

- [How to Use This Guide with the .python Files](#)
- [Introduction to Python Programming: Foundations for Object-Oriented Programming](#)
 - [Purpose of This Guide](#)
 - [Key Python Concepts Covered](#)
 - [1. Data Types and Structures](#)
 - [2. Control Flow](#)
 - [3. Functions and Modularity](#)
 - [4. Error Handling \(try-except\)](#)
 - [5. String and Data Formatting](#)
 - [6. Operators](#)
- [Python Modules: Organizing and Reusing Code](#)
 - [What is a Module?](#)
 - [Key Characteristics of Modules:](#)
 - [How Modules Work \(Based on Your Files\)](#)
 - [1. Creating Modules](#)
 - [2. Using Modules](#)
 - [3. Module Execution](#)
 - [Practical Benefits in Your Code](#)
 - [1. Separation of Concerns](#)
 - [2. Avoiding Duplication](#)
 - [3. Collaboration](#)
 - [Best Practices](#)
 - [Common Pitfalls](#)
 - [Try It Yourself](#)
- [Python Packages: Organizing Modules into Logical Groups](#)
 - [What is a Package?](#)
 - [Key Characteristics:](#)
 - [How Packages Work \(Based on Your Files\)](#)
 - [1. Package Structure](#)
 - [3. Importing from Packages](#)
 - [Practical Benefits in Your Code](#)
 - [1. Logical Grouping](#)
 - [2. Avoiding Name Collisions](#)
 - [3. Team Collaboration](#)
 - [Best Practices](#)
 - [Common Pitfalls](#)

- Try It Yourself
- Python Packages: Organizing Related Functionality
 - Package Structure in Your Project
 - Key Package Components Explained
 - 1. Module Organization
 - 2. Package Initialization
 - 3. Usage in Main Program
 - Best Practices Demonstrated
 - Practical Exercise
- Object-Oriented Programming (OOP) in Python: A Comprehensive Guide
 - Introduction to OOP Concepts
 - 1. Encapsulation
 - 2. Abstraction
 - 3. Inheritance
 - 4. Polymorphism
 - Core Components in Your Implementation
 - Classes and Objects
- From classe.py
 - Attributes and Methods
 - Special Methods
 - Project Structure Best Practices
 - Key Code Examples Explained
 - 1. Class Definition (classe.py)
 - 2. Attributes Initialization (atributo.py)
 - 3. Method Implementation (metodo.py)
 - 4. Comprehensive Example (completo.py)
 - Practical OOP Exercises
 - Common Pitfalls and Solutions
- Object-Oriented Relationships in Python: Association, Composition, and Aggregation
 - Understanding Object Relationships
 - 1. Association
 - 2. Composition
 - 3. Aggregation
 - Practical Implementation in Your Code
 - Composition in Action (carro.py + motor.py)
 - Association in Action (estudante.py + curso.py)
 - Visualizing the Relationships

- [When to Use Each Relationship](#)
- [Common Mistakes and Fixes](#)
- [Advanced Techniques](#)
- [Exercises to Reinforce Learning](#)
- [Understanding Aggregation in Object-Oriented Programming](#)
 - [The Aggregation Relationship](#)
 - [Key Characteristics:](#)
 - [Implementation in Your Code](#)
 - [1. Book Class \(livro.py\)](#)
 - [2. Librarian Class \(bibliotecario.py\)](#)
 - [3. Demonstration \(demonstracao_agregacao.py\)](#)
 - [Why This is Aggregation \(Not Composition\)](#)
 - [Practical Implications](#)
 - [Common Mistakes to Avoid](#)
 - [Real-World Analogies](#)
 - [Exercises to Reinforce Learning](#)

How to Use This Guide with the `.python` Files

1. **Read the Theory First** : Understand the concepts here before diving into code.
2. **Experiment** : Modify examples in the provided files to see how changes affect outcomes.
3. **Relate to OOP** : Ask: *How would this concept apply to a class or object?*

Introduction to Python Programming: Foundations for Object-Oriented Programming

Purpose of This Guide

This document serves as a **conceptual supplement** to the `.python` files provided in the course. While the original files demonstrate syntax and practical implementations, this guide explains the *why* and *how* behind Python's core concepts. By combining

both resources, you'll build a robust understanding of Python's fundamentals, preparing you for Object-Oriented Programming (OOP).

Key Python Concepts Covered

1. Data Types and Structures

Python is dynamically typed, meaning variables infer their type at runtime. However, explicit type hints (e.g., `var: int = 10`) improve code clarity. Key types include:

- **Primitives:** `int`, `float`, `bool`, `str`, `NoneType`.
- **Collections:**
 - **Lists:** Ordered, mutable sequences (`[1, 2, 3]`).
 - **Tuples:** Ordered, immutable sequences (`(1, "a", True)`).
 - **Sets:** Unordered, unique elements (`{1, 2, 3}`).
 - **Dictionaries:** Key-value pairs (`{"name": "Alice"}`).

Why this matters for OOP: Understanding types is essential for designing class attributes and method signatures.

2. Control Flow

Python uses conditional logic (`if-elif-else`) and loops (`for`, `while`) to control program execution.

- **Conditionals:** Evaluate boolean expressions to branch logic.
- **Loops:** Iterate over sequences or repeat actions until a condition is met.

Example from your files:

```
for nome in ["Alice", "Bob"]: # Iterates over a list
    print(f"Hello, {nome}!")
```

OOP Connection : Control flow structures are used within methods to implement object behaviors.

3. Functions and Modularity

Functions encapsulate reusable logic. Key features:

- **Parameters/Arguments** : Inputs to functions.
- **Return Values** : Outputs from functions.
- **Scope** : Variables exist only within their function unless declared `global`.

Why this matters : OOP extends this idea with *methods* (functions bound to objects) and *encapsulation* (controlling access to data).

4. Error Handling (`try-except`)

Python raises exceptions for runtime errors (e.g., `ValueError`, `ZeroDivisionError`). Use `try-except` to gracefully handle failures.

Example :

```
try:
    age = int(input("Enter age: "))
except ValueError:
    print("Invalid input! Must be a number.")
```

OOP Connection : Robust classes include error handling to protect object integrity.

5. String and Data Formatting

Python offers multiple ways to format strings:

- **f-strings** (modern): `f"Name: {name}"`.
- **`.format()`** : `"Name: {}".format(name)`.
- **`%` operator** : Legacy but useful for simple cases.

OOP Connection : Used in `__str__` methods to define how objects are printed.

6. Operators

- **Comparison** : `==`, `!=`, `>`, `<` (return `bool`).
- **Logical** : `and`, `or`, `not` (combine conditions).

Example :

```
if age >= 18 and has_license:  
    print("Can drive.")
```

OOP Connection : Operators can be overloaded in classes (e.g., `__eq__` for `==`).

Python Modules: Organizing and Reusing Code

What is a Module?

A **module** in Python is simply a file containing Python definitions and statements (functions, variables, classes, etc.). Modules allow you to logically organize your code and reuse it across multiple programs.

Key Characteristics of Modules:

- **Reusability**: Write once, use anywhere.
 - **Namespace Separation**: Avoid naming conflicts.
 - **Maintainability**: Update one file to fix/improve functionality everywhere it's used.
-

How Modules Work (Based on Your Files)

1. Creating Modules

Each `.py` file is a module. Your examples include:

- `calculadora.py`: Contains math operations (`somar` function).
- `saudacoes.py`: Handles greetings (`saudacao` function).

```
# calculadora.py
def somar(a, b):
    return a + b # Reusable in any file that imports this module
```

2. Using Modules

Import modules with `import` or `from ... import`. Example from `modulo.py`:

```
from saudacoes import saudacao # Imports only the `saudacao` function
from calculadora import somar   # Imports only the `somar` function
```

3. Module Execution

When imported:

- Python runs all top-level code in the module.
- Functions/variables become available in the importer's namespace.

Practical Benefits in Your Code

1. Separation of Concerns

- `saudacoes.py` : Dedicated to greeting logic.
- `calculadora.py` : Dedicated to math operations.
- `modulo.py` : Uses both without mixing their responsibilities.

2. Avoiding Duplication

Without modules, you'd copy-paste the `somar()` function everywhere—error-prone and hard to maintain.

3. Collaboration

Teams can work on different modules simultaneously (e.g., one developer on `calculadora.py`, another on `saudacoes.py`).

Best Practices

1. Naming Modules :

- Use lowercase with underscores (`saudacoes.py`, not `Saudacoes.py`).
- Avoid Python built-in names (e.g., `math.py`).

1. Import Statements :

- Group imports logically:

```
# Standard library imports first
import os

# Third-party next
import numpy

# Local modules last
from calculadora import somar
```

1. `__name__` Guard : Prevent code from running on import:


```
# In saudacoes.py
if __name__ == "__main__":
    print(saudacao("Teste")) # Runs only when file is executed directly
```

Common Pitfalls

1. **Circular Imports** : Module A imports Module B, which imports Module A → Causes errors. *Fix* : Restructure code or use local imports.
 2. **Shadowing** : Naming a module after a built-in (e.g., `json.py`) breaks imports. *Fix* : Rename the module.
-

Try It Yourself

1. Add a `subtrair(a, b)` function to `calculadora.py`.
2. Import and use it in `modulo.py`.
3. Run `modulo.py` to see both modules working together.

 **Next** : Packages extend this concept further by organizing modules into directories (coming soon).

Python Packages: Organizing Modules into Logical Groups

What is a Package?

A **package** is a collection of related Python modules organized in a directory hierarchy. Packages allow you to:

- Scale your project structure as it grows
- Avoid naming conflicts between modules
- Create a clear, maintainable architecture

Key Characteristics:

- A directory containing an `__init__.py` file (even empty)
 - Can contain sub-packages (nested directories with their own `__init__.py`)
 - Accessed using dot notation: `package.subpackage.module`
-

How Packages Work (Based on Your Files)

1. Package Structure

Your example demonstrates two packages:

strings_utils/ # Package |—— **init** .py # Optional (implicit namespace package) |——
contagem.py # Module with contar_palavras() |—— limpeza.py # Module with
remover_espacos_extra()

estruturas/ # Another package |—— **init** .py # Explicit imports |—— condicional.py |——
conjuntos.py |—— ... (other modules)

```
### 2. **The `__init__.py` File**  
This special file makes Python treat directories as packages. Your  
`estruturas/__init__.py` shows best practices:  
```python  
estruturas/__init__.py
from .controle import estrutura_controle # Relative imports
from .repeticao import repetir_for, repetir_while
...
```

- Acts as the package's "public interface"
- Controls what gets imported with `from package import *`

## 3. Importing from Packages

Your `Pacotes.py` demonstrates clean package usage:

```
from strings_utils.contagem import contar_palavras # Absolute import
from estruturas import classificar_idade # Import from __init__.py
```

---

## Practical Benefits in Your Code

---

# 1. Logical Grouping

- `strings_utils/`: Text manipulation functions
- `estructuras/`: Data structures and control flow
- Clear separation makes code self-documenting

## 2. Avoiding Name Collisions

You could have:

```
math_utils/ # Your custom math package
└─ stats.py # with mean(), median()
```

Without conflicting with Python's built-in `math` module.

## 3. Team Collaboration

Multiple developers can work on different packages simultaneously.

---

## Best Practices

---

### 1. Naming Packages :

- Use lowercase with underscores (`data_utils/`, not `DataUtils/`)
- Avoid Python built-in names (`json/`, `sys/`)

### 1. Import Statements :

- Prefer absolute imports (`from package.subpackage import module`)
- Use relative imports carefully within packages (`from . import sibling_module`)

### 1. `__init__.py` Usage :

- Can be empty (creates a "namespace package")

- Or define `__all__` to control `from package import *` behavior:

```
__init__.py
__all__ = ['contar_palavras', 'remover_espacos_extra']
```

## 1. Package Distribution :

- Add `setup.py` to make installable via pip
- Include requirements in `requirements.txt`

---

# Common Pitfalls

1. **Circular Imports** : Package A imports Package B which imports Package A → Runtime error *Fix* : Restructure or use local imports
2. **Shadowing Standard Library** : Naming a package `email/` conflicts with Python's `email` package *Fix* : Choose distinct names (`email_utils/`)
3. **Missing `__init__.py`** : Python won't recognize the directory as a package *Fix* : Create empty `__init__.py` (Python 3.3+ supports namespace packages without it)

---

# Try It Yourself

1. Create a new package `math_utils/` with:
  - `basic.py` (add `somar`, `subtrair`)
  - `advanced.py` (add `potencia`)
  - `__init__.py` that imports all functions
2. Use it in `Pacotes.py`:

```
from math_utils import potencia
print(potencia(2, 3)) # Should print 8
```

# Python Packages: Organizing Related Functionality

## Package Structure in Your Project

Your files demonstrate a well-organized package architecture:

project/ ├── strings\_utils/ # Text processing package | ├── init .py | ├──  
contagem.py # Word counting | ├── limpeza.py # Text cleaning | ├── estruturas/ #  
Data structures package | ├── init .py # Package interface | ├── condicional.py # Age  
classification | ├── conjuntos.py # Set operations | ├── controle.py # Number  
verification | ├── dicionarios.py # Student data handling | ├── listas.py # Fruit list  
operations | ├── repeticao.py # Loop examples | ├── tuplas.py # Color tuple handling

## Key Package Components Explained

### 1. Module Organization

Each module focuses on a specific data structure or operation:

- **tuplas.py**: Handles immutable sequences (tuples)

```
def mostrar_cores(tamanho_cores):
 cores = []
 for i in range(tamanho_cores):
 cor = input(f"Digite o nome da cor {i + 1}: ").strip()
 cores.append(cor)
 return tuple(cores) # Converts list to tuple
```

- **repeticao.py**: Demonstrates different loop types

```
def repetir_for(n):
 return [i for i in range(1, n + 1)] # List comprehension

def repetir_while(n):
 contador = 1
 resultado = []
```

```
while contador <= n: # Traditional while loop
 resultado.append(contador)
 contador += 1
return resultado
```

## 2. Package Initialization

The `__init__.py` in `estruturas` serves as the package's public interface:

```
from .controle import estrutura_controle
from .repeticao import repetir_for, repetir_while
from .listas import operacoes_lista
from .tuplas import mostrar_cores
from .conjuntos import animais_unicos
from .dicionarios import dados_alunos
from .condicional import classificar_idade
```

## 3. Usage in Main Program

`Pacotes.py` imports and uses these packages cleanly:

```
from strings_utils.contagem import contar_palavras
from estruturas import mostrar_cores, repetir_for

Example usage
cores = mostrar_cores(3) # Uses tuplas.py
numeros = repetir_for(5) # Uses repeticao.py
```

## Best Practices Demonstrated

### 1. Single Responsibility Principle

- Each module handles one specific data structure type
- Example: `tuplas.py` only deals with tuple operations

### 2. Clear Naming Conventions

- Package names: lowercase\_with\_underscores (`strings_utils`)
- Module names: descriptive (`repeticao.py`, `condicional.py`)

### 3. Controlled Exports

- `__init__.py` selectively exposes functions

- Avoids namespace pollution

#### 4. Consistent Interfaces

- Similar function signatures across modules (e.g., size parameters)
- Uniform error handling patterns


## Practical Exercise

---

1. Add a new module `matrizes.py` to `estruturas` with:

```
def criar_matriz(linhas, colunas):
 return [[0]*colunas for _ in range(linhas)]
```

2. Update `__init__.py` to export it
3. Use it in `Pacotes.py` to create a 2x2 matrix

 **Tip** : Run `python -m pytest` to test packages (install pytest first). This modular structure makes unit testing easier!

## Object-Oriented Programming (OOP) in Python: A Comprehensive Guide

---

### Introduction to OOP Concepts

---

Object-Oriented Programming is a paradigm that organizes code around "objects" rather than functions and logic. Your Python files demonstrate all four pillars of OOP:

#### 1. Encapsulation

Bundling data (attributes) and methods that operate on that data within a single unit (class).

Example from `atributo.py`:

```
class Pessoa:
 def __init__(self, nome, idade): # Encapsulates name and age
 self.nome = nome
 self.idade = idade
```

## 2. Abstraction

Hiding complex implementation details while exposing only necessary interfaces.

Example from `metodo.py`:

```
def apresentar(self): # Simple interface hides implementation
 print(f"Olá! Meu nome é {self.nome}.")
```

## 3. Inheritance

Creating new classes from existing ones (not shown in your files but implied by structure).

## 4. Polymorphism

Using a single interface for different data types (demonstrated in `completo.py` through various data structure methods).

---

# Core Components in Your Implementation

---

## Classes and Objects

- **Class** : Blueprint for creating objects (`classe.py`)
- **Object** : Instance of a class (`atributo.py` creates `Pessoa` instances)



# From classe.py

---

`class Pessoa: # Class definition pass`

`p = Pessoa() # Object instantiation`

## Attributes and Methods

- **Attributes** : Variables belonging to objects (`self.nome` in `atributo.py`)
- **Methods** : Functions defined within a class (`apresentar()` in `metodo.py`)

## Special Methods

- `__init__` : Constructor method called when object is created (shown in multiple files)

---

# Project Structure Best Practices

---

Your package organization follows Pythonic principles:

### 1. Modular Design

- `introducao/` package contains related OOP concepts
- Each concept in separate files: `classe.py`, `atributo.py`, etc.

### 2. Clean Imports

- `global_imports.py` centralizes imports
- `__init__.py` controls package visibility

### 3. Menu-Driven Interface

- Clear user navigation through `menu_principal()` and `menu_introducao()`

---

# Key Code Examples Explained

---

## 1. Class Definition (`classe.py`)

```
class Pessoa:
 pass # Minimal class definition
```

## 2. Attributes Initialization ([atributo.py](#))

```
def __init__(self, nome, idade):
 self.nome = nome # Instance attribute
 self.idade = idade
```

## 3. Method Implementation ([metodo.py](#))

```
def apresentar(self): # Instance method
 print(f"Olá! Meu nome é {self.nome}.")
```

## 4. Comprehensive Example ([completo.py](#))

Demonstrates multiple OOP concepts:

- Instance methods with different behaviors
- Interaction with various data structures
- Error handling

---

# Practical OOP Exercises

---

### 1. Extend the Pessoa Class :

```
class Estudante(Pessoa): # Inheritance
 def __init__(self, nome, idade, matricula):
 super().__init__(nome, idade)
 self.matricula = matricula
```

### 1. Add Class Methods :

```
@classmethod
def criar_anonimo(cls):
 return cls("Anônimo", 0)
```

## 1. Implement Magic Methods :

```
def __str__(self):
 return f"{self.nome} ({self.idade} anos)"
```

---

# Common Pitfalls and Solutions

---

## 1. Forgotten self parameter :

```
Wrong
def apresentar(): # Missing self
 print("Olá!")

Right
def apresentar(self):
 print(f"Olá! Meu nome é {self.nome}.")
```

## 1. Mutable Default Arguments :

```
Dangerous
def __init__(self, hobbies=[]):

Safe
def __init__(self, hobbies=None):
 self.hobbies = hobbies if hobbies else []
```

## 1. Overusing Class Variables :

```
class Pessoa:
 especie = "Humano" # Good class variable
 nomes = [] # Bad - shared across instances
```

---

# Object-Oriented Relationships in Python: Association, Composition, and Aggregation

---

## Understanding Object Relationships

---

Your code demonstrates three fundamental types of object relationships in OOP:

### 1. Association

A "uses-a" relationship where objects are independent but can interact.

Example from `demonstracao_associacao.py`:

```
Estudante and Curso exist independently
curso = Curso("Computer Science")
estudante = Estudante("Alice", 20)
curso.cadastrar_estudante(estudante) # Temporary relationship
```

Key Characteristics:

- Both objects have separate lifecycles
- Either can exist without the other
- Demonstrated in Student-Course relationship

### 2. Composition

A "has-a" relationship where the child cannot exist without the parent.

Example from `demonstracao_composicao.py`:

```
class Carro:
 def __init__(self):
 self.motor = Motor() # Motor is created WITH the car
```

Key Characteristics:

- Strong lifecycle dependency (destroy car → destroy motor)
- The part belongs to exactly one whole
- Demonstrated in Car-Engine relationship

## 3. Aggregation

A weaker "has-a" relationship where the child can exist independently.

**Example (conceptual - not in your files):**

```
class Department:
 def __init__(self):
 self.professors = [] # Professors can exist without department

class Professor:
 pass
```

Key Differences from Composition:

- Parts can be shared between wholes
- No ownership implied

---

## Practical Implementation in Your Code

### Composition in Action (**carro.py** + **motor.py**)

```
class Carro:
 def configurar_carro(self, marca, modelo, tipo_motor):
 self.motor_do_carro = Motor() # Composition - motor created inside car
 self.motor_do_carro.configurar_motor(tipo_motor)
```

When the car is deleted:

```
del meu_carro_novo # Automatically destroys its motor
```

# Association in Action (estudante.py + curso.py)

```
class Curso:
 def cadastrar_estudante(self, estudante):
 self.estudantes.append(estudante) # Association - student exists
independently
```

Students continue to exist even if the course is deleted.

## Visualizing the Relationships

Relationship	Lifetime Dependency	Example	UML Notation
Association	None	Student ↔ Course	Simple line
Composition	Strong	Car → Engine	Filled diamond
Aggregation	Weak	Department → Professor	Empty diamond

## When to Use Each Relationship

1. **Use Composition** when:
  - The part cannot reasonably exist without the whole
  - Example: Order → OrderItems (delete order → delete items)
2. **Use Aggregation** when:
  - The part can exist independently
  - Example: Playlist → Songs (songs exist without playlist)
3. **Use Association** when:
  - Objects need temporary collaboration
  - Example: Teacher ↔ Student (both exist independently)

## Common Mistakes and Fixes

## Mistake 1 : Using composition when aggregation is needed

```
Wrong (composition)
class University:
 def __init__(self):
 self.professors = [Professor()] # Professors die with university?

Right (aggregation)
class University:
 def __init__(self):
 self.professors = [] # Professors added later
```

## Mistake 2 : Creating circular dependencies

```
Problematic
class A:
 def __init__(self, b): self.b = b

class B:
 def __init__(self, a): self.a = a

Solution: Use weak references or redesign
```

---

# Advanced Techniques

### 1. Dependency Injection (Making composition flexible):

```
class Car:
 def __init__(self, engine): # Accept engine as parameter
 self.engine = engine
```

### 2. Interface Segregation (For cleaner associations):

```
from abc import ABC, abstractmethod

class GradeCalculator(ABC):
 @abstractmethod
 def calculate_grade(self): pass

class Estudante(GradeCalculator): # Now can associate with any grade calculator
 ...
```

---

# Exercises to Reinforce Learning

---

1. **Convert Association to Aggregation** : Modify `Curso` to accept pre-existing `Estudante` objects rather than creating them
2. **Implement a Weak Reference Example** : Create a `Classroom` class that weakly references `Student` objects
3. **Design a Composition Hierarchy** : Build a `Computer` class composed of `CPU`, `RAM`, and `Storage` components

## Understanding Aggregation in Object-Oriented Programming

---

### The Aggregation Relationship

---

Aggregation is a "has-a" relationship where one object (the whole) contains references to other objects (the parts), but the parts can exist independently of the whole. Your library system example perfectly demonstrates this concept.

### Key Characteristics:

- **Independent Lifecycles**: Books exist before and after being associated with a librarian
- **Weak Ownership**: Books aren't destroyed when a librarian leaves
- **Bidirectional Access**: Both objects maintain their identities

## Implementation in Your Code

---

### 1. Book Class (`livro.py`)



```
class Livro:
 def __init__(self, titulo, autor):
 self.titulo = titulo # Independent attributes
 self.autor = autor
 self.emprestado = False
```

## 2. Librarian Class (**bibliotecario.py**)

```
class Bibliotecario:
 def __init__(self, nome):
 self.nome = nome
 self.livros = [] # Aggregation - stores references to Book objects
```

## 3. Demonstration (**demonstracao\_agregacao.py**)

```
Books exist independently
livro1 = Livro("Dom Casmurro", "Machado de Assis")
livro2 = Livro("1984", "George Orwell")

Librarian aggregates books
bibliotecario = Bibliotecario("Maria")
bibliotecario.adicionar_livro(livro1) # Book added to collection
```

## Why This is Aggregation (Not Composition)

Scenario	Aggregation Behavior
Librarian leaves job	Books remain in library system
Book is removed from catalog	Can be reassigned to another librarian
Library system shuts down	Book records could be transferred

# Practical Implications

---

## 1. Database Modeling :

- Books would have their own table
- Librarians would have their own table
- A join table would manage the relationship

## 1. Memory Management :

```
del bibliotecario # Only removes librarian, books remain
```

## 3. Flexible Systems :

```
Same book can be referenced by multiple systems
class Biblioteca:
 def __init__(self):
 self.catalogo = [] # Could share book references with librarians
```

# Common Mistakes to Avoid

---

**Mistake** : Treating aggregation as composition

```
Wrong (composition approach)
class Bibliotecario:
 def __init__(self):
 self.livros = [Livro("Título", "Autor")] # Creates inseparable books
```

**Solution** : Always inject pre-existing objects

```
Correct (aggregation approach)
class Bibliotecario:
 def adicionar_livro(self, livro_existente): # Accepts external book
 self.livros.append(livro_existente)
```

# Real-World Analogies

---

## 1. University System :

- Professor (whole) ↔ Courses (parts)
- Courses continue if professor leaves

## 1. Playlist System :

- Playlist (whole) ↔ Songs (parts)
- Songs exist without playlists

# Exercises to Reinforce Learning

---

## 1. Extend the Library System :

- Add a **Biblioteca** class that also aggregates **Livro** objects
- Show how the same book can be in both librarian's list and library catalog

## 1. Implement Book Transfer :

```
def transferir_livro(self, outro_bibliotecario, titulo):
 # Find and transfer book to another librarian
```

## 1. Add Date Tracking :

- Modify **Livro** to track when it was added to a librarian's collection
- Show how this metadata doesn't affect the book's independent existence