UNIVERSIDAD
DE MURCIA

Tema 2. Miembros, Sobrecarga y Visibilidad

Tecnología de la Programación

Material original por L. Daniel Hernández (*ldaniel@um.es*)

Editado por:

- Sergio López Bernal (*slopez@um.es*)
- Javier Pastor Galindo (*javierpg@um.es*)

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia

14 de septiembre de 2025

Índice de contenidos

UNIVERSIDAD
DE MURCIA

Miembros de una clase

Introducción

- Una clase representa a un conjunto de objetos con la misma estructura.
- Una clase consta de una serie de elementos, llamados miembros.

```
class UnaClase {
   miembros
}
```

- Los miembros de una clase son los elementos de una estructura que la caracterizan y, por ende, permite definir a sus objetos:
  - Las **atributos**, que definirán el estado de cada objeto.
  - Los **métodos**, que establecen el comportamiento (o funcionalidad) de todos los objetos.
  - Pueden existir otros: constantes, eventos, clases internas, etc.
  - En cada lenguaje de programación los miembros pueden variar.
- Los constructores establecen el estado inicial de un objeto recién creado pero formalmente no se considera miembro de una clase.
- Las clases *protegen sus miembros* con restricciones de visibilidad para evitar que cualquier otra clase lo manipule directamente.

UNIVERSIDAD
DE MURCIA

Miembros de una clase

Atributos

Atributos de una clase

- Los atributos miembros definen el estado de la **clases** o sus **objetos**.
- También se llaman campos.
- Pueden ser de tipo simple (entero, booleanos...) o compuesto (listas, objetos...)
- Existen dos tipos de atributos: de clase y de instancia.
- **Atributos de instancia**. Define el estado particular de un objeto.
  Ejemplo. Sobre la clase `Estudiante`, un alumno tendrá:
  - Color de los ojos
  - Referencia al centro en el que estudia
  - Notas obtenidas
  - ...
- **Atributos de clase.** Define características generales de la clase.
  Ejemplo. Con la clase `Estudiante` podríamos guardar detalles como:
  - Número de estudiantes total (varía con el tiempo)
  - Sistema de calificaciones (constante en el tiempo)
  - ...

Atributos de instancia (u objeto)

Definición y uso interno con self

- Son miembros de una clase que definen el estado de un objeto.
- Se definen dentro del constructor con el prefijo `self.`$nombre_{a}$*tributo*.*Seutilizan*`internamente`*siemprecondichoprefijo*.

```
class Estudiante:

    def __init__(self, color_ojos: str, notas: list[float]):
        self._color_ojos = _color_ojos # Atributo de instancia
        self._notas = notas            # Atributo de instancia

    def imprimir_color_ojos(self):
        print(f'Estudiante con ojos {self._color_ojos}')
```

- Hay, al menos, tantos como objetos o instancias se hayan creado.

```
javi: Estudiante = Estudiante("verdes", [4.5, 6.0, 9.2])
sergio: Estudiante = Estudiante("marrones", [2.0, 8.5, 6.0, 4.4])
# Existen los dos colores y las dos listas, respectivamente
## javi._color_ojos y javi._notas
## sergio._color_ojos y sergio._notas
```

Atributos de instancia (u objeto)

Uso externo con la notación punto

- Los atributos de instancia se pueden acceder **externamente** con la notación `nombreObjeto.nombreAtributo`, pero es una **mala práctica**. La clase debería proveer métodos específicamente para ello.

```
color_ojos_javi : str = javi._color_ojos # no se debe acceder así
javi.consultar_color_ojos() # método que accede a atributo
```

- **CUIDADO**. En Python se crea un **nuevo atributo de instancia** si se realiza una asignación sobre un atributo no declarado en la clase.

```
javi._primer_apellido = "Pastor"
# crea el atributo _primer_apellido sólo para el objeto javi
```

- Para proteger a los objetos de declaraciones no deseadas o de erratas fuera del constructor, la instrucción `__slots__ =[identificadores]` en una clase fija los identificadores de atributos de instancia permitidos.

```
class Estudiante:

    __slots__ = ["_color_ojos", "_notas"]
    ... # Representa código arbitrario

javi._color_hojos = "verdes"     # AttributeError
```

Atributos de clase

- Son miembros de una clase que definen información de la clase en su conjunto (constantes, valores por defecto, contadores, ...).
- Definen atributos comunes a todos los objetos de la clase. Cada atributo existe una única vez, independientemente del número de instancias.
- Se definen fuera de los métodos y ubicándose en **memoria estática**, existiendo desde que se carga la clase y hasta el final de ejecución.
- Son accesibles a través del nombre de la clase con la notación punto: `NombreClase.nombreAtributo`

```
class Estudiante:
    num_estudiantes = 0                          # Atributo de clase

    def __init__(self,calificaciones: list[float]):
        self._calificaciones: list[float] = calificaciones
        Estudiante.num_estudiantes += 1          # Atributo de clase

estudiante: Estudiante = Estudiante([5.5,9.9])
Estudiante.num_estudiantes  # Acceso a atributo de clase
estudiante.num_estudiantes  # Acceso NO recomendado!
```

Al ser compartidas por todos los objetos de la clase, también se podría acceder a través de un objeto de la clase. **NO** se recomienda este uso. De hecho, la modificación a través de `nombre_objeto.nombre_atributo_clase` no modifica el atributo de clase, sino que crea un nuevo atributo de instancia `nombre_atributo_clase` para el objeto.

Otro tipo de variables

Variables locales

- No hay que confundir los **atributos de una clase** con otro tipo de **variables**.
- Además de los atributos de instancia y de clase, en los métodos se pueden usar **variables locales** y **globales** (no son miembros de la clase).

- **Variables locales**. Las auxiliares propias de un método o algoritmo, quedando su acceso limitado a un ámbito local (if, bucle, método, etc).

  Ejemplo. Sobre la clase Estudiante se pueden usar variables locales en el método para calcular la media de las notas de un estudiante.

```
class Estudiante:
    def __init__(self, nombre: str, notas: list[float], altura: float):
        self._nombre: str = nombre
        self._notas: list[float] = notas
        self._altura: float = altura

    def calcular_media(self) -> float:
        suma_notas: float = sum(self._notas)    # Variable local
        num_notas: int = len(self._notas)       # Variable local
        media: float = suma_notas / num_notas   # Variable local
        return media
```

Otro tipo de variables

Variables globales

- **Variables globales**. Definidas en niveles superiores de un programa pues representan conceptos que transcienden a una clase en particular. Son accesibles por cualquier clase o función en un ámbito amplio.

  Ejemplo. La clase Estudiante podría acceder a variables globales como el nombre del planeta donde viven (la Tierra no sólo existe para los estudiantes, es algo más amplio) o el aire que respiran (no es una característica diferenciadora de los estudiantes, pues el aire también es respirado por profesores o animales).

```python
# Variables globales
PLANETA: str = "Tierra"
AIRE: str = "Oxígeno"

class Estudiante:
    def __init__(self, nombre: str, edad: int, altura: float):
        self._nombre: str = nombre
        self._edad: int = edad
        self._altura: float = altura

    def informacion_global(self) -> str:
        return f"{self._nombre} vive en el planeta {PLANETA}
                    y respira {AIRE}."
```

Ámbito de las variables

- El ámbito de una variable hace referencia a las partes del programa en el que una variable es reconocida.
- Una variable es local respecto de un bloque de código si solo es reconocida en ese bloque. Fuera de él, la variable no es reconocida.
- Una variable es global respecto de un bloque de código si se reconoce tanto dentro de dicho bloque como fuera de él.
- En Python se tienen las siguientes situaciones:
  - Variables declaradas "fuera" de una función son globales para ella.
  - Los parámetros de una función son locales para ella.
  - Las variables declaradas en una función son siempre locales.

```
s: str = "global"       # Variable global
def f(param: object): #
    s = "local"         # Nueva variable local
```

  - Para modificar una variable como global dentro de una función debe especificarse la keyword `global`.

```
s: str = "global"     # Variable global
def f(param: object):
    global s          # Indicamos que "s" es global
    s = "global2"     # Modificamos la variable global
```

Ejemplo completo con atributos y variables

```
planeta: str = "La Tierra"                          # Var. Global
class Estudiante:
    num_estudiantes: int = 0                        # Atr. Clase
    def __init__(self, calificaciones: list[float]):
        self._calificaciones = calificaciones       # Atr. Instancia
        Estudiante.num_estudiantes += 1             # Atr. Clase
        print(f'Este vive en {planeta}')            # Var. Global
    def calificacion_media(self) -> float:
        sum: float = 0                              # Var. Local
        for i in range(0,len(self._calificaciones)):
            sum += self._calificaciones[i]  # Var.local y Atr. Instancia
        return sum/len(self._calificaciones)

est: Estudiante = Estudiante ([5,10])
est.calificaciones          # Atributo de objeto
Estudiante.num_estudiantes  # Atributo de clase
est.num_estudiantes         # EVITAR: Atributo de clase a través de objeto
```

- Observa que si sólo vamos a **leer** una variable global en una función (p.e., `planeta`), podemos evitar la keyword `global`. Si se quiere modificar (p.e., `planeta='Marte'`), entonces sí se debe usar la keyword `global` (de lo contrario se crearía una variable local `planeta`).
- Por otro lado, ¿te has preguntado qué ocurre si intentamos modificar el atributo de clase `num_estudiantes` a través del objeto `est` de la forma `est.num_estudiantes = N`? ¡Piensa y pruébalo!

Algunas aclaraciones

- Hemos distinguido 2 tipos de atributos y 2 tipos de variables.
- Cada lenguaje de programación implementa los atributos y variables de forma diferente:
  - Java, por ejemplo, trabaja con atributos de instancia y de clase, así como variables locales. Sin embargo, no se pueden definir variables globales fuera de una clase.
  - Hemos visto que Python sí permite trabajar con variables globales y locales, así como atributos de instancia y de clase.
- Recuerda que el ámbito de los parámetros de una función es local (es decir, fuera de la función no se reconocen). No obstante, en Python:
  - Los parámetros de una función se pasan por referencia (y no por copia de valor). La función tiene acceso a las posiciones de memoria originales.
  - Por otro lado, recuerda que sólo pueden modificarse los tipos de datos mutables.
  - Por tanto, las funciones pueden modificar directamente las **listas**, **conjuntos**, **diccionarios** y **objetos** de nuestros programas gracias al paso por referencia (la función recibe el puntero a memoria) y la mutabilidad (el tipo de dato nos permite la modificación).

Ejercicios de atributos

Ejercicio.

- Si tuvieses que definir constantes matemáticas como π, e, ... ¿de qué tipo de atributo serían? ¿qué nombre le pondrías a la clase?
- Si tienes una clase para los empleados públicos ¿el salario base sería clase o de instancia? ¿y los complementos por antigüedad?
- Considera una casa, donde se emplean las clases `Casa`, `Habitación` y `Silla`. Define, para cada clase, atributos de instancia y de clase. ¿Qué relación hay entre estas clases?

UNIVERSIDAD
DE MURCIA

Miembros de una clase

Métodos

Métodos de una clase

- Los métodos representan el comportamiento de los objetos.
- Un método está asociado con una acción que puede realizar un objeto.
- Siempre se coloca en "el interior" de la definición de una clase.

  Ejemplo.

```
class Coche:
    def __init__(self, gas: int):
        self._gas: int = gas  # Inicializa los litros de gas

    def recarga(self, n: int):  # MÉTODO
        self._gas += n  # Recarga los litros de gas
```

- Existen tres tipos de métodos:
  - de instancia: uno por cada objeto.
  - de clase: uno para toda la clase y común a todos los objetos.
  - estáticos: es independiente de clases y objetos.

Métodos de Instancia

- Los métodos de instancia son los que están asociados a un objeto.
- Se tiene que invocar a través de un objeto (existente) de la clase.
- Se llaman con esta notación punto: objeto.método_instancia().
- Un método de instancia puede invocar a otro con self.método_instancia()
- Accede a los atributos de instancia (ver pág. 10).
- En Python se reconocen porque tiene como primer parámetro self, que apunta al objeto que invoca al método.

  Ejemplo.

```
class Coche:
    def __init__(self, gas: int):# MÉTODO DE INSTANCIA (CONSTRUCTOR)
        self._gas: int = gas  # Inicializa los litros de gas

    def descargar(self, n: int): # MÉTODO DE INSTANCIA
        self._gas -= n  # Descarga los litros de gas
```

- Los métodos `__init__(self, ...)` o `descargar(self, ...)` son ejemplos de métodos de instancia.

Métodos de Clase

- Los métodos de clase son los que están asociados a una clase.
- Se pueden invocar sin existir ningún objeto de la clase.
- Se usa notación punto: NombreDeLaClase.nombre_método_clase()
- **No pueden acceder** a las variables de instancia.
- Operan solo sobre variables de la clase (afectará a todas las instancias de los objetos), por lo que también se puede usar objeto.nombre_método_clase()
- En Python se reconocen porque tienen el decorador `@classmethod` y como primer parámetro cls que apunta a la clase cuando el método es invocado.

Ejemplo.

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

    @classmethod
    def descripcion(cls) -> str:
        return "Una rueda es un objeto circular que gira sobre un eje."

# Usa el método de clase para obtener la descripción
print(Rueda.descripcion())
```

Métodos Estáticos

- Los métodos estáticos no están asociados ni a una clase ni a objetos.
- Se pueden invocar sin existir ningún objeto de la clase.
- Se usa notación punto: NombreDeLaClase.nombre_método_estático()
- No pueden acceder ni a las variables de clase ni a las de instancia.
- Por tanto son métodos independientes para crear métodos auxiliares o de utilidad (funciones útiles para usar en cualquier momento).
- En Python se reconocen porque tienen el decorador @staticmethod.

  Ejemplo. Imagina que tienes la clase Util que contiene funciones de conversión de medidas. Son útiles porque se podrían usar en cualquier momento (sin depender de un objeto o clase en particular).

```
class Coche:
    @staticmethod
    def convertir_a_galon(litros: int) -> float:  # MÉTODO ESTÁTICO
        return litros * 0.264172  # Convierte litros a galones
```

- Si bien se podría implementar este tipo de operaciones en funciones, los **métodos estáticos** permiten que estas funcionalidades queden agrupadas y organizadas dentro de clases (en lugar de distribuidas por todo un programa) aunque no accedan a atributos de clase o instancia.

Métodos Mágicos en `Python`

- En Python existen métodos especiales llamados métodos mágicos.
- **__new__(cls , ...)** . Se invoca cuando un objeto es creado. **No usar**.
- **__init__ (self)** Debe implementarse siempre
  - Es el método inicializador de instancia que se invoca siempre, una vez construido el objeto con __new__(cls). **SOLO HAY UNO.**
  - Inicializa los atributos de la instancia (objeto) recién creada.
- **__del__ (self)** No se recomienda su uso.
  - Es un método finalizador que se invoca cuando un objeto es recogido por el recolector de basura (garbage collected, GC)
  - Un objeto será recogido (destruido) cuando deja de tener referencias hacia él (es decir, variables apuntándolo).
- **__str__ (self)**
  - Es un método de acceso que retornar un string con información del objeto entendible por el usuario. Se invoca en `print()` y `str()`.
- __lt__ (self , other), __le__ (self , other), __eq__ (self , other), __ne__ (self , other), __gt__ (self , other), __ge__ (self , other).
  - Definen los operadores $<$, $\leq$, $==$, $>$ y $\geq$ entre objetos.
    `is` identidad: True sii a y b son el mismo objeto.
    `==` igualdad: True sii __eq__ es True (por defecto, identidad).

Ejemplo de Métodos Mágicos

```python
class Coche:
    def __init__(self, marca: str, gas: int):
        self._marca: str = marca
        self._gas: int = gas

    def __str__(self) -> str:  # Método mágico __str__
        return f"Coche: {self._marca}, Gasolina: {self._gas} litros"

    def __eq__(self, otro: 'Coche') -> bool:  # Por qué las comillas?
        return self._marca == otro._marca # no recomendado

    def __gt__(self, otro: 'Coche') -> bool:  # Método mágico __gt__
        return self._gas > otro._gas  # acceso a otro._gas no recomendado

coche1: Coche = Coche("Toyota", 50)
coche2: Coche = Coche("Honda", 60)
coche3: Coche = Coche("Toyota", 50)

# Uso de __str__
print(coche1)  # Coche: Toyota, Gasolina: 50 litros

# Comparaciones con __eq__ y __gt__
print(coche1 == coche2)  # False, diferente marca
print(coche1 == coche3)  # True, misma marca
print(coche1 is coche3)  # False, diferente objeto
print(coche2 > coche1)   # True, coche2 tiene más gasolina
print(coche1 > coche3)   # False, ambos tienen la misma gasolina
```

UNIVERSIDAD
DE MURCIA

Sobrecarga de métodos

Sobrecarga

- En ocasiones nos puede interesar que un objeto pueda realizar un mismo método con parámetros diferentes.
- Equivalentemente, nos gustaría mandar el mismo mensaje pero con diferente número o tipo de argumentos.

Ejemplo.

Para sumar dos números con una calculadora no parece razonable tener distintas operaciones de suma según sus argumentos. Todo lo contrario, todos los métodos deberían llamarse igual. Lo que cambian son los argumentos.

```
int    sumar(int a, int b) { return a+b; }
double sumar(double a, double b) { return a+b; }
Fraccion sumar(Fraccion a, Fraccion b) { ... }
Complejo sumar(Complejo a, Complejo b) { ... }
```

Listing 1: Distintas sumas

Definición de Sobrecarga

- La sobrecarga permite tener un mismo identificador (nombre) para métodos con distinto tipo o número de parámetros.
- Se distinguen los distintos métodos sobrecargados por sus parámetros, ya sea por su **cantidad** o sus **tipos**.

```
class Persona {
   ...
   float distancia(Persona p) { .... }
   float distancia(Casa casa) { ... }
}
```

Listing 2: Sobrecarga de métodos

- Por tanto, dos métodos sobrecargados con el mismo número de parámetros, tipos y órdenes se considerarán iguales.
- La sobrecarga también puede aplicarse a los constructores.

Sobrecarga de constructores

Existen dos tipos de constructores:

- Constructor implícito: Se invoca por defecto si no se define un constructor. El lenguaje garantiza así la construcción de objetos (sin parámetros de inicialización).
- Constructor explícito: Se define como método en la clase, con la parametrización necesaria para dar valor a los atributos de instancia. Siempre suele definirse, sobrecargando así el constructor implícito.

En **Python**, el método __init__() es un **constructor explícito** que inicializa el objeto. Si no se definiera, **Python** invocaría a un constructor implícito (inicialmente el objeto no tendría atributos de instancia).

```
class Biblioteca:
    total_libros: int = 0  # Atributo de clase

class Libro:
    def __init__(self, titulo: str, autor: str): # constructor explícito
        self.titulo: str = titulo  # Atributo de instancia
        self.autor: str = autor    # Atributo de instancia

biblioteca: Biblioteca = Biblioteca() # se invoca constructor implícito

libro1: Libro = Libro("1984", "George Orwell") # constructor explícito
```

Sobrecarga en Python

- Realmente en Python **no existe la sobrecarga** de funciones: no puede haber varios métodos con el mismo nombre (aunque tengan parámetros diferentes).
- Al tener varias funciones con el mismo nombre, será la última función la que sobreescriba la implementación de las anteriores.

```
def f(p1: int):
    print(p1*10)

def f(p1: float, p2: float):  # Sobreescribe la función f
    print(p1*p2)

f(1) #  Error: f() tiene dos parámetros
```

- Este comportamiento se denomina sobreescritura de una función.
- En **Python**, la sobrecarga (tener una función/método con un nombre y varias posibilidades de parámetros) podemos simularla de diferentes formas.

a. Simulando la sobrecarga con parámetros opcionales

- RECUERDA: Los k-últimos parámetros de una función pueden ser opcionales.
  - Los opcionales determinan un valor literal por defecto.
  - **Primero** los obligatorios **y después** los opcionales (o por defecto).

```
def fun(a:int, b: int, c: int = 3, d: int = 4):
    pass       #  2 posicionales + 2 opcionales.
```

- Para el siguiente código solo existirá la última función, la que tiene dos parámetros y no existe la función con un parámetro.

```
def f(p1):
    print(p1*10);

def f(p1, p2):
    print(p1*p2);
```

- Podemos simular la sobrecarga de la función con este código:

```
def f(p1, p2 = None):
    print(p1*p2) if p2 else print(p1*10)

f(1)
f(10, 100)
```

b. Simulando la sobrecarga con parámetros variables

- RECUERDA: Se puede definir una función con número variable de parámetros.
- Deberá considerar un parámetro que empieza con el signo *, colocándose siempre después de los parámetros opcionales de la función.

```
def f(p: str, *args): # args se usa por convención
    print(p)
    for val in otros:
        print (f"\t{val}")

f("El primero", 2, 3, "el cuarto")
```

- Usar * significa empaquetar argumentos variables (*packing arguments*) como un solo parámetro, recibiéndose dentro como una **tupla**.
- De esta forma, podemos obtener otra aproximación a la sobrecarga de funciones con uno o varios parámetros.

```
def f(*args):
    if len(p) == 1:
        print(p[0])
    elif len(p) == 2:
        print(p[0]+p[1])

f(1)
f(10, 100)
```

c. Simulando la sobrecarga con diccionarios

- El problema de la aproximación anterior es que los parámetros no tienen nombre, teniendo que jugar con el orden de aparición.
- Sin embargo, se puede invocar a una función suministrando una lista variable de parámetros con *keywords* con el operador **, colocándose siempre después de los parámetros opcionales de la función.

```
def fun(**kwargs): # kwargs se usa por convención
    print(kwargs) # Muestra el diccionario

fun(a=1, b=2, c=3, d=4)
```

- Usar ** significa empaquetar argumentos variables con keywords como un sólo parámetro, recibiéndose dentro de la función un **diccionario**.

```
def f(**kwargs):
    if 'name' in kwargs:
        print("Nombre:" , kwargs['name'])
    if 'phone' in kwargs:
        print("Teléfono:" , kwargs['phone'])

f(name = 'Luis', phone =  '868931234')
```

- Así también se puede simular la sobrecarga, accediendo al diccionario (clave-valor) y usando condicionales de la forma más adecuada.
- Problema: La función debe conocer las claves (parámetros).

Resumen de tipos de parámetros

- Existen 4 tipos de parámetros en Python
- Posicionales obligatorios.
  - Siempre aparecerán los primeros.
- Posicionales optativos.
  - Aparecerán después de los obligatorios con valores por defecto.
- Variables sin keyword.
  - Los argumentos variables se identifican con un solo parámetro
  - Aparecerá después de los opcionales.
  - Empezará con *.
  - El nombre usual es args, tratándose como una *tupla*.
- Variables con keyword.
  - Los argumentos variables se identifican con un solo parámetro
  - Aparecerá después de los variables sin kyword.
  - Empezará con **.
  - El nombre usual es kwargs, tratándose como un *diccionario*.

Ejemplo.

```
def funcion(ob1, ob2, op1='a', op2='b', *args, **kwargs):
    pass
```

UNIVERSIDAD
DE MURCIA

Visibilidad

Acceso a los miembros de una clase

Interacción entre clases: paso de mensajes

- RECUERDA: La forma en la que deben interactuar las clases entre sí es mediante paso de mensajes:
  - Un objeto envía a otro un mensaje (solicita al otro una acción)
    - El envío de mensaje se realiza con una llamada a un método.
    - El objeto receptor es el responsable de ejecutar el método invocado.
  - El objeto receptor reaccionará (dependiendo del mensaje):
    - Cambiando el estado. Es decir, modificando los atributos.
    - Retornando información sobre su estado.
    - Realizar una rutina concreta.
    - A su vez, puede verse obligado a enviar otros mensajes. Es decir, llamando a otros métodos de otros objetos.
- A continuación, profundizamos en cómo una clase especifica el tipo de acceso a sus atributos y métodos (esto es, indicar quién puede acceder al estado para leerlo o modificarlo).

Acceso a los miembros de una clase

- No se puede permitir que un objeto externo consulte o cambie directamente los valores de un atributo:
  - Podría asignar valores sin sentido. (p.e. una edad negativa)
  - Puede destruir objetos sin control. (p.e. maria.pareja = null)
  - Podría consultar variables auxiliares o detalles de implementación (p.e., consultar una constante interna).
  - No se deben retornar datos ocultos (p.e. obtener PIN de la tarjeta).
- Sólo nos interesa aquellos métodos que la clase exponga, abstrayéndonos del estado, funcionamiento o implementación de esa clase.
  Ejemplo.
  - La llave de un coche es el mecanismo para arrancar un coche.
  - La implementación de cómo se arranca nos da igual (es privado).
  - Además, solo se puede actuar sobre el arranque con la llave.
- **RESUMEN**: No se debe acceder directamente a los atributos de un objeto ni ejecutar libremente cualquier método existente.
- Un objeto **encapsulará** su estado poniendo restricciones de **visibilidad** de sus atributos y métodos mediante **modificadores de acceso**.

Visibilidad de los miembros

- La **visibilidad de un miembro** especifica el alcance o ámbito de un atributo o método a nivel de clase, subclase o paquete (desde dónde son accesibles con la notación punto).
- La interpretación del alcance de estos modificadores **varía en función del lenguaje de programación**. En **Python**:
  - **Alcance público**. El miembro es accesible desde cualquier lugar, tanto dentro de la clase, como desde otras clases y paquetes. Se utiliza cuando quieres que un atributo o método sea accesible de forma abierta.
  - **Alcance protegido**. El miembro de la clase es accesible desde la propia clase y las clases derivadas (subclases). Se utiliza cuando se quiere permitir acceso jerárquico a los miembros en herencia.
  - **Alcance privado**. El miembro de la clase solo es accesible dentro de la misma clase. Ninguna otra clase, ni siquiera las que heredan de esta, puede acceder a este miembro. Se utiliza para encapsular datos y evitar que otros objetos accedan directamente.
- Para una buena **encapsulación** añadiremos **modificadores de acceso** (público, protegido o privado) a los atributos y métodos de la clase.

Modificadores de acceso en Python

- En realidad, Python **no define explícitamente modificadores de acceso o visibilidad**.
- Esto quiere decir que los miembros de una clase son siempre públicos. Todas las clases tienen acceso directo a los miembros de otras clases.
- Ante la visibilidad total, la privacidad se expresa de manera informativa con el nombre que le damos al atributo o método.
- Por convenio, un guión bajo antes del nombre (_identificador) indica que el atributo, método o clase debe tratarse como protegida. No obstante, cualquier otra clase sigue pudiendo acceder a ellas.
- Si se utilizase el doble guión bajo antes del nombre (__identificador) provocará que el intérprete de Python modifique el nombre del miembro de la clase (*name mangling*) a _NombreClase__identificador. Dificulta el acceso, pero _NombreClase__identificador sigue siendo también público.

Buenas prácticas en Python

- **Miembros públicos** sin anteponer guiones (var_publica).
  - Pueden ser accedidos desde otras clases del programa.
  - Los **métodos públicos** son esenciales para permitir la correcta interacción entre clases y el acceso seguro a los atributos.

```
def consultar_nombre(self): # método público
    return self.nombre      # atributo público
```

- **Miembros protegidos** con un guión bajo (_var_protegida).
  - Pueden ser accedidos por la clase y sus subclases.
  - Los **atributos protegidos** se utilizan por defecto. Los **métodos protegidos** para funcionalidad interna a nivel de herencia.

```
def mostrar_informacion(self): # método público
    return f"{self.nombre}, {self._edad} años" # Atr. público y protegido
```

- **Miembros privados** con doble guión bajo (__var_privada).
  - Realmente usado para ocultar un miembro, incluido subclases.
  - No es común en Python y muy rígido con su *name mangling*.

```
def _get_matricula(self):  # método protegido
    return {self.__matricula} # atributo privado
```

Ejemplo completo en Python

```python
class Vehiculo:
    def __init__(self, llave: bool, tipo: str, motor: bool):
        self.llave: bool = llave          # Atributo público
        self._tipo: str = tipo            # Atributo protegido
        self.__motor: bool = motor        # Atributo privado

    def arrancar(self):                   # Método público
        if self.llave and self.__verificar_motor():
            print(f"El {self._tipo} está arrancado")
        else:
            print(f"Falta llave o el motor no está listo")

    def __verificar_motor(self):          # Método privado
        return self.__motor


# Uso
vehiculo = Vehiculo(True, "coche", True)
vehiculo.arrancar()  # Acceso a método público

# Acceso a atributos
print(vehiculo.llave)  # Atributo público, accesible directamente
print(vehiculo._tipo)  # Atributo protegido, accesible pero desaconsejado

# print(vehiculo.__motor) # Error, el atributo privado no es accesible
print(vehiculo._Vehiculo__motor) # Name mangling: acceso no recomendado
```

IMPORTANTE: Cómo usar los modificadores de acceso

Métodos públicos y protegidos/privados

- Los métodos públicos describen qué pueden hacer los objetos de la clase.
  - Son los métodos con los que los objetos interactúan entre sí.
  - El conjunto de métodos públicos componen su contrato público.

```
class Cafetera: ...
    def preparar_cafe(self, cantidad: int):      # Método público
        if self._puede_preparar(cantidad):       # Método protegido
            self._calentar_agua()                # Método protegido
            print(f"Preparando {cantidad} ml de café.")
            self._set_agua(self._agua-cantidad) # Método setter protegido
            self._set_cafe(self._cafe-cantidad) # Método setter protegido
        else:
            print("No hay suficiente agua o café.")

mi_cafetera: Cafetera = Cafetera(500, 300)
mi_cafetera.preparar_cafe(200) # Interacción mediante método público
```

- Los métodos protegidos/privados describen cómo lo hacen.
  - Codifican el funcionamiento interno.

```
    def _puede_preparar(self, cantidad: int) -> bool: # Método protegido
        return self._agua >= cantidad and self._cafe >= cantidad

    def _calentar_agua(self):  # Método protegido
        print("Calentando el agua...")
```

IMPORTANTE: Cómo usar los modificadores de acceso

Métodos Getter/Setter

Todo estado (atributo) de un objeto debería de ser protegido/privado

- Diseño correcto de POO: ocultar los detalles del objeto.
- **Métodos Getter/Setter**. Aquellos definidos para permitir el acceso seguro a los **atributos** de una clase.
  - El **método de consulta** get() permite obtener el valor del atributo. Sólo existirá cuando se permita la consulta del atributo.
  - El **método modificador** set() permite establecer el valor de un atributo. Se implementará si se va a permitir la modificación. Este método controlará que la asignación al atributo es coherente, rechazando o alterando el argumento de entrada si fuera necesario.
  - Estos métodos podrían tener niveles de visibilidad diferentes (público para ser usado por todos, protegido para clase/subclases).
  - Una vez definidos, deberían ser usados no sólo por clases externas, sino también por la propia clase (pues mantienen la coherencia y realizan control de errores).
  - Un uso común es definir **métodos Getter/Setter públicos para ofrecer acceso seguro a atributos protegidos/privados**.

Métodos Getter/Setter en Python

Uso de métodos públicos get() y set() para acceder al estado protegido/privado del objeto

```python
class Persona:
    def __init__(self, dni: str, años: int) -> None:
        self._dni: str = dni    # Atributo protegido
        self._años: int = 14    # Atributo protegido

    # Método público: getter para DNI
    # No hay setter pues no se puede cambiar el DNI
    def get_dni(self) -> str:
        return self._dni

    # Método público: getter para años (todos pueden ver los años)
    def get_años(self) -> int:
        return self._años

    # Método protegido: setter para años (clase o sus subclases)
    def _set_años(self, años: int):
        if 0 <= años <= 100:  # Control de valores válidos
            self._años = años

    # Método público: incrementar los años
    def cumple_años(self):
        # Método del contrato usando el setter (y el getter)
        self._set_años(self.get_años() + 1)
```

Ejercicio

Ejercicio.

Un personaje se caracteriza por el dinero que posee.

- Se construye suministrando la cantidad de dinero inicial.
- Construye los métodos Getter/Setter sabiendo que un personaje **solo puede añadir/quitar una moneda** cada vez.
- Construye el método para saber si un personaje tiene dinero.
- Construye el método por el que otro personaje le dé una moneda de las que tiene.

  ¡Piensa bien qué métodos públicos y protegidos necesitas!

- Unified Modeling Language (UML) es un lenguaje estándar para **modelar sistemas orientados a objetos**.
- Es un **lenguaje visual**, muy usado en ingeniería del software.
- Nos permite representar:
  - Cada una de las clases como entidades independientes.
  - Listado de atributos por clase y su visibilidad.
  - Listado de métodos por clase y su visibilidad.
  - Relaciones entre clases: asociación, herencia, etc. (**Tema 3**).
  - Cardinalidad entre clases: uno a uno, uno a muchos, etc. (**Tema 3**).
  - Diferentes tipos de clases: abstractas, interfaces (**Tema 4**).
- La visibilidad de atributos y métodos se representa de la siguiente forma:
  - $+$ Visibilidad pública
  - $\#$ Visibilidad protegida
  - $-$ Visibilidad privada

Representación UML

Representación de una clase individual

- Por convención, cada clase tiene tres apartados bien diferenciados: **nombre**, listado de **atributos** y listado de **métodos**.
- Nombre:
  - Representa el nombre de la clase, en notación CamelCase.
  - El nombre de la clase de indica en negrita.
- Atributos:
  - Indican su visibilidad, nombre y tipo de datos, en este orden
  - Los atributos de clase se subrayan
- Métodos:
  - Indican su visibilidad, nombre, nombre y tipos de parámetros, y el tipo de retorno.
  - Los métodos de clase y estáticos se subrayan. Para diferenciarlos, los de clase incluyen la notación $\{\text{class}\}$ al final.
- Las clases se representan en **amarillo** para diferenciarlas de otros tipos de elementos.
- Los métodos get/set no se incluyen en los diagramas UML por simplicidad.

Representación UML

Representación de una clase individual

- Podemos añadir comentarios para destacar aspectos clave del diagrama (usar con moderación).
- Es importante documentar los parámetros de los métodos, pues de un simple vistazo podemos ver la funcionalidad ofrecida.
- Podemos representar cómo las diferentes clases o módulos se agrupan en diferentes paquetes.

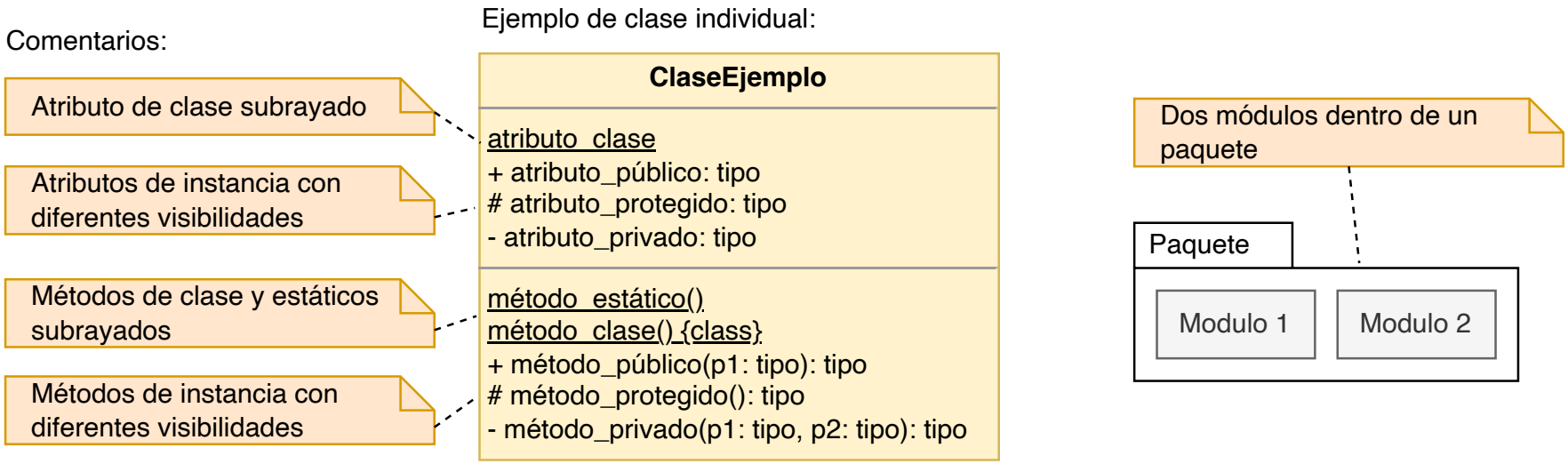

Figura: Representación UML para una clase y agrupación de clases en paquetes

UNIVERSIDAD
DE MURCIA

Visibilidad

Módulos, paquetes y espacio de nombres

Paquetes y módulos

Modularidad

- Un programa se divide en partes más pequeñas llamadas **módulos** que facilitan la reutilización y el mantenimiento (**modularidad**), pudiéndose organizar por **paquetes** de manera ordenada.
- Un **módulo** es un fichero que agrupa identificadores relacionados (variables, funciones y clases) que se pueden importar para reutilizar. La variable `__name__` contiene el nombre del módulo.

```python
# módulo: suma.py
def sumar(a: int, b: int) -> int:
    return a + b
```

```python
# módulo principal: calculadora.py
import suma
print(suma.sumar(2, 3))
```

- Un **paquete** organiza una colección de módulos (ficheros) en una jerarquía lógica de carpetas. Para ello hay que crear el fichero `__init__.py` (para nosotros estará vacío).

```
calculadora/          # paquete "calculadora"
    __init__.py       # especifica el paquete "calculadora"
    calculadora.py        # módulo (principal) "calculadora"
    operaciones/      # paquete "operaciones"
        __init__.py   # especifica el paquete "operaciones"
        suma.py           # módulo "suma"
        resta.py          # módulo "resta"
```

Espacio de nombres: namespaces

- Un **namespace** es el entorno abstracto donde quedan definidos los nombres de las variables, funciones y clases.
- Dentro de un *namespace* usamos las variables, funciones y clases sin conflictos de nombres, evitando colisiones.
- En **Python** existen tres tipos de *namespaces*:
  - **Incorporado** (*built-in*): Nombres por defecto de Python (`print()`, `len()`, excepciones, etc.) siempre disponibles.
  - **Global**: contiene los nombres definidos a nivel de un módulo.
    - Los identificadores de los módulos están aislados frente a otros módulos, pero se pueden importar para acceder a ellos.
    - La importación explícita (`import`) permite acceder a los nombres dentro de otro módulo de manera segura.
  - **Local**: nombres definidos dentro de una función o clase.

```python
# modulo_a.py
x = 5
```

```python
# modulo_b.py
x = 10
```

```python
# módulo principal
import modulo_a
import modulo_b
print(modulo_a.x)   # 5
print(modulo_b.x)   # 10
```

Visibilidad de los namespaces en Python

- Una **función** define sus identificadores en su *namespace local*. Además, la función tiene acceso al *namespace global* y *namespace incorporado*.
- Un **módulo** define sus identificadores en el *namespace global*. Puede tener acceso a identificadores de otros módulos mediante la importación (`import`). No tiene acceso a los *namespaces locales* de las funciones pero sí al *namespace incorporado*.
- El **namespace incorporado** define identificadores como `print` o `len` que pueden ser usadas en cualquier lugar del código.

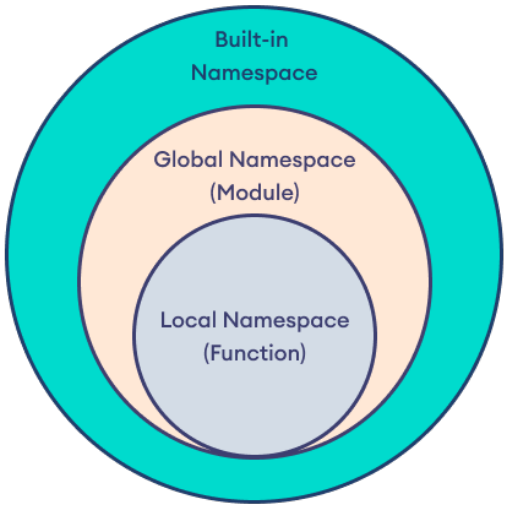

Figura: Ámbito de los *namespaces* en Python

Visibilidad de los namespaces en Python

```python
# --- NAMESPACE GLOBAL DE modulo.py ---
# Identificadores: mensaje, saludar
mensaje = "Hola desde modulo.py"

def saludar():
    return "Saludos desde modulo.py"
```

```python
import modulo # Importamos el módulo

# --- NAMESPACE GLOBAL DE main.py ---
# Identificadores: x, funcion_global, modulo

x = 10

def funcion_global():
    # --- NAMESPACE LOCAL DE funcion_global ---
    # Identificadores: y (diferente al de arriba)

    y = 5; print(f"Variable local y: {y}")  # local

    print(f"Valiable global x: {x}")        # global
    print(modulo.saludar())                 # función saludar() importada

# Llamada a la función print (NAMESPACE INCORPORADO)
print("Llamada a función incorporada")
```

Uso de módulos y paquetes en Python

Ejemplo de uso de módulos

```
import math
print("PI=", math.pi)
```

```
import math as m
print("PI=", m.pi)
```

```
from math import pi, cos
print("cos(PI)=", cos(pi))
```

Ejemplo de uso de paquetes

```
# Usa un modulo del mismo paquete
import unmodulo
# Usa un modulo de otro paquete
import Paquete.unmodulo
# Usa un modulo de un paquete interno
import Paquete.otropaquete.otromodulo

# Usa la función del mismo paquete.
unmodulo.funcA()
# Usa la función de otro paquete.
Paquete.unmodulo.funcB()
# Usa la función del modulo del paquete interno
Paquete.otropaquete.otromodulo.funcC()
```

```
# Usa una función del mismo paquete
from unmodulo import funcA
# Usa una función del modulo de otro paquete
from Paquete.unmodulo import funcB
# Usa una función del modulo de un paquete interno
from Paquete.otropaquete.otromodulo import funcC

funcA()
funcB()
funcC()
```

Modularidad

- Un módulo deber ser Cohesivo.
  - El propósito debe estar bien definido.
  - La cohesión hace más fácil:
    - Entender qué hace una clase y sus métodos.
    - Usar nombres más descriptivos.
    - Reutilizar mejor las clases y métodos.
- Un módulo deber ser Poco Acoplado.
  - El acoplamiento indica la dependencia entre clases.
  - Lo ideal es "independencia" para que una clase conozca lo mínimo esencial de otra clase (lo que indica cohesión).
  - Acoplamiento fuerte implica que las clases relacionadas tienen que conocer detalles unas de otras.
  - El poco acoplamiento hace más fácil:
    - Entender una clase sin leer otras.
    - Cambiar una clase sin afectar a otras.
    - El mantenimiento: se detectan antes los errores.

Principios de la POO

¿Los recuerdas?

- Abstracción[1]. Proceso mental de extracción de las características esenciales de un concepto o proceso descartando los detalles.
- Encapsulación[2]. Proceso por el que se ocultan los detalles de un objeto.
- Jerarquización[3]. Estructurar por niveles (jerarquía) los elementos que intervienen en el proceso.
  - Jerarquía de clasificación (**Herencia**).
  - Jerarquía de composición (**Asociación**).
- Modularidad[4]. Descomposición del sistema en conjunto de módulos poco acoplados (independientes) y cohesivos (con significado propio).

[1] Se trató en el tema anterior con clases y objetos
[2] Se ha estudiado en este tema con la visibilidad de clases
[3] Lo trataremos en el tema siguiente
[4] Se ha estudiado en este tema con la visibilidad de módulos