SDEV 1001

Programming Fundamentals

Classes and Objects - 1 and 2

A LEADING POLYTECHNIC COMMITTED TO YOUR SUCCESS

Expectations - What I expect from you

- No Late Assignments

- No Cheating

- Be a good classmate

- Don’t waste your time

- Show up to class

Agenda

On the right is what we will cover today.

- Object-Oriented Programming (OOP)
- Why is OOP Important? What are the benefits?
- Classes and Objects in Python
- Defining a Class
- Creating and Using Objects
- Adding Methods to a Class
- The `__str__` method and dunder methods
- The `__repr__` Method
- Working with Collections of Objects
- Summary

Object-Oriented Programming (OOP)

What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to represent data and methods to manipulate that data. It helps in organizing code, making it reusable, and modeling real-world entities.

Analogy

Think of OOP like a car factory. Each car is an object, and the factory (class) defines how cars are built and what features they have.

Why is OOP Important? What are the benefits?

- **Modularity**: Breaks down complex problems into smaller, manageable pieces.
- **Reusability**: Classes can be reused across different programs.
- **Encapsulation**: Keeps data safe within objects, exposing only what is necessary. Note this less true in Python than in some other languages.
- **Inheritance**: Allows new classes to inherit properties and methods from existing classes, promoting code reuse.
- **Polymorphism**: Enables objects to be treated as instances of their parent class, allowing for flexible code.

Over time you'll learn more about these concepts, but for now, just know that OOP is a powerful way to structure your code.

Classes and Objects in Python

Classes and objects help you organize code by modeling real-world things and their behaviors.

- **Class**: A blueprint for creating objects (like a recipe).
  - This is analogous to the car factory that defines how cars are built.
- **Object**: An instance of a class (like a cake made from a recipe).
  - This is analogous to a specific car made in the factory.

In the next few slides, we’ll explore how to define classes, create objects, and use methods in Python with this analogy in mind.

Defining a Class

A class is defined using the `class` keyword followed by the class name. Inside the class, you can define attributes (data) and methods (functions).

- A class is like a blueprint for creating objects.

Important concepts in this example:

- `__init__` is the constructor, called when you create a new object. This is the method called when you create an instance of a class.
- `self` refers to the current object, allowing you to access its attributes and methods in the class (more on this later).
  - `make`, `model`, and `year` are attributes of the class, that you can think of as the properties of a car, different cars can have different makes, models, and years.

Defining a Class

Here’s an example of the last slide in code form:

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
```

So here in this example, we define a `Car` class with an `__init__` method that initializes the attributes `make`, `model`, and `year`. When you create a new `Car` object, you provide these values.

In the next slide, we’ll see how to create objects from this class and access their attributes.

Creating and Using Objects

Using the class we defined, we can create objects (instances of the class) and access their attributes.

- Create an object by calling the class like a function.
  - Important note here is that you need to pass the parameters that you defined in the `__init__` method.
- Access attributes with dot notation. Note here you can access the attributes of an object using the dot notation, like `my_car.make` or `my_car.model`.
  - You shouldn't try to access variables that have an underscore at the start of their name, like `self._make`, as these are considered private attributes in Python. This is a convention to indicate that these attributes should not be accessed directly from outside the class.

Creating and Using Objects

Here’s the code to create and use objects from the `Car` class we defined earlier:

```
my_car = Car("Toyota", "Corolla", 2020)
print(my_car.make)   # Output: Toyota
print(my_car.model)  # Output: Corolla
second_car = Car("Honda", "Civic", 2021)
print(second_car.year)  # Output: 2021
print(second_car.make)  # Output: Honda
```

Note on other programming languages: To create instances of a class, you typically use the `new` keyword, but in Python, you simply call the class name as if it were a function.

Adding Methods to a Class

This is where classes become powerful. You can define methods (functions) inside a class to give objects behaviors based on their attributes.

- The methods are a special type of function that is defined inside a class. The first parameter is always `self`, which refers to the instance of the class.
  - this is passed automatically when you call the method on an object.
- When you're calling the method, you don't pass it, you just call it on the object and pass in the other parameters if needed.

Example:

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def honk(self):
        print(f"{self.make} {self.model} says beep beep!")

my_car = Car("Honda", "Civic")
my_car.honk()
# note you didn't have to pass `self` here, it is passed au
```

Here's the output of the above code:

```
Honda Civic says beep beep!
```

The `__str__` method and dunder methods

Notice that `__init__` is a special method (often called a dunder method, short for "double underscore"). Python has many such methods that allow you to customize how your objects behave in different situations.

Here we use the `__str__` method to define how the object should be represented as a string when printed. This is useful for debugging and logging.

There are a ton of other dunder methods that you can use to customize the behavior of your objects, such as `__eq__` for equality checks, `__len__` for length, and many more. We'll talk more about these in later classes.

Example:

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def __str__(self):
        return f"{self.make} {self.model}"

my_car = Car("Ford", "Focus")
print(my_car)  # Output: Ford Focus
```

Here's the output of the above code:

```
Ford Focus
```

WE ARE ESSENTIAL TO ALBERTA

NAIT

The `__repr__` Method

The `__repr__` method is another special (dunder) method in Python. It defines the "official" string representation of an object, which is useful for debugging and development.

- `__repr__` should return a string that, if possible, could be used to recreate the object.
- When you inspect an object in the Python shell or use `repr(obj)`, this method is called.
- If `__str__` is not defined, `print(obj)` will use `__repr__` as a fallback.

Note: It’s normally best practice to define both `__str__` and `__repr__` methods in your classes. `__str__` is for end-users, while `__repr__` is for developers and debugging.

Example:

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def __repr__(self):
        return f"Car(make='{self.make}', model='{self.model

my_car = Car("Subaru", "Impreza")
print(repr(my_car))  # Output: Car(make='Subaru', model='Im
```

The output of the above code:

```
Car(make='Subaru', model='Impreza')
```

Working with Collections of Objects

Attributes can by any data type, including other objects. You can create collections of objects to manage multiple instances of a class.

In the last slide, we define a `Garage` class that can hold multiple `Car` objects.

- The `add_car` method allows us to add a car to the garage, and the `list_cars` method prints all cars in the garage.
  - remember from the last slide that `print(car)` will call the `__str__` method of the `Car` class, so it will print the car's make and model.

Example:

```
class Garage:
    def __init__(self):
        self.cars = []

    def add_car(self, car):
        self.cars.append(car)

    def list_cars(self):
        for car in self.cars:
            print(car)

garage = Garage()
garage.add_car(Car("Mazda", "3"))
garage.add_car(Car("Tesla", "Model 3"))
garage.list_cars()
```

Here's the output of the `list_cars` method:

```
Mazda 3
Tesla Model 3
```

Summary

- Classes let you model real-world things in code.
- Use `__init__` for initialization and `__str__` for printing.
- Methods define behaviors for your objects.
- Organize related objects in collections for more complex programs.

Example

Let’s go run a few examples together

A LEADING POLYTECHNIC COMMITTED TO YOUR SUCCESS