class Car:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year
    def get_info(self):
        return f"{self.brand} - {self.year}"

my_car = Car("Toyota", 2020)
print(my_car.get_info())
```

Car - 2020

Error: __init__ missing required arguments

Toyota - 2020

Toyota

Modifying Attribute Values

You can change an attribute's value in different ways: you can change the value directly through an instance, set the value through a method, or increment the value (add a certain amount to it) through a method. The simplest way to modify the value of an attribute is to access the attribute directly through an instance.

You can change an attribute's value in different ways: you can change the value directly through an instance, set the value through a method, or increment the value (add a certain amount to it) through a method. The simplest way to modify the value of an attribute is to access the attribute directly through an instance.

In [13]:

```
my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

```
This car has 23 miles on it.
```

You can change an attribute's value in different ways: you can change the value directly through an instance, set the value through a method, or increment the value (add a certain amount to it) through a method. The simplest way to modify the value of an attribute is to access the attribute directly through an instance.

In [13]:
```
my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

```
This car has 23 miles on it.
```

It can be helpful to have methods that update certain attributes for you. Instead of accessing the attribute directly, you pass the new value to a method that handles the updating internally.

In [14]:

```
class Car:
    def __init__(self, make, model, year):
        self.make, self.model, self.year, self.odometer_reading = make, model
    def get_descriptive_name(self):
        long_name = f"Created in {self.year}, {self.make} {self.model}"
        return long_name.title()
    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    ## We add these there methods!
    def update_odometer(self, mileage):
        """
        1. Set the odometer reading to the given value. Reject the change if
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """2 . Add the given amount to the odometer reading."""
        self.odometer_reading += miles

    def fill_gas_tank(self):
        """3. Filling the gas tank."""
        print("The gas tank is now full!")
```

We also define the new method `increment_odometer()` takes in a number of miles and adds this value to `self.odometer_reading`. Finally, a method `fill_gas_tank()` is also added to the class.

We also define the new method `increment_odometer()` takes in a number of miles and adds this value to `self.odometer_reading`. Finally, a method `fill_gas_tank()` is also added to the class.

In [15]:
```
my_new_car = Car('audi', 'a4', 2023)
print(my_new_car.get_descriptive_name())

my_new_car.update_odometer(23)
my_new_car.read_odometer()

my_new_car.fill_gas_tank()
my_new_car.increment_odometer(100)
my_new_car.read_odometer()
```

```
Created In 2023, Audi A4
This car has 23 miles on it.
The gas tank is now full!
This car has 123 miles on it.
```

We also define the new method `increment_odometer()` takes in a number of miles and adds this value to `self.odometer_reading`. Finally, a method `fill_gas_tank()` is also added to the class.

In [15]:
```
my_new_car = Car('audi', 'a4', 2023)
print(my_new_car.get_descriptive_name())

my_new_car.update_odometer(23)
my_new_car.read_odometer()

my_new_car.fill_gas_tank()
my_new_car.increment_odometer(100)
my_new_car.read_odometer()
```

```
Created In 2023, Audi A4
This car has 23 miles on it.
The gas tank is now full!
This car has 123 miles on it.
```

You can use methods like this to control how users use your program by including additional logic!

`__repr__` and `__str__` method

Notice that when you evaluate the `my_new_car`, it will return a message that returns the address of the object:

Notice that when you evaluate the `my_new_car`, it will return a message that returns the address of the object:

In [16]:
```
my_new_car
```

Out[16]:
```
<__main__.Car at 0x2278a2fba00>
```

Notice that when you evaluate the `my_new_car`, it will return a message that returns the address of the object:

In [16]:
```
my_new_car
```

Out[16]:
```
<__main__.Car at 0x2278a2fba00>
```

When writing your classes, it's a good idea to have a method that returns a string containing useful information about a class instance. You can change this behavior by adding a special function `__repr__`.

In [17]:

```python
class Car:
    def __init__(self, make, model, year):
        self.make, self.model, self.year, self.odometer_reading = make, model
    def get_descriptive_name(self):
        long_name = f"Created in {self.year} {self.make} {self.model}"
        return long_name.title()
    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")
    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")
    def increment_odometer(self, miles):
        self.odometer_reading += miles
    def fill_gas_tank(self):
        print("The gas tank is now full!")

    ## We add these two methods!
    def __repr__(self):
        return f'Car(make={self.make}, model={self.model}, year={self.year})'

    def __str__(self):
        return self.get_descriptive_name()
```

In [18]:

```
my_new_car = Car('audi', 'a4', 2023)
my_new_car
```

Out[18]:

```
Car(make=audi, model=a4, year=2023)
```

In [18]:
```python
my_new_car = Car('audi', 'a4', 2023)
my_new_car
```

Out[18]:
```
Car(make=audi, model=a4, year=2023)
```

The `Python` documentation indicates that `__repr__` returns the "official" string representation of the object (It is also return when you call the built-in function `repr()`). We also define the `__str__` special method that is used to replace the behavior of `__repr__` in some cases. This method is called when you convert an object to a string with the built-in function `str()`, such as when you print an object or call `str()` explicitly.

In [18]:
```python
my_new_car = Car('audi', 'a4', 2023)
my_new_car
```

Out[18]:
```
Car(make=audi, model=a4, year=2023)
```

The `Python` documentation indicates that `__repr__` returns the "official" string representation of the object (It is also return when you call the built-in function `repr()`). We also define the `__str__` special method that is used to replace the behavior of `__repr__` in some cases. This method is called when you convert an object to a string with the built-in function `str()`, such as when you print an object or call `str()` explicitly.

In [19]:
```python
print(my_new_car)
str(my_new_car)
```

```
Created In 2023 Audi A4
```

Out[19]:
```
'Created In 2023 Audi A4'
```

In [18]:
```python
my_new_car = Car('audi', 'a4', 2023)
my_new_car
```

Out[18]:
```
Car(make=audi, model=a4, year=2023)
```

The `Python` documentation indicates that `__repr__` returns the "official" string representation of the object (It is also return when you call the built-in function `repr()`). We also define the `__str__` special method that is used to replace the behavior of `__repr__` in some cases. This method is called when you convert an object to a string with the built-in function `str()`, such as when you print an object or call `str()` explicitly.

In [19]:
```python
print(my_new_car)
str(my_new_car)
```

```
Created In 2023 Audi A4
```

Out[19]:
```
'Created In 2023 Audi A4'
```

Special methods like `__init__()`, `__str__()` and `__repr__` are called ***dunder methods*** (Double UNDERscore). There are many dunder methods that you can use to customize classes.

In [5]:
```python
display_quiz(path+"string_class.json", max_width=800)
```

What is printed by the following code?

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __repr__(self):
        return f"Person('{self.name}', {self.age})"
    def __str__(self):
        return f"{self.name} ({self.age} years old)"

p = Person("Bob", 25)
print(repr(p))
print(p)
```

```
Bob (25 years ol
d)
Person('Bob', 25)
```

```
Person('Bob', 25)
Bob (25 years ol
d)
```

```
Person('Bob', 2
5)
Person('Bob', 2
5)
```

```
Bob (25 years ol
d)
Bob (25 years ol
d)
```

> Exercise 1: Create a `Pokemon` class with three instance attributes: `name`, which stores the name of the Pokemon as a string, `type` which stores the type of Pokemon (e.g., ”Fire”, ”Water”, ”Grass”, etc.) as string and `total_species` as an integer. In addition, add the `__str__()` method to the class so that it can print out meaningful information as follows:

```
Pikachu (Electric, total species 320)
```

Complete the following class and execute the code cell to see which six Pokemon you get.

In [ ]:

```
import random
import json
import time
import requests

def slow_print(text, delay=0.05):
    for char in text:
        print(char, end='', flush=True)
        time.sleep(delay)
    print()

class Pokemon:
    # Your code here
    def __init__(,_,_,_):
        _______

    # Your code here
    def __str__(_):
        return f"{_} ({_}, total specis {_})"
```

In [ ]:

```python
## 1. Download the data
url = 'https://raw.githubusercontent.com/fanzeyi/pokemon.json/master/pokedex.
response = requests.get(url)

if response.status_code == 200:
    with open('pokedex.json', 'w', encoding = "utf-8") as f:
        f.write(response.text)
else:
    print(f"Failed to download the file. Status code: {response.status_code}"

with open('Pokedex.json', 'r' , encoding = "utf-8") as file:
    pokemon_data = json.load(file)

## 2. Randomly pick 6 pokemon

random.shuffle(pokemon_data)
picks = pokemon_data[:6]
```

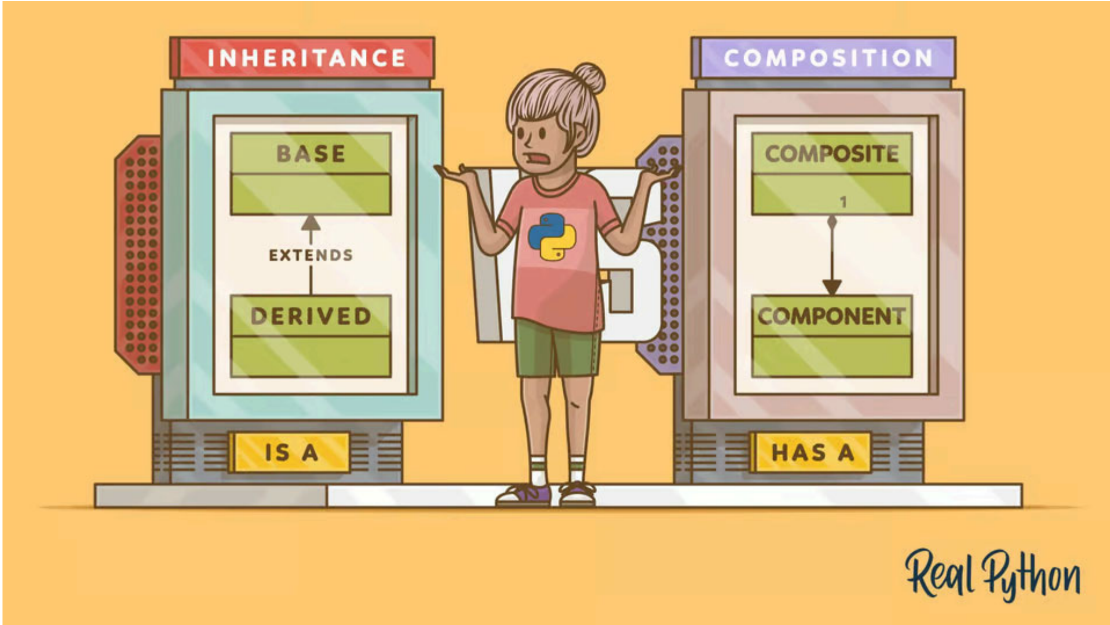
In [ ]:
```
## 3. Print the pokemon you got!
print("The pokemon you got are: ")
for i in range(6):
    pokemon = Pokemon(picks[i]['name']['english'], picks[i]['type'][0], sum([
    slow_print(str(pokemon))
```

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*** which is called **" *is a*"** releationship.

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*** which is called ***" is a"*** releationship.

When one class inherits from another, it takes on the attributes and methods of the first class. The original class is called the ***parent class (Base)***, and the new class is the ***child class (Derived)***. The child class can inherit any or all of the attributes and methods of its parent class, but it's also free to define new attributes and methods of its own.



source: https://realpython.com/inheritance-composition-python/

The `__init__()` method for a Child Class

When writing a new class based on an existing class, we will often want to call the `__init__()` method from the parent class. This will initialize any attributes that were defined in the parent `__init__()` method and make them available in the child class.

When writing a new class based on an existing class, we will often want to call the `__init__()` method from the parent class. This will initialize any attributes that were defined in the parent `__init__()` method and make them available in the child class.

As an example, let's model an electric car. An electric car is just a specific kind of car, so we can base our new `ElectricCar` class on the `Car` class we wrote about earlier. Then we'll only have to write code for the attributes and behaviors specific to electric cars.

When writing a new class based on an existing class, we will often want to call the `__init__()` method from the parent class. This will initialize any attributes that were defined in the parent `__init__()` method and make them available in the child class.

As an example, let's model an electric car. An electric car is just a specific kind of car, so we can base our new `ElectricCar` class on the `Car` class we wrote about earlier. Then we'll only have to write code for the attributes and behaviors specific to electric cars.

In [22]:
```
class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""
    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year) # Call the constructor of parent
        self.battery_size = 40

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")
```

1. When you create a child class, the parent class must be part of the current file and appear before the child class. We then define the child class, `ElectricCar`. The name of the parent class must be included in parentheses in the definition of a child class.

1. When you create a child class, the parent class must be part of the current file and appear before the child class. We then define the child class, `ElectricCar`. The name of the parent class must be included in parentheses in the definition of a child class.

2. The `__init__()` method takes in the information required to make a `Car` instance. **The `super()` function is a special function that allows you to call a method from the parent class.** This line tells Python to call the `__init__()` method from `Car`. The name `super` comes from a convention of calling the parent class a ***superclass (base class)*** and the child class a ***subclass (derived class)***.

1. When you create a child class, the parent class must be part of the current file and appear before the child class. We then define the child class, `ElectricCar`. The name of the parent class must be included in parentheses in the definition of a child class.

2. The `__init__()` method takes in the information required to make a `Car` instance. **The `super()` function is a special function that allows you to call a method from the parent class.** This line tells Python to call the `__init__()` method from `Car`. The name `super` comes from a convention of calling the parent class a ***superclass (base class)*** and the child class a ***subclass (derived class)***.

3. We also add a new attribute specific to electric cars (a `battery`) and a method to report on this attribute. We'll store the battery size and write a method that prints a description of the battery. This attribute/method will be associated with all instances created from the `ElectricCar` class but won't be associated with any instances of `Car`.

We make an instance of the `ElectricCar` class and assign it to `my_leaf`.

We make an instance of the `ElectricCar` class and assign it to `my_leaf`.

In [23]:
```
my_leaf = ElectricCar('nissan', 'leaf', 2023)
print(my_leaf.get_descriptive_name())
my_leaf.describe_battery()
```

```
Created In 2023 Nissan Leaf
This car has a 40-kWh battery.
```

When we need to know the type of an object, we can pass the object to the built-in `type()` function. But if we're doing a type check of an object, it's a better idea to use the more flexible `isinstance()` built-in function. **The `isinstance()` function will return `True` if the object is of the given class or a subclass of the given class.**

When we need to know the type of an object, we can pass the object to the built-in `type()` function. But if we're doing a type check of an object, it's a better idea to use the more flexible `isinstance()` built-in function. **The `isinstance()` function will return `True` if the object is of the given class or a subclass of the given class.**

In [24]:
```
type(my_leaf)
```

Out[24]:
```
__main__.ElectricCar
```

When we need to know the type of an object, we can pass the object to the built-in `type()` function. But if we're doing a type check of an object, it's a better idea to use the more flexible `isinstance()` built-in function. **The `isinstance()` function will return `True` if the object is of the given class or a subclass of the given class.**

In [24]:
```
type(my_leaf)
```

Out[24]:
```
__main__.ElectricCar
```

In [25]:
```
isinstance(my_leaf, ElectricCar)
```

Out[25]:
```
True
```

When we need to know the type of an object, we can pass the object to the built-in `type()` function. But if we're doing a type check of an object, it's a better idea to use the more flexible `isinstance()` built-in function. **The `isinstance()` function will return `True` if the object is of the given class or a subclass of the given class.**

In [24]:
```
type(my_leaf)
```

Out[24]:
```
__main__.ElectricCar
```

In [25]:
```
isinstance(my_leaf, ElectricCar)
```

Out[25]:
```
True
```

In [26]:
```
isinstance(my_leaf, Car)
```

Out[26]:
```
True
```

Overriding Methods from the Parent Class

You can ***override*** any method from the parent class that doesn't fit what you're trying to model with the child class. To do this, you define a method in the child class **with the same name as the method you want to override in the parent class.**

You can ***override*** any method from the parent class that doesn't fit what you're trying to model with the child class. To do this, you define a method in the child class **with the same name as the method you want to override in the parent class.**

Say the class `Car` had a method called `fill_gas_tank()`. This method is meaningless for an all-electric vehicle, so you might want to override this method. Here's one way to do that:

In [28]:

```
class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""
    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
        self.battery_size = 40

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    ## We overide the method here
    def fill_gas_tank(self):
        """Electric cars don't have gas tanks."""
        print("This car doesn't have a gas tank!")
```

Now if someone tries to call `fill_gas_tank()` with an electric car, `Python` will ignore the method `fill_gas_tank()` in `Car` and run this code instead.

Now if someone tries to call `fill_gas_tank()` with an electric car, `Python` will ignore the method `fill_gas_tank()` in `Car` and run this code instead.

In [29]:
```
my_leaf = ElectricCar('nissan', 'leaf', 2023)
my_leaf.fill_gas_tank()
```

```
This car doesn't have a gas tank!
```

Now if someone tries to call `fill_gas_tank()` with an electric car, `Python` will ignore the method `fill_gas_tank()` in `Car` and run this code instead.

In [29]:
```
my_leaf = ElectricCar('nissan', 'leaf', 2023)
my_leaf.fill_gas_tank()
```

```
This car doesn't have a gas tank!
```

When you use inheritance, you can make your child classes retain what you need and override anything you don't need from the parent class!

Use `composition` to organize the code

When modeling something from the real world in code, you may add more detail to a class. You'll find that you have a growing list of attributes and methods and that your files are becoming lengthy.

When modeling something from the real world in code, you may add more detail to a class. You'll find that you have a growing list of attributes and methods and that your files are becoming lengthy.

In these situations, you might recognize that part of one class can be written **as a separate class**. You can break your large class into smaller classes that work together; this approach is called ***composition***, which is sometimes referred to as the ***has a*** releationship.

When modeling something from the real world in code, you may add more detail to a class. You'll find that you have a growing list of attributes and methods and that your files are becoming lengthy.

In these situations, you might recognize that part of one class can be written **as a separate class**. You can break your large class into smaller classes that work together; this approach is called ***composition***, which is sometimes referred to as the ***has a*** releationship.

For example, if we continue adding detail to the `ElectricCar` class, we might notice that we're adding many attributes and methods specific to the car's battery. When we see this happening, we can stop and move those attributes and methods to a separate class called `Battery`. Then we can use a `Battery` instance as an attribute in the `ElectricCar` class.

In [31]:

```python
class Battery:
    """A simple attempt to model a battery for an electric car."""
    def __init__(self, battery_size=40):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 40:
            range = 150 # Class attributes
        elif self.battery_size == 65:
            range = 225
        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""
    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
        self.battery = Battery()
```

We define a new class called `Battery` that doesn't inherit from any other class. The `__init__()` method has one parameter, `battery_size`, in addition to `self`. This optional parameter sets the battery's size to 40 if no value is provided. The method `describe_battery()` has been moved to this class as well. A new method, `get_range()`, performs some simple analysis and is also added.

We define a new class called `Battery` that doesn't inherit from any other class. The `__init__()` method has one parameter, `battery_size`, in addition to `self`. This optional parameter sets the battery's size to 40 if no value is provided. The method `describe_battery()` has been moved to this class as well. A new method, `get_range()`, performs some simple analysis and is also added.

In the `ElectricCar` class, we now add an attribute called `self.battery`, tells `Python` to create a new instance of `Battery` (with a default size of 40) and assign that instance to the attribute `self.battery`.

We define a new class called `Battery` that doesn't inherit from any other class. The `__init__()` method has one parameter, `battery_size`, in addition to `self`. This optional parameter sets the battery's size to 40 if no value is provided. The method `describe_battery()` has been moved to this class as well. A new method, `get_range()`, performs some simple analysis and is also added.

In the `ElectricCar` class, we now add an attribute called `self.battery`, tells `Python` to create a new instance of `Battery` (with a default size of 40) and assign that instance to the attribute `self.battery`.

Any `ElectricCar` instance will now have a `Battery` instance created automatically. When we want to describe the battery, we need to work through the car's battery attribute:

In [32]:

```
my_leaf = ElectricCar('nissan', 'leaf', 2023)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()
my_leaf.battery.get_range()
```

```
Created In 2023 Nissan Leaf
This car has a 40-kWh battery.
This car can go about 150 miles on a full charge.
```

In [ ]:
```
display_quiz(path+"inheritance.json", max_width=800)
```

Encapsulation - Attributes for data access

Most object-oriented programming languages enable you to ***encapsulate*** (or hide) an object's data from the code. Such data in these languages are said to be ***private data***.

Most object-oriented programming languages enable you to ***encapsulate*** (or hide) an object's data from the code. Such data in these languages are said to be ***private data***.

`Python` does not have private data. Instead, you use **naming conventions** to design classes that encourage correct use.

Most object-oriented programming languages enable you to ***encapsulate*** (or hide) an object's data from the code. Such data in these languages are said to be ***private data***.

`Python` does not have private data. Instead, you use **naming conventions** to design classes that encourage correct use.

By convention, **Python programmers know that any attribute name beginning with an underscore (`_`) is for a class's internal use only.** Code should use the class's methods to interact with each object's internal-use data attributes. Attributes whose identifiers do not begin with an underscore (`_`) are considered publicly accessible.

Let's develop a `Time` class that stores the time in 24-hour clock format with `hours` in the range 0–23 and `minutes` and `seconds` each in the range 0–59:

Let's develop a `Time` class that stores the time in 24-hour clock format with `hours` in the range 0–23 and `minutes` and `seconds` each in the range 0–59:

```
class Time:
    """Class Time with read-write attributes."""
    def __init__(self, hour=0, minute=0, second=0):
        """Initialize each attribute."""
        self.set_hour(hour)      # 0-23, note that this line calls the
setter method hour
        self.set_minute(minute)  # 0-59, note that this line calls the
setter method minute
        self.set_second(second)  # 0-59, note that this line calls the
setter method second
    #getter
    def get_hour(self):
        """Return the hour."""
        print("getter is called")
        return self._hour # Private data
    #setter
    def set_hour(self, hour):
        """Set the hour."""
        print("setter is called")
        if not (0 <= hour < 24):
            raise ValueError(f'Hour ({hour}) must be 0-23')

        self._hour = hour
```

```python
#getter
    def get_minute(self):
        """Return the minute."""
        return self._minute # Private data
    #setter
    def set_minute(self, minute):
        """Set the minute."""
        if not (0 <= minute < 60):
            raise ValueError(f'Minute ({minute}) must be 0-59')

        self._minute = minute
    #getter
    def get_second(self):
        """Return the second."""
        return self._second # Private data
    #setter
    def set_second(self, second):
        """Set the second."""
        if not (0 <= second < 60):
            raise ValueError(f'Second ({second}) must be 0-59')

        self._second = second
```

1. Class `Time`'s `__init__` method specifies `hour`, `minute` and `second` parameters, each with a default argument of 0. The statements containing `self.set_hour()`, `self.set_minute()` and `self.set_second()` call methods that implement the class's ***setter***. Those methods then create attributes named `_hour`, `_minute` and `_second` that is meant for use only inside the class!

1. Class `Time`'s `__init__` method specifies `hour`, `minute` and `second` parameters, each with a default argument of 0. The statements containing `self.set_hour()`, `self.set_minute()` and `self.set_second()` call methods that implement the class's ***setter***. Those methods then create attributes named `_hour`, `_minute` and `_second` that is meant for use only inside the class!

2. Lines 10–21 define methods that manipulate a data attribute named `_hour`. The single-leading-underscore (_) naming convention indicates that we should not access `_hour` directly. We define a ***getter*** method which gets (that is, returns) a data attribute's value and a setter method, which sets a data attribute's value.

1. Class `Time`'s `__init__` method specifies `hour`, `minute` and `second` parameters, each with a default argument of 0. The statements containing `self.set_hour()`, `self.set_minute()` and `self.set_second()` call methods that implement the class's ***setter***. Those methods then create attributes named `_hour`, `_minute` and `_second` that is meant for use only inside the class!
2. Lines 10–21 define methods that manipulate a data attribute named `_hour`. The single-leading-underscore (_) naming convention indicates that we should not access `_hour` directly. We define a ***getter*** method which gets (that is, returns) a data attribute's value and a setter method, which sets a data attribute's value.

Here is how we initialize an object:

1. Class `Time`'s `__init__` method specifies `hour`, `minute` and `second` parameters, each with a default argument of 0. The statements containing `self.set_hour()`, `self.set_minute()` and `self.set_second()` call methods that implement the class's ***setter***. Those methods then create attributes named `_hour`, `_minute` and `_second` that is meant for use only inside the class!

2. Lines 10–21 define methods that manipulate a data attribute named `_hour`. The single-leading-underscore (_) naming convention indicates that we should not access `_hour` directly. We define a ***getter*** method which gets (that is, returns) a data attribute's value and a setter method, which sets a data attribute's value.

Here is how we initialize an object:

In [37]:
```
wake_up = Time(hour=8, minute=30)
```

```
setter is called
```

The following code expression invokes the getter method:

The following code expression invokes the getter method:

In [38]:
```
wake_up.get_hour()
# Instead of wake_up._hour
```

```
getter is called
```

Out[38]:
```
8
```

The following code expression invokes the getter method:

In [38]:
```
wake_up.get_hour()
# Instead of wake_up._hour
```

```
getter is called
```

Out[38]:
```
8
```

The following code expression invokes the setter by assigning a value to the attribute:

The following code expression invokes the getter method:

```
In [38]: wake_up.get_hour()
         # Instead of wake_up._hour
```

```
getter is called
```

```
Out[38]: 8
```

The following code expression invokes the setter by assigning a value to the attribute:

```
In [39]: wake_up.set_hour(8)
         # Instead of wake_up._hour = 8
```

```
setter is called
```

Class `Time`'s getter and setter define the class's public interface — that is, the set of attributes programmers should use to interact with objects of the class.

Class `Time`'s getter and setter define the class's public interface — that is, the set of attributes programmers should use to interact with objects of the class.

Just like the private attributes above, not all methods need to serve as part of a class's interface. Some serve as ***utility methods*** used only inside the class and are not intended to be part of the class's public interface used by others. Such methods should be named with a single leading underscore. In other object-oriented languages like C++, Java and C#, such methods typically are implemented as ***private methods***.

Class `Time`'s getter and setter define the class's public interface — that is, the set of attributes programmers should use to interact with objects of the class.

Just like the private attributes above, not all methods need to serve as part of a class's interface. Some serve as ***utility methods*** used only inside the class and are not intended to be part of the class's public interface used by others. Such methods should be named with a single leading underscore. In other object-oriented languages like C++, Java and C#, such methods typically are implemented as ***private methods***.

Note that although we define the public interface, the internal attribute can still be accessed.

Class `Time`'s getter and setter define the class's public interface — that is, the set of attributes programmers should use to interact with objects of the class.

Just like the private attributes above, not all methods need to serve as part of a class's interface. Some serve as ***utility methods*** used only inside the class and are not intended to be part of the class's public interface used by others. Such methods should be named with a single leading underscore. In other object-oriented languages like C++, Java and C#, such methods typically are implemented as ***private methods***.

Note that although we define the public interface, the internal attribute can still be accessed.

In [5]:
```
wake_up._hour
```

Out[5]:
```
8
```

Simulating ”Private” Attributes

In programming languages such as C++, Java and C#, classes state explicitly which class members are publicly accessible. Class members that may not be accessed outside a class definition are private and visible only within the class that defines them.

In programming languages such as C++, Java and C#, classes state explicitly which class members are publicly accessible. Class members that may not be accessed outside a class definition are private and visible only within the class that defines them.

Rather than `_hour`, we can name the attribute `__hour` with two leading underscores. This convention indicates that `__hour` is "private" and should not be access outside.

In programming languages such as C++, Java and C#, classes state explicitly which class members are publicly accessible. Class members that may not be accessed outside a class definition are private and visible only within the class that defines them.

Rather than `_hour`, we can name the attribute `__hour` with two leading underscores. This convention indicates that `__hour` is "private" and should not be access outside.

In [40]:

```
class PrivateClass:
    """Class with public and private attributes."""

    def __init__(self):
        """Initialize the public and private attributes."""
        self.public_data = "public"  # public attribute
        self._private_data = "private1"  # private attribute
        self.__private_data = "private2"  # private attribute
```

In [41]:

```python
my_object = PrivateClass()
my_object.public_data
```

Out[41]:

```
'public'
```

In [41]:
```
my_object = PrivateClass()
my_object.public_data
```
Out[41]:
```
'public'
```
In [8]:
```
my_object._private_data
```
Out[8]:
```
'private1'
```

In [41]:
```
my_object = PrivateClass()
my_object.public_data
```

Out[41]:
```
'public'
```

In [8]:
```
my_object._private_data
```

Out[8]:
```
'private1'
```

When we attempt to access `__private_data` directly, we get an `AttributeError` indicating that the class does not have an attribute by that name:

In [41]:
```
my_object = PrivateClass()
my_object.public_data
```

Out[41]:
```
'public'
```

In [8]:
```
my_object._private_data
```

Out[8]:
```
'private1'
```

When we attempt to access `__private_data` directly, we get an `AttributeError` indicating that the class does not have an attribute by that name:

In [9]:
```
my_object.__private_data
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9884\3291977527.py in <module>
----> 1 my_object.__private_data

AttributeError: 'PrivateClass' object has no attribute '__private_data'
```

```
In [74]: display_quiz(path+"encapsulation.json", max_width=800)
```

What is printed by the following code?

```
class Person:
    def __init__(self, name):
        self.__name = name  # private attribute

    def get_name(self):
        return self.__name

p = Person("Alice")
print(p.get_name())
print(p.__name)
```

```
AttributeError
AttributeError
```

```
Alice
Alice
```

```
Alice
AttributeError
```

```
Alice
None
```

Class Methods

Class methods are associated with a class rather than individual objects like regular methods are.

Class methods are associated with a class rather than individual objects like regular methods are.

You can recognize a class method in code when you see two markers:

1. The `@classmethod` decorator before the method's `def` statement.
2. The use of `cls` as the first parameter

Class methods are associated with a class rather than individual objects like regular methods are.

You can recognize a class method in code when you see two markers:

1. The `@classmethod` decorator before the method's `def` statement.
2. The use of `cls` as the first parameter

In [46]:
```
class ExampleClass:
    def exampleRegularMethod(self):
        print('This is a regular method.')
    # This is the "decorator" that takes another function as input,
    # extends or modifies its behavior, and returns a new function
    @classmethod
    def exampleClassMethod(cls):
        print('This is a class method.')
```

In [47]:
```python
ExampleClass.exampleRegularMethod() # This is not a valid statement
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_27100\3660119935.py in <module>
----> 1 ExampleClass.exampleRegularMethod() # This is not a valid statement

TypeError: exampleRegularMethod() missing 1 required positional argument: 'self'
```

In [47]:
```python
ExampleClass.exampleRegularMethod() # This is not a valid statement
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_27100\3660119935.py in <module>
----> 1 ExampleClass.exampleRegularMethod() # This is not a valid stat
ement

TypeError: exampleRegularMethod() missing 1 required positional argume
nt: 'self'
```

In [48]:
```python
# Call the class method without instantiating an object!
ExampleClass.exampleClassMethod()

obj = ExampleClass()
# Given the above line, these two lines are equivalent
# Note that we can also access the class method using obj!
obj.exampleClassMethod() # It will implicitly pass the class instead of the o
obj.__class__.exampleClassMethod()
```

```
This is a class method.
This is a class method.
This is a class method.
```

The `cls` parameter acts like `self` except `self` refers to an object, but the `cls` parameter refers to an object's class.

The `cls` parameter acts like `self` except `self` refers to an object, but the `cls` parameter refers to an object's class.

This means that the code in a class method cannot access an individual object's attributes or call an object's regular methods. Class methods can only call other class methods or access class attributes.

The `cls` parameter acts like `self` except `self` refers to an object, but the `cls` parameter refers to an object's class.

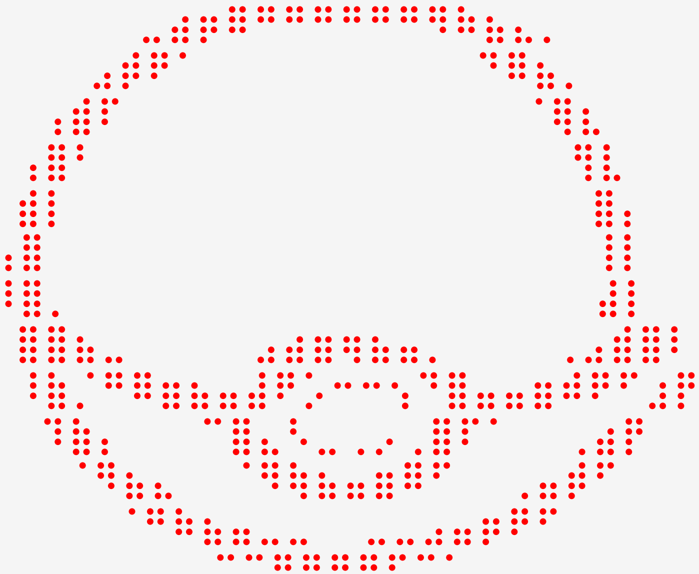
This means that the code in a class method cannot access an individual object's attributes or call an object's regular methods. Class methods can only call other class methods or access class attributes.

Class methods aren't commonly used. The most frequent use case is to provide **alternative constructor** methods besides `__init__()`. For example, what if a constructor function could accept either a `string` of data the new object needs or a `string` of a filename that contains the data the new object needs?

We don't want the `__init__()` method's parameters to be lengthy and confusing. Instead, let's use class methods to return a new object. For example, let's create an `AsciiArt` class.

We don't want the `__init__()` method's parameters to be lengthy and confusing. Instead, let's use class methods to return a new object. For example, let's create an `AsciiArt` class.

In [49]:
```
%%writefile pokeball.txt
```



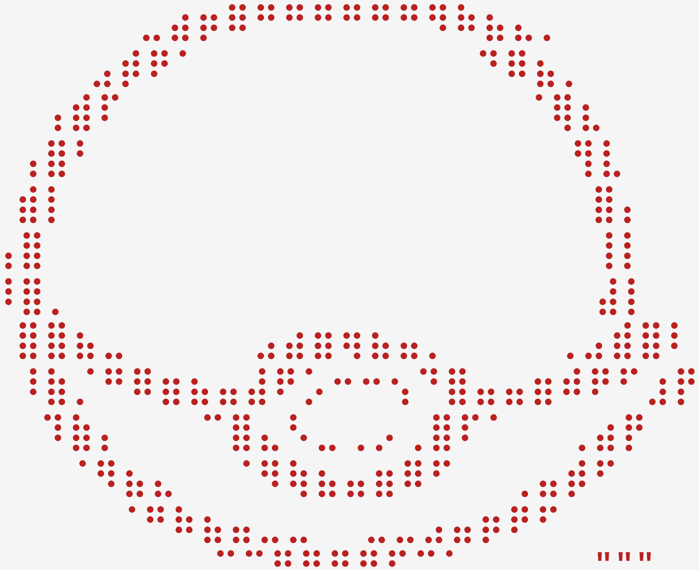
```
Overwriting pokeball.txt
```

In [51]:

```python
class AsciiArt:
    def __init__(self, characters):
        """ Approach1: Initialize it with string """
        self._characters = characters

    @classmethod
    def fromFile(cls, filename):
        """ Approach2: Initialize it with filename """
        with open(filename) as fileObj:
            characters = fileObj.read()
        return AsciiArt(characters) # This calls the __init__ function, notic

    def display(self):
        print(self._characters)

    # Other AsciiArt methods would go here...
```

In [ ]:

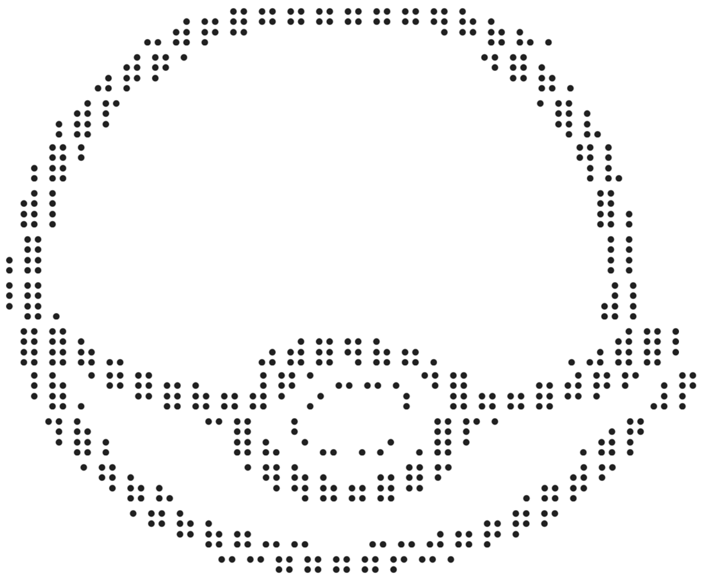
```
ball1 = AsciiArt("""
```



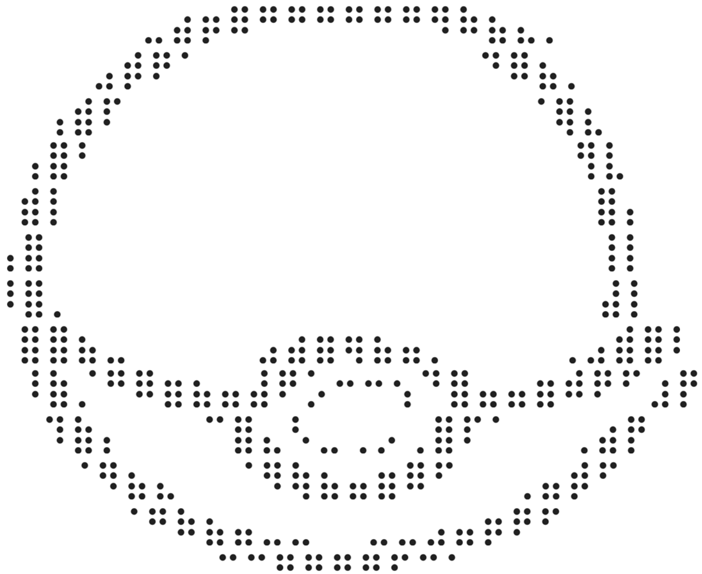
```
"""
)

ball1.display()
```

In [53]:

```
ball2 = AsciiArt.fromFile('pokeball.txt')
ball2.display()
```



In [53]:
```python
ball2 = AsciiArt.fromFile('pokeball.txt')
ball2.display()
```



The `AsciiArt` class has an `__init__()` method that can be passed the text characters of the image as a string. It also has a `fromFile()` class method that can be passed the filename string of a text file containing the ASCII art. Both methods create `AsciiArt` objects.

Class Attributes

A class attribute is a variable that belongs to the class rather than to an object. We create class attributes **inside the class but outside all methods**, just like we can create global variables in a `.py` file but outside all functions.

A class attribute is a variable that belongs to the class rather than to an object. We create class attributes **inside the class but outside all methods**, just like we can create global variables in a `.py` file but outside all functions.

Here's an example of a class attribute named `count`, which keeps track of how many `CreateCounter` objects have been created:

A class attribute is a variable that belongs to the class rather than to an object. We create class attributes **inside the class but outside all methods**, just like we can create global variables in a `.py` file but outside all functions.

Here's an example of a class attribute named `count`, which keeps track of how many `CreateCounter` objects have been created:

In [54]:

```
class CreateCounter:
    count = 0 # This is a class attribute.

    def __init__(self):
        CreateCounter.count += 1

print('Objects created:', CreateCounter.count)  # Prints 0.
a = CreateCounter()
b = CreateCounter()
c = CreateCounter()
print('Objects created:', CreateCounter.count)  # Prints 3.
```

```
Objects created: 0
Objects created: 3
```

In [75]:
```python
display_quiz(path+"class_method.json", max_width=800)
```

What is printed by the following code?

```python
class MyClass:
    count = 0

    def __init__(self):
        MyClass.count += 1

    @classmethod
    def get_count(cls):
        return cls.count

obj1 = MyClass()
obj2 = MyClass()
print(obj1.get_count())
print(MyClass.get_count())
```

2 2	2 0
1 1	0 2

Polymorphism

Polymorphism allows objects of one type to be treated as objects of another type (class).

1. For example, the `len()` function returns the length of the argument passed to it. You can pass a `string` to `len()` to see how many characters it has, but you can also pass a `list` or `dictionary` to `len()` to see how many items or key-value pairs it has, respectively. This polymorphism of function is called generic functions or ***method/function overloading*** because it can handle objects of many different types.

Polymorphism allows objects of one type to be treated as objects of another type (class).

1. For example, the `len()` function returns the length of the argument passed to it. You can pass a `string` to `len()` to see how many characters it has, but you can also pass a `list` or `dictionary` to `len()` to see how many items or key-value pairs it has, respectively. This polymorphism of function is called generic functions or ***method/function overloading*** because it can handle objects of many different types.

2. Polymorphism also includes ***operator overloading***, where operators (such as `+` or `*`) can behave differently based on the type of objects they're operating on. For example, the `+` operator does mathematical addition when operating on two `integer` or `float` values, but it does `string` concatenation when operating on two strings.

Polymorphism allows objects of one type to be treated as objects of another type (class).

1. For example, the `len()` function returns the length of the argument passed to it. You can pass a `string` to `len()` to see how many characters it has, but you can also pass a `list` or `dictionary` to `len()` to see how many items or key-value pairs it has, respectively. This polymorphism of function is called generic functions or ***method/function overloading*** because it can handle objects of many different types.

2. Polymorphism also includes ***operator overloading***, where operators (such as `+` or `*`) can behave differently based on the type of objects they're operating on. For example, the `+` operator does mathematical addition when operating on two `integer` or `float` values, but it does `string` concatenation when operating on two strings.

In `Python`, we can achieve method polymorphism by defining a method in a base class and then overriding it in the derived classes. Each derived class can then provide its implementation of the method.

In [58]:

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

    def speak(self):
        pass # Don't do anything and prevent error by using this keyword

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

In [ ]:

```python
def speak(animal):
    print(animal.speak())

animals = [Dog("Rufus"), Cat("Whiskers"), Dog("Buddy")]

# method overloading
for animal in animals:
    print(f'{animal.name} : {animal.speak()}')

# function overloading
for animal in animals:
    speak(animal)
```

In [ ]:

```python
def speak(animal):
    print(animal.speak())

animals = [Dog("Rufus"), Cat("Whiskers"), Dog("Buddy")]

# method overloading
for animal in animals:
    print(f'{animal.name} : {animal.speak()}')

# function overloading
for animal in animals:
    speak(animal)
```

In this example, the `Animal` class defines the `speak` method as a `pass` statement, **meaning it does nothing**. However, both `Dog` and `Cat` classes override the method with their implementation of the method. This is called ***method overriding*** and is also a form of polymorphism.

In [ ]:
```python
def speak(animal):
    print(animal.speak())

animals = [Dog("Rufus"), Cat("Whiskers"), Dog("Buddy")]

# method overloading
for animal in animals:
    print(f'{animal.name} : {animal.speak()}')

# function overloading
for animal in animals:
    speak(animal)
```

In this example, the `Animal` class defines the `speak` method as a `pass` statement, **meaning it does nothing**. However, both `Dog` and `Cat` classes override the method with their implementation of the method. This is called ***method overriding*** and is also a form of polymorphism.

The `speak()` function accepts any object that implements the `speak()` method, meaning it can handle animals of different types. Here, we can pass both `Dog` and `Cat` objects to the `speak()` function, as they both inherit the `speak()` method from the `Animal` class.

Operator overloading

`Python` has several ***dunder*** method. You're already familiar with the `__init__()` dunder method name, but `Python` has several more. We often use them for operator overloading — that is, adding custom behaviors that allow us to use objects of our classes with `Python` operators, such as `+` or `>=`.

`Python` has several ***dunder*** method. You're already familiar with the `__init__()` dunder method name, but `Python` has several more. We often use them for operator overloading — that is, adding custom behaviors that allow us to use objects of our classes with `Python` operators, such as `+` or `>=`.

In [59]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __eq__(self, other):
        return (self.x == other.x) and (self.y == other.y)
```

In [60]:

```python
p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
p4 = Point(4, 6)
print(p3.x, p3.y)  # Output: 4 6
print(p3 == p4)
```

```
4 6
True
```

In [60]:
```python
p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
p4 = Point(4, 6)
print(p3.x, p3.y)  # Output: 4 6
print(p3 == p4)
```

```
4 6
True
```

In this example, we define the `__add__` and `__eq__` method in the `Point` class to implement the addition and equality of two `Point` objects. When we use the `+` and `=` operators with two `Point` objects, the `__add__` and `__eq__` methods are called automatically to perform the addition and comparison.

For more information about overloading, see here.

```
In [76]: display_quiz(path+"poly.json", max_width=800)
```

What is printed by the following code?

```
class Shape:
    def area(self):
        return "No area"

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius * self.radius

class Square(Shape):
    def __init__(self, side):
        self.side = side
    def area(self):
        return self.side * self.side

shapes = [Circle(2), Square(3)]
for shape in shapes:
    print(shape.area())
```

```
12.56
9
```

```
12.56 9
```

```
9
12.56
```

```
No area
No area
```

> Exercise 2: Inherit from `Pokemon` class to create new classes, `firePokemon` and `waterPokemon`, that accept the same parameters when constructed. Add a new method `attack()` for the two derived classes that recieve a single parameter `attack_type` and print out the message like this:

```
Magmortar is attacking with flamethrower
```

In addition, define a function `PokemonAttack()`, which receives a Pokemon object and an `attack_type`, then call the method `attack()`. Complete the following class/function and execute the code cell.

In [61]:
```python
class Pokemon:
    def __init__(self, name, type, total_specis):
        self.name = name
        self.type = type
        self.total_specis = total_specis

    def __str__(self):
        return f"{self.name} ({self.type}, total specis {self.total_specis})"

class firePokemon(Pokemon):
    # Your code here
    def __init__(self, name, type, total_specis):
        self.name = name
        self.type = type
        self.total_specis = total_specis
    # Your code here
    def attack(self, attack_type):
        print(f"{self.name} is attacking with {attack_type}")
```

In [63]:
```python
import random
import json
import time
import requests

class waterPokemon(Pokemon):
    # Your code here
    def __init__(self, name, type, total_specis):
        super().__init__(name, type, total_specis)
    # Your code here
    def attack(self, attack_type):
        print(f"{self.name} is attacking with {attack_type}")

def PokemonAttack(pokemon, attack_type):
    # Your code here
    pokemon.attack(attack_type)

def slow_print(text, delay=0.05):
    for char in text:
        print(char, end='', flush=True)
        time.sleep(delay)
    print()
```

In [64]:

```python
## 1. Download the data
url = 'https://raw.githubusercontent.com/fanzeyi/pokemon.json/master/pokedex.
response = requests.get(url)

if response.status_code == 200:
    with open('pokedex.json', 'w', encoding = "utf-8") as f:
        f.write(response.text)
else:
    print(f"Failed to download the file. Status code: {response.status_code}"

with open('Pokedex.json', 'r', encoding = "utf-8") as file:
    pokemon_data = json.load(file)
```

In [65]:

```python
## 2. Get the pokemon
random.shuffle(pokemon_data)
getpokemon = []
i = 0
while True:
    if len(getpokemon) == 6:
        break
    if pokemon_data[i]['type'][0] != "Fire" and pokemon_data[i]['type'][0] !=
        i += 1
        continue
    else:
        if pokemon_data[i]['type'][0] == "Fire":
            pokemon = firePokemon(pokemon_data[i]['name']['english'], pokemon_
        else:
            pokemon = waterPokemon(pokemon_data[i]['name']['english'], pokemo
        getpokemon.append(pokemon)
        i += 1
```

In [66]:

```python
# 3. Print the pokemon and attack!
for i in range(6):
    if getpokemon[i].type == "Fire":
        attack = 'flamethrower'
    else:
        attack = 'hydro pump'
    PokemonAttack(getpokemon[i], attack)
```

```
Palkia is attacking with hydro pump
Kingdra is attacking with hydro pump
Seaking is attacking with hydro pump
Clamperl is attacking with hydro pump
Greninja is attacking with hydro pump
Azumarill is attacking with hydro pump
```

Summary

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that attributes and behaviors are bundled into individual objects.

- Inheritance promotes code reusability and organization by allowing derived classes to inherit attributes and methods from parent classes.

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that attributes and behaviors are bundled into individual objects.

- Inheritance promotes code reusability and organization by allowing derived classes to inherit attributes and methods from parent classes.
- Encapsulation improves maintainability and security by bundling data and methods within objects and controlling access to their internal state.

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that attributes and behaviors are bundled into individual objects.

- Inheritance promotes code reusability and organization by allowing derived classes to inherit attributes and methods from parent classes.
- Encapsulation improves maintainability and security by bundling data and methods within objects and controlling access to their internal state.
- Polymorphism enhances flexibility and extensibility by enabling a single interface to represent different types, allowing for interchangeable objects and easier code modification.

In [2]:

```
from jupytercards import display_flashcards
fpath= "https://raw.githubusercontent.com/phonchi/nsysu-math106A/refs/heads/m
display_flashcards(fpath + 'ch8.json')
```

Object Oriented Programming (OOP)

Next

>