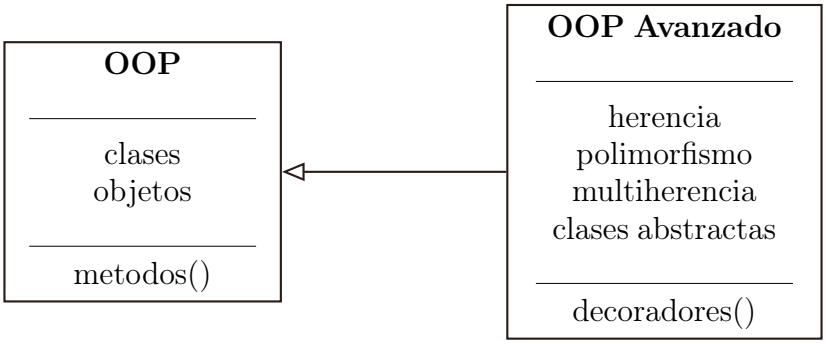
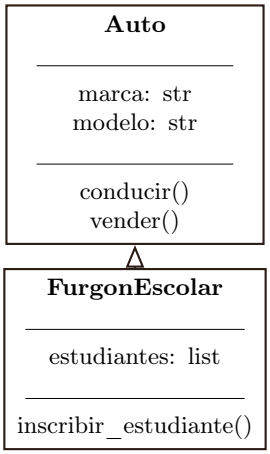


Resumen de Contenidos

IIC2233 — Programación Avanzada

Resumen elaborado por el equipo de ayudantía, en base al material oficial del curso



Contents

1 Introducción **2**

2 Herencia **3**
2.1 Ejemplo: especializando la clase `Auto` . . . 3
2.2 Sobreescritura de métodos: *overriding* . . . 4
2.3 Obtener la clase superior: `super()` . . . 5
2.4 Ejemplo: herencia con *built-ins* . . . 5

3 Polimorfismo **6**
3.1 *Overriding* como polimorfismo . . . 6
3.2 *Overloading* de operadores . . . 7
3.3 `__repr__` vs `__str__` . . . 8
3.4 *Duck typing* . . . 9

4 Multiherencia **9**
4.1 Diferencias, ventajas y desafíos . . . 10
4.2 Conflictos y el Orden de Resolución de Métodos (MRO) . . . 10
4.3 El problema del diamante . . . 11
4.4 Solucionando el diamante con `super()` . . . 12
4.5 Multiherencia con distintos argumentos: `*args` y `**kwargs` . . . 13
4.6 Ejemplo aplicado: la clase `Avatar` . . . 14

5 Clases abstractas **15**
5.1 En Python: *Abstract Base Class* . . . 15
5.2 Ejemplo aplicado: clase `Personaje` . . . 16
5.3 Ejemplo aplicado: figuras geométricas . . . 17

6 Decoradores **18**
6.1 Funciones como objetos de primera clase . . . 18
6.2 Sintaxis de los decoradores . . . 18
6.3 Funciones dentro de funciones . . . 19
6.4 *Properties*: `@property` . . . 19
6.4.1 Ejemplo: definiendo una clase `Email` . . . 20
6.5 Métodos estáticos: `@staticmethod` . . . 21



Felipín ha vuelto. Esta vez, en lugar de usar `list` para todo, a Felipín le dio por **copiar y pegar** código: cada vez que necesita una clase parecida a otra, vuelve a escribir todos los atributos y métodos desde cero, en lugar de reutilizarlos con herencia.

No seas como Felipín. Las herramientas de este resumen — herencia, polimorfismo, clases abstractas y decoradores — existen precisamente para que puedas **reutilizar, extender y organizar** tu código sin repetirlo.

1 Introducción

La **programación orientada a objetos** (OOP) no se detiene en definir clases con atributos y métodos: su verdadero poder aparece cuando empezamos a relacionar clases entre sí y a dotarlas de comportamientos flexibles. Este resumen cubre las herramientas que permiten llevar el modelamiento de un programa a un siguiente nivel:

■ Los cinco pilares de este resumen

- **Herencia**: reutilizar atributos y métodos de una clase en otra, estableciendo relaciones de especialización.
- **Polimorfismo**: usar objetos de distintos tipos con la misma interfaz (*overriding*, *overloading* y *duck typing*).
- **Multiherencia**: heredar de más de una clase a la vez, y resolver los conflictos que esto introduce (MRO, problema del diamante).
- **Clases abstractas**: definir *contratos* que obligan a las subclases a implementar cierto comportamiento.

- **Decoradores**: modificar el comportamiento de funciones y métodos de forma limpia y reutilizable (`@property`, `@staticmethod`, entre otros).

Todos estos conceptos comparten un mismo objetivo: escribir código **más reutilizable, extensible y mantenible**. Vamos a revisarlos uno por uno.

2 Herencia

La **herencia** (*inheritance*) es una relación de **especialización y generalización** entre clases: una **subclase** *hereda* atributos y métodos de una **superclase**. La subclase posee todo lo que tiene la superclase, pero además puede definir sus propios atributos y métodos, o modificar los heredados. Esto nos permite **reutilizar** código ya escrito en vez de duplicarlo.

2.1 Ejemplo: especializando la clase `Auto`

Consideremos una clase `Auto`, con atributos como `marca`, `modelo` y `color`, y métodos como `conducir`, `vender` y `leer_odometro`:

```python
class Auto:

    def __init__(self, marca, modelo, año, color, km):
        self.marca = marca
        self.modelo = modelo
        self.año = año
        self.color = color
        self._kilometraje = km
        self._dueño = None

    def conducir(self, kms):
        print(f"Conduciendo {kms} kilometros")
        self._kilometraje += kms

    def vender(self, nuevo_dueño):
        self._dueño = nuevo_dueño
        print(f"Auto vendido a {nuevo_dueño}")

    def leer_odometro(self):
        return self._kilometraje
```

Si necesitamos modelar un `FurgonEscolar`, es natural pensar que **es un tipo particular de** `Auto`: comparte todos sus atributos y métodos, pero además tiene una lista de estudiantes y un método para inscribirlos. En vez de reescribir todo desde cero, usamos herencia: en la definición de la clase, indicamos entre paréntesis de qué clase hereda.

```python
class FurgonEscolar(Auto): # Aqui se marca de donde hereda
    """Subclase de Auto"""

    def __init__(self, marca, modelo, año, color, km):
        # Delegamos la inicializacion de los atributos heredados
        Auto.__init__(self, marca, modelo, año, color, km)

        # Atributo unico de FurgonEscolar, no existe en Auto
        self.estudiantes = []

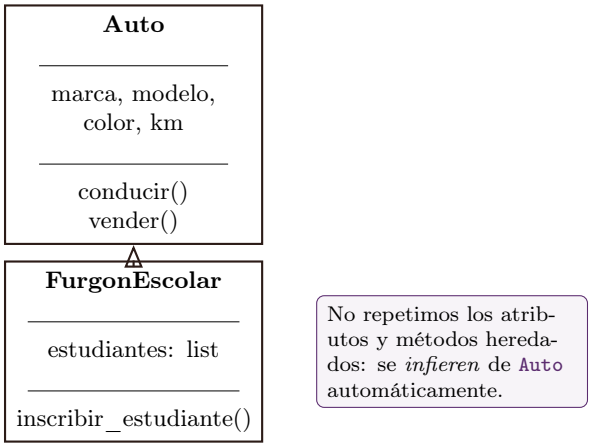
    # Metodo especifico de esta subclase
    def inscribir_estudiante(self, estudiante):
        self.estudiantes.append(estudiante)
```

Cuando `FurgonEscolar` hereda de `Auto`, decimos que:

■ Vocabulario de la herencia

- `FurgonEscolar` es una **especialización** de `Auto`.
- `FurgonEscolar` es una **subclase** (o clase hija) de `Auto`.
- `FurgonEscolar` **extiende** a la clase `Auto`.
- `Auto` es la **superclase** (o clase madre) de `FurgonEscolar`.

Gráficamente, esta relación se representa con una flecha de triángulo abierto (▷) que apunta desde la subclase hacia la superclase:



Un objeto de tipo `FurgonEscolar` tiene, entonces, **todo** lo que tenía `Auto`, más lo definido específicamente para él:

```
furgon = FurgonEscolar('Kia', 'Sportage', 2000, "Blanco", 135000)
furgon.conducir(5) # Metodo heredado de Auto
furgon.inscribir_estudiante('Jaimito') # Metodo propio de FurgonEscolar
print(furgon.marca, furgon.estudiantes) # Kia ['Jaimito']
```

2.2 Sobreescritura de métodos: *overriding*

A veces un método heredado no nos sirve *tal cual* en la subclase, y queremos reemplazarlo. En Python basta con **volver a definir un método con el mismo nombre** dentro de la subclase: al invocarlo desde un objeto de la subclase, se ejecuta la nueva versión y no la de la superclase. A esto se le llama ***overriding***.

```
class FurgonEscolar(Auto):
    """Subclase de Auto"""

    def __init__(self, marca, modelo, año, color, kms):
        Auto.__init__(self, marca, modelo, año, color, kms)
        self.estudiantes = []

    def inscribir_estudiante(self, estudiante):
        self.estudiantes.append(estudiante)

    # Overriding del metodo conducir original
    def conducir(self, distancia):
        # Aqui NO usamos la version original de conducir
        print(f"Conduciendo con cuidado {distancia} kilometros")
```

Ahora, si llamamos a `conducir` desde una instancia de `Auto` se ejecuta la versión original, pero si lo llamamos desde una instancia de `FurgonEscolar` se ejecuta la versión sobreescrita:

```
auto = Auto('Suzuki', 'Grand Nomade', 2015, "Naranjo", 35000)
furgon = FurgonEscolar('Kia', 'Sportage', 2000, "Blanco", 135000)

auto.conducir(12) # "Conduciendo 12 kilometros"
furgon.conducir(5) # "Conduciendo con cuidado 5 kilometros"
```

★ Mismo método, distinto comportamiento

Al ejecutar el **mismo** método sobre objetos de clases distintas, el resultado puede ser diferente según cuál clase implementa ese método. Este es un primer adelanto de lo que en la Sección 3 llamaremos **polimorfismo**.

2.3 Obtener la clase superior: `super()`

En el ejemplo anterior, para inicializar los atributos heredados llamamos explícitamente a `Auto.__init__(self, ...)`. Esto funciona, pero **acopla** el código de `FurgonEscolar` al nombre exacto de su superclase. Python ofrece la función `super()`, que permite invocar el método de la superclase **sin nombrarla explícitamente**: `super().metodo(argumentos)`.

```
class FurgonEscolar(Auto):
    """Subclase de Auto"""

    def __init__(self, marca, modelo, año, color, kms):
        # Usamos super() en vez de nombrar "Auto" directamente
        super().__init__(marca, modelo, año, color, kms)
        self.estudiantes = []

    def inscribir_estudiante(self, estudiante):
        self.estudiantes.append(estudiante)

    def conducir(self, distancia):
        print(f"Conduciendo con cuidado {distancia} kilometros")
```

★ ¿Por qué preferir `super()`?

Usar `super()` en vez de nombrar la clase madre explícitamente hace el código **más mantenible** (si cambia el nombre de la superclase, no hay que tocar nada más) y **evita problemas al usar multiherencia** (Sección 4), donde nombrar clases explícitamente puede provocar que un método se ejecute más de una vez.

2.4 Ejemplo: herencia con *built-ins*

La herencia no solo aplica a nuestras propias clases: también podemos extender clases *built-in* de Python, como `list`, `dict` o `str`. La subclase hereda todos los métodos originales, y puede agregar los suyos propios.

```
class ListaDeContactos(list):
    """Extiende list agregando busqueda por nombre"""

    def buscar(self, nombre):
        matches = []
        for contacto in self:
            if nombre in contacto.nombre:
                matches.append(contacto)
```

```
        return matches


class Contacto:
    def __init__(self, nombre, email):
        self.nombre = nombre
        self.email = email


class Familiar(Contacto):
    """Especializacion de Contacto con un tipo de relacion"""

    def __init__(self, nombre, email, relacion):
        super().__init__(nombre, email)
        self.relacion = relacion
```

Una vez creado el objeto `ListaDeContactos`, este posee **todos los métodos de** `list` (`append`, `pop`, `sort`, ...) más el método `buscar` que definimos:

```
contactos = ListaDeContactos()
contactos.append(Familiar("Catalina Kendrick", "cata.k@mail.com", "prima"))
contactos.append(Contacto("Ami Carol", "ami.carol@mail.com"))
contactos.append(Familiar("Carol Fernandez", "klaus@mail.com", "Hermana"))

nombres = [c.nombre for c in contactos.buscar("Carol")]
print(nombres) # ['Catalina Kendrick', 'Carol Fernandez']
```

3 Polimorfismo

El **polimorfismo** es la propiedad que permite enviar **mensajes sintácticamente iguales a objetos de tipos distintos**. En otras palabras: usar objetos de clases diferentes con la **misma interfaz**. Los dos mecanismos típicos para lograr polimorfismo son *overriding* y *overloading*.

■ Dos caminos hacia el polimorfismo

- ***Overriding***: una subclase sobreescribe la implementación de un método heredado desde su superclase.
- ***Overloading***: definir un método con el mismo nombre, pero con distinto número o tipo de parámetros, para que se comporte distinto según los argumentos recibidos.

! Python no soporta *function overloading*

A diferencia de Java, C# o C++, Python **no** permite definir la misma función más de una vez con distinto número de parámetros y esperar que ambas versiones coexistan: cada vez que Python ve una definición nueva de una función (o clase, o variable), **se queda con la más reciente**, y las anteriores quedan sobreescritas. Sin embargo, sí es posible "simular" esta capacidad usando parámetros con valores por defecto o `*args`/`**kwargs`.

3.1 *Overriding* como polimorfismo

El siguiente ejemplo define una superclase `Variable` que almacena datos en un atributo `data`, y tres subclases (`Ingresos`, `Comunas`, `Puestos`) que sobreescriben el método `representante` para calcular, respectivamente, el promedio, la moda, o el cargo de mayor jerarquía.

```
import statistics
```

```python
class Variable:
    def __init__(self, data):
        self.data = data

    def representante(self): # Implementacion vacia en la clase madre
        pass


class Ingresos(Variable):
    def representante(self):
        return statistics.mean(self.data)


class Comunas(Variable):
    def representante(self):
        return statistics.mode(self.data)


class Puestos(Variable):
    # Atributo de clase, compartido por todas las instancias
    categorias = ['Alumno en Practica', 'Analista', 'SubGerente', 'Gerente']

    def representante(self):
        puestos = [Puestos.categorias.index(c) for c in self.data]
        return Puestos.categorias[max(puestos)]
```

Aunque las tres subclases heredan de la misma superclase y guardan tipos de datos distintos, podemos llamar a `representante()` sobre **cualquiera** de ellas con la misma sintaxis, y cada una responde según su propia implementación:

```python
lista_pesos = Ingresos([50, 80, 90, 150, 45])
lista_comunas = Comunas(['Providencia', 'Macul', 'Providencia'])

print(lista_pesos.representante()) # 83.0
print(lista_comunas.representante()) # Providencia
```

★ Búsqueda del método

Cuando se invoca un método sobre un objeto, Python primero busca su implementación en la **clase del objeto**. Si no la encuentra, sube por la jerarquía hasta encontrarla en alguna superclase. Este mecanismo de búsqueda es la base de cómo *overriding* produce polimorfismo.

3.2 *Overloading* de operadores

Aunque Python no soporta *function overloading* tradicional, **sí** permite el *overloading* de sus **operadores built-in**. Por ejemplo, `+` suma números, concatena *strings* o mezcla listas dependiendo del tipo de los operandos. Podemos lograr este mismo comportamiento en nuestras propias clases implementando **métodos mágicos** (*dunder methods*).

```python
class Carro:

    def __init__(self, pan, leche, agua):
        self.pan = pan
        self.leche = leche
        self.agua = agua

    def __add__(self, otro): # Permite usar el operador "+"
        return Carro(self.pan + otro.pan,
```

```
                    self.leche + otro.leche,
                    self.agua + otro.agua)

    def __str__(self):
        return f"Pan:{self.pan}, Leche:{self.leche}, Agua:{self.agua}"


carro_1 = Carro(1, 2, 3)
carro_2 = Carro(3, 4, 5)
print(carro_1 + carro_2) # Pan:4, Leche:6, Agua:8
```

De la misma forma podemos personalizar comparaciones, implementando por ejemplo `__lt__` (*less than*) para el operador `<`:

```
import math

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

    def magnitud(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)

    def __lt__(self, otro):
        return self.magnitud() < otro.magnitud()

print(Vector(2, 4) < Vector(8, 3)) # True
```

Otros métodos mágicos de comparación

- `__le__`: "menor o igual que" (`<=`)
- `__gt__`: "mayor que" (`>`)
- `__ge__`: "mayor o igual que" (`>=`)
- `__eq__`: "igual que" (`==`)
- `__ne__`: "distinto de" (`!=`)

3.3 `__repr__` vs `__str__`

Ambos métodos entregan una representación en texto de un objeto, pero con objetivos distintos:

Diferencia de propósito

- `__str__`: representación **legible** (*human-readable*), pensada para un **usuario** del programa. La usa `print`.
- `__repr__`: representación **completa y sin ambigüedades**, pensada para un **desarrollador**. Si no existe `__str__`, `print` recurre a `__repr__`.

```
class Fraccion:
    def __init__(self, numerador, denominador):
        self.numerador = numerador
        self.denominador = denominador

    def __repr__(self):
        return f"Fraccion({self.numerador}, {self.denominador})"
```

```
    def __str__(self):
        return f"{self.numerador} / {self.denominador}"

frac = Fraccion(3, 4)
print(frac) # 3 / 4 -> usa __str__
print(repr(frac)) # Fraccion(3, 4) -> usa __repr__
```

! Al imprimir listas de objetos

Cuando se imprime una **lista** de objetos, Python usa `__repr__` de cada elemento, **no** `__str__`. Por eso, si una clase solo define `__str__`, imprimir `[frac1, frac2]` mostrará la dirección de memoria de cada objeto en vez de su forma legible. Definir `__repr__` es clave si se quiere imprimir listas de objetos de forma útil.

3.4 *Duck typing*

★ La frase que lo resume todo

"If it walks like a duck and quacks like a duck, then it is a duck" — no importa de qué tipo sea un objeto, mientras tenga el comportamiento (método) que se necesita.

El *duck typing* es una característica de los lenguajes con **tipado dinámico**, como Python, que permite comportamiento polimórfico **sin necesidad de herencia**. Si dos clases sin ninguna relación entre sí implementan los mismos métodos, una función puede operar sobre objetos de ambas sin distinguirlas.

```
class Pato:
    def gritar(self):
        print("Quack!")
    def caminar(self):
        print("Caminando como un pato")

class Persona:
    def gritar(self):
        print("¡Ahhh!")
    def caminar(self):
        print("Caminando como un humano")

def activar(pato): # No exige que "pato" sea de clase Pato
    pato.gritar()
    pato.caminar()

activar(Pato()) # Quack! / Caminando como un pato
activar(Persona()) # ¡Ahhh! / Caminando como un humano
```

La función `activar` no sabe (ni le interesa) el tipo real de su argumento: simplemente llama a `gritar` y `caminar`, y en tiempo de ejecución Python determina si el objeto entregado los implementa. En lenguajes con **tipado estático** (C, C++, Java, C#), este mecanismo no funciona de la misma manera, ya que el tipo de cada argumento debe declararse explícitamente.

4 Multiherencia

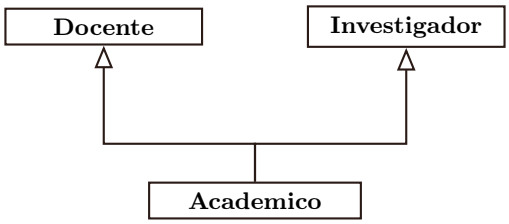
Así como una subclase puede heredar de una superclase, en Python es posible heredar de **más de una clase a la vez**. A esto se le llama **multiherencia**.

```
class Docente:
```

```
    pass # Metodos y atributos propios de la docencia

class Investigador:
    pass # Metodos y atributos propios de la investigacion

class Academico(Docente, Investigador):
    pass # Combina el comportamiento de ambas
```



4.1 Diferencias, ventajas y desafíos

■ Herencia simple vs. multiherencia

- **Herencia simple**: se sigue una única línea de herencia (por ejemplo, `Animal` → `Perro` → `Terrier`).
- **Multiherencia**: introduce una estructura de **árbol** más compleja, donde una clase derivada tiene varias "ramas" de comportamiento a considerar.

★ Ventajas

Flexibilidad y reutilización: permite combinar funcionalidades diversas sin reescribir código. **Representación de múltiples roles**: útil cuando un objeto cumple varios papeles a la vez (como un académico que es docente e investigador).

! Desafíos

Ambigüedades: si dos clases base tienen métodos con el mismo nombre, ¿cuál se ejecuta? **Orden de Resolución de Métodos (MRO)**: Python necesita un orden bien definido para buscar métodos en la jerarquía. **Complejidad**: usar multiherencia en exceso dificulta el mantenimiento y la comprensión del diseño.

★ Uso adecuado

Se recomienda la multiherencia solo cuando un objeto necesita cumplir **roles múltiples de forma natural**, y aporta claridad al diseño — no como un atajo para forzar reutilización de código. En casos de conflicto, hay que tener claro el MRO, o considerar alternativas como **composición**.

4.2 Conflictos y el Orden de Resolución de Métodos (MRO)

Supongamos que `Docente` e `Investigador` definen, cada una, un método `publicar()` distinto, y que `Academico` hereda de ambas:

```
class Docente:
    def publicar(self):
        print("Publicacion orientada a la docencia")

class Investigador:
    def publicar(self):
```

```
        print("Publicacion orientada a la investigacion")

class Academico(Docente, Investigador):
    pass

a = Academico()
a.publicar() # ¿Cual se ejecuta?
```

Cuando se llama a `a.publicar()`, Python sigue el **Orden de Resolución de Métodos** (*MRO, Method Resolution Order*):

■ Pasos del MRO

1. Busca el método en la propia clase derivada (`Academico`). No está definido aquí.
2. Busca en la **primera** clase base según el orden de declaración: en `class Academico(Docente, Investigador)`, primero `Docente`. Como `Docente.publicar()` existe, se usa **ese**.
3. Si `Docente` no lo tuviera, continuaría buscando en `Investigador`, y así sucesivamente.

Por lo tanto, `a.publicar()` imprime *"Publicación orientada a la docencia"*. Si invertimos el orden de las clases base (`class Academico(Investigador, Docente)`), el resultado cambia y se ejecuta la versión de `Investigador`.

★ Algoritmo C3 Linearization

La estrategia de resolución se basa en el algoritmo de **C3 Linearization**, que asegura que la jerarquía de clases se mantenga **consistente y predecible**: la primera clase base declarada tiene prioridad sobre las siguientes en caso de conflicto.

4.3 El problema del diamante

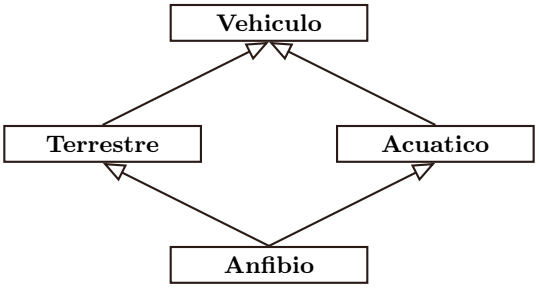
Una **jerarquía de diamante** ocurre cuando hay más de un "camino" en la jerarquía desde una clase inferior hasta una clase superior común. Consideremos:

```
class Vehiculo:
    def acelerar(self): pass

class Terrestre(Vehiculo):
    def acelerar(self): pass

class Acuatico(Vehiculo):
    def acelerar(self): pass

class Anfibio(Terrestre, Acuatico):
    def acelerar(self): pass
```



Si cada subclase llama **explícitamente** al método de su superclase (sin usar `super()`), `Vehiculo.acelerar()` termina llamándose **dos veces** al invocar `Anfibio().acelerar()`: una vez por la rama `Terrestre` y otra por la rama `Acuatico`.

```
class Vehiculo:
    num_llamadas_vehiculo = 0
    def acelerar(self):
        self.num_llamadas_vehiculo += 1

class Terrestre(Vehiculo):
    def acelerar(self):
        Vehiculo.acelerar(self) # Llamado explicito

class Acuatico(Vehiculo):
    def acelerar(self):
        Vehiculo.acelerar(self) # Llamado explicito

class Anfibio(Terrestre, Acuatico):
    def acelerar(self):
        Terrestre.acelerar(self)
        Acuatico.acelerar(self)

a = Anfibio()
a.acelerar()
print(a.num_llamadas_vehiculo) # 2 <-- ¡Problema!
```

! ¿Por qué es un problema?

Si el método de `Vehiculo` gastara combustible, registrara eventos o inicializara una conexión, estaríamos **duplicando** una operación que solo debía ocurrir una vez. En Python, además, **toda** clase hereda implícitamente de `object`, por lo que cualquier multiherencia sin cuidado puede generar este mismo problema, incluso sin una tercera superclase explícita.

4.4 Solucionando el diamante con `super()`

La solución consiste en que **cada clase llame a** `super()` en vez de nombrar explícitamente a su superclase. Python se encarga de que la llamada respete el **MRO**, garantizando que cada clase se ejecute **una sola vez**.

```
class ClaseB:
    num_llamadas_B = 0
    def llamar(self):
        self.num_llamadas_B += 1

class SubClaseIzquierda(ClaseB):
    def llamar(self):
        super().llamar()

class SubClaseDerecha(ClaseB):
    def llamar(self):
        super().llamar()

class SubClaseA(SubClaseIzquierda, SubClaseDerecha):
    def llamar(self):
        super().llamar()

s = SubClaseA()
s.llamar()
```

```
print(s.num_llamadas_B) # 1 <-- ¡Arreglado!
```

Podemos consultar el orden exacto que sigue `super()` con el atributo `__mro__` (*Method Resolution Order*), calculado mediante el algoritmo **C3**:

```
SubClaseA.__mro__
# (SubClaseA, SubClaseIzquierda, SubClaseDerecha, ClaseB, object)
```

! No toda multiherencia es válida

Si dos clases heredan de `X` e `Y` en **distinto orden** (por ejemplo, `A(X, Y)` y `B(Y, X)`), y luego una tercera clase hereda de ambas (`F(A, B)`), Python **no puede** construir un MRO consistente y lanza un `TypeError` al definir la clase. La solución es que las clases base compartan siempre el mismo orden de herencia.

4.5 Multiherencia con distintos argumentos: `*args` y `**kwargs`

El problema se complica cuando cada superclase espera **argumentos distintos** en su `__init__`. Si `Academico` hereda de `Docente` e `Investigador`, ¿qué argumentos le entregamos a `super().__init__()`?

```
# Version incorrecta: inicializa "object" dos veces
class Academico(Docente, Investigador):
    def __init__(self, nombre, oficina, area, depto):
        Investigador.__init__(self, area) # Llamado explicito
        Docente.__init__(self, depto) # Llamado explicito
        self.nombre = nombre
        self.oficina = oficina
```

La solución correcta es que **cada clase** reciba `**kwargs` y se lo reenvíe a `super().__init__(**kwargs)`. Así, cada inicializador **consume** del diccionario solo los argumentos con *nombre* (keyword) que necesita, y pasa el resto hacia el siguiente eslabón del MRO:

```
class Investigador:
    def __init__(self, area='', **kwargs):
        super().__init__(**kwargs)
        self.area = area
        self.num_publicaciones = 0

class Docente:
    def __init__(self, departamento='', **kwargs):
        super().__init__(**kwargs)
        self.departamento = departamento
        self.num_cursos = 3

class Academico(Docente, Investigador):
    def __init__(self, nombre, oficina, **kwargs):
        super().__init__(**kwargs)
        self.nombre = nombre
        self.oficina = oficina

p1 = Academico(
    "Emilia Donoso", oficina="05",
    area="Inteligencia de Maquina",
    departamento="Ciencia De La Computacion"
)
print(p1.area, p1.departamento) # Se inicializan ambos, una sola vez
```

★ ¿Por qué funciona?

Cada clase en el MRO recibe `**kwargs`, extrae con *nombre* lo que necesita, y reenvía el resto con `super().__init__(**kwargs)` al **siguiente** eslabón. Al llegar a la última clase antes de `object`, el diccionario ya está vacío — exactamente lo que `object.__init__()` espera. Así, cada superclase se inicializa **una sola vez**, sin importar cuántas ramas de la jerarquía existan.

4.6 Ejemplo aplicado: la clase `Avatar`

Modelemos a un `Avatar`, maestro de los 4 elementos, que hereda de cuatro clases `MaestroFuego`, `MaestroAgua`, `MaestroTierra` y `MaestroAire`, cada una con su propio método `saludar()`.

```
class Avatar(MaestroFuego, MaestroAgua, MaestroTierra, MaestroAire):
    def __init__(self, nombre):
        super().__init__(nombre)

Avatar("Aang").saludar()
# "Hola!! mi nombre es Aang y soy un maestro fuego" <-- solo UN saludo
```

Como las cuatro clases tienen un método con el **mismo nombre**, el MRO solo llega hasta la **primera** clase de la lista (`MaestroFuego`). Ni siquiera agregar `super().saludar()` dentro de `Avatar` soluciona esto por sí solo, porque `super()` respeta el MRO y solo avanza **un paso** en la cadena.

! Una solución riesgosa: encadenar `super()` en cada maestro

Se podría agregar `super().saludar()` dentro de `MaestroFuego`, `MaestroAgua` y `MaestroTierra` para forzar que cada uno llame al siguiente en el MRO. Esto **funciona**, pero solo si `MaestroAire` es la **última** clase heredada, y además **rompe** el comportamiento de `MaestroFuego` si se instancia **por sí sola** (su `super()` terminaría llamando a `object`, que no tiene `saludar`). Modificar 3 clases pequeñas solo para que una nueva clase funcione es una señal de mal diseño.

La solución más robusta es que sea `Avatar` quien llame **explícitamente** al método de cada clase padre, sin depender de `super()` para este caso particular:

```
class Avatar(MaestroFuego, MaestroAgua, MaestroTierra, MaestroAire):
    def __init__(self, nombre):
        super().__init__(nombre)

    def saludar(self):
        MaestroFuego.saludar(self)
        MaestroAgua.saludar(self)
        MaestroTierra.saludar(self)
        MaestroAire.saludar(self)

Avatar("Roku").saludar()
# Imprime los 4 saludos, y las clases Maestro siguen funcionando solas
```

★ Lección de este ejemplo

No existe una única forma "correcta" de resolver un conflicto de multiherencia: a veces `super()` es la solución (problema del diamante clásico), y a veces conviene **llamar directamente** al método de cada clase padre para no modificar el comportamiento de las clases originales. La clave es **reflexionar** antes de elegir, y preguntarse si conviene usar composición en su lugar.

5 Clases abstractas

Una **clase abstracta** representa un concepto que **no debería instanciarse directamente**, sino que sirve como molde para que otras clases lo especialicen. Por ejemplo, `Mamifero` es abstracta: describe características comunes a `Perro`, `Humano` o `Ballena`, pero no tiene sentido crear un objeto "mamífero genérico".

■ Una clase es abstracta cuando...

- No debería instanciarse directamente.
- Contiene uno o más **métodos abstractos**.
- Sus subclases están obligadas a implementar **todos** esos métodos abstractos.

Un **método abstracto** representa un comportamiento que **toda** subclase debe implementar, aunque cada una lo haga de forma distinta. Esto funciona como un **contrato**: la clase abstracta establece *qué* debe existir, y cada subclase decide *cómo* se implementa. Las clases abstractas también pueden tener métodos **normales** (no abstractos), que se heredan sin necesidad de reimplementarlos.

5.1 En Python: *Abstract Base Class*

Python no tiene una sintaxis nativa para clases abstractas, pero el módulo `abc` (*Abstract Base Class*) provee la clase `ABC` y el decorador `@abstractmethod` para lograrlo:

```python
from abc import ABC, abstractmethod

class Base(ABC): # Heredar de ABC marca la clase como abstracta

    @abstractmethod # metodo_1 pasa a ser obligatorio en subclases
    def metodo_1(self):
        pass

    @abstractmethod
    def metodo_2(self):
        pass
```

Si intentamos instanciar `Base` directamente, obtenemos un error, ¡porque es abstracta **y** contiene métodos abstractos!

```python
instancia = Base()
# TypeError: Can't instantiate abstract class Base
# with abstract methods metodo_1, metodo_2
```

Si una subclase implementa **todos** los métodos abstractos, sí puede instanciarse. Pero si deja alguno sin implementar, el error persiste — incluso en la subclase:

```python
class SubClase1(Base):
    def metodo_1(self): pass
    def metodo_2(self): pass

instancia = SubClase1() # Funciona: implementa ambos metodos

class SubClase2(Base):
    def metodo_1(self): pass # Falta metodo_2

instancia = SubClase2() # TypeError: falta implementar metodo_2
```

★ Métodos abstractos con implementación

Un método abstracto **puede** tener cuerpo (no solo `pass`). Una subclase puede reimplementarlo y, si quiere, acceder a la versión original con `super().metodo_2()`. Esto permite que la clase abstracta ofrezca una base común, mientras obliga a cada subclase a hacerla explícita.

5.2 Ejemplo aplicado: clase `Personaje`

Modelemos personajes de un videojuego: un `Jugador` y varios `Enemigo`. Ambos comparten estructura (`nombre`, posición, `energia`) y un método `simular` que los mueve y gasta energía cada segundo, pero **cómo** se mueven, saludan y gastan energía es distinto para cada uno.

```python
from abc import ABC, abstractmethod
from random import randint

class Personaje(ABC):

    def __init__(self, nombre, x, y, energia):
        self.nombre = nombre
        self.x, self.y = x, y
        self.energia = energia

    def simular(self): # Metodo NO abstracto, compartido
        while self.energia > 0:
            self.saludar()
            self.moverse()
            self.gastar_energia()

    @abstractmethod
    def moverse(self):
        pass

    @abstractmethod
    def gastar_energia(self):
        pass

    @abstractmethod
    def saludar(self):
        print(f"Soy {self.nombre}. Estoy en {(self.x, self.y)}.")
```

```python
class Jugador(Personaje):
    def moverse(self):
        self.x += 1
        self.y += 1

    def gastar_energia(self):
        cambio = randint(-1, 3) # A veces gana energia
        self.energia = max(0, self.energia - cambio)

    def saludar(self):
        super().saludar() # Reutiliza la version base


class Enemigo(Personaje):
    def moverse(self):
        self.x += randint(-1, 1) # Se mueve al azar
        self.y += randint(-1, 1)
```

```
    def gastar_energia(self):
        self.energia = max(0, self.energia - 1) # Tasa constante

    def saludar(self):
        print("¡Te atraparé!")
        super().saludar() # Agrega su propio grito
```

Ambas subclases reutilizan `__init__` y `simular` de `Personaje`, pero implementan su propia versión de los tres métodos abstractos — exactamente el comportamiento que esperábamos del "contrato".

5.3 Ejemplo aplicado: figuras geométricas

Otro caso típico es forzar que **toda** figura geométrica tenga forma de calcular su área y su perímetro:

```
import math
from abc import ABC, abstractmethod

class Figura(ABC):
    @abstractmethod
    def __init__(self):
        pass

    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimetro(self):
        pass


class Cuadrado(Figura):
    def __init__(self, lado):
        super().__init__()
        self.lado = lado

    def area(self):
        return math.pow(self.lado, 2)

    def perimetro(self):
        return 4 * self.lado


class Circulo(Figura):
    def __init__(self, radio):
        super().__init__()
        self.radio = radio

    def area(self):
        return math.pi * math.pow(self.radio, 2)

    def perimetro(self):
        return 2 * math.pi * self.radio
```

! El contrato se cumple, o no compila

Si intentáramos crear `Cuadrado` sin implementar `area()` o `perimetro()`, Python lanzaría un `TypeError` al instanciar, **antes** de que el error pudiera manifestarse más adelante como un `AttributeError` silencioso. Las clases abstractas transforman errores de diseño en errores **tempranos y explícitos**.

6 Decoradores

Un **decorador** es una función que toma **otra función** (o método) como entrada y retorna una versión modificada de ella. Permiten agregar funcionalidad a una función **sin modificar su código original**: registro (*logging*), validaciones, control de acceso, *caching*, y mucho más.

6.1 Funciones como objetos de primera clase

Los decoradores son posibles porque en Python las funciones son **objetos de primera clase**:

■ ¿Qué significa esto?

- Se pueden **asignar** a una variable, como un `int` o un `string`.
- Se pueden **pasar como argumento** a otra función (o devolver como resultado).
- Se pueden **almacenar** en listas, diccionarios, etc.
- Pueden tener **atributos** propios.

```
def decir_hola(nombre):
    return f"Hola {nombre}"

saludar = decir_hola # Sin parentesis: referencia a la funcion
saludar("Dani") # "Hola Dani" (se ejecuta al llamar)
```

! Referencia vs. llamada

Un nombre de función **sin** paréntesis (`decir_hola`) es una **referencia** a la función — no se ejecuta. Un nombre de función **con** paréntesis (`decir_hola()`) la **llama**, y hace referencia a su valor de retorno. Esta distinción es la base de cómo funcionan los decoradores.

6.2 Sintaxis de los decoradores

El símbolo `@` aplica un decorador a la función definida justo debajo. Esta sintaxis:

```
@decorador
def mi_funcion():
    pass
```

es exactamente equivalente a:

```
def mi_funcion():
    pass

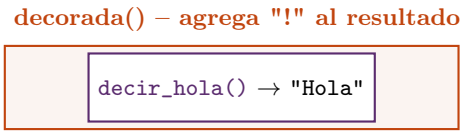
mi_funcion = decorador(mi_funcion)
```

Es decir: `decorador` recibe `mi_funcion` como argumento, y lo que retorne `decorador(mi_funcion)` se convierte en el **nuevo** valor de `mi_funcion`.

```python
def agregar_exclamacion(func):
    def decorada():
        return func() + "!"
    return decorada

@agregar_exclamacion
def decir_hola():
    return "Hola"

print(decir_hola()) # "Hola!"
```



6.3 Funciones dentro de funciones

Definir una función **dentro** de otra es válido en Python por la misma razón: las funciones son objetos de primera clase. Las funciones internas (*funciones anidadas*) solo existen dentro del ámbito de la función externa.

```python
def mi_decorador(func):
    def funcion_interna():
        print("Antes de la funcion original")
        func()
    return funcion_interna # Retorna la funcion modificada

@mi_decorador
def saludar():
    print("Hola!")

saludar()
# "Antes de la funcion original"
# "Hola!"
```

Resumiendo lo visto hasta ahora:

■ Anatomía de un decorador

- Es una función (*callable*) que toma un objeto función como entrada.
- Puede añadir comportamiento, envolver la llamada original, o modificarla.
- Devuelve la función original o una versión modificada de ella.

6.4 *Properties*: `@property`

En Python **todos** los atributos son públicos: anteponer un *underscore* (`_atributo`) es solo una **convención** para indicar "uso interno", no una protección real. Cuando queremos controlar el acceso de lectura/escritura de un atributo, usamos **getters** y **setters**.

La función *built-in* `property(fget, fset, fdel)` crea un atributo administrado a partir de funciones *getter*, *setter* y *deleter*:

```python
class Circulo:
    def __init__(self, radio):
        self._radio = radio # Atributo "protegido" por convencion
```

```
    def get_radio(self):
        return self._radio

    def set_radio(self, valor):
        if valor >= 0:
            self._radio = valor
        else:
            print("El radio no puede ser negativo")

    radio = property(get_radio, set_radio) # Definimos la property
```

La forma más común y elegante de lograr lo mismo es con el **decorador** `@property`, que permite acceder a un método **como si fuera un atributo**, sin paréntesis:

```
class Circulo:
    def __init__(self, radio):
        self._radio = radio

    @property
    def radio(self): # Getter
        return self._radio

    @radio.setter
    def radio(self, valor): # Setter, usando el nombre.setter
        if valor >= 0:
            self._radio = valor
        else:
            print("El radio no puede ser negativo")

    @property
    def area(self): # Property de solo lectura (sin setter)
        return 3.14 * self._radio ** 2

c = Circulo(5)
print(c.radio, c.area) # 5 78.5 (sin parentesis!)
c.radio = -3 # "El radio no puede ser negativo"
c.radio = 10 # Cambia el radio correctamente
```

★ Ventaja clave de `@property`

El código que **usa** la clase (`c.radio`) se ve idéntico a un acceso directo de atributo, pero por debajo se ejecuta lógica de validación. Esto permite empezar con atributos simples y **agregar validación después**, sin romper el código que ya usaba la clase.

6.4.1 Ejemplo: definiendo una clase `Email`

```
class Email:
    def __init__(self, address):
        self._email = address

    @property
    def email(self):
        return self._email

    @email.setter
    def email(self, value):
        if '@' not in value:
```

```
            print(f"El string {value} no parece un correo.")
        else:
            self._email = value

mail = Email("profe1@gmail.com")
mail.email = "profe2@gmail.com" # Valido, se actualiza
mail.email = "profe2.com" # Invalido, se rechaza con mensaje
```

6.5 Métodos estáticos: `@staticmethod`

Un **método estático** pertenece a una clase, pero **no depende** de sus atributos ni de una instancia particular: no recibe `self`. Se declara con el decorador `@staticmethod`, y por PEP8 es la forma correcta de agrupar funciones relacionadas dentro de una clase sin necesitar `self`.

```
class MiMath:

    @staticmethod
    def sumar(num1, num2):
        return num1 + num2

    @staticmethod
    def restar(num1, num2):
        return num1 - num2

print(MiMath.sumar(5, 6)) # 11 -- se llama sobre la CLASE
print(MiMath.restar(14, 1)) # 13 -- no se necesita instanciar
```

★ ¿Cuándo usar `@staticmethod`?

Cuando un método **no necesita** leer ni modificar el estado de una instancia (`self`) ni de la clase, pero conceptualmente pertenece a ella. Es un indicador claro, tanto para quien lee el código como para el intérprete, de que ese método es independiente del estado del objeto.

Fin del resumen.