The Master Guide to Object-Oriented Programming in Python

Part I: The Foundations of Object-Oriented Programming

Object-Oriented Programming (OOP) is more than a set of features in a programming language; it is a paradigm, a way of thinking about and structuring software to manage complexity. Before delving into the specific syntax of Python's class keyword, it is essential to first build a robust conceptual model of what OOP is and why it represents a fundamental advancement over earlier programming styles. This section establishes that foundational "why," providing the context needed to understand the practical "how" that follows.

Section 1.1: From Procedural to Object-Oriented Thinking

The history of programming is, in large part, a history of developing better tools to manage complexity. As software systems grew from simple scripts to vast, interconnected applications, the methods for organizing code had to evolve.

The Procedural Paradigm

Early programming was dominated by the procedural paradigm, which organizes code into procedures or functions that operate on data structures. In this model, data and the functions that manipulate it are separate entities. For instance, to manage a collection of users, a programmer might use a list of dictionaries, where each dictionary represents a user. A separate set of functions would be created to perform operations like adding a user, deactivating an account, or retrieving a user's details.
While simple and effective for small-scale tasks, this separation becomes a significant liability as a program grows. The programmer must mentally track which functions are intended to operate on which specific data structures. There is no inherent link between the user data and the functions designed to manipulate it. This can lead to several problems:

1. **Data Integrity Issues:** Since the data structures are passive and globally accessible, any function can potentially modify them, leading to inconsistent or invalid states.
2. **Low Cohesion:** Related data and behavior are scattered throughout the codebase, making it difficult to understand the complete state and capabilities of a conceptual entity like a "user."
3. **High Coupling:** Functions often become dependent on the specific internal structure of the data they operate on. A change to the data structure (e.g., renaming a key in the user dictionary) could require changes in numerous functions across the application.

The Object-Oriented Paradigm Shift

Object-Oriented Programming addresses these challenges by fundamentally restructuring the relationship between data and behavior. The core innovation of OOP is the concept of the

object, a self-contained unit that bundles together both data (called **attributes**) and the functions that operate on that data (called **methods**).
This approach directly models real-world entities. Consider a car. A car has properties (state) like its color, current speed, and fuel level. It also has behaviors (actions) like accelerating, braking, and turning. In OOP, we would model this as a Car object. The object itself would contain the data for its color and speed, and it would also contain the methods to accelerate() and brake(). The accelerate() method would inherently know how to modify the speed attribute of the specific car object it belongs to.
This co-location of data and behavior is the cornerstone of OOP's power. It provides a superior mental model for managing the inherent complexity of software systems. Instead of reasoning about a tangled web of separate data structures and global functions, developers can reason about discrete, self-contained components. This shift promotes modularity, enhances code reusability, and makes software systems easier to maintain and scale over time.

Section 1.2: The Core Vocabulary: Classes, Objects, Attributes, and Methods

To work with OOP, one must first master its fundamental vocabulary. These core terms form the building blocks of any object-oriented system.

The Class: A Blueprint

In Python, a **class** is a blueprint or template for creating objects. A class definition, created with the class keyword, does not contain any real data itself. Instead, it defines the structure and behavior that all objects created from it will share.
An effective analogy is a cookie cutter. The cookie cutter defines the shape of a cookie, but it is not a cookie itself. Similarly, a Dog class might specify that all dogs must have a name and an age, but the class itself does not represent any specific dog.
A minimal class definition in Python is simple:

```
# Defines a blueprint for creating Dog objects.
# The 'pass' keyword is a placeholder indicating an empty block.
class Dog:
    pass
```

By convention, Python class names are written in CapitalizedWords notation.

The Object (Instance): A Concrete Realization

While a class is the blueprint, an **object** (more formally called an **instance**) is a concrete entity created from that class, containing actual data. If the Dog class is the cookie cutter, then an actual cookie is an instance. Each instance is a unique entity that exists at a specific memory address in the computer.
The process of creating an object from a class is called **instantiation**. This is done by calling the class as if it were a function:

```
# Instantiation: creating two distinct objects from the Dog class.
dog1 = Dog()
dog2 = Dog()
```

Here, dog1 and dog2 are two separate instances of the Dog class. Even if they were to have identical data, they would remain distinct objects in memory.

Attributes and Methods: State and Behavior

Classes define two primary types of members: attributes and methods.

- **Attributes** are variables that belong to an object and store its data. They represent the **state** or properties of the object. For a Dog object, attributes might include its name, age, and breed.
- **Methods** are functions defined inside a class. They represent the **behavior** or actions an object can perform. For a Dog object, methods might include bark(), sit(), and fetch(). These methods typically operate on the object's attributes.

Together, attributes and methods allow a class to create a comprehensive model of a real-world concept, encapsulating both what it *is* and what it *does*.

Section 1.3: The Birth of an Object: The __init__ Method and self

An empty class is of limited use. The power of OOP is realized when objects are created with a specific initial state. Python facilitates this through a special method called __init__.

Introducing __init__: The Initializer

The __init__ method is a special method, often called a "dunder" (double underscore) method, that Python automatically calls every time a new instance of a class is created. Its primary purpose is to **initialize** the newly created object by setting the initial values of its attributes.

Demystifying self

The first parameter of any instance method in Python, including __init__, is, by strong convention, named self. This parameter is not a keyword and has no special meaning to the Python interpreter itself; however, adhering to this convention is critical for code readability. When a method is called on an instance, for example, dog1.bark(), Python automatically passes the instance on which the method was called (dog1) as the first argument to that method. This argument is received by the self parameter. This is the mechanism by which a method knows which specific instance's data it should operate on. self is the bridge between a method and an instance's unique attributes.

Creating Instance Attributes

Within the __init__ method, the self parameter is used to create and assign values to **instance attributes**. These are attributes that are unique to each specific instance of the class. The syntax self.attribute_name = value attaches an attribute to the instance represented by self. The following example demonstrates a complete, simple class definition:

```
class Dog:
    # The initializer method is called when a new Dog object is
created.
    def __init__(self, name, age):
```

```
        # 'self' refers to the specific instance being created (e.g.,
dog1 or dog2).

        # These lines create and initialize instance attributes.
        # The 'name' attribute on this specific instance is set to the
'name' value passed in.
        self.name = name
        # The 'age' attribute on this specific instance is set to the
'age' value passed in.
        self.age = age

    # An instance method that uses the instance's attributes.
    def bark(self):
        return f"{self.name} says Woof!"

# Instantiation:
# 1. A new Dog object is created.
# 2. The __init__ method is called automatically.
# 3. The new object is passed to 'self', "Buddy" is passed to 'name',
and 3 is passed to 'age'.
dog1 = Dog("Buddy", 3)

# A second, distinct Dog object is created and initialized.
dog2 = Dog("Lucy", 5)

# Accessing instance attributes and calling an instance method.
print(f"{dog1.name} is {dog1.age} years old.")  # Output: Buddy is 3
years old.
print(dog1.bark())  # Output: Buddy says Woof!

print(f"{dog2.name} is {dog2.age} years old.")  # Output: Lucy is 5
years old.
print(dog2.bark())  # Output: Lucy says Woof!
```

When dog1 = Dog("Buddy", 3) is executed, Python creates an empty Dog object and then calls __init__, effectively executing Dog.__init__(<the_new_dog1_object>, "Buddy", 3). Inside the method, self refers to <the_new_dog1_object>, so self.name = name becomes <the_new_dog1_object>.name = "Buddy". The same process occurs independently for dog2.

__init__ is an Initializer, Not a Constructor

A crucial distinction in Python, often misunderstood by those coming from languages like C++ or Java, is the precise role of __init__. While commonly referred to as a constructor, it is more accurately an **initializer**.
The object creation process in Python is actually a two-step sequence:

1. **Creation (__new__)**: Python first calls a different special method, __new__. This is the true constructor. Its responsibility is to allocate memory and return a new, empty instance of the class. Developers rarely need to implement __new__ directly.

2. **Initialization (__init__)**: After __new__ has created and returned the instance, Python then immediately calls __init__, passing that newly created instance as the self argument. The job of __init__ is not to create the object, but to populate it with its initial attributes.

For the vast majority of practical applications, a developer only needs to be concerned with __init__. However, understanding this two-stage process is vital for certain advanced programming tasks, such as creating immutable types or subclassing some of Python's built-in types. It clarifies that __init__'s role is to set the initial state of an object that already exists.

Part II: The Anatomy of a Python Class: A Detailed Map

To master object-oriented programming in Python, it is essential to have a precise mental map of how the different components of a class—its data and its behaviors—are defined, stored, and accessed. This section provides a detailed dissection of a class, exploring the distinct roles of class and instance attributes, the different types of methods, and the underlying mechanisms that govern their interaction.

Section 2.1: The Data Layer - Attributes

Attributes are the variables that store the state of a class and its instances. Python makes a critical distinction between attributes that are shared by all instances and those that are unique to each one.

2.1.1. Class Attributes: Shared State

- **Definition**: A class attribute is a variable defined directly in the body of a class, outside of any of its methods.
- **Role**: Class attributes belong to the class itself, not to any particular instance. Consequently, there is only one copy of a class attribute, and it is shared among all instances created from that class.
- **Use Cases**: This shared nature makes class attributes ideal for storing constants or default values that are relevant to all instances. For example, a Circle class could store the mathematical constant pi as a class attribute, or a Dog class could store the scientific name for the species.
- **Access**: Class attributes can be accessed directly through the class name (e.g., Dog.species) or through any instance of the class (e.g., my_dog.species).

```
class Dog:
    # This is a class attribute. It is shared by all instances of Dog.
    species = "Canis familiaris"

    def __init__(self, name):
        # This is an instance attribute.
        self.name = name

d1 = Dog("Buddy")
d2 = Dog("Lucy")
```

```
# Accessing the class attribute through the class and instances
print(Dog.species)  # Output: Canis familiaris
print(d1.species)   # Output: Canis familiaris
print(d2.species)   # Output: Canis familiaris
```

2.1.2. Instance Attributes: Unique State

- **Definition**: An instance attribute is a variable that is bound to a specific instance, typically within the __init__ method, using the self parameter.
- **Role**: Instance attributes represent the unique state of each object. Every instance has its own separate copy of these attributes, and changes to one instance's attributes do not affect any other instance.
- **Use Cases**: This is the standard way to store data that varies from object to object, such as a dog's name or a car's color.
- **Access**: Instance attributes can only be accessed through a specific instance of the class (e.g., d1.name).

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        # These are instance attributes. They are unique to each Dog
object.
        self.name = name
        self.age = age

d1 = Dog("Buddy", 3)
d2 = Dog("Lucy", 5)

# Accessing instance attributes
print(d1.name)  # Output: Buddy
print(d2.name)  # Output: Lucy

# Modifying an instance attribute only affects that instance
d1.age = 4
print(d1.age)   # Output: 4
print(d2.age)   # Output: 5 (unchanged)
```

2.1.3. The Attribute Lookup Chain

The ability for an instance to access both its own attributes and the class's attributes is governed by Python's attribute lookup chain. When an attribute is accessed, such as d1.species, the interpreter follows a specific order of precedence:

1. **Instance Namespace**: Python first checks if the attribute exists in the instance's own namespace. An object's instance attributes are typically stored in a special dictionary attribute called __dict__.
2. **Class Namespace**: If the attribute is not found on the instance, Python then looks for it in

the class's namespace (Dog.__dict__).
3. **Parent Classes**: If the attribute is still not found, Python continues the search up the inheritance hierarchy, following the Method Resolution Order (MRO), which will be discussed in Part III.

This chain explains why d1.species works: Python fails to find species in d1.__dict__ and proceeds to find it in Dog.__dict__. It also explains why d1.name works: Python finds name immediately in d1.__dict__ and stops the search.

2.1.4. The Shadowing Effect and the Mutable Attribute Pitfall

A deep understanding of the attribute lookup chain is critical for avoiding two common and subtle sources of error.

- **Shadowing with Immutable Attributes**: When a value is assigned to a class attribute *through an instance*, such as d1.species = "Felis catus", it does **not** modify the shared class attribute. Because assignment (=) always operates on the instance's namespace, this operation creates a new **instance attribute** on d1 that shares the same name as the class attribute. This new instance attribute effectively "shadows" or hides the class attribute for that specific instance. The original class attribute remains unchanged, and other instances are completely unaffected.

```
class Dog:
    species = "Canis familiaris"

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

d1 = Dog("Buddy")
d2 = Dog("Lucy")

print(f"Initial species for d1: {d1.species}") # Output: Canis
familiaris
print(f"Initial species for d2: {d2.species}") # Output: Canis
familiaris

# This creates a new INSTANCE attribute on d1 called 'species'.
# It does NOT change the CLASS attribute.
d1.species = "Felis catus" # Buddy is having an identity crisis.

print(f"New species for d1: {d1.species}")     # Output: Felis catus
(accesses the instance attribute)
print(f"Species for d2: {d2.species}")         # Output: Canis
familiaris (still accesses the class attribute)
print(f"Class species: {Dog.species}")         # Output: Canis
familiaris (the class attribute is unchanged)
```

- **The Mutable Attribute Pitfall**: This shadowing behavior leads to a significant and often unexpected pitfall when the class attribute is a mutable type, such as a list or dictionary. If an instance modifies this mutable object *in-place* (e.g., using list.append() or dict.update()), the change will be visible across **all** instances of the class.

This happens because the operation is not an assignment to the instance (=), but a method call on the object that the attribute refers to. Since all instances initially refer to the *same single list object* in memory, modifying it through any one instance alters that shared object for everyone.

```
class Dog:
    # A mutable class attribute - this is often a mistake!
    tricks =

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

    def add_trick(self, trick):
        self.tricks.append(trick)

d1 = Dog("Buddy")
d2 = Dog("Lucy")

d1.add_trick("roll over")
d2.add_trick("play dead")

# The 'tricks' list was modified in-place, and since it's a shared
# class attribute, the changes are reflected in both instances.
print(d1.tricks)  # Output: ['roll over', 'play dead']
print(d2.tricks)  # Output: ['roll over', 'play dead']
print(Dog.tricks) # Output: ['roll over', 'play dead']
```

This behavior is a direct and logical consequence of Python's core mechanics: the attribute lookup order, the nature of assignment versus in-place mutation, and the distinction between mutable and immutable types. To avoid this, mutable attributes that should be unique to each instance must be initialized within the __init__ method, making them instance attributes:

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name):
        self.name = name
        # Correctly initialize 'tricks' as an instance attribute
        self.tricks =

    def add_trick(self, trick):
        self.tricks.append(trick)
```

Section 2.2: The Behavior Layer - Methods

Methods define the actions an object can perform. Just as with attributes, Python provides different types of methods whose behavior depends on their relationship to the class and its instances.

2.2.1. Instance Methods: The Default

- **Definition**: An instance method is a regular function defined within a class that receives the instance object itself as its first, implicit argument, conventionally named self. This is the most common type of method.
- **Role**: The primary role of an instance method is to access and modify the state of a specific instance. It has full access to both instance attributes (via self.attribute) and class attributes (via self.__class__.attribute).

```
class Student:
    school_name = "Springfield University" # Class attribute

    def __init__(self, name, student_id):
        self.name = name # Instance attribute
        self.student_id = student_id # Instance attribute

    # This is an instance method. It operates on a specific student.
    def get_info(self):
        return f"Student: {self.name}, ID: {self.student_id}, School:
{self.__class__.school_name}"
```

2.2.2. Class Methods: Operating on the Class

- **Definition**: A class method is a method that is bound to the class, not the instance. It is marked with the @classmethod decorator and receives the class object itself as its first, implicit argument, conventionally named cls.
- **Role**: Class methods are used for operations that relate to the class as a whole. They can access and modify class state (class attributes via cls.attribute), but they cannot access instance-specific state because they do not receive the self argument.
- **Primary Use Case: Alternative Constructors**: The most powerful and common use for class methods is to define "factory methods" that provide alternative ways to create instances of the class. For example, a Person class could be instantiated with a name and age, but a class method could provide a way to create a Person object from their birth year instead. This pattern is particularly useful because cls will correctly refer to any subclass that calls the method, enabling polymorphic object creation.

```
from datetime import date

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

    # An instance method
    def display(self):
        return f"{self.name} is {self.age} years old."

    # A class method acting as an alternative constructor.
    @classmethod
```

```
    def from_birth_year(cls, name, birth_year):
        current_year = date.today().year
        age = current_year - birth_year
        # 'cls(name, age)' is equivalent to 'Person(name, age)'.
        # If a subclass calls this, 'cls' will be the subclass.
        return cls(name, age)

# Creating an instance using the standard initializer
person1 = Person("Alice", 30)

# Creating an instance using the class method factory
person2 = Person.from_birth_year("Bob", 1990)

print(person1.display()) # Output: Alice is 30 years old.
print(person2.display()) # Output: Bob is 34 years old (assuming
current year is 2024).
```

2.2.3. Static Methods: Namespaced Utility Functions

- **Definition**: A static method is a method that is not bound to either the instance or the class. It is marked with the @staticmethod decorator and does not receive any implicit first argument (self or cls).
- **Role**: A static method is essentially a regular function that is logically grouped within the class's namespace. Its primary purpose is organizational. It cannot access or modify the class or instance state directly.
- **Use Case**: Static methods are suitable for utility functions that have a logical connection to the class but do not depend on its state. For example, a TemperatureConverter class might have a static method to validate if a given temperature value is physically possible.

```
class MathUtils:
    # A static method. It has a logical connection to the class
    # but does not depend on any class or instance state.
    @staticmethod
    def average(numbers):
        if not numbers:
            return 0
        return sum(numbers) / len(numbers)

# Can be called on the class directly, without creating an instance.
avg = MathUtils.average()
print(avg) # Output: 20.0
```

Section 2.3: The Grand Unified Map of Class Components

The following table synthesizes the concepts from the preceding sections to provide the clear, comprehensive mapping of OOP components requested. It serves as a definitive reference for understanding the role, scope, and access mechanisms of each element within the Python class

structure.

Component	Definition	Scope	How It's Defined	Key Parameter(s)	How to Access	Primary Role
Class	A blueprint or template for creating objects.	Module-level	class ClassName:	N/A	By its name, ClassName.	To define the shared structure (attributes) and behavior (methods) for a type of object.
Object (Instance)	A concrete entity created from a class, containing real data.	Variable scope	my_obj = ClassName()	N/A	Through the variable it's assigned to, my_obj.	To represent a specific entity with its own unique state and identity.
__init__ (Initializer)	A special method automatically called upon instantiation.	Class	def __init__(self,...):	self	Called automatically by Python when an object is created.	To initialize the state of a new object by setting its instance attributes.
self	The conventional name for the first parameter of an instance method.	Method	The first parameter in an instance method definition.	self	Passed automatically by Python when a method is called on an instance.	To provide a reference to the specific instance, allowing methods to access its unique attributes and other methods.
Instance Attribute	A variable belonging to a single, specific instance.	Instance	Inside a method (usually __init__) using self.attr = value.	N/A	Through an instance: my_obj.attr.	To store the unique state or data for each individual object.
Class Attribute	A variable belonging to the class itself, shared by all instances.	Class	Directly in the class body, outside any method.	N/A	Through the class (ClassName.attr) or an instance (my_obj.attr).	To store constants, default values, or any data shared across all

Component	Definition	Scope	How It's Defined	Key Parameter(s)	How to Access	Primary Role
						instances of a class.
Instance Method	A function defined within a class that operates on an instance.	Class	def method_name(self,...):	self	Through an instance: my_obj.method_name().	To define the behavior of an object, often by reading or modifying its instance attributes.
@classmethod	A decorator that transforms a method into a class method.	Class	@classmethod def method_name(cls,...):	cls	Through the class (ClassName.method_name()) or an instance (my_obj.method_name()).	To perform operations related to the class as a whole, such as creating alternative constructors (factory methods).
cls	The conventional name for the first parameter of a class method.	Method	The first parameter in a class method definition.	cls	Passed automatically by Python when a class method is called.	To provide a reference to the class object itself, allowing methods to access class attributes or call other class methods.
@staticmethod	A decorator that transforms a method into a static method.	Class	@staticmethod def method_name(...):	None (no implicit parameter)	Through the class (ClassName.method_name()) or an instance (my_obj.method_name()).	To group utility functions within a class's namespace that are logically related but do not depend on class or instance state.

Part III: The Four Pillars of OOP in Practice

With a detailed map of a class's internal structure established, the focus now shifts to the high-level principles that guide object-oriented design. These four pillars—Encapsulation, Inheritance, Polymorphism, and Abstraction—are not merely language features but are foundational concepts for building robust, scalable, and maintainable software. This section explores how each pillar is realized in Python, emphasizing idiomatic practices.

Section 3.1: Encapsulation - Protecting Your Data

Encapsulation is the principle of bundling data (attributes) and the methods that operate on that data within a single unit, the class, and restricting direct access to some of the object's components. This serves two main purposes: it protects an object's internal state from accidental or unauthorized modification, and it hides implementation details, allowing the internal workings of a class to change without affecting the code that uses it.

The Pythonic Philosophy of Access Control

Unlike languages such as Java or C++, Python does not have keywords like private or protected to strictly enforce data hiding. Instead, it operates on a philosophy often summarized as, "we are all consenting adults here". This means that access control is primarily based on convention and programmer discipline rather than rigid language-level restrictions.
Python uses naming conventions with leading underscores to signal the intended visibility of an attribute or method:

- **Public**: Attributes without a leading underscore (e.g., name) are considered part of the public API and are meant to be accessed freely.
- **Protected**: Attributes with a single leading underscore (e.g., _balance) are, by convention, treated as non-public. This signals to other developers that the attribute is intended for internal use within the class or its subclasses and should not be accessed directly from outside. The interpreter does not enforce this; it is purely a convention.
- **Private**: Attributes with a double leading underscore (e.g., __api_key) trigger a mechanism called **name mangling**.

Name Mangling Explained

When the Python interpreter encounters an attribute name starting with two or more underscores and at most one trailing underscore, it automatically renames it to _ClassName__attributeName. For example, in a class named BankAccount, an attribute __balance would be internally stored as _BankAccount__balance.
This is not true privacy, as the attribute can still be accessed using its mangled name. The primary purpose of name mangling is to prevent accidental name collisions when a subclass defines an attribute with the same name as one in its parent class. It makes it harder, but not impossible, to access the attribute from outside the class.

```
class BankAccount:
    def __init__(self, initial_deposit):
        # This attribute will be name-mangled to _BankAccount__balance
        self.__balance = initial_deposit
```

```
    def get_balance(self):
        return self.__balance

account = BankAccount(1000)

# Direct access fails due to name mangling
try:
    print(account.__balance)
except AttributeError as e:
    print(e)  # Output: 'BankAccount' object has no attribute
'__balance'

# Access is still possible via the mangled name (discouraged)
print(account._BankAccount__balance) # Output: 1000
```

Managed Attributes: The @property Decorator

The most Pythonic and powerful way to implement encapsulation and controlled attribute access is not through explicit get_...() and set_...() methods, but by using the built-in @property decorator. This feature allows a method to be accessed as if it were a regular attribute, providing a clean and intuitive API while hiding the internal implementation logic.
The @property decorator turns a method into a "getter." Corresponding "setter" and "deleter" logic can be attached using the @<property_name>.setter and @<property_name>.deleter decorators, respectively. This approach allows a developer to start with a simple public attribute and later add access logic without changing the class's public interface, thus preventing breaking changes in code that uses the class.

```
class Employee:
    def __init__(self, name, salary):
        self._name = name
        # Use the setter for initial validation
        self.salary = salary

    @property
    def salary(self):
        """The 'getter' for the salary property."""
        return self._salary

    @salary.setter
    def salary(self, value):
        """The 'setter' for the salary property with validation."""
        if value < 0:
            raise ValueError("Salary cannot be negative.")
        self._salary = value

    @salary.deleter
    def salary(self):
```

```
        """The 'deleter' for the salary property."""
        del self._salary

emp = Employee("Jane Doe", 50000)

# Accessing the salary as if it were a public attribute (calls the
getter)
print(emp.salary)  # Output: 50000

# Setting the salary (calls the setter, which performs validation)
emp.salary = 60000
print(emp.salary)  # Output: 60000

# This will trigger the validation in the setter and raise an error
try:
    emp.salary = -100
except ValueError as e:
    print(e)  # Output: Salary cannot be negative.
```

This pattern perfectly embodies Pythonic encapsulation: the interface is simple and direct (emp.salary), but the implementation is controlled and protected.

Section 3.2: Inheritance - Building Hierarchies

Inheritance is a mechanism that allows a new class (the **subclass** or **child class**) to be based on an existing class (the **superclass** or **parent class**), inheriting its attributes and methods. This models an **"is-a"** relationship—for example, a Manager *is a* type of Employee. Inheritance is a cornerstone of code reuse and the creation of logical, hierarchical class structures.

Method Overriding and Extending

A subclass is not limited to only using the methods it inherits. It can provide its own specific implementation for a method that is already defined in its superclass. This is known as **method overriding**. When the method is called on an instance of the subclass, the subclass's version is executed.
Often, the goal is not to completely replace the parent's method, but to **extend** it by adding new functionality while still executing the original logic from the parent. This is achieved using the super() function.

The super() Function

The super() function provides a way to call methods from a parent class. It returns a temporary proxy object of the superclass that allows access to its methods. This is particularly crucial within __init__, where a subclass must call the superclass's __init__ to ensure that all inherited attributes are properly initialized.

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

```
        self.employee_id = employee_id

    def display(self):
        return f"ID: {self.employee_id}, Name: {self.name}"

class Manager(Employee):
    def __init__(self, name, employee_id, department):
        # Call the parent's __init__ to handle name and employee_id
        super().__init__(name, employee_id)
        # Add subclass-specific attribute
        self.department = department

    # Override the display method to add more information
    def display(self):
        # Extend the parent's method by calling it with super()
        base_info = super().display()
        return f"{base_info}, Department: {self.department}"

mgr = Manager("John Smith", "MGR01", "Engineering")
print(mgr.display()) # Output: ID: MGR01, Name: John Smith,
Department: Engineering
```

Multiple Inheritance and Method Resolution Order (MRO)

Python is one of the few mainstream languages that supports **multiple inheritance**, where a class can inherit from more than one parent class. This can be powerful but also introduces complexity, particularly the "diamond problem": if class D inherits from B and C, and both B and C inherit from A, which version of a method defined in A should D use?
Python resolves this ambiguity deterministically using the **Method Resolution Order (MRO)**. The MRO is a linearized, predictable sequence of all superclasses that Python follows when searching for an attribute or method. This order is calculated using an algorithm called **C3 linearization**. The algorithm ensures that if a class inherits from multiple parents, the parents are checked in the order they are listed, and each ancestor class appears only once in the sequence.
A class's MRO can be inspected using the __mro__ attribute or the mro() method.

```
class A:
    def who_am_i(self):
        print("I am an A")

class B(A):
    def who_am_i(self):
        print("I am a B")

class C(A):
    def who_am_i(self):
        print("I am a C")
```

```
class D(B, C):
    pass

d = D()
d.who_am_i() # Output: I am a B

# Inspect the MRO for class D
print(D.mro())
# Output:
```

The MRO for D is ``. When d.who_am_i() is called, Python searches this list in order. It finds the method in B first and executes it.
This leads to a critical understanding of super(). In the context of multiple inheritance, super() does not simply mean "call the parent." It means "call the **next** method in the MRO chain from the current class's position". This sophisticated behavior allows for the creation of "cooperative" methods in complex hierarchies, where each method in the chain calls super() to ensure that all implementations in the MRO are executed in the correct order.

Section 3.3: Polymorphism - One Interface, Many Forms

Polymorphism, from the Greek for "many forms," is the principle that a single interface can be used to represent different underlying data types. It allows objects of different classes to be treated as objects of a common type, enabling functions to be written in a general way that works with a variety of objects.

Duck Typing: Python's Primary Polymorphic Style

Python's dynamic nature leads to a unique and powerful style of polymorphism known as **duck typing**. The name comes from the phrase, "If it walks like a duck and quacks like a duck, then it must be a duck".
In practice, this means Python does not check an object's explicit type to determine if it is suitable for an operation. Instead, it only cares whether the object has the necessary attributes or methods to perform the operation. If a function expects an object with a .fly() method, it will work with any object that provides that method, regardless of its class or inheritance hierarchy.

```
class Duck:
    def fly(self):
        print("Duck flying")

class Airplane:
    def fly(self):
        print("Airplane flying")

class Whale:
    def swim(self):
        print("Whale swimming")

# This function demonstrates polymorphism via duck typing.
# It doesn't care about the type of 'thing', only that it has a 'fly'
```

```
method.
def make_it_fly(thing):
    thing.fly()

duck = Duck()
plane = Airplane()
whale = Whale()

make_it_fly(duck)   # Output: Duck flying
make_it_fly(plane)  # Output: Airplane flying

try:
    make_it_fly(whale) # This will raise an AttributeError
except AttributeError as e:
    print(e) # Output: 'Whale' object has no attribute 'fly'
```

This approach promotes loose coupling and flexibility, as new classes can work with existing functions without needing to inherit from a specific base class.

Polymorphism through Inheritance

Python also supports the more traditional form of polymorphism found in statically-typed languages, which is achieved through inheritance and method overriding. In this model, a function is designed to work with a base class type. It can then accept any object of a subclass of that base class, and when a method is called, Python will dynamically execute the correct overridden version from the subclass.

```
class Document:
    def render(self):
        raise NotImplementedError("Subclass must implement this
method")

class PdfDocument(Document):
    def render(self):
        return "Rendering PDF document."

class WordDocument(Document):
    def render(self):
        return "Rendering Word document."

# This function expects any object that is a 'Document'.
def display_document(doc):
    print(doc.render())

documents =

for doc in documents:
    display_document(doc)
# Output:
```

```
# Rendering PDF document.
# Rendering Word document.
```

Section 3.4: Abstraction - Hiding Complexity

Abstraction is the principle of hiding complex implementation details and exposing only the essential, relevant features of an object. It simplifies interaction with an object by providing a clean, high-level interface. An abstract class serves as a formal definition of this interface, acting as a "contract" that concrete classes must follow.

Abstract Base Classes (ABCs)

While duck typing provides an informal contract based on behavior, Python's abc module allows for the creation of formal **Abstract Base Classes (ABCs)**. An ABC is a class that cannot be instantiated itself. Its purpose is to serve as a blueprint for subclasses, enforcing that they implement a specific set of methods.

The @abstractmethod Decorator

The @abstractmethod decorator is used within an ABC to declare methods that must be implemented by any concrete subclass. If a subclass inherits from an ABC but fails to override all of its abstract methods, it too is considered abstract, and any attempt to instantiate it will result in a TypeError.
This provides the best of both worlds: it allows for the flexibility of polymorphism while adding a layer of safety that guarantees all objects conforming to the interface will have the required methods.

```
from abc import ABC, abstractmethod

# Define an Abstract Base Class for a data source.
# It defines the 'contract' for all data sources.
class DataSource(ABC):
    @abstractmethod
    def read_data(self):
        """Read and return data from the source."""
        pass

    @abstractmethod
    def write_data(self, data):
        """Write data to the source."""
        pass

# A concrete implementation of the contract.
class DatabaseSource(DataSource):
    def read_data(self):
        print("Reading data from the database...")
        return {"data": "db_content"}
```

```
    def write_data(self, data):
        print(f"Writing {data} to the database...")

# Another concrete implementation.
class FileSource(DataSource):
    def read_data(self):
        print("Reading data from a file...")
        return {"data": "file_content"}

    def write_data(self, data):
        print(f"Writing {data} to a file...")

# This class fails to implement the full contract.
class IncompleteSource(DataSource):
    def read_data(self):
        return {"data": "incomplete"}
    # Missing the write_data method.

db = DatabaseSource()
db.write_data("new_record") # Output: Writing new_record to the
database...

# Attempting to instantiate the incomplete class will fail.
try:
    incomplete = IncompleteSource()
except TypeError as e:
    print(e)
    # Output: Can't instantiate abstract class IncompleteSource with
abstract method write_data
```

By using ABCs, developers can create robust frameworks that ensure plugins or user-provided components adhere to a required API, catching errors early in the development process rather than at runtime.

Part IV: Advanced Concepts and Pythonic OOP

Having mastered the foundational principles and internal mechanics of Python classes, the final step toward expertise involves learning to make custom objects behave as naturally and intuitively as Python's built-in types. This is achieved through the implementation of special "dunder" methods. Furthermore, advanced object-oriented design involves making strategic choices about how to structure relationships between classes, often favoring more flexible patterns over rigid inheritance hierarchies.

Section 4.1: Dunder (Magic) Methods - Customizing Class Behavior

In Python, methods with names that start and end with double underscores (e.g., __init__, __add__) are known as **dunder methods** or **magic methods**. These are not meant to be called

directly by the programmer. Instead, Python calls them implicitly in response to specific syntax or built-in functions. For example, the expression a + b is translated by the interpreter into a call to a.__add__(b). By implementing these dunder methods in a custom class, an object can be made to support idiomatic Python operations, making its usage more intuitive.

Object Representation: __str__ vs. __repr__

Two of the most important dunder methods control how an object is represented as a string. They serve different audiences and purposes:

- **__repr__(self)**: This method should return an **unambiguous**, official string representation of the object. It is intended for the developer during debugging and logging. The primary goal is to provide enough information to, ideally, recreate the object. The built-in repr() function and the interactive console's output call this method. A common convention is to return a string that looks like a constructor call, e.g., Point(x=1, y=2).
- **__str__(self)**: This method should return an **informal**, user-friendly string representation of the object. It is intended for the end-user of the program. The print() function and the str() type constructor call this method.

A crucial rule of thumb is to always implement __repr__. If __str__ is not defined for a class, Python will fall back and use the __repr__ implementation instead.

```
import datetime

class Event:
    def __init__(self, name, date):
        self.name = name
        self.date = date

    def __str__(self):
        # User-friendly, readable output.
        return f"Event '{self.name}' on {self.date.strftime('%B %d,
%Y')}"

    def __repr__(self):
        # Unambiguous, developer-focused output.
        # Should ideally be a valid Python expression to recreate the
object.
        return f"Event('{self.name}', datetime.date({self.date.year},
{self.date.month}, {self.date.day}))"

# Create an instance
e = Event("Team Meeting", datetime.date(2024, 9, 15))

# The print() function uses __str__
print(e)  # Output: Event 'Team Meeting' on September 15, 2024

# The interactive console or repr() function uses __repr__
print(repr(e)) # Output: Event('Team Meeting', datetime.date(2024, 9,
15))
```

```
# A list's __str__ method uses the __repr__ of its contents
print([e]) # Output:
```

Operator Overloading

Operator overloading allows custom objects to work with Python's built-in operators like +, *, ==, and <. This is achieved by implementing the corresponding dunder methods.

- **Arithmetic Operators**: __add__(self, other), __sub__(self, other), __mul__(self, other), etc.
- **Comparison Operators**: __eq__(self, other) for ==, __ne__(self, other) for !=, __lt__(self, other) for <, __gt__(self, other) for >, etc.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

    # Overload the + operator
    def __add__(self, other):
        if not isinstance(other, Vector):
            return NotImplemented
        return Vector(self.x + other.x, self.y + other.y)

    # Overload the == operator
    def __eq__(self, other):
        if not isinstance(other, Vector):
            return False
        return self.x == other.x and self.y == other.y

v1 = Vector(2, 3)
v2 = Vector(3, 4)
v3 = Vector(5, 7)

# Python translates 'v1 + v2' into 'v1.__add__(v2)'
result = v1 + v2
print(result)  # Output: Vector(5, 7)

# Python translates 'result == v3' into 'result.__eq__(v3)'
print(result == v3) # Output: True
```

Emulating Containers and Other Behaviors

Dunder methods also allow custom objects to emulate the behavior of built-in container types or other Python constructs:

- **Container Methods**: __len__(self) for len(), __getitem__(self, key) for index/key access (obj[key]), __setitem__(self, key, value) for assignment (obj[key] = value), and __delitem__(self, key) for deletion (del obj[key]).
- **Callable Objects**: Implementing __call__(self, *args, **kwargs) allows an instance of a class to be called as if it were a function.
- **Context Managers**: Implementing __enter__(self) and __exit__(self, exc_type, exc_val, exc_tb) allows an object to be used with the with statement, which is ideal for managing resources like file handles or network connections.

Section 4.2: Designing for Success

Writing object-oriented code is not just about using the class keyword; it is about designing systems of interacting objects that are flexible, maintainable, and easy to understand.

Composition over Inheritance

While inheritance is a powerful tool for modeling "is-a" relationships, it can lead to rigid, deeply nested hierarchies that are difficult to manage. An alternative and often more flexible approach to code reuse is **composition**.
Composition models a **"has-a"** relationship. Instead of a class inheriting features from a parent, it is "composed" of other objects, holding instances of them as attributes. For example, a Car does not inherit from an Engine; rather, a Car *has an* Engine.

```
class Engine:
    def __init__(self, horsepower):
        self.horsepower = horsepower

    def start(self):
        print(f"Engine with {self.horsepower}hp starting.")

class Car:
    def __init__(self, make, model, horsepower):
        # Composition: The Car object contains an Engine object.
        self.engine = Engine(horsepower)
        self.make = make
        self.model = model

    def start(self):
        print(f"Starting the {self.make} {self.model}...")
        # Delegation: The Car delegates the 'start' behavior to its
engine component.
        self.engine.start()

my_car = Car("Ford", "Mustang", 450)
my_car.start()
# Output:
# Starting the Ford Mustang...
# Engine with 450hp starting.
```

Favoring composition over inheritance often leads to more modular and flexible designs. Components can be easily swapped (e.g., replacing the Engine object in the Car with a different type of engine) without altering the class hierarchies.

Best Practices for Pythonic Classes

To conclude this guide, here is a summary of best practices for writing effective, idiomatic, and expert-level object-oriented code in Python:

1. **Keep Classes Focused**: Adhere to the Single Responsibility Principle. Each class should have one primary responsibility and encapsulate it entirely. A class that manages database connections should not also handle file I/O.
2. **Use Meaningful Names**: Class, attribute, and method names should be descriptive and clear. A class representing a user should be named User, not DataContainer.
3. **Prefer Properties Over Getters/Setters**: Avoid writing explicit get_x() and set_x() methods. Start with public attributes and introduce @property decorators only when you need to add logic to attribute access, ensuring a clean and stable public API.
4. **Distinguish Class vs. Instance Attributes**: Use class attributes for constants and shared state. Use instance attributes for data unique to each object. Be wary of using mutable types for class attributes.
5. **Use Dunder Methods for Integration**: Implement dunder methods (__str__, __repr__, __eq__, etc.) to make your custom objects behave like native Python types. This creates intuitive and Pythonic APIs.
6. **Favor Composition Over Inheritance**: Use inheritance for true "is-a" relationships. For "has-a" relationships and code reuse, prefer composition for its greater flexibility and reduced coupling.
7. **Use ABCs for Formal Interfaces**: When designing a framework or library that requires other classes to conform to a specific API, use Abstract Base Classes from the abc module to create formal, enforceable contracts.

By internalizing these principles and patterns, a developer can move beyond simply knowing the syntax of classes to truly mastering the art of object-oriented design in Python, creating code that is not only functional but also elegant, robust, and maintainable.

Works cited

1. The Python Tutorial — Python 3.13.7 documentation, https://docs.python.org/3/tutorial/index.html 2. Basic Object Oriented Programming (OOPs) Concepts in Python - Analytics Vidhya, https://www.analyticsvidhya.com/blog/2020/09/object-oriented-programming/ 3. Python OOP Concepts - GeeksforGeeks, https://www.geeksforgeeks.org/python/python-oops-concepts/ 4. 9. Classes — Python 3.13.7 documentation, https://docs.python.org/3/tutorial/classes.html 5. Object-Oriented Programming (OOP) in Python – Real Python, https://realpython.com/python3-object-oriented-programming/ 6. Mastering Object-Oriented Programming in Python: Applications and Key Concepts, https://www.quickstart.com/blog/software-engineering/mastering-oop-python-applications/ 7. Python OOP Fundamentals: Understanding the Basics of Object Oriented Programming | by Sadaf Saleem | Medium, https://medium.com/@sadafsaleem5815/python-oop-fundamentals-understanding-the-basics-of-object-oriented-programming-c649552ac490 8. Python Basics: Object-Oriented Programming,

https://realpython.com/courses/python-basics-oop/ 9. Introduction to Python Object Oriented Programming: Python Basics - YouTube, https://www.youtube.com/watch?v=9-7khkt3ReY 10. What Is Object-Oriented Programming (OOP)? (Video) - Real Python, https://realpython.com/videos/what-object-oriented-programming-oop/ 11. Python __init__: An Overview - Great Learning, https://www.mygreatlearning.com/blog/python-init/ 12. What Is the Purpose of __init__ in Python? - StrataScratch, https://www.stratascratch.com/blog/what-is-the-purpose-of-__init__-in-python/ 13. Understanding Python's init Method: Object Initialization in Depth - Data Science & Beyond, https://ishanjainoffical.medium.com/understanding-pythons-init-method-object-initialization-in-depth-cc16f6e1e322 14. Understanding the Use Case of __init__ in Python | by Piyush Kashyap | Medium, https://medium.com/@piyushkashyap045/understanding-the-use-case-of-init-in-python-3de27bf9b529 15. What does __init__ and self do in python? : r/learnpython - Reddit, https://www.reddit.com/r/learnpython/comments/19aghsb/what_does_init_and_self_do_in_python/ 16. Understanding classes, init and self - Python Discussions, https://discuss.python.org/t/understanding-classes-init-and-self/46089 17. Understanding Class and Instance Attributes in Python | by Eunice Adewusi - Medium, https://medium.com/@euniceadewusic/understanding-class-and-instance-attributes-in-python-e3ae12eb3e66 18. realpython.com, https://realpython.com/python-repr-vs-str/#:~:text=__repr__()%20method%20returns%20a,the%20user%20of%20the%20program. 19. Python Classes: The Power of Object-Oriented Programming – Real ..., https://realpython.com/python-classes/ 20. Python Attributes: Class vs. Instance Explained - Built In, https://builtin.com/software-engineering-perspectives/python-attributes 21. Python Attributes: Class Vs. Instance Explained - GeeksforGeeks, https://www.geeksforgeeks.org/python/python-attributes-class-vs-instance-explained/ 22. python - What is the difference between class and instance attributes ..., https://stackoverflow.com/questions/207000/what-is-the-difference-between-class-and-instance-attributes 23. Class and Instance Attributes in Python - Analytics Vidhya, https://www.analyticsvidhya.com/blog/2024/02/class-and-instance-attributes-in-python/ 24. Class and Instance Attributes in Python - GeeksforGeeks, https://www.geeksforgeeks.org/python/class-instance-attributes-python/ 25. Understanding Object and Class Attributes in Python | by Shukufa Bayramzada - Medium, https://medium.com/@shukufabayramzada/understanding-object-and-class-attributes-in-python-1da88bf1a9de 26. Python's Instance, Class, and Static Methods Demystified – Real ..., https://realpython.com/instance-class-and-static-methods-demystified/ 27. OOP Method Types in Python: @classmethod vs @staticmethod vs Instance Methods, https://realpython.com/courses/python-method-types/ 28. Built-in Functions — Python 3.13.7 documentation, https://docs.python.org/3/library/functions.html 29. Class method vs Static method in Python - GeeksforGeeks, https://www.geeksforgeeks.org/python/class-method-vs-static-method-python/ 30. Class Method vs Static Method vs Instance Method in Python ..., https://www.geeksforgeeks.org/python/class-method-vs-static-method-vs-instance-method-in-python/ 31. python - Meaning of @classmethod and @staticmethod for beginner ..., https://stackoverflow.com/questions/12179271/meaning-of-classmethod-and-staticmethod-for-beginner 32. staticmethod — Python Reference (The Right Way) 0.1 documentation - Read the Docs, https://python-reference.readthedocs.io/en/latest/docs/functions/staticmethod.html 33. Python: The four pillars of Object-Oriented Programming - Medium,

https://medium.com/codex/python-the-four-pillars-of-object-oriented-programming-f6058d5471ba 34. Encapsulation in Python - GeeksforGeeks, https://www.geeksforgeeks.org/python/encapsulation-in-python/ 35. Encapsulation in Python: A Comprehensive Guide - DataCamp, https://www.datacamp.com/tutorial/encapsulation-in-python-object-oriented-programming 36. Should I use name mangling in Python? - Stack Overflow, https://stackoverflow.com/questions/7456807/should-i-use-name-mangling-in-python 37. Python Private Methods Explained | DataCamp, https://www.datacamp.com/tutorial/python-private-methods-explained 38. name mangling | Python Glossary, https://realpython.com/ref/glossary/name-mangling/ 39. encapsulation in python - Pythonspot, https://pythonspot.com/encapsulation/ 40. Is name mangling still still the best way to best, the only way and the future way for private instance variables? - Python Discussions, https://discuss.python.org/t/is-name-mangling-still-still-the-best-way-to-best-the-only-way-and-the-future-way-for-private-instance-variables/51272 41. Are getters and setters unpythonic? : r/Python - Reddit, https://www.reddit.com/r/Python/comments/37q38d/are_getters_and_setters_unpythonic/ 42. Getters and Setters: Manage Attributes in Python – Real Python, https://realpython.com/python-getter-setter/ 43. Python Property Decorator - TutorialsTeacher, https://www.tutorialsteacher.com/python/property-decorator 44. Python Property Decorator - @property - GeeksforGeeks, https://www.geeksforgeeks.org/python/python-property-decorator-property/ 45. Getting started with classes in Python Part 7: getters and setters | Ansys Developer Portal, https://developer.ansys.com/blog/getting-started-classes-python-part-7-getters-and-setters 46. Inheritance and Composition: A Python OOP Guide – Real Python, https://realpython.com/inheritance-composition-python/ 47. inheritance | Python Glossary, https://realpython.com/ref/glossary/inheritance/ 48. Method Overriding in Python (47/100 Days of Python) | by Martin Mirakyan | Medium, https://martinxpn.medium.com/method-overriding-in-python-47-100-days-of-python-24c1a34d27d1?source=user_profile---------8---------- 49. Method Overriding in Python - GeeksforGeeks, https://www.geeksforgeeks.org/python/method-overriding-in-python/ 50. Method overriding in Python - The Digital Cat, https://www.thedigitalcatonline.com/blog/2014/05/19/method-overriding-in-python/ 51. Python super() - GeeksforGeeks, https://www.geeksforgeeks.org/python/python-super/ 52. Supercharge Your Classes With Python super(), https://realpython.com/python-super/ 53. Python | super() function with multilevel inheritance - GeeksforGeeks, https://www.geeksforgeeks.org/python/python-super-function-with-multilevel-inheritance/ 54. How does python resolve conflicts in multiple inheritance? | by Shashi Kant - Medium, https://medium.com/@shashikantrbl123/how-does-python-resolve-conflicts-in-multiple-inheritance-a407839be714 55. Method resolution order in Python Inheritance - GeeksforGeeks, https://www.geeksforgeeks.org/python/method-resolution-order-in-python-inheritance/ 56. Understanding Method Resolution Order in Python - Codefinity, https://codefinity.com/blog/Understanding-Method-Resolution-Order-in-Python 57. method resolution order (MRO) | Python Glossary, https://realpython.com/ref/glossary/mro/ 58. Polymorphism in Python - GeeksforGeeks, https://www.geeksforgeeks.org/python/polymorphism-in-python/ 59. Polymorphism - Python Tutorial - pythonspot, https://pythonspot.com/polymorphism/ 60. Duck Typing in Python: Writing Flexible and Decoupled Code – Real ..., https://realpython.com/duck-typing-python/ 61. Duck

Typing in Python - GeeksforGeeks, https://www.geeksforgeeks.org/python/duck-typing-in-python/ 62. What is the difference between polymorphism and duck typing? - Stack Overflow, https://stackoverflow.com/questions/11502433/what-is-the-difference-between-polymorphism-and-duck-typing 63. Examining Duck Typing and Polymorphism (Video) - Real Python, https://realpython.com/videos/examining-duck-typing-polymorphism/ 64. abc — Abstract Base Classes — Python 3.13.7 documentation, https://docs.python.org/3/library/abc.html 65. Understanding Abstract Classes and Abstract Methods in Python | CodeSignal Learn, https://codesignal.com/learn/courses/revisiting-oop-concepts-in-python/lessons/understanding-abstract-classes-and-abstract-methods-in-python 66. Python ABCs- The Complete Guide to Abstract Base Classes - Machine Learning Plus, https://www.machinelearningplus.com/python/python-abcs-the-complete-guide-to-abstract-base-classes/ 67. abstract base class (ABC) | Python Glossary – Real Python, https://realpython.com/ref/glossary/abstract-base-class/ 68. Abstract Classes in Python - GeeksforGeeks, https://www.geeksforgeeks.org/python/abstract-classes-in-python/ 69. python @abstractmethod decorator - Stack Overflow, https://stackoverflow.com/questions/7196376/python-abstractmethod-decorator 70. Python's Magic Methods: Leverage Their Power in Your Classes - Real Python, https://realpython.com/python-magic-methods/ 71. Every dunder method in Python - Reddit, https://www.reddit.com/r/Python/comments/1bioxer/every_dunder_method_in_python/ 72. When to Use .__repr__() vs .__str__() in Python (Summary), https://realpython.com/lessons/repr-vs-str-summary/ 73. [Question] about str() vs repr() : r/learnpython - Reddit, https://www.reddit.com/r/learnpython/comments/1hjb24/question_about_str_vs_repr/ 74. __str__ & __repr__: Change String Representation In Python : r/Python - Reddit, https://www.reddit.com/r/Python/comments/12kroqt/str_repr_change_string_representation_in_python/ 75. What is the difference between __str__ and __repr__? - Stack Overflow, https://stackoverflow.com/questions/1436703/what-is-the-difference-between-str-and-repr 76. What Are Inheritance and Composition? (Video) - Real Python, https://realpython.com/videos/what-are-inheritance-and-composition/ 77. OOP in Python. Hello and welcome! Object-oriented… | by Taylor Berukoff | Medium, https://medium.com/@taylorberukoff/object-oriented-principles-in-python-ffb44453270c