Python OOP: The Missing Pieces

Table of Contents

The "Java-style" approach (works, but not Pythonic)

Usage feels awkward

Usage feels natural

Now we have validated attribute access

Usage

Usage

Usage

Usage

None of these classes inherit from each other

But they all work because they have the same methods

Works with actual files

Works with StringIO (fake file in memory)

Works with BytesIO

All of these work because they’re all "iterable"

Now it works with for loops!
And with list()
Usage
Both work with the same function thanks to duck typing

---

1. Introduction

So you've learned about classes, objects, the three pillars of OOP, and even touched on composition vs inheritance. That's a solid foundation. But Python has a few more tricks up its sleeve that make working with objects feel natural and "Pythonic."

Let's dive into three topics that often get glossed over but are essential for writing clean, professional Python code.

2. The @property Decorator

Remember encapsulation? We learned that we can make attributes "private" using double underscores to protect them from outside interference. But here's the thing: sometimes you *do* need controlled access to those private attributes.

In languages like Java, you'd write explicit `getX()` and `setX()` methods. It works, but it's verbose and clunky:

PYTHON

```
# The "Java-style" approach (works, but not Pythonic)
class Circle:
    def __init__(self, radius):
        self.__radius = radius

    def get_radius(self):
        return self.__radius

    def set_radius(self, value):
        if value > 0:
            self.__radius = value
        else:
            raise ValueError("Radius must be positive")

# Usage feels awkward
c = Circle(5)
print(c.get_radius())  # 5
c.set_radius(10)
```

Python offers a more elegant solution: the `@property` decorator. It lets you define methods that *look* like simple attribute access but actually run your custom code behind the scenes.

2.1. Creating a Read-Only Property

PYTHON

```
class Circle:
    def __init__(self, radius):
        self._radius = radius  # Single underscore: "protected" by convention

    @property
    def radius(self):
        """The radius property (read-only for now)."""
        return self._radius

    @property
    def area(self):
        """Calculated property — no stored value needed."""
        return 3.14159 * self._radius ** 2

# Usage feels natural
c = Circle(5)
print(c.radius)  # 5 — looks like an attribute, but it's a method!
print(c.area)    # 78.53975 — calculated on the fly
```

Notice how we access `radius` and `area` without parentheses. From the outside, they look like regular attributes. But behind the scenes, Python is calling our methods.

2.2. Adding a Setter: Controlled Write Access

What if we want to allow changing the radius, but with validation? We add a setter using `@property_name.setter`:

PYTHON

```
class Circle:
    def __init__(self, radius):
        self._radius = None  # Will be set by the setter
        self.radius = radius  # Use the setter for validation

    @property
    def radius(self):
        """Get the radius."""
        return self._radius

    @radius.setter
    def radius(self, value):
        """Set the radius with validation."""
        if value <= 0:
            raise ValueError("Radius must be positive")
        self._radius = value

    @property
    def area(self):
        """Calculated property."""
        return 3.14159 * self._radius ** 2

# Now we have validated attribute access
c = Circle(5)
print(c.radius)    # 5

c.radius = 10      # Works fine
print(c.radius)    # 10

c.radius = -5      # Raises ValueError: Radius must be positive
```

2.3. The Deleter: Cleaning Up

For completeness, you can also define what happens when someone tries to delete the attribute:

PYTHON

```
@radius.deleter
def radius(self):
    """Handle deletion of radius."""
    print("Deleting radius...")
    self._radius = None

# Usage
del c.radius  # Prints: Deleting radius...
```

> **NOTE** The deleter is rarely needed, but it's available when you need it.

2.4. Why Use Properties?

Properties offer several advantages:

- **Clean interface**: Users of your class interact with simple attributes, not method calls
- **Validation**: You can enforce rules when values are set
- **Calculated attributes**: Derive values on-the-fly without storing them
- **Backward compatibility**: You can start with a simple attribute and later add a property without changing the interface

2.5. Complete Example: Temperature Converter

PYTHON

```
class Temperature:
    """A temperature that can be accessed in Celsius or Fahrenheit."""

    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        """Temperature in Celsius."""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature cannot be below absolute zero")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Temperature in Fahrenheit (calculated)."""
        return (self._celsius * 9/5) + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        """Set temperature using Fahrenheit."""
        celsius_value = (value - 32) * 5/9
        if celsius_value < -273.15:
            raise ValueError("Temperature cannot be below absolute zero")
        self._celsius = celsius_value

    def __repr__(self):
        return f"Temperature({self._celsius}°C / {self.fahrenheit}°F)"


# Usage
temp = Temperature(25)
print(temp)              # Temperature(25°C / 77.0°F)

temp.fahrenheit = 100
print(temp)              # Temperature(37.77...°C / 100°F)

print(temp.celsius)      # 37.77...
print(temp.fahrenheit)   # 100
```

Both `celsius` and `fahrenheit` feel like simple attributes, but they're actually properties with logic behind them. The user doesn't need to know or care about the implementation.

3. Understanding super()

We touched on inheritance and mentioned `super()`, but let's really dig into what it does and why it matters.

When a child class inherits from a parent, sometimes you want to *extend* the parent's behavior rather than completely replace it. That's where `super()` comes in.

3.1. The Problem: Repeating Yourself

Imagine you're building on the `Animal` example:

PYTHON

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.is_alive = True

    def speak(self):
        raise NotImplementedError("Subclass must implement")

class Dog(Animal):
    def __init__(self, name, age, breed):
        # Without super(), you'd have to repeat the parent's work:
        self.name = name      # Duplicated!
        self.age = age        # Duplicated!
        self.is_alive = True  # Duplicated!
        self.breed = breed    # Only this is new

    def speak(self):
        return f"{self.name} says Woof!"
```

This works, but it violates DRY (Don't Repeat Yourself). If `Animal.__init__` changes, you'd have to update every child class. That's a maintenance nightmare.

3.2. The Solution: Using super()

PYTHON

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.is_alive = True

    def speak(self):
        raise NotImplementedError("Subclass must implement")

    def describe(self):
        return f"{self.name} is {self.age} years old"

class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, age)  # Call the parent's __init__
        self.breed = breed           # Add dog-specific attribute

    def speak(self):
        return f"{self.name} says Woof!"

    def describe(self):
        # Extend the parent's describe method
        base_description = super().describe()
        return f"{base_description} and is a {self.breed}"


# Usage
rex = Dog("Rex", 5, "German Shepherd")

print(rex.name)        # Rex (set by parent)
print(rex.age)         # 5 (set by parent)
print(rex.is_alive)    # True (set by parent)
print(rex.breed)       # German Shepherd (set by child)

print(rex.speak())     # Rex says Woof!
print(rex.describe())  # Rex is 5 years old and is a German Shepherd
```

3.3. What super() Actually Returns

When you call `super()`, Python returns a special proxy object that delegates method calls to the parent class. It's essentially saying: "Give me a way to call methods from my parent."

PYTHON

```
super().__init__(name, age)
```

This is equivalent to the older, more explicit syntax:

PYTHON

```
Animal.__init__(self, name, age)
```

TIP

The `super()` syntax is preferred because:

- It's cleaner and more readable
- It works correctly with multiple inheritance (more advanced topic)
- If you rename the parent class, you don't have to update the child

3.4. Extending Methods Beyond *init*

You can use `super()` in any method, not just `*init*`:

PYTHON

```
class Animal:
    def eat(self, food):
        print(f"{self.name} is eating {food}")
        self.hunger = 0

class Dog(Animal):
    def eat(self, food):
        if food == "chocolate":
            print(f"No! Chocolate is toxic for dogs!")
            return
        super().eat(food)  # Call parent's eat method
        print(f"{self.name} wags tail happily")


rex = Dog("Rex", 5, "German Shepherd")
rex.hunger = 100

rex.eat("chocolate")
# Output: No! Chocolate is toxic for dogs!

rex.eat("kibble")
# Output: Rex is eating kibble
#         Rex wags tail happily
```

3.5. Practical Example: Building a User System

PYTHON

```
class User:
    """Base user class."""

    def __init__(self, username, email):
        self.username = username
        self.email = email
        self.is_active = True
        self.created_at = "2025-01-01"  # Simplified

    def get_permissions(self):
        return ["read"]

    def __repr__(self):
        return f"User({self.username})"


class AdminUser(User):
    """Admin with elevated permissions."""

    def __init__(self, username, email, admin_level=1):
        super().__init__(username, email)  # Set up base user stuff
        self.admin_level = admin_level

    def get_permissions(self):
        # Start with base permissions, then add more
        base_permissions = super().get_permissions()
        admin_permissions = ["write", "delete"]
        if self.admin_level >= 2:
            admin_permissions.append("manage_users")
        return base_permissions + admin_permissions

    def __repr__(self):
        return f"AdminUser({self.username}, level={self.admin_level})"


class SuperAdmin(AdminUser):
    """Super admin with all permissions."""

    def __init__(self, username, email):
        super().__init__(username, email, admin_level=3)

    def get_permissions(self):
        return super().get_permissions() + ["system_config", "view_logs"]


# Usage
regular = User("alice", "alice@example.com")
admin = AdminUser("bob", "bob@example.com", admin_level=2)
super_admin = SuperAdmin("charlie", "charlie@example.com")

print(regular.get_permissions())
# ['read']

print(admin.get_permissions())
# ['read', 'write', 'delete', 'manage_users']

print(super_admin.get_permissions())
# ['read', 'write', 'delete', 'manage_users', 'system_config', 'view_logs']
```

Each level builds on the previous one, and `super()` makes it seamless.

4. Duck Typing

The presentation mentioned duck typing briefly, but it deserves a deeper look because it's fundamental to how Python thinks about objects.

4.1. The Philosophy

The name comes from the saying:

> *If it walks like a duck and quacks like a duck, then it probably is a duck.*

In Python terms: **we don't care what type an object *is*, only what it can *do*.**

Unlike languages like Java or C++, Python doesn't require objects to inherit from a common parent or implement a formal interface. If an object has the method you need, you can use it.

4.2. A Simple Example

PYTHON

```
class Duck:
    def speak(self):
        return "Quack!"

    def swim(self):
        return "Duck is swimming"

class Person:
    def speak(self):
        return "Hello!"

    def swim(self):
        return "Person is swimming"

class Robot:
    def speak(self):
        return "Beep boop!"

    def swim(self):
        return "Robot is short-circuiting!"

def make_it_speak(thing):
    """This function doesn't care about the type of 'thing'."""
    print(thing.speak())

def pool_party(participants):
    """Everyone goes swimming!"""
    for participant in participants:
        print(participant.swim())


# None of these classes inherit from each other
# But they all work because they have the same methods
duck = Duck()
person = Person()
robot = Robot()

make_it_speak(duck)    # Quack!
make_it_speak(person)  # Hello!
make_it_speak(robot)   # Beep boop!

pool_party([duck, person, robot])
# Duck is swimming
# Person is swimming
# Robot is short-circuiting!
```

The `make_it_speak` function works with *any* object that has a `speak` method. It doesn't check types, it just tries to use the method.

4.3. Duck Typing vs Traditional Polymorphism

With traditional inheritance-based polymorphism, you'd write:

PYTHON

```
class Speakable:
    def speak(self):
        raise NotImplementedError

class Duck(Speakable):
    def speak(self):
        return "Quack!"

class Person(Speakable):
    def speak(self):
        return "Hello!"
```

This works, but Python's duck typing lets you skip the formal hierarchy entirely. Your classes don't need to know about each other or share a parent.

4.4. Real-World Duck Typing: File-Like Objects

Python's standard library uses duck typing extensively. Consider reading data:

PYTHON

```
def process_data(data_source):
    """Process data from anything that has a read() method."""
    content = data_source.read()
    return content.upper()

# Works with actual files
with open("myfile.txt") as f:
    result = process_data(f)

# Works with StringIO (fake file in memory)
from io import StringIO
fake_file = StringIO("hello world")
result = process_data(fake_file)  # "HELLO WORLD"

# Works with BytesIO
from io import BytesIO
byte_file = BytesIO(b"hello world")
# (would need .decode() but you get the idea)
```

The `process_data` function doesn't check if it received a "real" file. It just needs something with a `read()` method.

4.5. Duck Typing with Iteration

You've been using duck typing since Chapter 2 without realizing it:

PYTHON

```
def print_all(items):
    """Print each item. Works with anything iterable."""
    for item in items:
        print(item)

# All of these work because they're all "iterable"
print_all([1, 2, 3])           # List
print_all((1, 2, 3))           # Tuple
print_all({1, 2, 3})           # Set
print_all("abc")               # String
print_all(range(3))            # Range
print_all({"a": 1, "b": 2})    # Dict (iterates keys)
```

The `for` loop doesn't check types. It just needs an object that supports iteration (has `__iter__` method).

4.6. Making Your Classes Duck-Type Friendly

Want your custom class to work with `for` loops? Just implement the right methods:

PYTHON

```
class Countdown:
    """A countdown that can be iterated."""

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

    def __iter__(self):
        """Make this class iterable."""
        self.current = self.start
        return self

    def __next__(self):
        """Return the next value."""
        if self.current < 0:
            raise StopIteration
        value = self.current
        self.current -= 1
        return value


# Now it works with for loops!
for num in Countdown(5):
    print(num)
# Output: 5, 4, 3, 2, 1, 0

# And with list()
numbers = list(Countdown(3))  # [3, 2, 1, 0]
```

4.7. Handling Duck Typing Failures

What happens when duck typing fails? The object doesn't have the method you expected:

PYTHON

```
class Rock:
    pass  # Rocks don't speak

def make_it_speak(thing):
    print(thing.speak())

rock = Rock()
make_it_speak(rock)  # AttributeError: 'Rock' object has no attribute 'speak'
```

You have options for handling this:

4.7.1. Option 1: EAFP (Easier to Ask Forgiveness than Permission)

This is the Pythonic way:

```python
def make_it_speak(thing):
    try:
        print(thing.speak())
    except AttributeError:
        print(f"{type(thing).__name__} cannot speak")

make_it_speak(Rock())  # "Rock cannot speak"
```

4.7.2. Option 2: LBYL (Look Before You Leap)

Check first using `hasattr()`:

```python
def make_it_speak(thing):
    if hasattr(thing, 'speak'):
        print(thing.speak())
    else:
        print(f"{type(thing).__name__} cannot speak")
```

4.7.3. Option 3: Use callable() for Methods

```python
def make_it_speak(thing):
    speak_method = getattr(thing, 'speak', None)
    if callable(speak_method):
        print(speak_method())
    else:
        print(f"{type(thing).__name__} cannot speak")
```

NOTE

EAFP (try/except) is generally preferred in Python because:

- It's often faster when the method usually exists
- It's more Pythonic
- It handles edge cases better

4.8. Duck Typing Summary

Concept	Description
Core Idea	Care about behavior, not type
Benefit	Flexible, loosely-coupled code
Risk	Runtime errors if object lacks expected method
Best Practice	Use try/except (EAFP) for graceful handling
Common Uses	Iteration, file operations, context managers

5. Putting It All Together

Let's build a small example that combines all three concepts:

PYTHON

```
class DataSource:
    """Base class for data sources using properties and meant for duck typing."""

    def __init__(self, name):
        self._name = name
        self._data = []

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

    @property
    def record_count(self):
        """Calculated property."""
        return len(self._data)

    def read(self):
        """Duck typing target: anything with read() can be a data source."""
        raise NotImplementedError


class CSVSource(DataSource):
    """Reads data from a CSV-like format."""

    def __init__(self, name, raw_text):
        super().__init__(name)  # Call parent's __init__
        self._raw_text = raw_text
    = Python OOP: The Missing Pieces
:toc:
:toc-placement: left
:toclevels: 3
:sectnums:
:source-highlighter: pygments

== Introduction

So you've learned about classes, objects, the three pillars of OOP, and even touched on composition vs
inheritance. That's a solid foundation. But Python has a few more tricks up its sleeve that make working with
objects feel natural and "Pythonic."

Let's dive into three topics that often get glossed over but are essential for writing clean, professional Python
code.

== The @property Decorator

Remember encapsulation? We learned that we can make attributes "private" using double underscores to protect them
from outside interference. But here's the thing: sometimes you _do_ need controlled access to those private
attributes.

In languages like Java, you'd write explicit `getX()` and `setX()` methods. It works, but it's verbose and clunky:

[source,python]
```

The "Java-style" approach (works, but not Pythonic)

class Circle: def *init*(self, radius): self.__radius = radius

```
def get_radius(self):
    return self.__radius


def set_radius(self, value):
    if value > 0:
        self.__radius = value
    else:
        raise ValueError("Radius must be positive")
```

Usage feels awkward

c = Circle(5) print(c.get_radius()) # 5 c.set_radius(10)

```
Python offers a more elegant solution: the `@property` decorator. It lets you define methods that _look_ like
simple attribute access but actually run your custom code behind the scenes.

=== Creating a Read-Only Property

[source,python]
```

class Circle: def *init*(self, radius): self._radius = radius # Single underscore: "protected" by convention

```
@property
def radius(self):
    """The radius property (read-only for now)."""
    return self._radius


@property
def area(self):
    """Calculated property – no stored value needed."""
    return 3.14159 * self._radius ** 2
```

Usage feels natural

c = Circle(5) print(c.radius) # 5 — looks like an attribute, but it’s a method! print(c.area) # 78.53975 — calculated on the fly

```
Notice how we access `radius` and `area` without parentheses. From the outside, they look like regular attributes.
But behind the scenes, Python is calling our methods.

=== Adding a Setter: Controlled Write Access

What if we want to allow changing the radius, but with validation? We add a setter using `@property_name.setter`:

[source,python]
```

class Circle: def *init*(self, radius): self._radius = None # Will be set by the setter self.radius = radius # Use the setter for validation

```
    @property
    def radius(self):
        """Get the radius."""
        return self._radius


    @radius.setter
    def radius(self, value):
        """Set the radius with validation."""
        if value <= 0:
            raise ValueError("Radius must be positive")
        self._radius = value


    @property
    def area(self):
        """Calculated property."""
        return 3.14159 * self._radius ** 2
```

Now we have validated attribute access

c = Circle(5) print(c.radius) # 5

c.radius = 10 # Works fine print(c.radius) # 10

c.radius = -5 # Raises ValueError: Radius must be positive

```
=== The Deleter: Cleaning Up

For completeness, you can also define what happens when someone tries to delete the attribute:

[source,python]
```

@radius.deleter def radius(self): """Handle deletion of radius.""" print("Deleting radius...") self._radius = None

Usage

del c.radius # Prints: Deleting radius...

```
[NOTE]
====
The deleter is rarely needed, but it's available when you need it.
====

=== Why Use Properties?

Properties offer several advantages:

* **Clean interface**: Users of your class interact with simple attributes, not method calls
* **Validation**: You can enforce rules when values are set
* **Calculated attributes**: Derive values on-the-fly without storing them
* **Backward compatibility**: You can start with a simple attribute and later add a property without changing the
interface

=== Complete Example: Temperature Converter

[source,python]
```

class Temperature: """A temperature that can be accessed in Celsius or Fahrenheit."""

```
def __init__(self, celsius=0):
    self._celsius = celsius



@property
def celsius(self):
    """Temperature in Celsius."""
    return self._celsius



@celsius.setter
def celsius(self, value):
    if value < -273.15:
        raise ValueError("Temperature cannot be below absolute zero")
    self._celsius = value



@property
def fahrenheit(self):
    """Temperature in Fahrenheit (calculated)."""
    return (self._celsius * 9/5) + 32



@fahrenheit.setter
def fahrenheit(self, value):
    """Set temperature using Fahrenheit."""
    celsius_value = (value - 32) * 5/9
    if celsius_value < -273.15:
        raise ValueError("Temperature cannot be below absolute zero")
    self._celsius = celsius_value



def __repr__(self):
    return f"Temperature({self._celsius}°C / {self.fahrenheit}°F)"
```

Usage

temp = Temperature(25) print(temp) # Temperature(25°C / 77.0°F)

temp.fahrenheit = 100 print(temp) # Temperature(37.77...°C / 100°F)

print(temp.celsius) # 37.77...print(temp.fahrenheit) # 100

```
Both `celsius` and `fahrenheit` feel like simple attributes, but they're actually properties with logic behind
them. The user doesn't need to know or care about the implementation.

== Understanding super()

We touched on inheritance and mentioned `super()`, but let's really dig into what it does and why it matters.

When a child class inherits from a parent, sometimes you want to _extend_ the parent's behavior rather than
completely replace it. That's where `super()` comes in.

=== The Problem: Repeating Yourself

Imagine you're building on the `Animal` example:

[source,python]
```

class Animal: def *init*(self, name, age): self.name = name self.age = age self.is_alive = True

```
def speak(self):
    raise NotImplementedError("Subclass must implement")
```

class Dog(Animal): def *init*(self, name, age, breed): # Without super(), you'd have to repeat the parent's work: self.name = name # Duplicated! self.age = age # Duplicated! self.is_alive = True # Duplicated! self.breed = breed # Only this is new

```
def speak(self):
    return f"{self.name} says Woof!"


This works, but it violates DRY (Don't Repeat Yourself). If `Animal.__init__` changes, you'd have to update every
child class. That's a maintenance nightmare.

=== The Solution: Using super()

[source,python]
```

class Animal: def *init*(self, name, age): self.name = name self.age = age self.is_alive = True

```
def speak(self):
    raise NotImplementedError("Subclass must implement")


def describe(self):
    return f"{self.name} is {self.age} years old"
```

class Dog(Animal): def *init*(self, name, age, breed): super().*init*(name, age) # Call the parent's *init* self.breed = breed # Add dog-specific attribute

```
def speak(self):
    return f"{self.name} says Woof!"


def describe(self):
    # Extend the parent's describe method
    base_description = super().describe()
    return f"{base_description} and is a {self.breed}"
```

Usage

rex = Dog("Rex", 5, "German Shepherd")

print(rex.name) # Rex (set by parent) print(rex.age) # 5 (set by parent) print(rex.is_alive) # True (set by parent) print(rex.breed) # German Shepherd (set by child)

print(rex.speak()) # Rex says Woof! print(rex.describe()) # Rex is 5 years old and is a German Shepherd

```
=== What super() Actually Returns

When you call `super()`, Python returns a special proxy object that delegates method calls to the parent class.
It's essentially saying: "Give me a way to call methods from my parent."

[source,python]
```

super().*init*(name, age)

```
This is equivalent to the older, more explicit syntax:

[source,python]
```

Animal.*init*(self, name, age)

```
[TIP]
====
The `super()` syntax is preferred because:

* It's cleaner and more readable
* It works correctly with multiple inheritance (more advanced topic)
* If you rename the parent class, you don't have to update the child
====

=== Extending Methods Beyond __init__

You can use `super()` in any method, not just `__init__`:

[source,python]
```

class Animal: def eat(self, food): print(f"{self.name} is eating {food}") self.hunger = 0

class Dog(Animal): def eat(self, food): if food == "chocolate": print(f"No! Chocolate is toxic for dogs!") return super().eat(food) # Call parent's eat method print(f"{self.name} wags tail happily")

rex = Dog("Rex", 5, "German Shepherd") rex.hunger = 100

rex.eat("chocolate") # Output: No! Chocolate is toxic for dogs!

rex.eat("kibble") # Output: Rex is eating kibble # Rex wags tail happily

```
=== Practical Example: Building a User System

[source,python]
```

class User: """Base user class."""

```
def __init__(self, username, email):
    self.username = username
    self.email = email
    self.is_active = True
    self.created_at = "2025-01-01"  # Simplified


def get_permissions(self):
    return ["read"]


def __repr__(self):
    return f"User({self.username})"
```

class AdminUser(User): """Admin with elevated permissions."""

```
def __init__(self, username, email, admin_level=1):
    super().__init__(username, email)  # Set up base user stuff
    self.admin_level = admin_level


def get_permissions(self):
    # Start with base permissions, then add more
    base_permissions = super().get_permissions()
    admin_permissions = ["write", "delete"]
    if self.admin_level >= 2:
        admin_permissions.append("manage_users")
    return base_permissions + admin_permissions


def __repr__(self):
    return f"AdminUser({self.username}, level={self.admin_level})"
```

class SuperAdmin(AdminUser): """Super admin with all permissions."""

```
def __init__(self, username, email):
    super().__init__(username, email, admin_level=3)
```

```
    def get_permissions(self):
        return super().get_permissions() + ["system_config", "view_logs"]
```

Usage

regular = User("alice", "alice@example.com") admin = AdminUser("bob", "bob@example.com", admin_level=2) super_admin = SuperAdmin("charlie", "charlie@example.com")

print(regular.get_permissions()) # ['read']

print(admin.get_permissions()) # ['read', 'write', 'delete', 'manage_users']

print(super_admin.get_permissions()) # ['read', 'write', 'delete', 'manage_users', 'system_config', 'view_logs']

```
Each level builds on the previous one, and `super()` makes it seamless.

== Duck Typing

The presentation mentioned duck typing briefly, but it deserves a deeper look because it's fundamental to how
Python thinks about objects.

=== The Philosophy

The name comes from the saying:

[quote]
____
If it walks like a duck and quacks like a duck, then it probably is a duck.
____

In Python terms: *we don't care what type an object _is_, only what it can _do_.*

Unlike languages like Java or C++, Python doesn't require objects to inherit from a common parent or implement a
formal interface. If an object has the method you need, you can use it.

=== A Simple Example

[source,python]
```

class Duck: def speak(self): return "Quack!"

```
    def swim(self):
        return "Duck is swimming"
```

class Person: def speak(self): return "Hello!"

```
    def swim(self):
        return "Person is swimming"
```

class Robot: def speak(self): return "Beep boop!"

```
    def swim(self):
        return "Robot is short-circuiting!"
```

def make_it_speak(thing): """This function doesn’t care about the type of 'thing'.""" print(thing.speak())

def pool_party(participants): """Everyone goes swimming!""" for participant in participants: print(participant.swim())

None of these classes inherit from each other

But they all work because they have the same methods

duck = Duck() person = Person() robot = Robot()

make_it_speak(duck) # Quack! make_it_speak(person) # Hello! make_it_speak(robot) # Beep boop!

pool_party([duck, person, robot]) # Duck is swimming # Person is swimming # Robot is short-circuiting!

```
The `make_it_speak` function works with _any_ object that has a `speak` method. It doesn't check types, it just
tries to use the method.

=== Duck Typing vs Traditional Polymorphism

With traditional inheritance-based polymorphism, you'd write:

[source,python]
```

class Speakable: def speak(self): raise NotImplementedError

class Duck(Speakable): def speak(self): return "Quack!"

class Person(Speakable): def speak(self): return "Hello!"

```
This works, but Python's duck typing lets you skip the formal hierarchy entirely. Your classes don't need to know
about each other or share a parent.

=== Real-World Duck Typing: File-Like Objects

Python's standard library uses duck typing extensively. Consider reading data:

[source,python]
```

def process_data(data_source): """Process data from anything that has a read() method.""" content = data_source.read() return content.upper()

Works with actual files

with open("myfile.txt") as f: result = process_data(f)

Works with StringIO (fake file in memory)

from io import StringIO fake_file = StringIO("hello world") result = process_data(fake_file) # "HELLO WORLD"

Works with BytesIO

from io import BytesIO byte_file = BytesIO(b"hello world") # (would need .decode() but you get the idea)

```
The `process_data` function doesn't check if it received a "real" file. It just needs something with a `read()`
method.

=== Duck Typing with Iteration

You've been using duck typing since Chapter 2 without realizing it:

[source,python]
```

def print_all(items): """Print each item. Works with anything iterable.""" for item in items: print(item)

All of these work because they're all "iterable"

print_all([1, 2, 3]) # List print_all1, 2, 3 # Tuple print_all({1, 2, 3}) # Set print_all("abc") # String print_all(range(3)) # Range print_all({"a": 1, "b": 2}) # Dict (iterates keys)

```
The `for` loop doesn't check types. It just needs an object that supports iteration (has `__iter__` method).

=== Making Your Classes Duck-Type Friendly

Want your custom class to work with `for` loops? Just implement the right methods:

[source,python]
```

class Countdown: """A countdown that can be iterated."""

```
def __init__(self, start):
    self.start = start


def __iter__(self):
    """Make this class iterable."""
    self.current = self.start
    return self


def __next__(self):
    """Return the next value."""
    if self.current < 0:
        raise StopIteration
    value = self.current
    self.current -= 1
    return value
```

Now it works with for loops!

for num in Countdown(5): print(num) # Output: 5, 4, 3, 2, 1, 0

And with list()

numbers = list(Countdown(3)) # [3, 2, 1, 0]

```
=== Handling Duck Typing Failures

What happens when duck typing fails? The object doesn't have the method you expected:

[source,python]
```

class Rock: pass # Rocks don’t speak

def make_it_speak(thing): print(thing.speak())

rock = Rock() make_it_speak(rock) # AttributeError: 'Rock' object has no attribute 'speak'

```
You have options for handling this:

==== Option 1: EAFP (Easier to Ask Forgiveness than Permission)

This is the Pythonic way:

[source,python]
```

def make_it_speak(thing): try: print(thing.speak()) except AttributeError: print(f"{type(thing).*name*} cannot speak")

make_it_speak(Rock()) # "Rock cannot speak"

```
==== Option 2: LBYL (Look Before You Leap)

Check first using `hasattr()`:

[source,python]
```

def make_it_speak(thing): if hasattr(thing, 'speak'): print(thing.speak()) else: print(f"{type(thing).*name*} cannot speak")

```
==== Option 3: Use callable() for Methods

[source,python]
```

def make_it_speak(thing): speak_method = getattr(thing, 'speak', None) if callable(speak_method): print(speak_method()) else: print(f"{type(thing).*name*} cannot speak")

```
[NOTE]
====
EAFP (try/except) is generally preferred in Python because:

* It's often faster when the method usually exists
* It's more Pythonic
* It handles edge cases better
====

=== Duck Typing Summary

[cols="1,3"]
|===
|Concept |Description

|**Core Idea**
|Care about behavior, not type

|**Benefit**
|Flexible, loosely-coupled code

|**Risk**
|Runtime errors if object lacks expected method

|**Best Practice**
|Use try/except (EAFP) for graceful handling

|**Common Uses**
|Iteration, file operations, context managers
|===

== Putting It All Together

Let's build a small example that combines all three concepts:

[source,python]
```

class DataSource: """Base class for data sources using properties and meant for duck typing."""

```
def __init__(self, name):
    self._name = name
    self._data = []


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


@property
def record_count(self):
    """Calculated property."""
    return len(self._data)


def read(self):
    """Duck typing target: anything with read() can be a data source."""
    raise NotImplementedError
```

class CSVSource(DataSource): """Reads data from a CSV-like format."""

```
    def __init__(self, name, raw_text):
        super().__init__(name)  # Call parent's __init__
        self._raw_text = raw_text


    def read(self):
        """Parse CSV and return records."""
        lines = self._raw_text.strip().split('\n')
        headers = lines[0].split(',')


    self._data = []
    for line in lines[1:]:
        values = line.split(',')
        record = dict(zip(headers, values))
        self._data.append(record)


    return self._data
```

class JSONSource(DataSource): """Reads data from JSON format."""

```
    def __init__(self, name, json_data):
        super().__init__(name)
        self._json_data = json_data


    def read(self):
        """Parse JSON and return records."""
        import json
        self._data = json.loads(self._json_data)
        return self._data
```

def analyze_data(source): """ Works with ANY object that has read() and record_count. This is duck typing in action. """ try: data = source.read() print(f"Source: {source.name}") print(f"Records: {source.record_count}") print(f"First record: {data[0] if data else 'No data'}") print() except AttributeError as e: print(f"Invalid data source: {e}")

Usage

csv_data = """name,age,city Alice,30,New York Bob,25,Boston Charlie,35,Chicago"""

json_data = '[{"name": "Diana", "score": 95}, {"name": "Eve", "score": 87}]'

csv_source = CSVSource("Employee CSV", csv_data) json_source = JSONSource("Scores JSON", json_data)

Both work with the same function thanks to duck typing

analyze_data(csv_source) # Source: Employee CSV # Records: 3 # First record: {'name': 'Alice', 'age': '30', 'city': 'New York'}

analyze_data(json_source) # Source: Scores JSON # Records: 2= Python OOP: The Missing Pieces :toc: :toc-placement: left :toclevels: 3 :sectnums: :source-highlighter: pygments

1. Introduction

So you've learned about classes, objects, the three pillars of OOP, and even touched on composition vs inheritance. That's a solid foundation. But Python has a few more tricks up its sleeve that make working with objects feel natural and "Pythonic."

Let's dive into three topics that often get glossed over but are essential for writing clean, professional Python code.

2. The @property Decorator

Remember encapsulation? We learned that we can make attributes "private" using double underscores to protect them from outside interference. But here's the thing: sometimes you *do* need controlled access to those private attributes.

In languages like Java, you'd write explicit `getX()` and `setX()` methods. It works, but it's verbose and clunky:

PYTHON

```
# The "Java-style" approach (works, but not Pythonic)
class Circle:
    def __init__(self, radius):
        self.__radius = radius

    def get_radius(self):
        return self.__radius

    def set_radius(self, value):
        if value > 0:
            self.__radius = value
        else:
            raise ValueError("Radius must be positive")

# Usage feels awkward
c = Circle(5)
print(c.get_radius())  # 5
c.set_radius(10)
```

Python offers a more elegant solution: the `@property` decorator. It lets you define methods that *look* like simple attribute access but actually run your custom code behind the scenes.

2.1. Creating a Read-Only Property

PYTHON

```
class Circle:
    def __init__(self, radius):
        self._radius = radius  # Single underscore: "protected" by convention

    @property
    def radius(self):
        """The radius property (read-only for now)."""
        return self._radius

    @property
    def area(self):
        """Calculated property — no stored value needed."""
        return 3.14159 * self._radius ** 2

# Usage feels natural
c = Circle(5)
print(c.radius)  # 5 — looks like an attribute, but it's a method!
print(c.area)    # 78.53975 — calculated on the fly
```

Notice how we access `radius` and `area` without parentheses. From the outside, they look like regular attributes. But behind the scenes, Python is calling our methods.

2.2. Adding a Setter: Controlled Write Access

What if we want to allow changing the radius, but with validation? We add a setter using `@property_name.setter`:

PYTHON

```
class Circle:
    def __init__(self, radius):
        self._radius = None  # Will be set by the setter
        self.radius = radius  # Use the setter for validation

    @property
    def radius(self):
        """Get the radius."""
        return self._radius

    @radius.setter
    def radius(self, value):
        """Set the radius with validation."""
        if value <= 0:
            raise ValueError("Radius must be positive")
        self._radius = value

    @property
    def area(self):
        """Calculated property."""
        return 3.14159 * self._radius ** 2

# Now we have validated attribute access
c = Circle(5)
print(c.radius)    # 5

c.radius = 10      # Works fine
print(c.radius)    # 10

c.radius = -5      # Raises ValueError: Radius must be positive
```

2.3. The Deleter: Cleaning Up

For completeness, you can also define what happens when someone tries to delete the attribute:

PYTHON

```
@radius.deleter
def radius(self):
    """Handle deletion of radius."""
    print("Deleting radius...")
    self._radius = None

# Usage
del c.radius  # Prints: Deleting radius...
```

> NOTE
>
> The deleter is rarely needed, but it's available when you need it.

2.4. Why Use Properties?

Properties offer several advantages:

- **Clean interface**: Users of your class interact with simple attributes, not method calls
- **Validation**: You can enforce rules when values are set
- **Calculated attributes**: Derive values on-the-fly without storing them
- **Backward compatibility**: You can start with a simple attribute and later add a property without changing the interface

2.5. Complete Example: Temperature Converter

PYTHON

```
class Temperature:
    """A temperature that can be accessed in Celsius or Fahrenheit."""

    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        """Temperature in Celsius."""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature cannot be below absolute zero")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Temperature in Fahrenheit (calculated)."""
        return (self._celsius * 9/5) + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        """Set temperature using Fahrenheit."""
        celsius_value = (value - 32) * 5/9
        if celsius_value < -273.15:
            raise ValueError("Temperature cannot be below absolute zero")
        self._celsius = celsius_value

    def __repr__(self):
        return f"Temperature({self._celsius}°C / {self.fahrenheit}°F)"


# Usage
temp = Temperature(25)
print(temp)              # Temperature(25°C / 77.0°F)

temp.fahrenheit = 100
print(temp)              # Temperature(37.77...°C / 100°F)

print(temp.celsius)      # 37.77...
print(temp.fahrenheit)   # 100
```

Both `celsius` and `fahrenheit` feel like simple attributes, but they're actually properties with logic behind them. The user doesn't need to know or care about the implementation.

3. Understanding super()

We touched on inheritance and mentioned `super()`, but let's really dig into what it does and why it matters.

When a child class inherits from a parent, sometimes you want to *extend* the parent's behavior rather than completely replace it. That's where `super()` comes in.

3.1. The Problem: Repeating Yourself

Imagine you're building on the `Animal` example:

PYTHON

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.is_alive = True

    def speak(self):
        raise NotImplementedError("Subclass must implement")

class Dog(Animal):
    def __init__(self, name, age, breed):
        # Without super(), you'd have to repeat the parent's work:
        self.name = name      # Duplicated!
        self.age = age        # Duplicated!
        self.is_alive = True  # Duplicated!
        self.breed = breed    # Only this is new

    def speak(self):
        return f"{self.name} says Woof!"
```

This works, but it violates DRY (Don't Repeat Yourself). If `Animal.__init__` changes, you'd have to update every child class. That's a maintenance nightmare.

3.2. The Solution: Using super()

PYTHON

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.is_alive = True

    def speak(self):
        raise NotImplementedError("Subclass must implement")

    def describe(self):
        return f"{self.name} is {self.age} years old"

class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, age)  # Call the parent's __init__
        self.breed = breed           # Add dog-specific attribute

    def speak(self):
        return f"{self.name} says Woof!"

    def describe(self):
        # Extend the parent's describe method
        base_description = super().describe()
        return f"{base_description} and is a {self.breed}"


# Usage
rex = Dog("Rex", 5, "German Shepherd")

print(rex.name)        # Rex (set by parent)
print(rex.age)         # 5 (set by parent)
print(rex.is_alive)    # True (set by parent)
print(rex.breed)       # German Shepherd (set by child)

print(rex.speak())     # Rex says Woof!
print(rex.describe())  # Rex is 5 years old and is a German Shepherd
```

3.3. What super() Actually Returns

When you call `super()`, Python returns a special proxy object that delegates method calls to the parent class. It's essentially saying: "Give me a way to call methods from my parent."

PYTHON

```
super().__init__(name, age)
```

This is equivalent to the older, more explicit syntax:

PYTHON

```
Animal.__init__(self, name, age)
```

TIP

The `super()` syntax is preferred because:

- It's cleaner and more readable
- It works correctly with multiple inheritance (more advanced topic)
- If you rename the parent class, you don't have to update the child

3.4. Extending Methods Beyond *init*

You can use `super()` in any method, not just `*init*`:

PYTHON

```
class Animal:
    def eat(self, food):
        print(f"{self.name} is eating {food}")
        self.hunger = 0

class Dog(Animal):
    def eat(self, food):
        if food == "chocolate":
            print(f"No! Chocolate is toxic for dogs!")
            return
        super().eat(food)  # Call parent's eat method
        print(f"{self.name} wags tail happily")


rex = Dog("Rex", 5, "German Shepherd")
rex.hunger = 100

rex.eat("chocolate")
# Output: No! Chocolate is toxic for dogs!

rex.eat("kibble")
# Output: Rex is eating kibble
#         Rex wags tail happily
```

3.5. Practical Example: Building a User System

PYTHON

```
class User:
    """Base user class."""

    def __init__(self, username, email):
        self.username = username
        self.email = email
        self.is_active = True
        self.created_at = "2025-01-01"  # Simplified

    def get_permissions(self):
        return ["read"]

    def __repr__(self):
        return f"User({self.username})"


class AdminUser(User):
    """Admin with elevated permissions."""

    def __init__(self, username, email, admin_level=1):
        super().__init__(username, email)  # Set up base user stuff
        self.admin_level = admin_level

    def get_permissions(self):
        # Start with base permissions, then add more
        base_permissions = super().get_permissions()
        admin_permissions = ["write", "delete"]
        if self.admin_level >= 2:
            admin_permissions.append("manage_users")
        return base_permissions + admin_permissions

    def __repr__(self):
        return f"AdminUser({self.username}, level={self.admin_level})"


class SuperAdmin(AdminUser):
    """Super admin with all permissions."""

    def __init__(self, username, email):
        super().__init__(username, email, admin_level=3)

    def get_permissions(self):
        return super().get_permissions() + ["system_config", "view_logs"]


# Usage
regular = User("alice", "alice@example.com")
admin = AdminUser("bob", "bob@example.com", admin_level=2)
super_admin = SuperAdmin("charlie", "charlie@example.com")

print(regular.get_permissions())
# ['read']

print(admin.get_permissions())
# ['read', 'write', 'delete', 'manage_users']

print(super_admin.get_permissions())
# ['read', 'write', 'delete', 'manage_users', 'system_config', 'view_logs']
```

Each level builds on the previous one, and `super()` makes it seamless.

4. Duck Typing

The presentation mentioned duck typing briefly, but it deserves a deeper look because it's fundamental to how Python thinks about objects.

4.1. The Philosophy

The name comes from the saying:

> *If it walks like a duck and quacks like a duck, then it probably is a duck.*

In Python terms: **we don't care what type an object *is*, only what it can *do*.**

Unlike languages like Java or C++, Python doesn't require objects to inherit from a common parent or implement a formal interface. If an object has the method you need, you can use it.

4.2. A Simple Example

PYTHON

```
class Duck:
    def speak(self):
        return "Quack!"

    def swim(self):
        return "Duck is swimming"

class Person:
    def speak(self):
        return "Hello!"

    def swim(self):
        return "Person is swimming"

class Robot:
    def speak(self):
        return "Beep boop!"

    def swim(self):
        return "Robot is short-circuiting!"

def make_it_speak(thing):
    """This function doesn't care about the type of 'thing'."""
    print(thing.speak())

def pool_party(participants):
    """Everyone goes swimming!"""
    for participant in participants:
        print(participant.swim())


# None of these classes inherit from each other
# But they all work because they have the same methods
duck = Duck()
person = Person()
robot = Robot()

make_it_speak(duck)    # Quack!
make_it_speak(person)  # Hello!
make_it_speak(robot)   # Beep boop!

pool_party([duck, person, robot])
# Duck is swimming
# Person is swimming
# Robot is short-circuiting!
```

The `make_it_speak` function works with *any* object that has a `speak` method. It doesn't check types, it just tries to use the method.

4.3. Duck Typing vs Traditional Polymorphism

With traditional inheritance-based polymorphism, you'd write:

PYTHON

```
class Speakable:
    def speak(self):
        raise NotImplementedError

class Duck(Speakable):
    def speak(self):
        return "Quack!"

class Person(Speakable):
    def speak(self):
        return "Hello!"
```

This works, but Python's duck typing lets you skip the formal hierarchy entirely. Your classes don't need to know about each other or share a parent.

4.4. Real-World Duck Typing: File-Like Objects

Python's standard library uses duck typing extensively. Consider reading data:

PYTHON

```
def process_data(data_source):
    """Process data from anything that has a read() method."""
    content = data_source.read()
    return content.upper()

# Works with actual files
with open("myfile.txt") as f:
    result = process_data(f)

# Works with StringIO (fake file in memory)
from io import StringIO
fake_file = StringIO("hello world")
result = process_data(fake_file)  # "HELLO WORLD"

# Works with BytesIO
from io import BytesIO
byte_file = BytesIO(b"hello world")
# (would need .decode() but you get the idea)
```

The `process_data` function doesn't check if it received a "real" file. It just needs something with a `read()` method.

4.5. Duck Typing with Iteration

You've been using duck typing since Chapter 2 without realizing it:

PYTHON

```
def print_all(items):
    """Print each item. Works with anything iterable."""
    for item in items:
        print(item)

# All of these work because they're all "iterable"
print_all([1, 2, 3])           # List
print_all((1, 2, 3))           # Tuple
print_all({1, 2, 3})           # Set
print_all("abc")               # String
print_all(range(3))            # Range
print_all({"a": 1, "b": 2})    # Dict (iterates keys)
```

The `for` loop doesn't check types. It just needs an object that supports iteration (has `__iter__` method).

4.6. Making Your Classes Duck-Type Friendly

Want your custom class to work with `for` loops? Just implement the right methods:

PYTHON

```
class Countdown:
    """A countdown that can be iterated."""

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

    def __iter__(self):
        """Make this class iterable."""
        self.current = self.start
        return self

    def __next__(self):
        """Return the next value."""
        if self.current < 0:
            raise StopIteration
        value = self.current
        self.current -= 1
        return value


# Now it works with for loops!
for num in Countdown(5):
    print(num)
# Output: 5, 4, 3, 2, 1, 0

# And with list()
numbers = list(Countdown(3))  # [3, 2, 1, 0]
```

4.7. Handling Duck Typing Failures

What happens when duck typing fails? The object doesn't have the method you expected:

PYTHON

```
class Rock:
    pass  # Rocks don't speak

def make_it_speak(thing):
    print(thing.speak())

rock = Rock()
make_it_speak(rock)  # AttributeError: 'Rock' object has no attribute 'speak'
```

You have options for handling this:

4.7.1. Option 1: EAFP (Easier to Ask Forgiveness than Permission)

This is the Pythonic way:

```python
def make_it_speak(thing):
    try:
        print(thing.speak())
    except AttributeError:
        print(f"{type(thing).__name__} cannot speak")

make_it_speak(Rock())  # "Rock cannot speak"
```

4.7.2. Option 2: LBYL (Look Before You Leap)

Check first using `hasattr()`:

```python
def make_it_speak(thing):
    if hasattr(thing, 'speak'):
        print(thing.speak())
    else:
        print(f"{type(thing).__name__} cannot speak")
```

4.7.3. Option 3: Use callable() for Methods

```python
def make_it_speak(thing):
    speak_method = getattr(thing, 'speak', None)
    if callable(speak_method):
        print(speak_method())
    else:
        print(f"{type(thing).__name__} cannot speak")
```

NOTE

EAFP (try/except) is generally preferred in Python because:

- It's often faster when the method usually exists
- It's more Pythonic
- It handles edge cases better

4.8. Duck Typing Summary

Concept	Description
Core Idea	Care about behavior, not type
Benefit	Flexible, loosely-coupled code
Risk	Runtime errors if object lacks expected method
Best Practice	Use try/except (EAFP) for graceful handling
Common Uses	Iteration, file operations, context managers

5. Putting It All Together

Let's build a small example that combines all three concepts:

PYTHON

```python
class DataSource:
    """Base class for data sources using properties and meant for duck typing."""

    def __init__(self, name):
        self._name = name
        self._data = []

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

    @property
    def record_count(self):
        """Calculated property."""
        return len(self._data)

    def read(self):
        """Duck typing target: anything with read() can be a data source."""
        raise NotImplementedError


class CSVSource(DataSource):
    """Reads data from a CSV-like format."""

    def __init__(self, name, raw_text):
        super().__init__(name)  # Call parent's __init__
        self._raw_text = raw_text

    def read(self):
        """Parse CSV and return records."""
        lines = self._raw_text.strip().split('\n')
        headers = lines[0].split(',')

        self._data = []
        for line in lines[1:]:
            values = line.split(',')
            record = dict(zip(headers, values))
            self._data.append(record)

        return self._data


class JSONSource(DataSource):
    """Reads data from JSON format."""

    def __init__(self, name, json_data):
        super().__init__(name)
        self._json_data = json_data

    def read(self):
        """Parse JSON and return records."""
        import json
        self._data = json.loads(self._json_data)
        return self._data


def analyze_data(source):
    """
    Works with ANY object that has read() and record_count.
    This is duck typing in action.
    """
    try:
        data = source.read()
```

```
        print(f"Source: {source.name}")
        print(f"Records: {source.record_count}")
        print(f"First record: {data[0] if data else 'No data'}")
        print()
    except AttributeError as e:
        print(f"Invalid data source: {e}")


# Usage
csv_data = """name,age,city
Alice,30,New York
Bob,25,Boston
Charlie,35,Chicago"""

json_data = '[{"name": "Diana", "score": 95}, {"name": "Eve", "score": 87}]'

csv_source = CSVSource("Employee CSV", csv_data)
json_source = JSONSource("Scores JSON", json_data)

# Both work with the same function thanks to duck typing
analyze_data(csv_source)
# Source: Employee CSV
# Records: 3
# First record: {'name': 'Alice', 'age': '30', 'city': 'New York'}

analyze_data(json_source)
# Source: Scores JSON
# Records: 2
# First record: {'name': 'Diana', 'score': 95}
```

6. Summary

In this section, you learned:

- `@property` lets you create attributes with built-in logic — validation, calculation, or transformation — while keeping a clean interface
- `super()` is your tool for building on parent class functionality without repeating yourself. Use it in *`init`* and any other method you want to extend
- **Duck typing** is Python's philosophy of caring about what an object can *do*, not what it *is*. Write functions that expect behaviors (methods), not specific types

These three concepts will make your Python code more Pythonic, maintainable, and flexible. They're the difference between code that merely works and code that feels natural to write and read.

7. Practice Exercises

1. Create a `BankAccount` class with a `balance` property that prevents negative balances. Add a `@property` for `is_overdrawn` that returns `True` if balance is zero.
2. Build a `Vehicle` base class with `make`, `model`, and `year`. Create `Car` and `Motorcycle` child classes that use `super()` to initialize the parent and add their own attributes (`num_doors` for Car, `has_sidecar` for Motorcycle).
3. Write a function `get_length(thing)` that uses duck typing to return the length of any object. It should work with strings, lists, dictionaries, and any custom class that implements *`len`*. Handle objects that don't have a length gracefully.

4. Create a `Rectangle` class with `width` and `height` properties that validate positive values. Add calculated properties for `area` and `perimeter`. Include a `@property` setter that allows setting `area` by adjusting the width while keeping the aspect ratio.
5. Build a simple plugin system using duck typing. Create a `PluginManager` that accepts any object with `activate()` and `deactivate()` methods. Test it with different "plugin" classes that don't share a common parent. # First record: {'name': 'Diana', 'score': 95}

== Summary

In this section, you learned:

* **`@property`** lets you create attributes with built-in logic — validation, calculation, or transformation — while keeping a clean interface
* **`super()`** is your tool for building on parent class functionality without repeating yourself. Use it in `__init__` and any other method you want to extend
* **Duck typing** is Python's philosophy of caring about what an object can _do_, not what it _is_. Write functions that expect behaviors (methods), not specific types

These three concepts will make your Python code more Pythonic, maintainable, and flexible. They're the difference between code that merely works and code that feels natural to write and read.

== Practice Exercises

1. Create a `BankAccount` class with a `balance` property that prevents negative balances. Add a `@property` for `is_overdrawn` that returns `True` if balance is zero.

2. Build a `Vehicle` base class with `make`, `model`, and `year`. Create `Car` and `Motorcycle` child classes that use `super()` to initialize the parent and add their own attributes (`num_doors` for Car, `has_sidecar` for Motorcycle).

3. Write a function `get_length(thing)` that uses duck typing to return the length of any object. It should work with strings, lists, dictionaries, and any custom class that implements `__len__`. Handle objects that don't have a length gracefully.

4. Create a `Rectangle` class with `width` and `height` properties that validate positive values. Add calculated properties for `area` and `perimeter`. Include a `@property` setter that allows setting `area` by adjusting the width while keeping the aspect ratio.

5. Build a simple plugin system using duck typing. Create a `PluginManager` that accepts any object with `activate()` and `deactivate()` methods. Test it with different "plugin" classes that don't share a common parent.
    def read(self):
        """Parse CSV and return records."""
        lines = self._raw_text.strip().split('\n')
        headers = lines[0].split(',')

        self._data = []
        for line in lines[1:]:
            values = line.split(',')
            record = dict(zip(headers, values))
            self._data.append(record)

        return self._data

class JSONSource(DataSource):
    """Reads data from JSON format."""

    def __init__(self, name, json_data):
        super().__init__(name)
        self._json_data = json_data

    def read(self):
        """Parse JSON and return records."""
        import json
        self._data = json.loads(self._json_data)
        return self._data

def analyze_data(source):
    """

```
    Works with ANY object that has read() and record_count.
    This is duck typing in action.
    """
    try:
        data = source.read()
        print(f"Source: {source.name}")
        print(f"Records: {source.record_count}")
        print(f"First record: {data[0] if data else 'No data'}")
        print()
    except AttributeError as e:
        print(f"Invalid data source: {e}")


# Usage
csv_data = """name,age,city
Alice,30,New York
Bob,25,Boston
Charlie,35,Chicago"""

json_data = '[{"name": "Diana", "score": 95}, {"name": "Eve", "score": 87}]'

csv_source = CSVSource("Employee CSV", csv_data)
json_source = JSONSource("Scores JSON", json_data)

# Both work with the same function thanks to duck typing
analyze_data(csv_source)
# Source: Employee CSV
# Records: 3
# First record: {'name': 'Alice', 'age': '30', 'city': 'New York'}

analyze_data(json_source)
# Source: Scores JSON
# Records: 2
# First record: {'name': 'Diana', 'score': 95}
```

8. Summary

In this section, you learned:

- `@property` lets you create attributes with built-in logic — validation, calculation, or transformation — while keeping a clean interface
- `super()` is your tool for building on parent class functionality without repeating yourself. Use it in `__init__` and any other method you want to extend
- **Duck typing** is Python's philosophy of caring about what an object can *do*, not what it *is*. Write functions that expect behaviors (methods), not specific types

These three concepts will make your Python code more Pythonic, maintainable, and flexible. They're the difference between code that merely works and code that feels natural to write and read.

9. Practice Exercises

1. Create a `BankAccount` class with a `balance` property that prevents negative balances. Add a `@property` for `is_overdrawn` that returns `True` if balance is zero.
2. Build a `Vehicle` base class with `make`, `model`, and `year`. Create `Car` and `Motorcycle` child classes that use `super()` to initialize the parent and add their own attributes (`num_doors` for Car, `has_sidecar` for Motorcycle).

3. Write a function `get_length(thing)` that uses duck typing to return the length of any object. It should work with strings, lists, dictionaries, and any custom class that implements *`len`*. Handle objects that don't have a length gracefully.
4. Create a `Rectangle` class with `width` and `height` properties that validate positive values. Add calculated properties for `area` and `perimeter`. Include a `@property` setter that allows setting `area` by adjusting the width while keeping the aspect ratio.
5. Build a simple plugin system using duck typing. Create a `PluginManager` that accepts any object with `activate()` and `deactivate()` methods. Test it with different "plugin" classes that don't share a common parent.

Last updated 2025-11-30 14:46:26 +0800