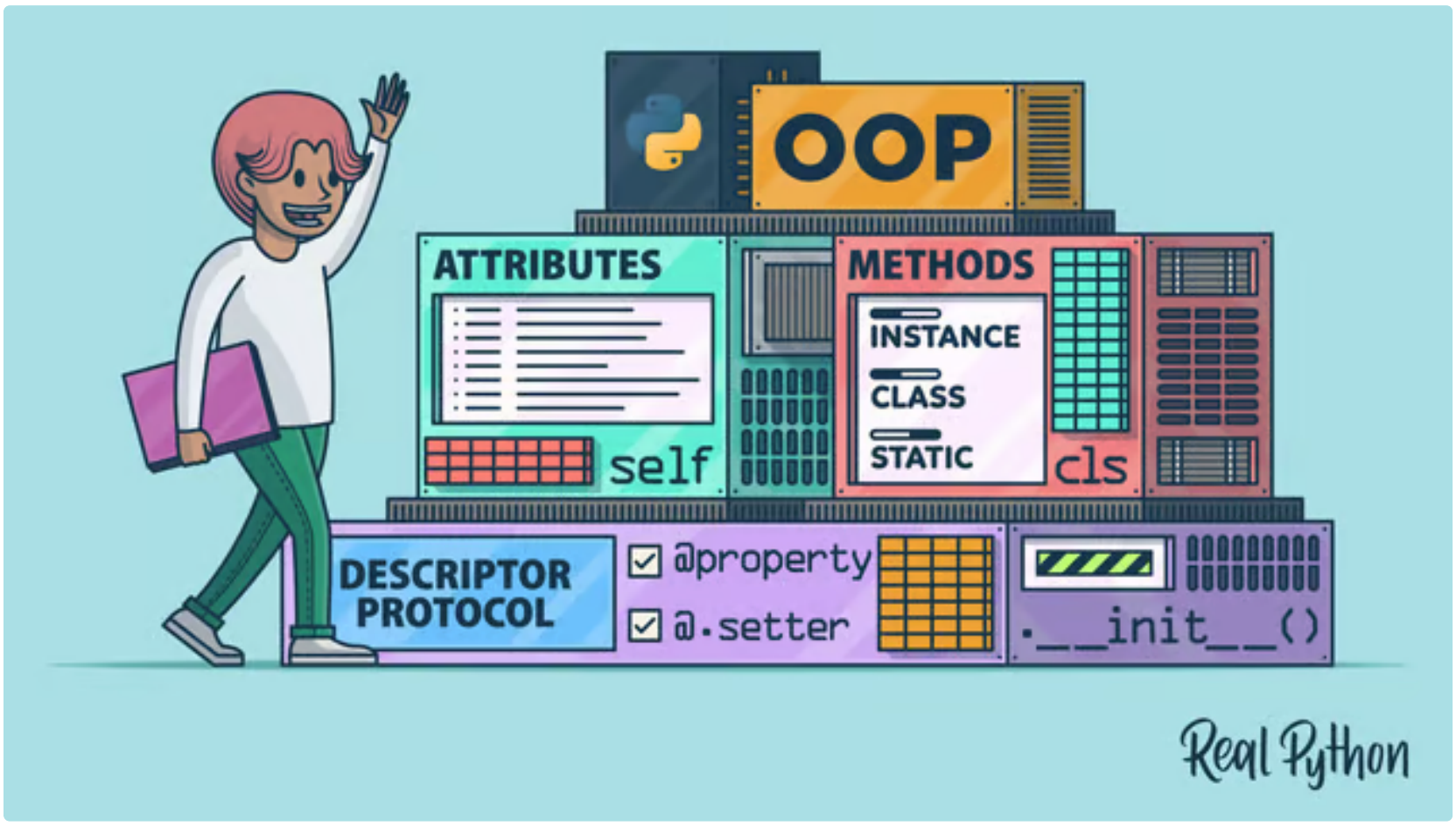


Python Classes: The Power of Object-Oriented Programming

by Leodanis Pozo Ramos Apr 26, 2023 22 Comments

intermediate python

Mark as Completed

Tweet Share Email

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Inheritance and Internals: Object-Oriented Programming in Python**

Python supports the **object-oriented** programming paradigm through **classes**. They provide an elegant way to define reusable pieces of code that encapsulate data and behavior in a single entity. With classes, you can quickly and intuitively model real-world objects and solve complex problems.

If you're new to classes, need to refresh your knowledge, or want to dive deeper into them, then this tutorial is for you!

In this tutorial, you'll learn how to:

- Define Python classes with the `class` **keyword**
- Add state to your classes with **class** and **instance attributes**
- Provide **behavior** to your classes with **methods**
- Use **inheritance** to build hierarchies of classes
- Provide **interfaces** with **abstract classes**

To get the most out of this tutorial, you should know about Python variables, data types, and functions. Some experience with object-oriented programming (OOP) is also a plus. Don't worry if you're not an OOP expert yet. In this tutorial, you'll learn the key concepts that you need to get started and more. You'll also write several practical examples to help reinforce your knowledge of Python classes.

Free Bonus: Click here to download your sample code for building powerful object blueprints with classes in Python.

Getting Started With Python Classes

Python is a multiparadigm programming language that supports object-oriented programming (OOP) through classes that you can define with the `class` keyword. You can think of a class as a piece of code that specifies the **data** and **behavior** that represent and model a particular type of object.

What is a class in Python? A common analogy is that a class is like the blueprint for a house. You can use the blueprint to create several houses and even a complete neighborhood. Each concrete house is an **object** or **instance** that's derived from the blueprint.

Each instance can have its own properties, such as color, owner, and interior design. These properties carry what's commonly known as the object's state. Instances can also have different behaviors, such as locking the doors and windows, opening the garage door, turning the lights on and off, watering the garden, and more.

In OOP, you commonly use the term **attributes** to refer to the properties or data associated with a specific object of a given class. In Python, attributes are variables defined inside a class with the purpose of storing all the required data for the class to work.

Similarly, you'll use the term **methods** to refer to the different behaviors that objects will show. Methods are functions that you define within a class. These functions typically operate on or with the attributes of the underlying instance or class. Attributes and methods are collectively referred to as **members** of a class or object.

You can write fully functional classes to model the real world. These classes will help you better organize your code and solve complex programming problems.

For example, you can use classes to create objects that emulate people, animals, vehicles, books, buildings, cars, or other objects. You can also model virtual objects, such as a web server, directory tree, chatbot, file manager, and more.

Finally, you can use classes to build **class hierarchies**. This way, you'll promote code reuse and remove repetition throughout your codebase.

In this tutorial, you'll learn a lot about classes and all the cool things that you can do with them. To kick things off, you'll start by defining your first class in Python. Then you'll dive into other topics related to instances, attributes, and methods.

Defining a Class in Python

To define a class, you need to use the `class` keyword followed by the class name and a colon, just like you'd do for other compound statements in Python. Then you must define the class body, which will start at the next indentation level:

Python

```
class ClassName:
    # Class body
    pass
```

In a class body, you can define attributes and methods as needed. As you already learned, attributes are variables that hold the class data, while methods are functions that provide behavior and typically act on the class data.

> **Note:** In Python, the body of a given class works as a namespace where attributes and methods live. You can only access those attributes and methods through the class or its objects.

As an example of how to define attributes and methods, say that you need a `Circle` class to model different circles in a drawing application. Initially, your class will have a single attribute to hold the radius. It'll also have a method to calculate the circle's area:

Python

```python
# circle.py

import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return round(math.pi * self.radius ** 2, 2)
```

In this code snippet, you define `Circle` using the `class` keyword. Inside the class, you write two methods. The `.__init__()` method has a special meaning in Python classes. This method is known as the object initializer because it defines and sets the initial values for your attributes. You'll learn more about this method in the Instance Attributes section.

The second method of `Circle` is conveniently named `.calculate_area()` and will compute the area of a specific circle by using its radius. It's common for method names to contain a verb, such as *calculate*, to describe an action the method performs. In this example, you've used the `math` module to access the `pi` constant as it's defined in that module.

> **Note:** In Python, the first argument of most methods is `self`. This argument holds a reference to the current object so that you can use it inside the class. You'll learn more about this argument in the section on instance methods with `self`.

Cool! You've written your first class. Now, how can you use this class in your code to represent several concrete circles? Well, you need to instantiate your class to create specific circle objects from it.

Creating Objects From a Class in Python

The action of creating concrete objects from an existing class is known as **instantiation**. With every instantiation, you create a new object of the target class. To get your hands dirty, go ahead and make a couple of instances of `Circle` by running the following code in a Python REPL session:

Python >>>

```
>>> from circle import Circle

>>> circle_1 = Circle(42)
>>> circle_2 = Circle(7)

>>> circle_1
<__main__.Circle object at 0x102b835d0>
>>> circle_2
<__main__.Circle object at 0x1035e3910>
```

To create an object of a Python class like `Circle`, you must call the `Circle()` class constructor with a pair of parentheses and a set of appropriate arguments. What arguments? In Python, the class constructor accepts the same arguments as the `.__init__()` method. In this example, the `Circle` class expects the `radius` argument.

Calling the class constructor with different argument values will allow you to create different objects or instances of the target class. In the above example, `circle_1` and `circle_2` are separate instances of `Circle`. In other words, they're two different and concrete circles, as you can conclude from the code's output.

Great! You already know how to create objects of an existing class by calling the class constructor with the required arguments. Now, how can you access the attributes and methods of a given class? That's what you'll learn in the next section.

Accessing Attributes and Methods

In Python, you can access the attributes and methods of an object by using **dot notation** with the **dot operator**. The following snippet of code shows the required syntax:

Python

```
obj.attribute_name

obj.method_name()
```

Note that the dot (`.`) in this syntax basically means *give me the following attribute or method from this object*. The first line returns the value stored in the target attribute, while the second line accesses the target method so that you can call it.

> **Note:** Remember that to call a function or method, you need to use a pair of parentheses and a series of arguments, if applicable.

Now get back to your circle objects and run the following code:

Python >>>

```
>>> from circle import Circle

>>> circle_1 = Circle(42)
>>> circle_2 = Circle(7)

>>> circle_1.radius
42
>>> circle_1.calculate_area()
5541.77

>>> circle_2.radius
7
>>> circle_2.calculate_area()
153.94
```

In the first couple of lines after the instantiation, you access the `.radius` attribute on your `circle_1` object. Then you call the `.calculate_area()` method to calculate the circle's area. In the second pair of statements, you do the same but on the `circle_2` object.

You can also use dot notation and an assignment statement to change the current value of an attribute:

Python >>>

```
>>> circle_1.radius = 100
>>> circle_1.radius
100
>>> circle_1.calculate_area()
31415.93
```

Now the radius of `circle_1` is entirely different. When you call `.calculate_area()`, the result immediately reflects this change. You've changed the object's internal state or data, which typically impacts its behaviors or methods.

Naming Conventions in Python Classes

Before continuing diving into classes, you'll need to be aware of some important naming conventions that Python uses in the context of classes. Python is a flexible language that loves freedom and doesn't like to have explicit restrictions. Because of that, the language and the community rely on conventions rather than restrictions.

> **Note:** Most Python programmers follow the snake_case naming convention, which involves using underscores (_) to separate multiple words. However, the recommended naming convention for Python classes is the PascalCase, where each word is capitalized.

In the following two sections, you'll learn about two important naming conventions that apply to class attributes.

Public vs Non-Public Members

The first naming convention that you need to know about is related to the fact that Python doesn't distinguish between **private**, **protected**, and **public** attributes like Java and other languages do. In Python, all attributes are accessible in one way or another. However, Python has a well-established naming convention that you should use to communicate that an attribute or method isn't intended for use from outside its containing class or object.

The naming convention consists of adding a leading underscore to the member's name. So, in a Python class, you'll have the following convention:

Member	Naming	Examples
Public	Use the normal naming pattern.	`radius`, `calculate_area()`
Non-public	Include a leading underscore in names.	`_radius`, `_calculate_area()`

Public members are part of the official **interface** or **API** of your classes, while **non-public** members aren't intended to be part of that API. This means that you shouldn't use non-public members outside their defining class.

It's important to note that the second naming convention only *indicates* that the attribute isn't intended to be used directly from outside the containing class. It doesn't prevent direct access, though. For example, you can run `obj._name`, and you'll access the content of `._name`. However, this is bad practice, and you should avoid it.

Non-public members exist only to support the internal implementation of a given class and may be removed at any time, so you shouldn't rely on them. The existence of these members depends on how the class is implemented. So, you shouldn't use them directly in client code. If you do, then your code could break at any moment.

When writing classes, sometimes it's hard to decide if an attribute should be public or non-public. This decision will depend on how you want your users to use your classes. In most cases, attributes should be non-public to guarantee the safe use of your classes. A good approach will be to start with all your attributes as non-public and only make them public if real use cases appear.

Name Mangling

Another naming convention that you can see and use in Python classes is to add two leading underscores to attribute and method names. This naming convention triggers what's known as **name mangling**.

Name mangling is an automatic name transformation that prepends the class's name to the member's name, like in `_ClassName__attribute` or `_ClassName__method`. This results in name hiding. In other words, mangled names aren't available for direct access. They're not part of a class's public API.

For example, consider the following sample class:

Python >>>

```
>>> class SampleClass:
...     def __init__(self, value):
...         self.__value = value
...     def __method(self):
...         print(self.__value)
...

>>> sample_instance = SampleClass("Hello!")
>>> vars(sample_instance)
{'_SampleClass__value': 'Hello!'}

>>> vars(SampleClass)
mappingproxy({
    ...
    '__init__': <function SampleClass.__init__ at 0x105dfd4e0>,
    '_SampleClass__method': <function SampleClass.__method at 0x105dfd760>,
    '__dict__': <attribute '__dict__' of 'SampleClass' objects>,
    ...
})

>>> sample_instance = SampleClass("Hello!")
>>> sample_instance.__value
Traceback (most recent call last):
    ...
AttributeError: 'SampleClass' object has no attribute '__value'

>>> sample_instance.__method()
Traceback (most recent call last):
    ...
AttributeError: 'SampleClass' object has no attribute '__method'
```

In this class, `.__value` and `.__method()` have two leading underscores, so their names are mangled to `._SampleClass__value` and `._SampleClass__method()`, as you can see in the highlighted lines. Python has automatically added the prefix `_SampleClass` to both names. Because of this internal renaming, you can't access the attributes from outside the class using their original names. If you try to do it, then you get an `AttributeError`.

> **Note:** In the above example, you use the built-in `vars()` function, which returns a dictionary of all the members associated with the given object. This dictionary plays an important role in Python classes. You'll learn more about it in the `.__dict__` attribute section.

This internal behavior hides the names, creating the illusion of a private attribute or method. However, they're not strictly private. You can access them through their mangled names:

Python >>>

```
>>> sample_instance._SampleClass__value
'Hello!'

>>> sample_instance._SampleClass__method()
Hello!
```

It's still possible to access named-mangled attributes or methods using their mangled names, although this is bad practice, and you should avoid it in your code. If you see a name that uses this convention in someone's code, then don't try to force the code to use the name from outside its containing class.

Name mangling is particularly useful when you want to ensure that a given attribute or method won't get accidentally overwritten. It's a way to avoid naming conflicts between classes or subclasses. It's also useful to prevent subclasses from overriding methods that have been optimized for better performance.

Wow! Up to this point, you've learned the basics of Python classes and a bit more. You'll dive deeper into how Python classes work in a moment. But first, it's time to jump into some reasons why you should learn about classes and use them in your Python projects.

Understanding the Benefits of Using Classes in Python

Is it worth using classes in Python? Absolutely! Classes are the building blocks of object-oriented programming in Python. They allow you to leverage the power of Python while writing and organizing your code. By learning about classes, you'll be able to take advantage of all the benefits that they provide. With classes, you can:

- **Model and solve complex real-world problems:** You'll find many situations where the objects in your code map to real-world objects. This can help you think about complex problems, which will result in better solutions to your programming problems.

- **Reuse code and avoid repetition:** You can define hierarchies of related classes. The base classes at the top of a hierarchy provide common functionality that you can reuse later in the subclasses down the hierarchy. This allows you to reduce code duplication and promote code reuse.

- **Encapsulate related data and behaviors in a single entity:** You can use Python classes to bundle together related attributes and methods in a single entity, the object. This helps you better organize your code using modular and autonomous entities that you can even reuse across multiple projects.

- **Abstract away the implementation details of concepts and objects:** You can use classes to abstract away the implementation details of core concepts and objects. This will help you provide your users with intuitive interfaces (APIs) to process complex data and behaviors.

- **Unlock polymorphism with common interfaces:** You can implement a particular interface in several slightly different classes and use them interchangeably in your code. This will make your code more flexible and adaptable.

In short, Python classes can help you write more organized, structured, maintainable, reusable, flexible, and user-friendly code. They're a great tool to have under your belt. However, don't be tempted to use classes for everything in Python. In some situations, they'll overcomplicate your solutions.

> **Note:** In Python, the public attributes and methods of a class make up what you'll know as the class's **interface** or **application programming interface (API)**. You'll learn more about interfaces throughout this tutorial. Stay tuned!

In the following section, you'll explore some situations where you should avoid using classes in your code.

Deciding When to Avoid Using Classes

Python classes are pretty cool and powerful tools that you can use in multiple scenarios. Because of this, some people tend to overuse classes and solve all their coding problems using them. However, sometimes using a class isn't the best solution. Sometimes a couple of functions are enough.

In practice, you'll encounter a few situations in which you should avoid classes. For example, you shouldn't use regular classes when you need to:

- Store **only data**. Use a data class or a named tuple instead.
- Provide a **single method**. Use a function instead.

Data classes, enumerations, and named tuples are specially designed to store data. So, they might be the best solution if your class doesn't have any behavior attached.

If your class has a single method in its API, then you may not require a class. Instead, use a function unless you need to retain a certain state between calls. If more methods appear later, then you can always create a class. Remember the Python principle:

> Simple is better than complex. (Source)

Additionally, you should avoid creating custom classes to wrap up functionality that's available through built-in types or third-party classes. Use the type or third-party class directly.

You'll find many other general situations where you may not need to use classes in your Python code. For example, classes aren't necessary when you're working with:

- A small and **simple program** or **script** that doesn't require complex data structures or logic. In this case, using classes may be overkill.
- A **performance-critical** program. Classes add overhead to your program, especially when you need to create many objects. This may affect your code's general performance.
- A **legacy codebase**. If an existing codebase doesn't use classes, then you shouldn't introduce them. This will break the current coding style and disrupt the code's consistency.
- A team with a **different coding style**. If your current team doesn't use classes, then stick with their coding style. This will ensure consistency across the project.
- A codebase that uses **functional programming**. If a given codebase is currently written with a functional approach, then you shouldn't introduce classes. This will break the underlying coding paradigm.

You may find yourself in many other situations where using classes will be overkill. Classes are great, but don't turn them into a one-size-fits-all type of tool. Start your code as simply as possible. If the need for a class appears, then go for it.

Attaching Data to Classes and Instances

As you've learned, classes are great when you must bundle data and behavior together in a single entity. The data will come in the form of attributes, while the behavior will come as methods. You already have an idea of what an attribute is. Now it's time to dive deeper into how you can add, access, and modify attributes in your custom classes.

First, you need to know that your classes can have two types of attributes in Python:

1. **Class attributes:** A class attribute is a variable that you define in the class body directly. Class attributes belong to their containing class. Their data is common to the class and all its instances.
2. **Instance attributes:** An instance is a variable that you define inside a method. Instance attributes belong to a concrete instance of a given class. Their data is only available to that instance and defines its state.

Both types of attributes have their specific use cases. Instance attributes are, by far, the most common type of attribute that you'll use in your day-to-day coding, but class attributes also come in handy.

Class Attributes

Class attributes are variables that you define directly in the class body but outside of any method. These attributes are tied to the class itself rather than to particular objects of that class.

All the objects that you create from a particular class share the same class attributes with the same original values. Because of this, if you change a class attribute, then that change affects all the derived objects.

As an example, say that you want to create a class that keeps an internal count of the instances you've created. In that case, you can use a class attribute:

Python >>>

```
>>> class ObjectCounter:
...     num_instances = 0
...     def __init__(self):
...         ObjectCounter.num_instances += 1
...

>>> ObjectCounter()
<__main__.ObjectCounter object at 0x10392d810>
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x1039810d0>
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x10395b750>
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x103959810>

>>> ObjectCounter.num_instances
4

>>> counter = ObjectCounter()
>>> counter.num_instances
5
```

`ObjectCounter` keeps a `.num_instances` class attribute that works as a counter of instances. When Python parses this class, it initializes the counter to zero and leaves it alone. Creating instances of this class means automatically calling the `.__init__()` method and incremementing `.num_instances` by one.

Note: In the above example, you've used the class name to access `.num_instances` inside `.__init__()`. However, using the built-in `type()` function is best because it'll make your class more flexible:

Python >>>

```
>>> class ObjectCounter:
...     num_instances = 0
...     def __init__(self):
...         type(self).num_instances += 1
...
```

The built-in `type()` function returns the class or type of `self`, which is `ObjectCounter` in this example. This subtle change makes your class more robust and reliable by avoiding hard-coding the class that provides the attribute.

It's important to note that you can *access* class attributes using either the class or one of its instances. That's why you can use the `counter` object to retrieve the value of `.num_instances`. However, if you need to *modify* a class attribute, then you must use the class itself rather than one of its instances.

For example, if you use `self` to modify `.num_instances`, then you'll be overriding the original class attribute by creating a new instance attribute:

Python >>>

```
>>> class ObjectCounter:
...     num_instances = 0
...     def __init__(self):
...         self.num_instances += 1
...

>>> ObjectCounter()
<__main__.ObjectCounter object at 0x103987550>
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x1039c5890>
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x10396a890>
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x1036fa110>

>>> ObjectCounter.num_instances
0
```

You can't modify class attributes through instances of the containing class. Doing that will create new instance attributes with the same name as the original class attributes. That's why `ObjectCounter.num_instances` returns `0` in this example. You've overridden the class attribute in the highlighted line.

In general, you should use class attributes for sharing data between instances of a class. Any changes on a class attribute will be visible to all the instances of that class.

Instance Attributes

Instance attributes are variables tied to a particular object of a given class. The value of an instance attribute is attached to the object itself. So, the attribute's value is specific to its containing instance.

Python lets you dynamically attach attributes to existing objects that you've already created. However, you most often define instance attributes inside **instance methods**, which are those methods that receive `self` as their first argument.

> **Note:** Even though you can define instance attributes inside any instance method, it's best to define all of them in the `.__init__()` method, which is the instance initializer. This ensures that all of the attributes have the correct values when you create a new instance. Additionally, it makes the code more organized and easier to debug.

Consider the following `Car` class, which defines a bunch of instance attributes:

Python

```
# car.py

class Car:
    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color
        self.started = False
        self.speed = 0
        self.max_speed = 200
```

In this class, you define a total of seven instance attributes inside `.__init__()`. The attributes `.make`, `.model`, `.year`, and `.color` take values from the arguments to `.__init__()`, which are the arguments that you must pass to the class constructor, `Car()`, to create concrete objects.

Then, you explicitly initialize the attributes `.started`, `.speed`, and `.max_speed` with sensible values that don't come from the user.

Note: Inside a class, you must access all instance attributes through the `self` argument. This argument holds a reference to the **current instance**, which is where the attributes belong and live. The `self` argument plays a fundamental role in Python classes. You'll learn more about `self` in the section Instance Methods With `self`.

Here's how your `Car` class works in practice:

Python >>>

```
>>> from car import Car

>>> toyota_camry = Car("Toyota", "Camry", 2022, "Red")
>>> toyota_camry.make
'Toyota'
>>> toyota_camry.model
'Camry'
>>> toyota_camry.color
'Red'
>>> toyota_camry.speed
0

>>> ford_mustang = Car("Ford", "Mustang", 2022, "Black")
>>> ford_mustang.make
'Ford'
>>> ford_mustang.model
'Mustang'
>>> ford_mustang.year
2022
>>> ford_mustang.max_speed
200
```

In these examples, you create two different instances of `Car`. Each instance takes specific input arguments at instantiation time to initialize part of its attributes. Note how the values of the associated attributes are different and specific to the concrete instance.

Unlike class attributes, you can't access instance attributes through the class. You need to access them through their containing instance:

Python >>>

```
>>> Car.make
Traceback (most recent call last):
    ...
AttributeError: type object 'Car' has no attribute 'make'
```

Instance attributes are specific to a concrete instance of a given class. So, you can't access them through the class object. If you try to do that, then you get an `AttributeError` exception.

The `.__dict__` Attribute

In Python, both classes and instances have a special attribute called `.__dict__`. This attribute holds a dictionary containing the writable members of the underlying class or instance. Remember, these members can be attributes or methods. Each key in `.__dict__` represents an attribute name. The value associated with a given key represents the value of the corresponding attribute.

In a class, `.__dict__` will contain class attributes and methods. In an instance, `.__dict__` will hold instance attributes.

When you access a class member through the class object, Python automatically searches for the member's name in the class `.__dict__`. If the name isn't there, then you get an `AttributeError`.

Similarly, when you access an instance member through a concrete instance of a class, Python looks for the member's name in the instance `.__dict__`. If the name doesn't appears there, then Python looks in the class `.__dict__`. If the name isn't found, then you get a `NameError`.

Here's a toy class that illustrates how this mechanism works:

Python

```
# sample_dict.py

class SampleClass:
    class_attr = 100

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

    def method(self):
        print(f"Class attribute: {self.class_attr}")
        print(f"Instance attribute: {self.instance_attr}")
```

In this class, you define a class attribute with a value of `100`. In the `.__init__()` method, you define an instance attribute that takes its value from the user's input. Finally, you define a method to print both attributes.

Now it's time to check the content of `.__dict__` in the class object. Go ahead and run the following code:

Python >>>

```
>>> from sample_dict import SampleClass

>>> SampleClass.class_attr
100

>>> SampleClass.__dict__
mappingproxy({
    '__module__': '__main__',
    'class_attr': 100,
    '__init__': <function SampleClass.__init__ at 0x1036c62a0>,
    'method': <function SampleClass.method at 0x1036c56c0>,
    '__dict__': <attribute '__dict__' of 'SampleClass' objects>,
    '__weakref__': <attribute '__weakref__' of 'SampleClass' objects>,
    '__doc__': None
})

>>> SampleClass.__dict__["class_attr"]
100
```

The highlighted lines show that both the class attribute and the method are in the class `.__dict__` dictionary. Note how you can use `.__dict__` to access the value of class attributes by specifying the attribute's name in square brackets, as you usually access keys in a dictionary.

> **Note:** You can access the same dictionary by calling the built-in `vars()` function on your class or instance, as you did before.

In instances, the `.__dict__` dictionary will contain instance attributes only:

Python >>>

```
>>> instance = SampleClass("Hello!")

>>> instance.instance_attr
'Hello!'
>>> instance.method()
Class attribute: 100
Instance attribute: Hello!

>>> instance.__dict__
{'instance_attr': 'Hello!'}

>>> instance.__dict__["instance_attr"]
'Hello!'

>>> instance.__dict__["instance_attr"] = "Hello, Pythonista!"
>>> instance.instance_attr
'Hello, Pythonista!'
```

The instance `.__dict__` dictionary in this example holds `.instance_attr` and its specific value for the object at hand. Again, you can access any existing instance attribute using `.__dict__` and the attribute name in square brackets.

You can modify the instance `.__dict__` dynamically. This means that you can change the value of existing instance attributes through `.__dict__`, as you did in the final example above. You can even add new attributes to an instance using its `.__dict__` dictionary.

Using `.__dict__` to change the value of instance attributes will allow you to avoid `RecursionError` exceptions when you're wiring descriptors in Python. You'll learn more about descriptors in the Property and Descriptor-Based Attributes section.

Dynamic Class and Instance Attributes

In Python, you can add new attributes to your classes and instances dynamically. This possibility allows you to attach new data and behavior to your classes and objects in response to changing requirements or contexts. It also allows you to adapt existing classes to specific and dynamic needs.

For example, you can take advantage of this Python feature when you don't know the required attributes of a given class at the time when you're defining that class itself.

Consider the following class, which aims to store a row of data from a database table or a CSV file:

Python >>>

```
>>> class Record:
...     """Hold a record of data."""
...
```

In this class, you haven't defined any attributes or methods because you don't know what data the class will store. Fortunately, you can add attributes and even methods to this class dynamically.

For example, say that you've read a row of data from an `employees.csv` file using `csv.DictReader`. This class reads the data and returns it in a dictionary-like object. Now suppose that you have the following dictionary of data:

Python >>>

```
>>> john = {
...     "name": "John Doe",
...     "position": "Python Developer",
...     "department": "Engineering",
...     "salary": 80000,
...     "hire_date": "2020-01-01",
...     "is_manager": False,
... }
```

Next, you want to add this data to an instance of your `Record` class, and you need to represent each data field as an instance attribute. Here's how you can do it:

Python >>>

```
>>> john_record = Record()

>>> for field, value in john.items():
...     setattr(john_record, field, value)
...

>>> john_record.name
'John Doe'
>>> john_record.department
'Engineering'

>>> john_record.__dict__
{
    'name': 'John Doe',
    'position': 'Python Developer',
    'department': 'Engineering',
    'salary': 80000,
    'hire_date': '2020-01-01',
    'is_manager': False
}
```

In this code snippet, you first create an instance of `Record` called `john_record`. Then you run a `for` loop to iterate over the items of your dictionary of data, `john`. Inside the loop, you use the built-in `setattr()` function to sequentially add each field as an attribute to your `john_record` object. If you inspect `john_record`, then you'll note that it stores all the original data as attributes.

You can also use dot notation and an assignment to add new attributes and methods to a class dynamically:

Python >>>

```
>>> class User:
...     pass
...

>>> # Add instance attributes dynamically
>>> jane = User()
>>> jane.name = "Jane Doe"
>>> jane.job = "Data Engineer"
>>> jane.__dict__
{'name': 'Jane Doe', 'job': 'Data Engineer'}

>>> # Add methods dynamically
>>> def __init__(self, name, job):
...     self.name = name
...     self.job = job
...
>>> User.__init__ = __init__

>>> User.__dict__
mappingproxy({
    ...
    '__init__': <function __init__ at 0x1036ccae0>
})

>>> linda = User("Linda Smith", "Team Lead")
>>> linda.__dict__
{'name': 'Linda Smith', 'job': 'Team Lead'}
```

Here, you first create a minimal `User` class with no custom attributes or methods. To define the class's body, you've just used a `pass` statement as a placeholder, which is Python's way of doing nothing.

Then you create an object called `jane`. Note how you can use dot notation and an assignment to add new attributes to the instance. In this example, you add `.name` and `.job` attributes with appropriate values.

Then you provide the `User` class with an initializer or `.__init__()` method. In this method, you take the `name` and `job` arguments, which you turn into instance attributes in the method's body. Then you add the method to `User` dynamically. After this addition, you can create `User` objects by passing the name and job to the class constructor.

As you can conclude from the above example, you can construct an entire Python class dynamically. Even though this capability of Python may seem neat, you must use it carefully because it can make your code difficult to understand and reason about.

Property and Descriptor-Based Attributes

Python allows you to add function-like behavior on top of existing instance attributes and turn them into **managed attributes**. This type of attribute prevents you from introducing breaking changes into your APIs.

In other words, with managed attributes, you can have function-like behavior and attribute-like access at the same time. You don't need to change your APIs by replacing attributes with method calls, which can potentially break your users' code.

To create a managed attribute with function-like behavior in Python, you can use either a property or a descriptor, depending on your specific needs.

> **Note:** To dive deeper into Python properties, check out Python's `property()`: Add Managed Attributes to Your Classes.

As an example, get back to your `Circle` class and say that you need to validate the radius to ensure that it only stores positive numbers. How would you do that without changing your class interface? The quickest approach to this problem is to use a property and implement the validation logic in the setter method.

Here's what your new version of `Circle` can look like:

Python

```
# circle.py

import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if not isinstance(value, int | float) or value <= 0:
            raise ValueError("positive number expected")
        self._radius = value

    def calculate_area(self):
        return round(math.pi * self._radius**2, 2)
```

To turn an existing attribute like `.radius` into a property, you typically use the `@property` decorator to write the getter method. The getter method must return the value of the attribute. In this example, the getter returns the circle's radius, which is stored in the non-public `._radius` attribute.

To define the setter method of a property-based attribute, you need to use the decorator `@attr_name.setter`. In the example, you use `@radius.setter`. Then you need to define the method itself. Note that property setters need to take an argument providing the value that you want to store in the underlying attribute.

Inside the setter method, you use a conditional to check whether the input value is an integer or a floating-point number. You also check if the value is less than or equal to `0`. If either is true, then you raise a `ValueError` with a descriptive message about the actual issue. Finally, you assign `value` to `._radius`, and that's it. Now, `.radius` is a

property-based attribute.

Here's an example of this new version of `Circle` in action:

Python >>>

```
>>> from circle import Circle

>>> circle_1 = Circle(100)
>>> circle_1.radius
100
>>> circle_1.radius = 500
>>> circle_1.radius = 0
Traceback (most recent call last):
    ...
ValueError: positive number expected

>>> circle_2 = Circle(-100)
Traceback (most recent call last):
    ...
ValueError: positive number expected

>>> circle_3 = Circle("300")
Traceback (most recent call last):
    ...
ValueError: positive number expected
```

The first instance of `Circle` in this example takes a valid value for its radius. Note how you can continue working with `.radius` as a regular attribute rather than as a method. If you try to assign an invalid value to `.radius`, then you get a `ValueError`.

> **Note:** Remember to reload the `circle.py` module if you're working on the same REPL session as before. This recommendation will also be valid for all the examples in this tutorial where you change modules that you defined in previous examples.

It's important to note that the validation also runs at instantiation time when you call the class constructor to create new instances of `Circle`. This behavior is consistent with your validation strategy.

Using a descriptor to create managed attributes is another powerful way to add function-like behavior to your instance attributes without changing your APIs. Like properties, descriptors can also have getter, setter, and other types of methods.

> **Note:** To learn more about descriptors and how to use them, check out Python Descriptors: An Introduction.

To explore how descriptors work, say that you've decided to continue creating classes for your drawing application, and now you have the following `Square` class:

Python

```
# square.py

class Square:
    def __init__(self, side):
        self.side = side

    @property
    def side(self):
        return self._side

    @side.setter
    def side(self, value):
        if not isinstance(value, int | float) or value <= 0:
            raise ValueError("positive number expected")
        self._side = value

    def calculate_area(self):
        return round(self._side**2, 2)
```

This class uses the same pattern as your `Circle` class. Instead of using `radius`, the `Square` class takes a `side` argument and computes the area using the appropriate expression for a square.

This class is pretty similar to `Circle`, and the repetition starts looking odd. Then you think of using a descriptor to abstract away the validation process. Here's what you come up with:

Python

```
# shapes.py

import math

class PositiveNumber:
    def __set_name__(self, owner, name):
        self._name = name

    def __get__(self, instance, owner):
        return instance.__dict__[self._name]

    def __set__(self, instance, value):
        if not isinstance(value, int | float) or value <= 0:
            raise ValueError("positive number expected")
        instance.__dict__[self._name] = value

class Circle:
    radius = PositiveNumber()

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

    def calculate_area(self):
        return round(math.pi * self.radius**2, 2)

class Square:
    side = PositiveNumber()

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

    def calculate_area(self):
        return round(self.side**2, 2)
```

The first thing to notice in this example is that you moved all the classes to a `shapes.py` file. In that file, you define a descriptor class called `PositiveNumber` by implementing the `.__get__()` and `.__set__()` special methods, which are part of the descriptor protocol.

Next, you remove the `.radius` property from `Circle` and the `.side` property from `Square`. In `Circle`, you add a `.radius` class attribute, which holds an instance of `PositiveNumber`. You do something similar in `Square`, but the class attribute is appropriately named `.side`.

Here are a few examples of how your classes work now:

Python >>>

```
>>> from shapes import Circle, Square

>>> circle = Circle(100)
>>> circle.radius
100
>>> circle.radius = 500
>>> circle.radius
500
>>> circle.radius = 0
Traceback (most recent call last):
    ...
ValueError: positive number expected

>>> square = Square(200)
>>> square.side
200
>>> square.side = 300
>>> square.side
300
>>> square.side = -100
Traceback (most recent call last):
    ...
ValueError: positive number expected
```

Python descriptors provide a powerful tool for adding function-like behavior on top of your instance attributes. They can help you remove repetition from your code, making it cleaner and more maintainable. They also promote code reuse.

Lightweight Classes With `.__slots__`

In a Python class, using the `.__slots__` attribute can help you reduce the memory footprint of the corresponding instances. This attribute prevents the automatic creation of an instance `.__dict__`. Using `.__slots__` is particularly handy when you have a class with a fixed set of attributes, and you'll use that class to create a large number of objects.

In the example below, you have a `Point` class with a `.__slots__` attribute that consists of a tuple of allowed attributes. Each attribute will represent a Cartesian coordinate:

Python >>>

```
>>> class Point:
...     __slots__ = ("x", "y")
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...

>>> point = Point(4, 8)
>>> point.__dict__
Traceback (most recent call last):
    ...
AttributeError: 'Point' object has no attribute '__dict__'
```

This `Point` class defines `.__slots__` as a tuple with two items. Each item represents the name of an instance attribute. So, they must be strings holding valid Python identifiers.

> **Note:** Although `.__slots__` can hold a `list` object, you should use a `tuple` object instead. Even if changing the list in `.__slots__` after processing the class body had no effect, it'd be misleading to use a mutable sequence there.

Instances of your `Point` class don't have a `.__dict__`, as the code shows. This feature makes them memory-efficient. To illustrate this efficiency, you can measure the memory consumption of an instance of `Point`. To do this, you can use the Pympler library, which you can install from PyPI using the `pip install pympler` command.

Once you've installed Pympler with `pip`, then you can run the following code in your REPL:

Python >>>

```
>>> from pympler import asizeof

>>> asizeof.asizeof(Point(4, 8))
112
```

The `asizeof()` function from Pympler says that the object `Point(4, 8)` occupies 112 bytes in your computer's memory. Now get back to your REPL session and redefine `Point` without providing a `.__slots__` attribute. With this update in place, go ahead and run the memory check again:

Python >>>

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

>>> asizeof.asizeof(Point(4, 8))
528
```

The same object, `Point(4, 8)`, now consumes 528 bytes of memory. This number is over four times greater than what you got with the original implementation of `Point`. Imagine how much memory `.__slots__` would save if you had to create a million points in your code.

The `.__slots__` attribute adds a second interesting behavior to your custom classes. It prevents you from adding new instance attributes dynamically:

Python >>>

```
>>> class Point:
...     __slots__ = ("x", "y")
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...

>>> point = Point(4, 8)
>>> point.z = 16
Traceback (most recent call last):
  ...
AttributeError: 'Point' object has no attribute 'z'
```

Adding a `.__slots__` to your classes allows you to provide a series of *allowed* attributes. This means that you won't be able to add new attributes to your instances dynamically. If you try to do it, then you'll get an `AttributeError` exception.

A word of caution is in order, as many of Python's built-in mechanisms implicitly assume that objects have the `.__dict__` attribute. When you use `.__slots__()`, then you waive that assumption, which means that some of those mechanisms might not work as expected anymore.

Providing Behavior With Methods

Python classes allow you to bundle data and behavior together in a single entity through attributes and methods, respectively. You'll use the data to define the object's current state and the methods to operate on that data or state.

A method is just a function that you define inside a class. By defining it there, you make the relationship between the class and the method explicit and clear.

Because they're just functions, methods can take arguments and return values as functions do. However, the syntax for calling a method is a bit different from the syntax for calling a function. To call a method, you need to specify the class or instances in which that method is defined. To do this, you need to use dot notation. Remember that classes are namespaces, and their members aren't directly accessible from the outside.

In a Python class, you can define three different types of methods:

1. **Instance methods**, which take the current instance, `self`, as their first argument
2. **Class methods**, which take the current class, `cls`, as their first argument
3. **Static methods**, which take neither the class nor the instance

Every type of method has its own characteristics and specific use cases. Instance methods are, by far, the most common methods that you'll use in your custom Python classes.

> **Note:** To learn more about instance, class, and static methods, check out Python's Instance, Class, and Static Methods Demystified.

In the following sections, you'll dive into how each of these methods works and how to create them in your classes. To get started, you'll begin with instance methods, which are the most common methods that you'll define in your classes.

Instance Methods With `self`

In a class, an instance method is a function that takes the current instance as its first argument. In Python, this first argument is called `self` by convention.

> **Note:** Naming the current instance `self` is a strong convention in Python. It's so strong that it may look like `self` is one of the Python keywords. However, you could use any other name instead of `self`.
>
> Even though it's possible to use any name for the first argument of an instance method, using `self` is definitely the right choice because it'll make your code look like Python code in the eyes of other developers.

The `self` argument holds a reference to the current instance, allowing you to access that instance from within methods. More importantly, through `self`, you can access and modify the instance attributes and call other methods within the class.

To define an instance method, you just need to write a regular function that accepts `self` as its first argument. Up to this point, you've already written some instance methods. To continue learning about them, turn back to your `Car` class.

Now say that you want to add methods to start, stop, accelerate, and brake the car. To kick things off, you'll begin by writing the `.start()` and `.stop()` methods:

Python

```
# car.py

class Car:
    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color
        self.started = False
        self.speed = 0
        self.max_speed = 200

    def start(self):
        print("Starting the car...")
        self.started = True

    def stop(self):
        print("Stopping the car...")
        self.started = False
```

The `.start()` and `.stop()` methods are pretty straightforward. They take the current instance, `self`, as their first argument. Inside the methods, you use `self` to access the `.started` attribute on the current instance using dot notation. Then you change the current value of this attribute to `True` in `.start()` and to `False` in `.stop()`. Both methods print informative messages to illustrate what your car is doing.

> **Note:** Instance methods should act on instance attributes by either accessing them or changing their values. If you find yourself writing an instance method that doesn't use `self` in its body, then that may not be an instance method. In this case, you should probably use a class method or a static method, depending on your specific needs.

Now you can add the `.accelerate()` and `.brake()` methods, which will be a bit more complex:

Python

```
# car.py

class Car:
    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color
        self.started = False
        self.speed = 0
        self.max_speed = 200

    # ...

    def accelerate(self, value):
        if not self.started:
            print("Car is not started!")
            return
        if self.speed + value <= self.max_speed:
            self.speed += value
        else:
            self.speed = self.max_speed
        print(f"Accelerating to {self.speed} km/h...")

    def brake(self, value):
        if self.speed - value >= 0:
            self.speed -= value
        else:
            self.speed = 0
        print(f"Braking to {self.speed} km/h...")
```

The `.accelerate()` method takes an argument that represents the increment of speed that occurs when you call the method. For simplicity, you haven't set any validation for the input increment of speed, so using negative values can cause issues. Inside the method, you first check whether the car's engine is started, returning immediately if it's not.

Then you check if the incremented speed is less than or equal to the allowed maximum speed for your car. If this condition is true, then you increment the speed. Otherwise, you set the speed to the allowed limit.

The `.brake()` method works similarly. This time, you compare the decremented speed to `0` because cars can't have a negative speed. If the condition is true, then you decrement the speed according to the input argument. Otherwise, you set the speed to its lower limit, `0`. Again, you have no validation on the input decrement of speed, so be careful with negative values.

Your class now has four methods that operate on and with its attributes. It's time for a drive:

Python >>>

```
>>> from car import Car

>>> ford_mustang = Car("Ford", "Mustang", 2022, "Black")
>>> ford_mustang.start()
Starting the car...

>>> ford_mustang.accelerate(100)
Accelerating to 100 km/h...
>>> ford_mustang.brake(50)
Braking to 50 km/h...
>>> ford_mustang.brake(80)
Braking to 0 km/h...
>>> ford_mustang.stop()
Stopping the car...

>>> ford_mustang.accelerate(100)
Car is not started!
```

Great! Your `Car` class works nicely! You can start your car's engine, increment the speed, brake, and stop the car's engine. You're also confident that no one can increment the car's speed if the car's engine is stopped. How does that sound for minimal modeling of a car?

It's important to note that when you call an instance method on a concrete instance like `ford_mustang` using dot notation, you don't have to provide a value for the `self` argument. Python takes care of that step for you. It automatically passes the target instance to `self`. So, you only have to provide the rest of the arguments.

However, you can manually provide the desired instance if you want. To do this, you need to call the method on the class:

Python >>>

```
>>> ford_mustang = Car("Ford", "Mustang", 2022, "Black")

>>> Car.start(ford_mustang)
Starting the car...
>>> Car.accelerate(ford_mustang, 100)
Accelerating to 100 km/h...
>>> Car.brake(ford_mustang, 100)
Braking to 0 km/h...
>>> Car.stop(ford_mustang)
Stopping the car...

>>> Car.start()
Traceback (most recent call last):
  ...
TypeError: Car.start() missing 1 required positional argument: 'self'
```

In this example, you call instance methods directly on the class. For this type of call to work, you need to explicitly provide an appropriate value to the `self` argument. In this example, that value is the `ford_mustang` instance. Note that if you don't provide a suitable instance, then the call fails with a `TypeError`. The error message is pretty clear. There's a missing positional argument, `self`.

Special Methods and Protocols

Python supports what it calls special methods, which are also known as **dunder** or **magic** methods. These methods are typically instance methods, and they're a fundamental part of Python's internal class mechanism. They have an important feature in common: *Python calls them automatically in response to specific operations.*

Python uses these methods for many different tasks. They provide a great set of tools that will allow you to unlock the power of classes in Python.

You'll recognize special methods because their names start and end with a double underscore, which is the origin of their other name, dunder methods (**d**ouble **under**score). Arguably, `.__init__()` is the most common special method in Python classes. As you already know, this method works as the instance initializer. Python automatically calls it when you call a class constructor.

You've already written a couple of `.__init__()` methods. So, you're ready to learn about other common and useful special methods. For example, the `.__str__()` and `.__repr__()` methods provide **string representations** for your objects.

Go ahead and update your `Car` class to add these two methods:

Python

```
# car.py

class Car:
    # ...

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

    def __repr__(self):
        return (
            f"{type(self).__name__}"
            f'(make="{self.make}", '
            f'model="{self.model}", '
            f"year={self.year}, "
            f'color="{self.color}")'
        )
```

The `.__str__()` method provides what's known as the **informal string representation** of an object. This method must return a string that represents the object in a user-friendly manner. You can access an object's informal string representation using either `str()` or `print()`.

The `.__repr__()` method is similar, but it must return a string that allows you to re-create the object if possible. So, this method returns what's known as the **formal string representation** of an object. This string representation is mostly targeted at Python programmers, and it's pretty useful when you're working in an interactive REPL session.

In interactive mode, Python falls back to calling `.__repr__()` when you access an object or evaluate an expression, issuing the formal string representation of the resulting object. In script mode, you can access an object's formal string representation using the built-in `repr()` function.

Run the following code to give your new methods a try. Remember that you need to restart your REPL or reload `car.py`:

Python >>>

```
>>> from car import Car

>>> toyota_camry = Car("Toyota", "Camry", 2022, "Red")

>>> str(toyota_camry)
'Toyota, Camry, Red: (2022)'
>>> print(toyota_camry)
Toyota, Camry, Red: (2022)

>>> toyota_camry
Car(make="Toyota", model="Camry", year=2022, color="Red")
>>> repr(toyota_camry)
'Car(make="Toyota", model="Camry", year=2022, color="Red")'
```

When you use an instance of `Car` as an argument to `str()` or `print()`, you get a user-friendly string representation of the car at hand. This informal representation comes in handy when you need your programs to present your users with information about specific objects.

If you access an instance of `Car` directly in a REPL session, then you get a formal string representation of the object. You can copy and paste this representation to re-create the object in an appropriate environment. That's why this string representation is intended to be useful for developers, who can take advantage of it while debugging and testing their code.

Python **protocols** are another fundamental topic that's closely related to special methods. Protocols consist of one or more special methods that support a given feature or functionality. Common examples of protocols include:

Protocol	Provided Feature	Special Methods
Iterator	Allows you to create iterator objects	`.__iter__()` and `.__next__()`
Iterable	Makes your objects iterable	`.__iter__()`
Descriptor	Lets you write managed attributes	`.__get__()` and optionally `.__set__()`, `.__delete__()`, and `.__set_name__()`
Context manager	Enables an object to work on `with` statements	`.__enter__()` and `.__exit__()`

Of course, Python has many other protocols that support cool features of the language. You already coded an example of using the descriptor protocol in the Property and Descriptor-Based Attributes section.

Here's an example of a minimal `ThreeDPoint` class that implements the iterable protocol:

Python >>>

```
>>> class ThreeDPoint:
...     def __init__(self, x, y, z):
...         self.x = x
...         self.y = y
...         self.z = z
...     def __iter__(self):
...         yield from (self.x, self.y, self.z)
...

>>> list(ThreeDPoint(4, 8, 16))
[4, 8, 16]
```

This class takes three arguments representing the space coordinates of a given point. The `.__iter__()` method is a generator function that returns an iterator. The resulting iterator yields the coordinates of `ThreeDPoint` on demand.

The call to `list()` iterates over the attributes `.x`, `.y`, and `.z`, returning a `list` object. You don't need to call `.__iter__()` directly. Python calls it automatically when you use an instance of `ThreeDPoint` in an iteration.

Class Methods With `@classmethod`

You can also add **class methods** to your custom Python classes. A class method is a method that takes the class object as its first argument instead of taking `self`. In this case, the argument should be called `cls`, which is also a strong convention in Python. So, you should stick to it.

You can create class methods using the `@classmethod` decorator. Providing your classes with multiple constructors is one of the most common use cases of class methods in Python.

For example, say you want to add an alternative constructor to your `ThreeDPoint` so that you can quickly create points from tuples or lists of coordinates:

Python

```
# point.py

class ThreeDPoint:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def __iter__(self):
        yield from (self.x, self.y, self.z)

    @classmethod
    def from_sequence(cls, sequence):
        return cls(*sequence)

    def __repr__(self):
        return f"{type(self).__name__}({self.x}, {self.y}, {self.z})"
```

In the `.from_sequence()` class method, you take a sequence of coordinates as an argument, create a `ThreeDPoint` object from it, and return the object to the caller. To create the new object, you use the `cls` argument, which holds an implicit reference to the current class, which Python injects into your method automatically.

Here's how this class method works:

Python >>>

```
>>> from point import ThreeDPoint

>>> ThreeDPoint.from_sequence((4, 8, 16))
ThreeDPoint(4, 8, 16)

>>> point = ThreeDPoint(7, 14, 21)
>>> point.from_sequence((3, 6, 9))
ThreeDPoint(3, 6, 9)
```

In this example, you use the `ThreeDPoint` class directly to access the class method `.from_sequence()`. Note that you can also access the method using a concrete instance, like `point` in the example. In each of the calls to `.from_sequence()`, you'll get a completely new instance of `ThreeDPoint`. However, class methods should be accessed through the corresponding class name for better clarity and to avoid confusion.

Static Methods With `@staticmethod`

Your Python classes can also have **static methods**. These methods don't take the instance or the class as an argument. So, they're regular functions defined within a class. You could've also defined them outside the class as stand-alone function.

You'll typically define a static method instead of a regular function outside the class when that function is closely related to your class, and you want to bundle it together for convenience or for consistency with your code's API. Remember that calling a function is a bit different from calling a method. To call a method, you need to specify a class or object that provides that method.

If you want to write a static method in one of your custom classes, then you need to use the `@staticmethod` decorator. Check out the `.show_intro_message()` method below:

Python

```
# point.py

class ThreeDPoint:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def __iter__(self):
        yield from (self.x, self.y, self.z)

    @classmethod
    def from_sequence(cls, sequence):
        return cls(*sequence)

    @staticmethod
    def show_intro_message(name):
        print(f"Hey {name}! This is your 3D Point!")

    def __repr__(self):
        return f"{type(self).__name__}({self.x}, {self.y}, {self.z})"
```

The `.show_intro_message()` static method takes a `name` as an argument and prints a message on the screen. Note that this is only a toy example of how to write static methods in your classes.

Static methods like `.show_intro_message()` don't operate on the current instance, `self`, or the current class, `cls`. They work as independent functions enclosed in a class. You'll typically put them inside a class when they're closely related to that class but don't necessarily affect the class or its instances.

Here's how the method works:

Python >>>

```
>>> from point import ThreeDPoint

>>> ThreeDPoint.show_intro_message("Pythonista")
Hey Pythonista! This is your 3D Point!

>>> point = ThreeDPoint(2, 4, 6)
>>> point.show_intro_message("Python developer")
Hey Python developer! This is your 3D Point!
```

As you already know, the `.show_intro_message()` method takes a name as an argument and prints a message to your screen. Note that you can call the method using the class or any of its instances. As with class methods, you should generally call static methods through the corresponding class instead of one of its instances.

Getter and Setter Methods vs Properties

Programming languages like Java and C++ don't expose attributes as part of their classes' public APIs. Instead, these programming languages make extensive use of getter and setter methods to give you access to attributes.

> **Note:** To dive deeper into the getter and setter pattern and how Python approaches it, check out Getters and Setters: Manage Attributes in Python.

Using methods to access and update attributes promotes encapsulation. Encapsulation is a fundamental OOP principle that recommends protecting an object's state or data from the outside world, preventing direct access. The object's state should only be accessible through a public interface consisting of getter and setter methods.

For example, say that you have a `Person` class with a `.name` instance attribute. You can make `.name` a non-public attribute and provide getter and setter methods to access and change that attribute:

Python

```
# person.py

class Person:
    def __init__(self, name):
        self.set_name(name)

    def get_name(self):
        return self._name

    def set_name(self, value):
        self._name = value
```

In this example, `.get_name()` is the getter method and allows you to access the underlying `._name` attribute. Similarly, `.set_name()` is the setter method and allows you to change the current value of `._name`. The `._name` attribute is non-public and is where the actual data is stored.

Here's how you can use your `Person` class:

Python >>>

```
>>> from person import Person

>>> jane = Person("Jane")
>>> jane.get_name()
'Jane'

>>> jane.set_name("Jane Doe")
>>> jane.get_name()
'Jane Doe'
```

Here, you create an instance of `Person` using the class constructor and `"Jane"` as the required name. That means you can use the `.get_name()` method to access Jane's name and the `.set_name()` method to update it.

The getter and setter pattern is common in languages like Java and C++. Besides promoting encapsulation and APIs centered on method calls, this pattern also allows you to quickly add function-like behavior to your attributes without introducing breaking changes in your APIs.

However, this pattern is less popular in the Python community. In Python, it's completely normal to expose attributes as part of an object's public API. If you ever need to add function-like behavior on top of a public attribute, then you can turn it into a property instead of breaking the API by replacing the attribute with a method.

Here's how most Python developers would write the `Person` class:

Python

```
# person.py

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

This class doesn't have getter and setter methods for the `.name` attribute. Instead, it exposes the attribute as part of its API. So, you can use it directly:

Python >>>

```
>>> from person import Person

>>> jane = Person("Jane")
>>> jane.name
'Jane'

>>> jane.name = "Jane Doe"
>>> jane.name
'Jane Doe'
```

In this example, instead of using a setter method to change the value of `.name`, you use the attribute directly in an assignment statement. This is common practice in Python code. If your `Person` class evolves to a point where you need to add function-like behavior on top of `.name`, then you can turn the attribute into a property.

For example, say that you need to store the attribute in uppercase letters. Then you can do something like the following:

Python

```
# person.py

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

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value.upper()
```

This class defines `.name` as a property with appropriate getter and setter methods. Python will automatically call these methods, respectively, when you access or update the attribute's value. The setter method takes care of uppercasing the input value before assigning it back to `._name`:

Python >>>

```
>>> from person import Person

>>> jane = Person("Jane")
>>> jane.name
'JANE'

>>> jane.name = "Jane Doe"
>>> jane.name
'JANE DOE'
```

Python properties allow you to add function-like behavior to your attributes while you continue to use them as normal attributes instead of as methods. Note how you can still assign new values to `.name` using an assignment instead of a method call. Running the assignment triggers the setter method, which uppercases the input value.

Summarizing Class Syntax and Usage: A Complete Example

Up to this point, you've learned a lot about Python classes: how to create them, when and how to use them in your code, and more. In this section, you'll review that knowledge by writing a class that integrates most of the syntax and features you've learned so far.

Your class will represent an employee of a given company and will implement attributes and methods to manage some related tasks like keeping track of personal information and computing the employee's age. To kick things off, go ahead and fire up your favorite code editor or IDE and create a file called `employee.py`. Then add the following code to it:

Python

```
# employee.py

class Employee:
    company = "Example, Inc."

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

In this `Employee` class, you define a class attribute called `.company`. This attribute will hold the company's name, which is common to all employees on the payroll.

Then you define the initializer, `.__init__()`, which takes the employee's name and birth date as arguments. Remember that you must pass appropriate values for both arguments when you call the class constructor, `Employee()`.

Inside `.__init__()`, you define two public instance attributes to store the employee's name and birth date. These attributes will be part of the class API because they're public attributes.

Now say that you want to turn `.birth_date` into a property to automatically convert the input date in ISO format to a `datetime` object:

Python

```
# employee.py

from datetime import datetime

class Employee:
    # ...

    @property
    def birth_date(self):
        return self._birth_date

    @birth_date.setter
    def birth_date(self, value):
        self._birth_date = datetime.fromisoformat(value)
```

Here, you define the `.birth_date` property through the `@property` decorator. The getter method returns the content of `._birth_date`. This non-public attribute will hold the concrete data.

To define the setter method, you use the `@birth_date.setter` decorator. In this method, you assign a `datetime.datetime` object to `._birth_date`. In this example, you don't run any validation on the input data, which should be a string holding the date in ISO format. You can implement the validation as an exercise.

Next, say you want to write a regular instance method to compute the employee's age from their birth date:

Python

```
# employee.py

from datetime import datetime

class Employee:
    # ...

    def compute_age(self):
        today = datetime.today()
        age = today.year - self.birth_date.year
        birthday = datetime(
            today.year,
            self.birth_date.month,
            self.birth_date.day
        )
        if today < birthday:
            age -= 1
        return age
```

Here, `.compute_age()` is an instance method because it takes the current instance, `self`, as its first argument. Inside the method, you compute the employee's age using the `.birth_date` property as a starting point.

Now say that you'll often build instances of `Employee` from dictionaries containing the data of your employees. You can add a convenient class method to quickly build objects that way:

Python

```
# employee.py

from datetime import datetime

class Employee:
    # ...

    @classmethod
    def from_dict(cls, data_dict):
        return cls(**data_dict)
```

In this code snippet, you define a class method using the `@classmethod` decorator. The method takes a dictionary object containing the data of a given employee. Then it builds an instance of `Employee` using the `cls` argument and unpacking the dictionary.

Finally, you'll add suitable `.__str__()` and `.__repr__()` special methods to make your class friendly to users and developers, respectively:

Python

```
# employee.py

from datetime import datetime

class Employee:
    # ...

    def __str__(self):
        return f"{self.name} is {self.compute_age()} years old"

    def __repr__(self):
        return (
            f"{type(self).__name__}("
            f"name='{self.name}', "
            f"birth_date='{self.birth_date.strftime('%Y-%m-%d')}')"
        )
```

The `.__str__()` method returns a string describing the current employee in a user-friendly manner. Similarly, the `.__repr__()` method returns a string that will allow you to re-create the current object, which is great from a developer's perspective.

Here's how you can use `Employee` in your code:

Python >>>

```
>>> from employee import Employee

>>> john = Employee("John Doe", "1998-12-04")
>>> john.company
Example, Inc.
>>> john.name
'John Doe'
>>> john.compute_age()
24
>>> print(john)
John Doe is 24 years old
>>> john
Employee(name='John Doe', birth_date='1998-12-04')

>>> jane_data = {"name": "Jane Doe", "birth_date": "2001-05-15"}
>>> jane = Employee.from_dict(jane_data)
>>> print(jane)
Jane Doe is 21 years old
```

Cool! Your `Employee` class works great so far! It allows you to represent employees, access their attributes, and compute their ages. It also provides neat string representations that will make your class look polished and reliable. Great job! Do you have any ideas of cool features that you could add to `Employee`?

Debugging Python Classes

Debugging often represents a large portion of your coding time. You'll probably spend long hours tracking errors in the code that you're working on and trying to fix them to make the code more robust and reliable. When you start working with classes and objects in Python, you're likely to encounter some new exceptions.

For example, if you try to access an attribute or method that doesn't exist, then you'll get an `AttributeError`:

Python >>>

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

>>> point = Point(4, 8)
>>> point.z
Traceback (most recent call last):
  ...
AttributeError: 'Point' object has no attribute 'z'
```

The `Point` class doesn't define a `.z` instance attribute, so you get an `AttributeError` if you try to access that attribute.

You'll find a few exceptions that can occur when working with Python classes. These are some of the most common ones:

- An `AttributeError` occurs when the specified object doesn't define the attribute or method that you're trying to access. Take, for example, accessing `.z` on the `Point` class defined in the above example.
- A `TypeError` occurs when you apply an operation or function to an object that doesn't support that operation. For example, consider calling the built-in `len()` function with a number as an argument.
- A `NotImplementedError` occurs when an abstract method isn't implemented in a concrete subclass. You'll learn more about this exception in the section Creating Abstract Base Classes (ABC) and Interfaces.

These are just a few examples of exceptions that can occur when you're working with Python classes. You'll also find some common mistakes that people sometimes make when they start to write their own classes:

- Forgetting to include the **`self` argument** in instance methods
- Forgetting to **instantiate** the class by calling its constructor with appropriate arguments

- Confusing and misusing **class** and **instance attributes**
- Not following or respecting **naming conventions** for members
- Accessing **non-public members** from outside the containing class
- Overusing and misusing **inheritance**

These are just a few common mistakes that people might make when they're getting started with Python classes. From this list, you haven't learned about inheritance yet. Don't worry about it for now. Inheritance is an advanced topic that you'll study later in this tutorial.

Exploring Specialized Classes From the Standard Library

In the Python standard library, you'll find many tools that solve different problems and deal with different challenges. Among all these tools, you'll find a few that will make you more productive when writing custom classes.

For example, if you want a tool that saves you from writing a lot of class-related boilerplate code, then you can take advantage of data classes and the `dataclasses` module.

Similarly, if you're looking for a tool that allows you to quickly create class-based enumerations of constants, then you can turn your eye to the `enum` module and its different types of enumeration classes.

In the following sections, you'll learn the basics of using data classes and enumerations to efficiently write robust, reliable, and specialized classes in Python.

Data Classes

Python's data classes specialize in storing data. However, they're also code generators that produce a lot of class-related boilerplate code for you behind the scenes.

For example, if you use the data class infrastructure to write a custom class, then you won't have to implement special methods like `.__init__()`, `.__repr__()`, `.__eq__()`, and `.__hash__()`. The data class will write them for you. More importantly, the data class will write these methods applying best practices and avoiding potential errors.

> **Note:** To learn more about data classes in Python, check out Data Classes in Python 3.7+ (Guide).

As you already know, special methods support important functionalities in Python classes. In the case of data classes, you'll have accurate string representation, comparison capabilities, hashability, and more.

Even though the name *data class* may suggest that this type of class is limited to containing data, it also offers methods. So, data classes are like regular classes but with superpowers.

To create a data class, go ahead and import the `@dataclass` decorator from the `dataclasses` module. You'll use this decorator in the definition of your class. This time, you won't write an `.__init__()` method. You'll just define data fields as class attributes with type hints.

For example, here's how you can write the `ThreeDPoint` class as a data class::

Python

```
# point.py

from dataclasses import dataclass

@dataclass
class ThreeDPoint:
    x: int | float
    y: int | float
    z: int | float

    @classmethod
    def from_sequence(cls, sequence):
        return cls(*sequence)

    @staticmethod
    def show_intro_message(name):
        print(f"Hey {name}! This is your 3D Point!")
```

This new implementation of `ThreeDPoint` uses Python's `@dataclass` decorator to turn the regular class into a data class. Instead of defining an `.__init__()` method, you list the instance attributes with their corresponding types. The data class will take care of writing a proper initializer for you. Note that you don't define `.__iter__()` or `.__repr__()` either.

Note: Data classes are pretty flexible when it comes to defining their fields or attributes. You can declare them with the type annotation syntax. You can initialize them with a sensible default value. You can also combine both approaches depending on your needs:

Python

```
from dataclasses import dataclass

@dataclass
class ThreeDPoint:
    x: int | float
    y = 0.0
    z: int | float = 0.0
```

In this code snippet, you declare the first attribute using the type annotation syntax. The second attribute has a default value with no type annotation. Finally, the third attribute has both type annotation and a default value. However, when you don't specify a type hint for an attribute, then Python won't automatically generate the corresponding code for that attribute.

Once you've defined the data fields or attributes, you can start adding the methods that you need. In this example, you keep the `.from_sequence()` class method and the `.show_intro_message()` static method.

Go ahead and run the following code to check the additional functionality that `@dataclass` has added to this version of `ThreeDPoint`:

Python >>>

```
>>> from dataclasses import astuple
>>> from point import ThreeDPoint

>>> point_1 = ThreeDPoint(1.0, 2.0, 3.0)
>>> point_1
ThreeDPoint(x=1.0, y=2.0, z=3.0)
>>> astuple(point_1)
(1.0, 2.0, 3.0)

>>> point_2 = ThreeDPoint(2, 3, 4)
>>> point_1 == point_2
False

>>> point_3 = ThreeDPoint(1, 2, 3)
>>> point_1 == point_3
True
```

Your `ThreeDPoint` class works pretty well! It provides a suitable string representation with an automatically generated `.__repr__()` method. You can iterate over the fields using the `astuple()` function from the `dataclasses` module. Finally, you can compare two instances of the class for equality (==). As you can conclude, this new version of `ThreeDPoint` has saved you from writing several lines of tricky boilerplate code.

Enumerations

An **enumeration**, or just **enum**, is a data type that you'll find in several programming languages. Enums allow you to create sets of named constants, which are known as **members** and can be accessed through the enumeration itself.

Python doesn't have a built-in enum data type. Fortunately, Python 3.4 introduced the `enum` module to provide the `Enum` class for supporting general-purpose enumerations.

Days of the week, months and seasons of the year, HTTP status codes, colors in a traffic light, and pricing plans of a web service are all great examples of constants that you can group in an enum. In short, you can use enums to represent variables that can take one of a *limited set of possible values*.

The `Enum` class, among other similar classes in the `enum` module, allows you to quickly and efficiently create custom enumerations or groups of similar constants with neat features that you don't have to code yourself. Apart from member constants, enums can also have methods to operate with those constants.

> **Note:** To learn more about how to create and use enumerations in your Python code, check out Build Enumerations of Constants With Python's Enum.

To define a custom enumeration, you can subclass the `Enum` class. Here's an example of an enumeration that groups the days of the week:

Python >>>

```
>>> from enum import Enum

>>> class WeekDay(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...
```

In this code example, you define `WeekDay` by subclassing `Enum` from the `enum` module. This specific enum groups seven constants representing the days of the week. These constants are the enum members. Because they're constants, you should follow the convention for naming any constant in Python: uppercase letters and, if applicable, underscores between words.

Enumerations have a few cool features that you can take advantage of. For example, their members are strict constants, so you can't change their values. They're also iterable by default:

Python >>>

```
>>> WeekDay.MONDAY = 0
Traceback (most recent call last):
    ...
AttributeError: cannot reassign member 'MONDAY'

>>> list(WeekDay)
[
    <WeekDay.MONDAY: 1>,
    <WeekDay.TUESDAY: 2>,
    <WeekDay.WEDNESDAY: 3>,
    <WeekDay.THURSDAY: 4>,
    <WeekDay.FRIDAY: 5>,
    <WeekDay.SATURDAY: 6>,
    <WeekDay.SUNDAY: 7>
]
```

If you try to change the value of an enum member, then you get an `AttributeError`. So, enum members are strictly constants. You can iterate over the members directly because enumerations support iteration by default.

You can directly access their members using different syntax:

Python >>>

```
>>> # Dot notation
>>> WeekDay.MONDAY
<WeekDay.MONDAY: 1>

>>> # Call notation
>>> WeekDay(2)
<WeekDay.TUESDAY: 2>

>>> # Dictionary notation
>>> WeekDay["WEDNESDAY"]
<WeekDay.WEDNESDAY: 3>
```

In the first example, you access an enum member using **dot notation**, which is pretty intuitive and readable. In the second example, you access a member by **calling** the enumeration with that member's value as an argument. Finally, you use a dictionary-like syntax to access another member by name.

If you want fine-grain access to a member's components, then you can use the `.name` and `.value` attributes, which are pretty handy in the context of iteration:

Python >>>

```
>>> WeekDay.THURSDAY.name
'THURSDAY'
>>> WeekDay.THURSDAY.value
4

>>> for day in WeekDay:
...     print(day.name, "->", day.value)
...
MONDAY -> 1
TUESDAY -> 2
WEDNESDAY -> 3
THURSDAY -> 4
FRIDAY -> 5
SATURDAY -> 6
SUNDAY -> 7
```

In these examples, you access the `.name` and `.value` attributes of specific members of `WeekDay`. These attributes provide access to each member's component.

Finally, you can also add custom behavior to your enumerations. To do that, you can use methods as you'd do with regular classes:

Python

```
# week.py

from enum import Enum

class WeekDay(Enum):
    MONDAY = 1
    TUESDAY = 2
    WEDNESDAY = 3
    THURSDAY = 4
    FRIDAY = 5
    SATURDAY = 6
    SUNDAY = 7

    @classmethod
    def favorite_day(cls):
        return cls.FRIDAY

    def __str__(self):
        return f"Current day: {self.name}"
```

After saving your code to `week.py`, you add a class method called `.favorite_day()` to your `WeekDay` enumeration. This method will just return your favorite day of the week, which is Friday, of course! Then you add a `.__str__()` method to provide a user-friendly string representation for the current day.

Here's how you can use these methods in your code:

Python >>>

```
>>> from week import WeekDay

>>> WeekDay.favorite_day()
<WeekDay.FRIDAY: 5>

>>> print(WeekDay.FRIDAY)
Current day: FRIDAY
```

You've added new functionality to your enumeration through class and instance methods. Isn't that cool?

Using Inheritance and Building Class Hierarchies

Inheritance is a powerful feature of object-oriented programming. It consists of creating hierarchical relationships between classes, where child classes inherit attributes and methods from their parent class. In Python, one class can have multiple parents or, more broadly, ancestors.

This is called **implementation inheritance**, which allows you to reduce duplication and repetition by code reuse. It can also make your code more modular, better organized, and more scalable. However, classes also **inherit the interface** by becoming more specialized kinds of their ancestors. In some cases, you'll be able to use a child instance where an ancestor is expected.

In the following sections, you'll learn how to use inheritance in Python. You'll start with simple inheritance and continue with more complex concepts. So, get ready! This is going to be fun!

Simple Inheritance

When you have a class that inherits from a single parent class, then you're using **single-base inheritance** or just **simple inheritance**. To make a Python class inherit from another, you need to list the parent class's name in parentheses after the child class's name in the definition.

To make this clearer, here's the syntax that you must use:

Python

```
class Parent:
    # Parent's definition goes here...
    pass

class Child(Parent):
    # Child definitions goes here...
    pass
```

In this code snippet, `Parent` is the class you want to inherit from. Parent classes typically provide generic and common functionality that you can reuse throughout multiple child classes. `Child` is the class that inherits features and code from `Parent`. The highlighted line shows the required syntax.

> **Note:** In this tutorial, you'll use the terms **parent class**, **superclass**, and **base class** interchangeably to refer to the class that you inherit from.
>
> Similarly, you'll use the terms **child class**, **derived class**, and **subclass** to refer to classes that inherit from other classes.

Here's a practical example to get started with simple inheritance and how it works. Suppose you're building an app to track vehicles and routes. At first, the app will track cars and motorcycles. You think of creating a `Vehicle` class and deriving two subclasses from it. One subclass will represent a car, and the other will represent a motorcycle.

The `Vehicle` class will provide common attributes, such as `.make`, `.model`, and `.year`. It'll also provide the `.start()` and `.stop()` methods to start and stop the vehicle engine, respectively:

Python

```
# vehicles.py

class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self._started = False

    def start(self):
        print("Starting engine...")
        self._started = True

    def stop(self):
        print("Stopping engine...")
        self._started = False
```

In this code, you define the `Vehicle` class with attributes and methods that are common to all your current vehicles. You can say that `Vehicle` provides a common interface for your vehicles. You'll inherit from this class to reuse this interface and its functionality in your subclasses.

Now you can define the `Car` and `Motorcycle` classes. Both of them will have some unique attributes and methods specific to the vehicle type. For example, the `Car` will have a `.num_seats` attribute and a `.drive()` method:

Python

```
# vehicles.py

# ...

class Car(Vehicle):
    def __init__(self, make, model, year, num_seats):
        super().__init__(make, model, year)
        self.num_seats = num_seats

    def drive(self):
        print(f'Driving my "{self.make} - {self.model}" on the road')

    def __str__(self):
        return f'"{self.make} - {self.model}" has {self.num_seats} seats'
```

Your `Car` class uses `Vehicle` as its parent class. This means that `Car` will automatically inherit the `.make`, `.model`, and `.year` attributes, as well as the non-public `._started` attribute. It'll also inherit the `.start()` and `.stop()` methods.

> **Note:** Like inheritance in nature, inheritance in OOP goes in a single direction, from the parents to the children. In other words, children inherit from their parents and not the other way around.

The class defines a `.num_seats` attribute. As you already know, you should define and initialize instance attributes in `.__init__()`. This requires you to provide a custom `.__init__()` method in `Car`, which will shadow the superclass initializer.

How can you write an `.__init__()` method in `Car` and still guarantee that you initialize the `.make`, `.model`, and `.year` attributes? That's where the built-in `super()` function comes on the scene. This function allows you to access members in the superclass, as its name suggests.

> **Note:** To learn more about using `super()` in your classes, check out Supercharge Your Classes With Python `super()`.

In `Car`, you use `super()` to call the `.__init__()` method on `Vehicle`. Note that you pass the input values for `.make`, `.model`, and `.year` so that `Vehicle` can initialize these attributes correctly. After this call to `super()`, you add and initialize the `.num_seats` attributes, which is specific to the `Car` class.

Finally, you write the `.drive()` method, which is also specific to `Car`. This method is just a demonstrative example, so it only prints a message to your screen.

Now it's time to define the `Motorcycle` class, which will inherit from `Vehicle` too. This class will have a `.num_wheels` attribute and a `.ride()` method:

Python

```
# vehicles.py

# ...

class Motorcycle(Vehicle):
    def __init__(self, make, model, year, num_wheels):
        super().__init__(make, model, year)
        self.num_wheels = num_wheels

    def ride(self):
        print(f'Riding my "{self.make} - {self.model}" on the road')

    def __str__(self):
        return f'"{self.make} - {self.model}" has {self.num_wheels} wheels'
```

Again, you call `super()` to initialize `.make`, `.model`, and `.year`. After that, you define and initialize the `.num_wheels` attribute. Finally, you write the `.ride()` method. Again, this method is just a demonstrative example.

With this code in place, you can start using `Car` and `Motorcycle` right away:

Python >>>

```
>>> from vehicles import Car, Motorcycle

>>> tesla = Car("Tesla", "Model S", 2022, 5)
>>> tesla.start()
Starting engine...
>>> tesla.drive()
Driving my "Tesla - Model S" on the road
>>> tesla.stop()
Stopping engine...
>>> print(tesla)
"Tesla - Model S" has 5 seats

>>> harley = Motorcycle("Harley-Davidson", "Iron 883", 2021, 2)
>>> harley.start()
Starting engine...
>>> harley.ride()
Riding my "Harley-Davidson - Iron 883" on the road.
>>> harley.stop()
Stopping engine...
>>> print(harley)
"Harley-Davidson - Iron 883" has 2 wheels
```

Cool! Your Tesla and your Harley-Davidson work nicely. You can start their engines, drive or ride them, and so on. Note how you can use both the inherited and specific attributes and methods in both classes.

You'll typically use single inheritance or inheritance in general when you have classes that share common attributes and behaviors and want to reuse them in derived classes. So, inheritance is a great tool for code reuse. Subclasses will inherit and reuse functionality from their parent.

Subclasses will frequently extend their parents' interface with new attributes and methods. You can use them as a new starting point to create another level of inheritance. This practice will lead to the creation of class hierarchies.

Class Hierarchies

Using inheritance, you can design and build class hierarchies, also known as inheritance trees. A class hierarchy is a set of closely related classes that are connected through inheritance and arranged in a tree-like structure.

The class or classes at the top of the hierarchy are the base classes, while the classes below are derived classes or subclasses. Inheritance-based hierarchies express an **is-a-type-of** relationship between subclasses and their base classes.

Each level in the hierarchy will inherit attributes and behaviors from the above levels. Therefore, classes at the top of the hierarchy are generic classes with common functionality, while classes down the hierarchy are more specialized. They'll inherit attributes and behaviors from their superclasses and will also add their own.

Taxonomic classification of animals is a commonly used example to explain class hierarchies. In this hierarchy, you'll have a generic `Animal` class at the top. Below this class, you can have subclasses like `Mammal`, `Bird`, `Fish`, and so on. These subclasses are more specific classes than `Animal` and inherit the attributes and methods from it. They can also have their own attributes and methods.

To continue with the hierarchy, you can subclass `Mammal`, `Bird`, and `Fish` and create derived classes with even more specific characteristics. Here's a short toy example:

Python

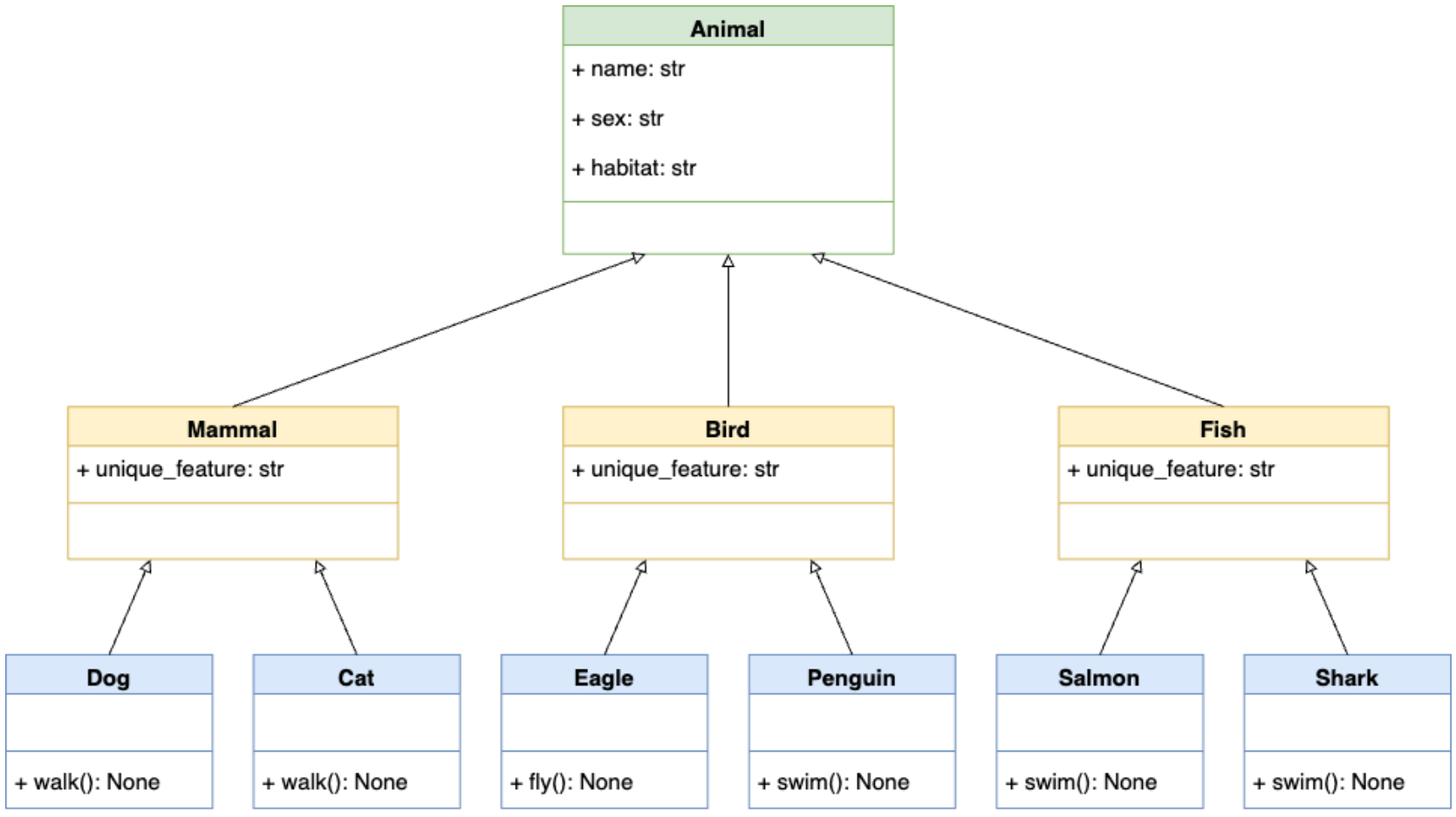
```python
# animals.py

class Animal:
    def __init__(self, name, sex, habitat):
        self.name = name
        self.sex = sex
        self.habitat = habitat

class Mammal(Animal):
    unique_feature = "Mammary glands"

class Bird(Animal):
    unique_feature = "Feathers"

class Fish(Animal):
    unique_feature = "Gills"

class Dog(Mammal):
    def walk(self):
        print("The dog is walking")

class Cat(Mammal):
    def walk(self):
        print("The cat is walking")

class Eagle(Bird):
    def fly(self):
        print("The eagle is flying")

class Penguin(Bird):
    def swim(self):
        print("The penguin is swimming")

class Salmon(Fish):
    def swim(self):
        print("The salmon is swimming")

class Shark(Fish):
    def swim(self):
        print("The shark is swimming")
```

At the top of the hierarchy, you have the `Animal` class. This is the base class of your hierarchy. It has the `.name`, `.sex`, and `.habitat` attributes, which will be string objects. These attributes are common to all animals.

Then you define the `Mammal`, `Bird`, and `Fish` classes by inheriting from `Animal`. These classes have a `.unique_feature` class attribute that holds the distinguishing characteristic of each group of animals.

Then you create concrete mammals like `Dog` and `Cat`. These classes have specific methods that are common to all dogs and cats, respectively. Similarly, you define two classes that inherit from `Bird` and two more that inherit from `Fish`.

Here's a tree-like class diagram that will help you see the hierarchical relationship between classes:



Each level in the hierarchy can—and typically will—add new attributes and functionality on top of those that its parents already provide. If you walk through the diagram from top to button, then you'll move from generic to specialized classes.

These latter classes implement new methods that are specific to the class at hand. In this example, the methods just print some information to the screen and automatically return `None`, which is the null value in Python.

> **Note:** You can create class diagrams to represent class hierarchies that are based on inheritance. However, that's not the only relationship that can appear between your classes.
>
> With class diagrams, you can also represent other types of relationships, including:
>
> - **Composition**, which expresses a strong **has-a** relationship. For example, a robot has an arm. If the robot stops existing, then the arm stops existing too.
> - **Aggregation**, which expresses a softer **has-a** relationship. For example, a university has an instructor. If the university stops existing, the instructor doesn't stop existing.
> - **Association**, which expresses a **uses-a** relationship. For example, a student may be associated with a course. They will use the course. This relationship is common in database systems where you have one-to-one, one-to-many, and many-to-many associations.
>
> You'll learn more about some of these types of relationships in the section called Using Alternatives to Inheritance.

That's how you design and create class hierarchies to reuse code and functionality. Such hierarchies also allow you to give your code a modular organization, making it more maintainable and scalable.

Extended vs Overridden Methods

When you're using inheritance, you can face an interesting and challenging issue. In some situations, a parent class may provide a given functionality only at a basic level, and you may want to extend that functionality in your subclasses. In other situations, the feature in the parent class isn't appropriate for the subclass.

In these situations, you can use one of the following strategies, depending on your specific case:

- **Extending** an inherited method in a subclass, which means that you'll reuse the functionality provided by the superclass and add new functionality on top
- **Overriding** an inherited method in a subclass, which means that you'll completely discard the functionality from the superclass and provide new functionality in the subclass

Here's an example of a small class hierarchy that applies the first strategy to provide extended functionality based on the inherited one:

Python

```
# aircrafts.py

class Aircraft:
    def __init__(self, thrust, lift, max_speed):
        self.thrust = thrust
        self.lift = lift
        self.max_speed = max_speed

    def show_technical_specs(self):
        print(f"Thrust: {self.thrust} kW")
        print(f"Lift: {self.lift} kg")
        print(f"Max speed: {self.max_speed} km/h")

class Helicopter(Aircraft):
    def __init__(self, thrust, lift, max_speed, num_rotors):
        super().__init__(thrust, lift, max_speed)
        self.num_rotors = num_rotors

    def show_technical_specs(self):
        super().show_technical_specs()
        print(f"Number of rotors: {self.num_rotors}")
```

In this example, you define `Aircraft` as the base class. In `.__init__()`, you create a few instance attributes. Then you define the `.show_technical_specs()` method, which prints information about the aircraft's technical specifications.

Next, you define `Helicopter`, inheriting from `Aircraft`. The `.__init__()` method of `Helicopter` extends the corresponding method of `Aircraft` by calling `super()` to initialize the `.thrust`, `.lift`, and `.max_speed` attributes. You already saw something like this in the previous section.

`Helicopter` also extends the functionality of `.show_technical_specs()`. In this case, you first call `.show_technical_specs()` from `Aircraft` using `super()`. Then you add a new call to `print()` that adds new information to the technical description of the helicopter at hand.

Here's how `Helicopter` instances work in practice:

Python >>>

```
>>> from aircrafts import Helicopter

>>> sikorsky_UH60 = Helicopter(1490, 9979, 278, 2)
>>> sikorsky_UH60.show_technical_specs()
Thrust: 1490 kW
Lift: 9979 kg
Max speed: 278 km/h
Number of rotors: 2
```

When you call `.show_technical_specs()` on a `Helicopter` instance, you get the information provided by the base class, `Aircraft`, and also the specific information added by `Helicopter` itself. You've extended the functionality of `Aircraft` in its subclass `Helicopter`.

Now it's time to take a look at how you can override a method in a subclass. As an example, say that you have a base class called `Worker` that defines several attributes and methods like in the following example:

Python

```python
# workers.py

class Worker:
    def __init__(self, name, address, hourly_salary):
        self.name = name
        self.address = address
        self.hourly_salary = hourly_salary

    def show_profile(self):
        print("== Worker profile ==")
        print(f"Name: {self.name}")
        print(f"Address: {self.address}")
        print(f"Hourly salary: {self.hourly_salary}")

    def calculate_payroll(self, hours=40):
        return self.hourly_salary * hours
```

In this class, you define a few instance attributes to store important data about the current worker. You also provide the `.show_profile()` method to display relevant information about the worker. Finally, you write a generic `.calculate_payroll()` method to compute the salary of workers from their hourly salary and the number of hours worked.

Later in the development cycle, some requirements change. Now you realize that managers compute their salaries in a different way. They'll have an hourly bonus that you must add to the normal hourly salary before computing the final amount.

After thinking a bit about the problem, you decide that `Manager` has to override `.calculate_payroll()` completely. Here's the implementation that you come up with:

Python

```python
# workers.py

# ...

class Manager(Worker):
    def __init__(self, name, address, hourly_salary, hourly_bonus):
        super().__init__(name, address, hourly_salary)
        self.hourly_bonus = hourly_bonus

    def calculate_payroll(self, hours=40):
        return (self.hourly_salary + self.hourly_bonus) * hours
```

In the `Manager` initializer, you take the hourly bonus as an argument. Then you call the parent's `.__init__()` method as usual and define the `.hourly_bonus` instance attribute. Finally, you override `.calculate_payroll()` with a completely different implementation that doesn't reuse the inherited functionality.

Multiple Inheritance

In Python, you can use **multiple inheritance**. This type of inheritance allows you to create a class that inherits from several parents. The subclass will have access to attributes and methods from all its parents.

Multiple inheritance allows you to reuse code from several existing classes. However, you must manage the complexity of multiple inheritance with care. Otherwise, you can face issues like the diamond problem. You'll learn more about this topic in the Method Resolution Order (MRO) section.

Here's a small example of multiple inheritance in Python:

Python

```python
# crafts.py

class Vehicle:
    def __init__(self, make, model, color):
        self.make = make
        self.model = model
        self.color = color

    def start(self):
        print("Starting the engine...")

    def stop(self):
        print("Stopping the engine...")

    def show_technical_specs(self):
        print(f"Make: {self.make}")
        print(f"Model: {self.model}")
        print(f"Color: {self.color}")

class Car(Vehicle):
    def drive(self):
        print("Driving on the road...")

class Aircraft(Vehicle):
    def fly(self):
        print("Flying in the sky...")

class FlyingCar(Car, Aircraft):
    pass
```

In this example, you write a `Vehicle` class with `.make`, `.model`, and `.color` attributes. The class also has the `.start()`, `.stop()`, and `.show_technical_specs()` methods. Then you create a `Car` class that inherits from `Vehicle` and extends it with a new method called `.drive()`. You also create an `Aircraft` class that inherits from `Vehicle` and adds a `.fly()` method.

Finally, you define a `FlyingCar` class to represent a car that you can drive on the road or fly in the sky. Isn't that cool? Note that this class includes both `Car` and `Aircraft` in its list of parent classes. So, it'll inherit functionality from both superclasses.

Here's how you can use the `FlyingCar` class:

Python >>>

```
>>> from crafts import FlyingCar

>>> space_flyer = FlyingCar("Space", "Flyer", "Black")
>>> space_flyer.show_technical_specs()
Make: Space
Model: Flyer
Color: Black

>>> space_flyer.start()
Starting the engine...
>>> space_flyer.drive()
Driving on the road...
>>> space_flyer.fly()
Flying in the sky...
>>> space_flyer.stop()
Stopping the engine...
```

In this code snippet, you first create an instance of `FlyingCar`. Then you call all its methods, including the inherited ones. As you can see, multiple inheritance promotes code reuse, allowing you to use functionality from several base classes at the same time. By the way, if you get this `FlyingCar` to really fly, then make sure you don't stop the engine while you're flying!

Method Resolution Order (MRO)

When you're using multiple inheritance, you can face situations where one class inherits from two or more classes that have the same base class. This is known as the diamond problem. The real issue appears when multiple parents provide specific versions of the same method. In this case, it'd be difficult to determine which version of that method the subclass will end up using.

Python deals with this issue using a specific method resolution order (MRO). So, what is the method resolution order in Python? It's an algorithm that tells Python how to search for inherited methods in a multiple inheritance context. Python's MRO determines which implementation of a method or attribute to use when there are multiple versions of it in a class hierarchy.

Python's MRO is based on the order of parent classes in the subclass definition. For example, `Car` comes before `Aircraft` in the `FlyingCar` class from the previous section. MRO also considers the inheritance relationships between classes. In general, Python searches for methods and attributes in the following order:

1. The current class
2. The leftmost superclasses
3. The superclass listed next, from left to right, up to the last superclass
4. The superclasses of inherited classes
5. The `object` class

It's important to note that subclasses come first in the search. Additionally, if you have multiple parents that implement a given method or attributes, then Python will search them in the same order that they're listed in the class definition.

To illustrate the MRO, consider the following sample class hierarchy:

Python

```python
# mro.py

class A:
    def method(self):
        print("A.method")

class B(A):
    def method(self):
        print("B.method")

class C(A):
    def method(self):
        print("C.method")

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

In this example, `D` inherits from `B` and `C`, which inherit from `A`. All the superclasses in the hierarchy define a different version of `.method()`. Which of these versions will `D` end up calling? To answer this question, go ahead and call `.method()` on a `D` instance:

Python >>>

```
>>> from mro import D

>>> D().method()
B.method
```

When you call `.method()` on an instance of `D`, you get `B.method` on your screen. This means that Python found `.method()` on the `B` class first. That's the version of `.method()` that you end up calling. You ignore the versions from `C` and `A`.

> **Note:** Sometimes, you may run into complex inheritance relationships where Python won't be able to create a consistent method resolution order. In those cases, you'll get a `TypeError` pointing out the issue.

You can check the current MRO of a given class by using the `.__mro__` special attribute:

Python >>>

```
>>> D.__mro__
(
    <class '__main__.D'>,
    <class '__main__.B'>,
    <class '__main__.C'>,
    <class '__main__.A'>,
    <class 'object'>
)
```

In the output, you can see that Python searches for methods and attributes in `D` by going through `D` itself, then `B`, then `C`, then `A`, and finally, `object`, which is the base class of all Python classes.

The `.__mro__` attribute can help you tweak your classes and define the specific MRO that you want your class to use. The way to tweak this is by moving and reordering the parent classes in the subclass definition until you get the desired MRO.

Mixin Classes

A mixin class provides methods that you can reuse in many other classes. Mixin classes don't define new types, so they're not intended to be instantiated. You use their functionality to attach extra features to other classes quickly.

You can access the functionality of a mixin class in different ways. One of these ways is inheritance. However, inheriting from mixin classes doesn't imply an **is-a** relationship because these classes don't define concrete types. They just bundle specific functionality that's intended to be reused in other classes.

To illustrate how to use mixin classes, say that you're building a class hierarchy with a `Person` class at the top. From this class, you'll derive classes like `Employee`, `Student`, `Professor`, and several others. Then you realize that all the subclasses of `Person` need methods that serialize their data into different formats, including JSON and pickle.

With this in mind, you think of writing a `SerializerMixin` class that takes care of this task. Here's what you come up with:

Python

```
# mixins.py

import json
import pickle

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

class SerializerMixin:
    def to_json(self):
        return json.dumps(self.__dict__)

    def to_pickle(self):
        return pickle.dumps(self.__dict__)

class Employee(SerializerMixin, Person):
    def __init__(self, name, age, salary):
        super().__init__(name, age)
        self.salary = salary
```

In this example, `Person` is the parent class, and `SerializerMixin` is a mixin class that provides serialization functionality. The `Employee` class inherits from both `SerializerMixin` and `Person`. Therefore, it'll inherit the `.to_json()` and `.to_pickle()` methods, which you can use to serialize instances of `Employee` in your code.

In this example, `Employee` is a `Person`. However, it's not a `SerializerMixin` because this class doesn't define a type of object. It's just a mixin class that packs serialization capabilities.

> **Note:** Because of the method resolution order (MRO), which you learned about earlier, placing your mixin classes *before* the base classes on the list of parents is often necessary. It's especially true for class-based views in the Django web framework, which uses mixins to modify the behavior of a base view class.

Here's how `Employee` works in practice:

Python >>>

```
>>> from mixins import Employee

>>> john = Employee("John Doe", 30, 50000)
>>> john.to_json()
'{"name": "John", "age": 30, "salary": 50000}'

>>> john.to_pickle()
b'...\x04name\x94\x8c\x08John Doe\x94\x8c\x03age\x94K\x1e\x8c\x06salary...'
```

Now your `Employee` class is able to serialize its data using JSON and pickle formats. That's great! Can you think of any other useful mixin classes?

Up to this point, you've learned a lot about simple and multiple inheritance in Python. In the following section, you'll go through some of the advantages of using inheritance when writing and organizing your code.

Benefits of Using Inheritance

Inheritance is a powerful tool that you can use to model and solve many real-world problems in your code. Some benefits of using inheritance include the following:

- **Reusability**: You can quickly inherit and reuse working code from one or more parent classes in as many subclasses as you need.
- **Modularity**: You can use inheritance to organize your code in hierarchies of related classes.
- **Maintainability**: You can quickly fix issues or add features to a parent class. These changes will be automatically available in all its subclasses. Inheritance also reduces code duplication.
- **Polymorphism**: You can create subclasses that can replace their parent class, providing the same or equivalent functionality.
- **Extensibility**: You can quickly extend an exiting class by adding new data and behavior to its subclasses.

You can also use inheritance to define a uniform API for all the classes that belong to a given hierarchy. This promotes consistency and leverages polymorphism.

Using classes and inheritance, you can make your code more modular, reusable, and extensible. Inheritance enables you to apply good design principles, such as separation of concerns. This principle states that you should organize code in small classes that each take care of a single task.

Even though inheritance comes with several benefits, it can also end up causing issues. If you overuse it or use it incorrectly, then you can:

- Artificially increase your code's complexity with multiple inheritance or multiple levels of inheritance
- Face issues like the diamond problem where you'll have to deal with the method resolution order
- End up with fragile base classes where changes to a parent class produce unexpected behaviors in subclasses

Of course, these aren't the only potential pitfalls. For example, having multiple levels of inheritance can make your code harder to reason about, which may impact your code's maintainability in the long term.

Another drawback of inheritance is that inheritance is defined at compile time. So, there's no way to change the inherited functionality at runtime. Other techniques, like composition, allow you to dynamically change the functionality of a given class by replacing its components.

Using Alternatives to Inheritance

Inheritance, and especially multiple inheritance, can be a complex and hard-to-grasp topic. Fortunately, inheritance isn't the only technique that allows you to reuse functionality in object-oriented programming. You also have composition, which represents a **has-a** relationship between classes.

Composition allows you to build an object from its components. The **composite** object doesn't have direct access to each component's interface. However, it can leverage each component's implementation.

Delegation is another technique that you can use to promote code reuse in your OOP programs. With delegation, you can represent **can-do** relationships, where an object relies on another object to perform a given task.

In the following sections, you'll learn more about these techniques and how they can make your object-oriented code more robust and flexible.

Composition

As you already know, you can use **composition** to model a **has-a** relationship between objects. In other words, through composition, you can create complex objects by combining objects that will work as **components**. Note that these components may not make sense as stand-alone classes.

Favoring composition over inheritance leads to more flexible class designs. Unlike inheritance, composition is defined at runtime, which means that you can dynamically replace a current component with another component of the same type. This characteristic makes it possible to change the composite's behavior at runtime.

In the example below, you use composition to create an `IndustrialRobot` class from the `Body` and `Arm` components:

Python

```python
# robot.py

class IndustrialRobot:
    def __init__(self):
        self.body = Body()
        self.arm = Arm()

    def rotate_body_left(self, degrees=10):
        self.body.rotate_left(degrees)

    def rotate_body_right(self, degrees=10):
        self.body.rotate_right(degrees)

    def move_arm_up(self, distance=10):
        self.arm.move_up(distance)

    def move_arm_down(self, distance=10):
        self.arm.move_down(distance)

    def weld(self):
        self.arm.weld()

class Body:
    def __init__(self):
        self.rotation = 0

    def rotate_left(self, degrees=10):
        self.rotation -= degrees
        print(f"Rotating body {degrees} degrees to the left...")

    def rotate_right(self, degrees=10):
        self.rotation += degrees
        print(f"Rotating body {degrees} degrees to the right...")

class Arm:
    def __init__(self):
        self.position = 0

    def move_up(self, distance=1):
        self.position += 1
        print(f"Moving arm {distance} cm up...")

    def move_down(self, distance=1):
        self.position -= 1
        print(f"Moving arm {distance} cm down...")

    def weld(self):
        print("Welding...")
```

In this example, you build an `IndustrialRobot` class out of its components, `Body` and `Arm`. The `Body` class provides horizontal movements, while the `Arm` class represents the robot's arm and provides vertical movement and welding functionality.

Here's how you can use `IndustrialRobot` in your code:

Python >>>

```
>>> from robot import IndustrialRobot

>>> robot = IndustrialRobot()

>>> robot.rotate_body_left()
Rotating body 10 degrees to the left...
>>> robot.move_arm_up(15)
Moving arm 15 cm up...
>>> robot.weld()
Welding...

>>> robot.rotate_body_right(20)
Rotating body 20 degrees to the right...
>>> robot.move_arm_down(5)
Moving arm 5 cm down...
>>> robot.weld()
Welding...
```

Great! Your robot works as expected. It allows you to move its body and arm according to your movement needs. It also allows you to weld different mechanical pieces together.

An idea to make this robot even cooler is to implement several types of arms with different welding technologies. Then you can change the arm by running `robot.arm = NewArm()`. You can even add a `.change_arm()` method to your robot class. How does that sound as a learning exercise?

Unlike inheritance, composition doesn't expose the entire interface of components, so it preserves encapsulation. Instead, the composite objects access and use only the required functionality from their components. This characteristic makes your class design more robust and reliable because it won't expose unneeded members.

Following the robot example, say you have several different robots in a factory. Each robot can have different capabilities like welding, cutting, shaping, polishing, and so on. You also have several independent arms. Some of them can perform all those actions. Some of them can perform just a subset of the actions.

Now say that a given robot can only weld. However, this robot can use different arms with different welding technologies. If you use inheritance, then the robot will have access to other operations like cutting and shaping, which can cause an accident or breakdown.

If you use composition, then the welder robot will only have access to the arm's welding feature. That said, composition can help you protect your classes from unintended use.

Delegation

Delegation is another technique that you can use as an alternative to inheritance. With delegation, you can model **can-do** relationships, where an object hands a task over to another object, which takes care of executing the task. Note that the delegated object can exist independently from the delegator.

You can use delegation to achieve code reuse, separation of concerns, and modularity. For example, say that you want to create a stack data structure. You think of taking advantage of Python's `list` as a quick way to store and manipulate the underlying data.

Here's how you end up writing your `Stack` class:

Python

```
# stack.py

class Stack:
    def __init__(self, items=None):
        if items is None:
            self._items = []
        else:
            self._items = list(items)

    def push(self, item):
        self._items.append(item)

    def pop(self):
        return self._items.pop()

    def __repr__(self) -> str:
        return f"{type(self).__name__}({self._items})"
```

In `.__init__()`, you define a `list` object called `._items` that can take its initial data from the `items` argument. You'll use this list to store the data in the containing `Stack`, so you delegate all the operations related to storing, adding, and deleting data to this list object. Then you implement the typical `Stack` operations, `.push()` and `.pop()`.

Note how these operations conveniently delegate their responsibilities on `._items.append()` and `._items.pop()`, respectively. Your `Stack` class has handed its operations over to the `list` object, which already knows how to perform them.

It's important to notice that this class is pretty flexible. You can replace the list object in `._items` with any other object as long as it implements the `.pop()` and `.append()` methods. For example, you can use a `deque` object from the `collections` module.

Because you've used delegation to write your class, the internal implementation of `list` isn't visible or directly accessible in `Stack`, which preserves encapsulation:

Python >>>

```
>>> from stack import Stack

>>> stack = Stack([1, 2, 3])
>>> stack
Stack([1, 2, 3])
>>> stack.push(4)
>>> stack
Stack([1, 2, 3, 4])
>>> stack.pop()
>>> stack.pop()
>>> stack
Stack([1, 2])

>>> dir(stack)
[
    ...
    '_items',
    'pop',
    'push'
]
```

The public interface of your `Stack` class only contains the stack-related methods `.pop()` and `.push()`, as you can see in the `dir()` function's output. This prevents the users of your class from using list-specific methods that aren't compatible with the classic stack data structure.

If you use inheritance, then your child class, `Stack`, will inherit all the functionality from its parent class, `list`:

Python >>>

```
>>> class Stack(list):
...     def push(self, item):
...         self.append(item)
...     def pop(self):
...         return super().pop()
...     def __repr__(self) -> str:
...         return f"{type(self).__name__}({super().__repr__()})"
...

>>> stack = Stack()
>>> dir(stack)
[
    ...
    'append',
    'clear',
    'copy',
    'count',
    'extend',
    'index',
    'insert',
    'pop',
    'push',
    'remove',
    'reverse',
    'sort'
]
```

In this example, your `Stack` class has inherited all the methods from `list`. These methods are exposed as part of your class's public API, which may lead to incorrect uses of the class and its instances.

With inheritance, the internals of parent classes are visible to subclasses, which breaks encapsulation. If some of the parent's functionality isn't appropriate for the child, then you run the risk of incorrect use. In this situation, composition and delegation are safer options.

Finally, in Python, you can quickly implement delegation through the `.__getattr__()` special method. Python calls this method automatically whenever you access an instance attribute or method. You can use this method to redirect the request to another object that can provide the appropriate method or attribute.

To illustrate this technique, get back to the mixin example where you used a mixin class to provide serialization capabilities to your `Employee` class. Here's how to rewrite the example using delegation:

Python

```python
# serializer_delegation.py

import json
import pickle

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

class Serializer:
    def __init__(self, instance):
        self.instance = instance

    def to_json(self):
        return json.dumps(self.instance.__dict__)

    def to_pickle(self):
        return pickle.dumps(self.instance.__dict__)

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

    def __getattr__(self, attr):
        return getattr(Serializer(self), attr)
```

In this new implementation, the serializer class takes the instance that provides the data as an argument. `Employee` defines a `.__getattr__()` method that uses the built-in `getattr()` function to access the methods in the `Serializer` class.

For example, if you call `.to_json()` on an instance of `Employee`, then that call will be automatically redirected to calling `.to_json()` on the instance of `Serializer`. Go ahead and try it out! This is a pretty cool Python feature.

You've tried your hand at a quick example of delegation in Python to learn how a class can delegate some of its responsibilities to another class, achieving code reuse and separation of concerns. Again, you should note that this technique indirectly exposes all the delegated attributes and methods. So, use it with care.

Dependency Injection

Dependency injection is a design pattern that you can use to achieve loose coupling between a class and its components. With this technique, you can provide an object's dependencies from the outside, rather than inheriting or implementing them in the object itself. This practice allows you to create flexible classes that are able to change their behavior dynamically, depending on the injected functionality.

In your robot example, you can use dependency injection to decouple the `Arm` and `Body` classes from `IndustrialRobot`, which will make your code more flexible and versatile.

Here's the updated example:

Python

```python
# robot.py

class IndustrialRobot:
    def __init__(self, body, arm):
        self.body = body
        self.arm = arm

    # ...

# ...
```

In this new version of `IndustrialRobot`, you only made two small changes to `.__init__()`. Now this method takes `body` and `arm` as arguments and assigns their values to the corresponding instance attributes, `.body` and `.arm`. This allows you to inject appropriate body and arm objects into the class so that it can do its work.

Here's how you can use `IndustrialRobot` with this new implementation:

Python >>>

```
>>> from robot import Arm, Body, IndustrialRobot

>>> robot = IndustrialRobot(Body(), Arm())

>>> robot.rotate_body_left()
Rotating body 10 degrees to the left...
>>> robot.move_arm_up(15)
Moving arm 15 cm up...
>>> robot.weld()
Welding...

>>> robot.rotate_body_right(20)
Rotating body 20 degrees to the right...
>>> robot.move_arm_down(5)
Moving arm 5 cm down...
>>> robot.weld()
Welding...
```

Overall, the class's functionality remains the same as in your first version. The only difference is that now you have to pass the body and arm objects to the class constructor. This step is a common way of implementing dependency injection.

Now that you know about a few techniques that you can use as alternatives to inheritance, it's time for you to learn about abstract base classes (ABCs) in Python. These classes allow you to define consistent APIs for your classes.

Creating Abstract Base Classes (ABCs) and Interfaces

Sometimes, you want to create a class hierarchy in which all the classes implement a predefined interface or API. In other words, you want to define the specific set of public methods and attributes that all the classes in the hierarchy must implement. In Python, you can do this using what's known as an **abstract base class (ABC)**.

The `abc` module in the standard library exports a couple of ABCs and other related tools that you can use to define custom base classes that require all their subclasses to implement specific interfaces.

You can't instantiate ABCs directly. You must subclass them. In a sense, ABCs work as templates for other classes to inherit from.

To illustrate how to use Python's ABCs, say that you want to create a class hierarchy to represent different shapes, such as `Circle`, `Square`, and so on. You decide that all the classes should have the `.get_area()` and `.get_perimeter()` methods. In this situation, you can start with the following base class:

Python

```
# shapes_abc.py

from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def get_area(self):
        pass

    @abstractmethod
    def get_perimeter(self):
        pass
```

The `Shape` class inherits from `abc.ABC`, which means it's an abstract base class. Then you define the `.get_area()` and `.get_perimeter()` methods using the `@abstractmethod` decorator. By using the `@abstractmethod` decorator, you declare that these two methods are the common interface that all the subclasses of `Shape` must implement.

Now you can create the `Circle` class. Here's the first approach to this class:

Python

```python
# shapes_abc.py

from abc import ABC, abstractmethod
from math import pi

# ...

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def get_area(self):
        return pi * self.radius ** 2
```

In this code snippet, you define the `Circle` class by inheriting from `Shape`. At this point, you've added the `.get_area()` method only. Now go ahead and run the following code:

Python >>>

```
>>> from shapes_abc import Circle

>>> circle = Circle(100)
Traceback (most recent call last):
    ...
TypeError: Can't instantiate abstract class Circle
    with abstract method get_perimeter
```

What just happened? You can't instantiate `Circle`. That's what ABCs are for. To be able to instantiate `Circle`, you must provide suitable implementations for all its abstract methods, which means you need to define a consistent interface:

Python

```python
# shapes_abc.py

from abc import ABC, abstractmethod
from math import pi

# ...

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def get_area(self):
        return pi * self.radius ** 2

    def get_perimeter(self):
        return 2 * pi * self.radius
```

This time, your `Circle` class implements all the required methods. These methods will be common to all the classes in your shape hierarchy. Once you've defined suitable custom implementations for all the abstract methods, you can proceed to instantiate `Circle` as in the following example:

Python >>>

```
>>> from shapes_abc import Circle

>>> circle = Circle(100)
>>> circle.radius
100
>>> circle.get_area()
31415.926535897932
>>> circle.get_perimeter()
628.3185307179587
```

Once you've implemented custom methods to replace the abstract implementations of `.get_area()` and `.get_perimeter()`, then you can instantiate and use `Circle` in your code.

If you want to add a `Square` class to your shape hierarchy, then that class must have custom implementations of the `.get_area()` and `.get_perimeter()` methods:

Python

```
# shapes_abc.py

from abc import ABC, abstractmethod
from math import pi

# ...

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

    def get_area(self):
        return self.side ** 2

    def get_perimeter(self):
        return 4 * self.side
```

This example demonstrates how you can use ABCs to define a common interface for a group of related classes. Each subclass must provide its own implementation of the abstract methods in the base class. Note that in recent Python versions, you can use static duck typing as an alternative to abstract base classes.

Unlocking Polymorphism With Common Interfaces

In the previous section, you learned about abstract base classes and explored how to use them to promote the use of a common public interface across several related classes. Having a set of classes to implement the same interface with specific behaviors for concrete classes is a great way to unlock polymorphism.

Polymorphism is when you can use objects of different classes interchangeably because they share a common interface. For example, Python strings, lists, and tuples are all sequence data types. This means that they implement an interface that's common to all sequences.

Because of this common interface, you can use them in similar ways. For example, you can:

- Use them in loops because they provide the `.__iter__()` method
- Access their items by index because they implement the `.__getitem__()` method
- Determine their number of items because they include the `.__len__()` method

These are just a few examples of common features of sequence data types. Note that you can run all these operations and more without caring about which specific type you're actually using in your code. That's possible because of polymorphism.

Consider the following examples, which use the built-in `len()` function:

Python >>>

```
>>> message = "Hello!"
>>> numbers = [1, 2, 3]
>>> letters = ("A", "B", "C")

>>> len(message)
6
>>> len(numbers)
3
>>> len(letters)
3
```

In these examples, you use `len()` with three different types of objects: a string, a list, and a tuple. Even though these types are quite different, all of them implement the `.__len__()` method, which provides support for `len()`.

You can unlock polymorphism in your custom classes and classes by making them share common attributes and methods.

For example, take a look at your `Vehicle` class hierarchy. The `Car` class has a method called `.drive()`, and the `Motorcycle` class has a method called `.ride()`. This API inconsistency breaks polymorphism. To fix the issue and use these classes in a polymorphic way, you can slightly change `Motorcycle` by renaming its `.ride()` method to `.drive()`:

Python

```
# vehicles.py

# ...

class Motorcycle(Vehicle):
    def __init__(self, make, model, year, num_wheels):
        super().__init__(make, model, year)
        self.num_wheels = num_wheels

    def drive(self):
        print(f'Riding my "{self.make} - {self.model}" on the road')

    def __str__(self):
        return f'"{self.make} - {self.model}" has {self.num_wheels} wheels'
```

Here, you rename the `.ride()` method to `.drive()` in the definition of `Motorcycle`. This small change makes your classes have a common interface. So, you can use them interchangeably in your code:

Python >>>

```
>>> from vehicles import Car, Motorcycle

>>> toyota = Car("Toyota", "Corolla", 2022, 5)
>>> honda = Car("Honda", "Civic", 2022, 4)
>>> harley = Motorcycle("Harley-Davidson", "Iron 883", 2022, 2)
>>> indian = Motorcycle("Indian", "Scout", 2022, 2)

>>> for vehicle in [toyota, honda, harley, indian]:
...     vehicle.drive()
...
Driving my "Toyota - Corolla" on the road
Driving my "Honda - Civic" on the road
Riding my "Harley-Davidson - Iron 883" on the road
Riding my "Indian - Scout" on the road
```

Now you can drive either a car or a motorcycle without having to worry about an `AttributeError` because one of them doesn't have the appropriate method. You've just made your classes work in a polymorphic way, which is a great way to add flexibility to your code.

Conclusion

You now know a lot about Python **classes** and how to use them to make your code more reusable, modular, flexible, and maintainable. Classes are the building blocks of object-oriented programming in Python. With classes, you can solve complex problems by modeling real-world objects, their properties, and their behaviors. Classes provide an intuitive and human-friendly approach to complex programming problems, which will make your life more pleasant.

In this tutorial, you've learned how to:

- Write Python classes using the **`class` keyword**
- Add state to your classes with **class** and **instance attributes**
- Give concrete **behaviors** to your classes with different types of **methods**
- Build hierarchies of classes using **inheritance**
- Create **interfaces** with **abstract classes**

With all this knowledge, you can leverage the power of Python classes in your code. Now you're ready to start writing your own classes in Python.

Free Bonus: Click here to download your sample code for building powerful object blueprints with classes in