Object-Oriented Programming

Overview

In this lesson, students will be introduced to Object-Oriented Programming, centered around the use of classes and inheritance.

Duration
120 minutes

Learning Objectives

In this lesson, students will:

- Explain what it means that Python is an Object-Oriented Programming Language.
- Define a class
- Instantiate an object from a class.
- Create classes with default instance variables.

Pre-Class Materials and Preparation

For remote classrooms: Virtual breakout rooms and Slack may be needed to facilitate the partner exercise and discussions. As you plan for your lesson:

- Consider how you'll create pairs for the partner exercise (randomly, or with pre-assigned partners).
- Determine how (if at all) exercise timing may need to be adjusted.
- For helpful tips, keep an eye out for the **For remote classrooms** tag in the speaker notes.
- Prepare screenshots and answers to exercises in advance so that they can be easily shared in Slack during your lecture.

Suggested Agenda

Time	Activity
0:00–0:30	**Classes**
0:30–0:45	**First Class**
0:45–1:15	**House of Cards**
1:30–1:50	**Getting the Band Together**
1:50–2:00	**Wrapping Up, Q&A, and Exit Ticket Completion**

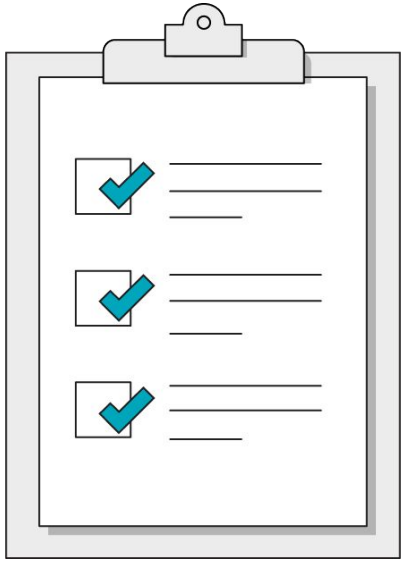
Jupyter Notebook

The exercises referenced in this lesson can be found in the Python Workbooks + Data folder.

Object-Oriented Programming

Our Learning Goals

- Explain what it means that Python is an Object-Oriented Programming Language.
- Define a class
- Instantiate an object from a class.
- Create classes with default instance variables.



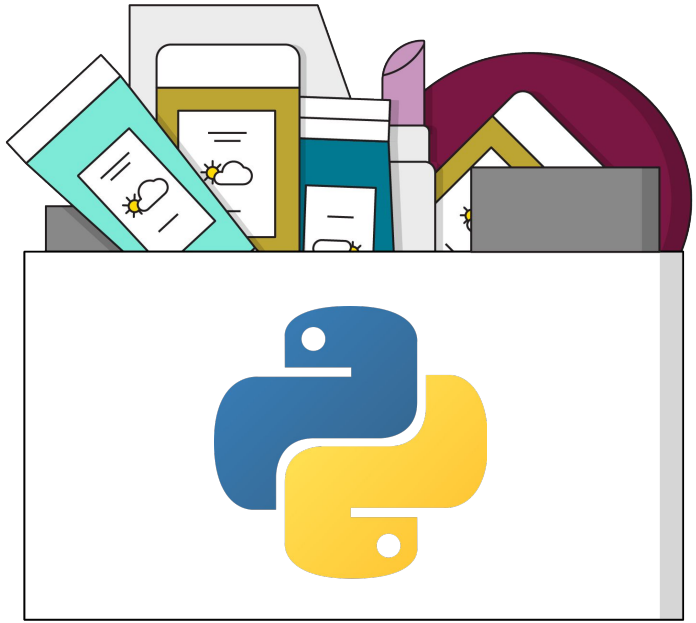
Object-Oriented Programming

Classes

Objects Are Everywhere

What if we told you that every variable you've ever declared in Python has been one type all along: **objects.**

This makes Python an **Object-Oriented** language.



Discussion:

Properties and Methods

All objects have **properties** and **methods**.

Methods are functions that objects can perform, such as when a list uses `.append()` or `.pop()`

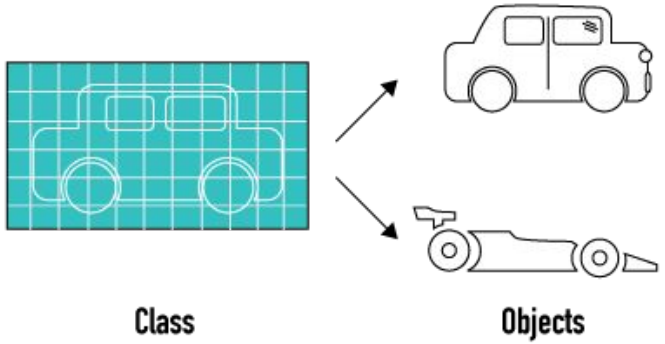
Let's say we're trying to represent a car as a Python object. What properties and methods would a car have?

What properties would **all** cars have?

Classes Ensure Consistency Among Objects

All cars have things that make them a **`Car`**. Although the details might be different, every type of car has the same basic properties and methods. This pattern is known as a **class.**

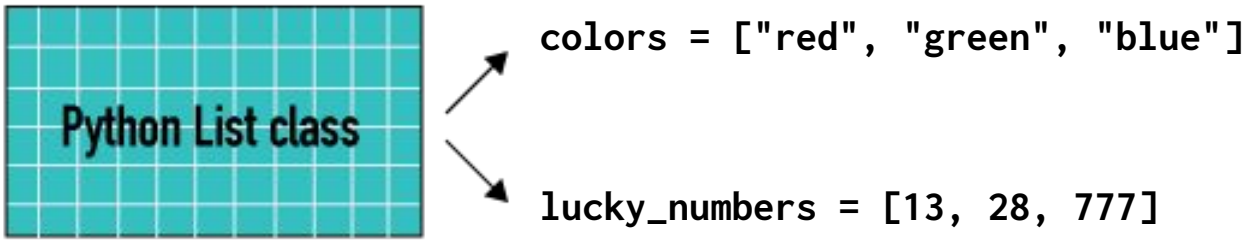
- Property: A shape
- Property: A color
- Property: Number of seats
- Method: Can drive.
- Method: Can park.



We've Been Using Classes All Along!

Data types are the most foundational classes in programming: String, Integer, List, Dictionary...these are all classes that Python uses under the hood.

For example, every List we've made has the same methods: append, pop, prepend, and so on.



Custom Classes for Custom Data

A class helps us ensure consistency among all the objects of a specific type.

To start, we're going to:

1. Create a class
2. Decide what properties the objects should have
    a. These properties might have default values, which we can also set up
3. Create, or instantiate, objects that belong to our class

Discussion:

Classes in Applications

Think about a standard social media application: what kinds of data might this app use Classes to represent?

Defining a Class

To start defining a function, we simply use the class keyword like so:

```
class Dog:
    # Anything indented beneath the class will define the class
```

We're going to eventually use the class like a function to create new objects:

```
fido = Dog()
poochie = Dog()
```

This would create two different objects of the Dog class.

Setting Up the Init Method

Then, the first part of ANY class will be the **__init__** method. This method will define the process of creating, or **instantiating**, a new object of the class.

Within the init method, we have access to a **self** object that represents the individual object we're creating. It **must** be the first parameter in every init.

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

fido = Dog("Fido", 8)
```


Discussion:

Useful Patterns in the Init Method

Within the init method, we can perform all sorts of useful validating and standardizing procedures. What's the logic behind this init method?

```
class Dog:
    def __init__(self, name, age):
        if age < 0:
            self.age = 0
        else:
            self.age = age
        self.good_dog = True
        self.name = name
```


Default Values in Methods

Just like regular functions, method parameters can be given default arguments in case the user doesn't provide one.

```
def __init__(self, name="poochie", age=0):
    self.name = name
    self.age = age
```

This means that any user-provided arguments will be applied as normal, but we get to have fall-back values if the method is called without arguments.

```
default_dog = Dog()
# default_dog looks like: {name: "poochie", age: 0}
```

Solo Exercise:

First Class

15 minutes

In this exercise, we'll practice setting up a class for musicians, then create several instances of this class.

Create a class for musical artists called Musician.

Musicians should have name and genre properties set by parameters in the init method. The default genre should be "Pop".

Musicians should also have an albums_sold property that always starts at 0.

Group Exercise:
House of Cards

30 minutes

Now, let's use classes to programmatically create a full deck of playing cards.

First, create a Card class with two properties:

- Suit
  - Possible suits are Hearts, Clubs, Spades, or Diamonds
- Face
  - Possible face values are 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace

Then, fill up a list called cards with all 52 combinations of the above properties.

Bonus: Create a Deck class that, when instantiated, creates the 52 cards and stores them in a property called cards.

What about Defining Methods in a Class?

For every method, including the magic __init__ method we've already been using, the first parameter must be self.

After that, it's up to you! Defining a method works just like defining a function.

```
def speak(self):
    print(f"Bark! Bark! My name is {self.name}")

def count_to(self, x):
    for i in range(x):
        print("Woof!")
```


Our Dog has Properties and Methods

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

    def speak(self):
        print(f"Bark! Bark! My name is {self.name}")

poochy = Dog("poochy", 3)
poochy.speak()
```


A Common Error of Python Classes

It's easy to forget to include self as the first parameter.

```
def speak():
    print("woof woof")
```

We get an error that's not exactly clear about what it means:

```
TypeError: speak() takes 0 positional arguments but 1 was given
```

Python secretly provides **self** to every method that gets called. If our methods aren't defined to expect this, we get the error above.

Discussion:

Tea Time

What might a class for a TeaCup look like?

What properties and methods could the class have?

Accessing Properties Within Methods

```
class TeaCup:
  def __init__(self, capacity):
    self.capacity = capacity # Total ounces the cup holds.
    self.amount = 0 # Current ounces in the cup. All cups start empty!

  def fill(self):
    self.amount = self.capacity

  def empty(self):
    self.amount = 0

  def drink(self, amount_to_drink):
     self.amount -= amount_to_drink
```

Partner Exercise:

Getting the Band Back Together

30 minutes

Let's create a class for Bands. A Band should have the following properties and methods:

- name: String
- members: a list of Strings, defaults to an empty list
- introduce_lineup(): a method that prints all of the strings in members
- add_member(new_member): a method that adds a new member to the members
- kick_out(old_member): a method that removes the given member from the members list. If the members list is empty, add a disbanded property equal to True.

Object-Oriented Programming

Wrapping Up

Recap

In today's class, we...

- Defined a class
- Instantiated an object from a class.
- Created classes with default instance variables.

Looking Ahead

On your own:

- Continue on with OOP with the optional Inheritance challenges

Next Class:

Error Handling and Debugging

Don’t Forget: Exit Tickets!


GA

Object-Oriented Programming Bonus Content

Inheritance

Moving From General to Specific Classes

Nearly every application will have the concept of a User. However, there are more specific types of users with different functionality and properties:

- Admin users have access to nearly everything behind the scenes
- Moderators might have editing capabilities over user-created content
- Team leaders have authority over a specific group of user accounts

Depending on the application, a wide range of more specific Users could be necessary to restrict or enhance a given user's abilities.

Creating Specific Sub-Classes with Inheritance

One of the most powerful concepts of Object-Oriented Programming is the ability for a class to **inherit** base functionality from a parent class, then **extend** in more specific ways.

The User class would contain everything common to all users.

The Admin class would have all the same functionality as a regular User, plus more specific properties and methods.

Inheritance Example

```
class User:
    all_users = []
    def __init__(self, username):
        self.username = username
        self.can_post = True
        all_users.append(self)
```

```
class Admin(User):
    def __init__(self, username):
        super().__init__(username)

    def suspend_user(username):
        for user in User.all_users:
            if user.username == username:
                user.can_post = False
```

Note the **super** method. It accesses the parent class. Why do we want to call the parent's __init__ function?

Solo Exercise:

Bands Part 2

20 minutes

Create subclasses that inherit from the Band class created earlier according to specifications described in the Jupyter Notebook.

Solo Exercise:

RPG Characters

45 minutes

Use inheritance to set up a combat simulation for a Role-Playing Game. The Character class will set up default properties for every character in the game, while the more specific classes will create more interesting, playable characters.