

Miembros, Sobrecarga y Visibilidad

Tipos de métodos y variables: Encapsulamiento

L. Daniel Hernández < *ldaniel@um.es* >

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
2 de octubre de 2023



¹ Imagen: Hombre feliz escondiéndose detrás de la pizarra. https://www.freepik.es/fotos-premium/hombre-feliz-escondiendose-detras-pizarra_1635302.htm

Índice de Contenidos

- 1 Introducción
- 2 Métodos y Variables
 - Métodos
 - Variables
- 3 Sobrecarga
 - Python no admite la sobrecarga
- 4 Visibilidad
 - Espacio de nombres: Modularidad
 - Modificadores de acceso
 - Representación UML
- 5 Resumen



1 Introducción

2 Métodos y Variables

Métodos
Variables

3 Sobrecarga

Python no admite la sobrecarga

4 Visibilidad

Espacio de nombres: Modularidad
Modificadores de acceso
Representación UML

5 Resumen



Introducción

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

```
1 class UnaClase {  
2     miembros  
3 }
```

- Los **miembros de una clase** son los elementos de una estructura que la caracterizan y, por ende, permite definir a sus objetos.
- Ya conocemos algunos de sus miembros:
 - Las **variables**, que definirán el estado de cada objeto.
 - Los **métodos**, que establecen el comportamiento (o funcionalidad) de todos los objetos.
- Pueden **existir otros**: constantes, propiedades, eventos, clases internas, etc.
- En cada lenguaje de programación los miembros pueden variar.
- Formalmente, los **constructores**, que indican cómo debe establecer el estado inicial de un objeto recién creado, no se consideran miembros de una clase (Realmente hay discusión sobre el tema, que si debe heredarse, que si debe ser estático, ...)
- Para todos ellos no se puede permitir que cualquier pueda acceder a cualquier miembro de una clase. Hay que imponer restricciones de **visibilidad**.

1 Introducción

2 Métodos y Variables

Métodos

Variables

3 Sobrecarga

Python no admite la sobrecarga

4 Visibilidad

Espacio de nombres: Modularidad

Modificadores de acceso

Representación UML

5 Resumen



Profundizamos en ...

1 Introducción

2 Métodos y Variables

Métodos

Variables

3 Sobrecarga

Python no admite la sobrecarga

4 Visibilidad

Espacio de nombres: Modularidad

Modificadores de acceso

Representación UML

5 Resumen



Métodos Miembros

- Hemos indicado que 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: Pseudocódigo

```
1  class Coche {  
2      ...  
3      None recarga (int n) {  
4          self.gas ← n;    // Estable los litros de gas  
5      }  
6      ...  
7  }
```

- Existen tres tipos de métodos:
 - **de instancia**: uno para todos los objetos.
 - **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.nombreMétodoInstancia()`
- **Accede a las variables de instancia** (ver pág. 15).
- En **Python** se reconocen porque tiene como primer parámetro **self** que apunta al objeto que invoca al método.

No se recomienda este uso generalizado

Ejemplo:

Los métodos mágicos `__init__(self, parámetros)` o `__str__(self)` son ejemplos de métodos de instancia.

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


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.nombreMétodoClase()`
- No pueden acceder a las variables de instancia (ver pág. 15).
- **Operan solo sobre variables de la clase** (afectará a todas las instancias de los objetos), por lo que también se puede usar `objeto.nombreMétodoClase()`
- En **Python** se reconocen porque tienen el decorador `@classmethod` y tiene como primer parámetro `cls` que apunta a la clase cuando el método es invocado.
 - Un claro uso de los métodos de clase es usarlos como **Funciones Factoría**, que son las que permiten obtener instancias de la clase.

Ejemplo:

```
>>> class Rueda:
...     def __init__(self, radio):
...         self.radio = radio
...     @classmethod
...     def grande(cls):
...         return cls(500)
... 
```

```
>>> # Fabrica una rueda
>>> mi_rueda = Rueda.grande()
>>> # Muestra el radio
>>> print(mi_rueda.radio)
500
```

Métodos Estáticos

- Los **métodos estáticos** son los que no están asociados ni a una clase ni a un objeto.
- Se pueden invocar sin existir ningún objeto de la clase.
- Se usa notación punto: `NombreDeLaClase.nombreMétodoEstático()`
- **No pueden acceder ni a las variables de clase ni a las de instancia** (ver pág. 15).
- Por tanto son métodos independientes de las clases/objetos y útiles para crear métodos “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). Serían adecuadas definir las como métodos estáticos `Util.celsius_a_fahrenheit()`, ...

```
>>> class Rueda:
...     @staticmethod
...     def superficieAproximada(radio):
...         return 3.14*radio**2
...
>>> Rueda.superficieAproximada(8) # Me valdría para cualquier círculo
200.96
```

Algunas aclaraciones

- Hemos distinguido 3 tipos de métodos:
 - Los **métodos de instancias** son los que están asociados a un objeto.
 - Los **métodos de clase** son los que están asociados a una clase.
 - Los **métodos estáticos** son los que **no** están asociados ni a instancia ni a clases
- Hay lenguajes que distinguen los 3 tipos de métodos. Como en **Python**.
 - Los **métodos de instancias** tienen como primer parámetro **self**.
 - Los **métodos de clase** usa **@classmethod** y el primer parámetro es **cls**.
 - Los **métodos estáticos** usa **@staticmethod**.
- Hay lenguajes que distinguen solo 2 grupos de métodos. Como en **Java**.
 - Los **métodos de instancias** son los que **no tienen** el modificador **static**.
 - Los métodos que sí tienen el modificador **static** y se pueden interpretar como **métodos de clase** o **métodos estáticos**. De hecho son términos sinónimos en este lenguaje.

```
1 public class Car {  
2     public double consumo(double metros) { } // De instancia  
3     public static void main(String[] args) { } // Estático. De Clase  
4 }
```

Ejercicio.

Indica cómo deberían ser los siguientes métodos (si estáticos, de clase o de instancia)

- Aplicar una función matemática sobre un número. P.e. `abs()`, `log()`, `sqrt()`, ...
- Retornar el número de objetos que se han construido de una clase.
- Calcular la suma de dos complejos.

Profundizamos en ...

1 Introducción

2 Métodos y Variables

Métodos

Variables

3 Sobrecarga

Python no admite la sobrecarga

4 Visibilidad

Espacio de nombres: Modularidad

Modificadores de acceso

Representación UML

5 Resumen



Variables Miembros

- Son **variables miembros** las que definen el estado de la **clases** o sus **objetos**.
- También se llaman **atributos** o **campos**.
- Pueden ser **simples** o **compuestas** (p.e. arrays u otros objetos)

Ejemplo: **Pseudocódigo**

```
class Persona{
    final String nombrePadre # Estructurado y No cambia (final, constante)
    int edad # Tipo de Dato simple
    ...
}
class Primitiva {
    int[] numeros # Estructurado: Array de enteros (TD simple)
    ...
}
class Persona {
    Persona pareja # Estructurado: Objeto (En POO es lo usual)
    ...
}
class Biblioteca {
    Libro[] libros # Estructurado: Array de objetos (Normal en POO)
    ...
}
```

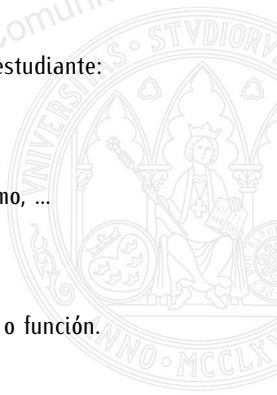
- No hay que confundir las **variables miembros** con otras variables.

Tipos de Variables

- Existen 4 tipos de variables: de clase, de instancia, locales y globales

Ejemplo: Sobre la clase **Estudiante** se pueden definir estas variables:

- **Variables de clase.** Evalúan atributos de toda la clase.
 - Número de estudiantes total (varía con el tiempo)
 - Sistema de calificaciones (constante en el tiempo)
 - ...
- **Variables de instancia** (u objeto). Definen el estado para un estudiante:
 - Color de los ojos
 - Referencia al centro en el que estudia
 - ...
- **Variables locales.** Las auxiliares propias de un método, algoritmo, ...
 - Las auxiliares para calcular la nota media de un estudiante
 - Las usadas para ordenar los estudiantes por altura
 - ...
- **Variables globales.** Las que son accesibles por cualquier clase o función.
 - El planeta donde viven los estudiantes.
 - El aire que respiran los estudiantes
 - ...



Variables Locales y Globales

- 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 variables 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 **variables globales** para esa función.
 - Los parámetros de una función/método son variable locales a la función/método
 - Las variables declaradas en el bloque de una función son variables locales a la función.
 - **En un bloque todas las variables son locales salvo que se diga lo contrario**

```
>>> s = "global"           # Variable global (está fuera de cualquier función)
>>> def f(param):          # Un parámetro es una variable local
...     #global s          # Descomenta !!
...     s = "local"        # NO ES REASIGNACIÓN!! - Nueva variable !
...     print(s, id(s))
...
>>> f(2)                   # En la invocación s será "local"
local 4299840688
>>> print(s, id(s))        # s vale "global"
global 4300326384
```


Variables de Instancia u Objeto

- **Son variables miembro**
- Definen el **estado** de un objeto.
- Hay, al menos, tantas como objetos o instancias se hayan creado.
- Se destruyen cuando se destruye el objeto.
- Se usa la notación punto: **nombreObjeto.nombreVariable**

Solo son accesibles a través de un método del objeto

- Se usan en métodos de instancia (pág 8).
- No se pueden usar ni en métodos estáticos ni en métodos de clase (págs 9, 10)
Recíprocamente, estos métodos no te dejará trabajar con este tipo de variables.
- En **Python** se crea una cada vez que se realiza una asignación

```
self.valor = valor # P.e. en el método __init__()  
objeto.valor = valor # En cualquier lugar
```

Ya que **objeto.valor** se puede realizar en cualquier lugar, **se debe usar** `__slots__ = [variables]` en la definición de la clase para indicar los identificadores permitidos para las variables de instancia.

Variables de Clase o Estáticas

- **Son variables miembro**
- Hacen referencia a **información de la clase** (en su conjunto)
Por tanto, definen atributos comunes (no propios) de los objetos.
- Son las que **se ubican estáticamente** (e.d. la memoria se ha reservado en tiempo de compilación) por lo que tienen que estar declaradas en la clase fuera de cualquier método.
- Existen en el momento de cargarse la clase (antes de cualquier objeto)
- **No se destruyen** durante la ejecución del programa.
- Solo existe **una por clase**, independientemente del número de objetos que existan.
- Usa la notación punto: **NombreDeLaClase.nombreVariable**
Son accesibles a través de una clase
- Son **compartidas por todos los objetos** de la clase, por lo que también se puede usar el nombre de un objeto en vez del nombre de la clase, pero NO se recomienda este uso.
- En **Python** se pueden usar en métodos estáticos, de clase (a través de **cls**) y de instancia (a través del nombre de la clase)
Notar que **NombreClase.var = valor** no define una variable de clase si **var** no estaba declarada, sino una variable de instancia del objeto que referencia a la clase.

Ejemplo de cómo se usan los distintos tipos de Variables

```
>>> planeta = "La Tierra"                                # Var. Global
>>> class Estudiante:
...     num_estudiantes = 0                                # Var. Clase
...     def __init__(self, calificaciones):
...         self.calificaciones = calificaciones          # Var. Instancia
...         Estudiante.num_estudiantes += 1               # Var. Clase
...         print(f'Este vive en {planeta}')               # Var. Global
...     def calificacion_media(self):
...         sum = 0                                         # Var. Local
...         for i in range(0, len(self.calificaciones)):
...             sum += self.calificaciones[i]              # Var. local y objeto
...         return sum/len(self.calificaciones)
...
>>> est = Estudiante ([5,10])
Este vive en La Tierra
>>> Estudiante.num_estudiantes    # Variable de clase
1
>>> est.num_estudiantes           # Este uso con objeto no se recomienda
1
>>> est.calificaciones            # Variable de objeto
[5, 10]
```

Algunas aclaraciones

- Hemos distinguido 4 tipos de variables.
- **Python** permite trabajar con globales, locales, de instancia y de clase.
 - En las **variables de instancia** se antepone la palabra **self**.
 - En las **variables de clase** se antepone la palabra **cls** o el nombre de la clase.
- **Java** solo trabaja con locales, de instancia y de clase.
 - Es decir, no se pueden definir variables fuera de una clase.
 - Los **atributos de instancias** son los que **no tienen** el modificador **static**.
 - Los atributo que sí tienen el modificador **static** son **atributos de clase o estáticos**.
 - Una vez declaradas no se requiere anteponer nada para su uso.

```
1 public class Car {  
2     public static int numCoches; // atributo estático  
3     public int numKilometros;    // atributo de instancia  
5 }
```

Ejercicio.

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

1 Introducción

2 Métodos y Variables

- Métodos
- Variables

3 Sobrecarga

- Python no admite la sobrecarga

4 Visibilidad

- Espacio de nombres: Modularidad
- Modificadores de acceso
- Representación UML

5 Resumen



- En ocasiones nos puede interesar que un objeto pueda realizar métodos con parámetros diferentes.
- Equivalentemente, nos gustaría mandar el mismo mensaje pero con argumentos diferentes.

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.

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

Definición de Sobrecarga

- La sobrecarga permite usar **un mismo identificador** para representar distintos **métodos** con distinto tipo y número de parámetros, todos dentro de la misma clase.
- Se distinguen** los distintos métodos sobrecargados **por sus parámetros**, ya sea por su cantidad, los tipos o su órdenes.

```
1 class Persona {  
2     ...  
3     float distancia(Persona p) { .... }  
4     float distancia(Casa casa) { ... }  
5 }
```

- Por tanto, dos métodos sobrecargados con el mismo número de parámetros, tipos y órdenes se considerarán iguales.
- Java** diferencia los métodos sobrecargados con base en el número y tipo de parámetros o argumentos que tiene el método y **no por el tipo** que devuelve.
- La sobrecarga también puede aplicarse a los **constructores**.
- No hay que confundir sobrecarga con sobreescritura (se verá en herencia).

Sobrecarga de constructores

- Algunos lenguajes de POO usan **this** para referirse al **objeto actual**.
- Para la sobrecarga de constructores suele usarse el constructor **this()** para **llamar a otro constructor de la misma clase**.
- Se distinguen dos tipos de constructores:
 - **Constructor implícito**: aquel que al ser llamado asigna un estado inicial por defecto a la instancia de la clase. No tiene parámetros.
 - **Constructor explícito**: aquel en el que se requiere indicar de forma explícita el valor de un atributo para instanciar la clase. Tiene, al menos, un parámetro.
- Se puede usar **this()** en un constructor para llamar a un constructor “más explícito”.

```
1 // El constructor más explícito. Contiene todos los atributos.
2 private Car(boolean lights, String color) {
3     this.lights ← lights;
4     this.color ← color;
5 }

7 // Constructor menos explícito. Contiene un atributo.
8 public Car(boolean lights) { this (lights, "white"); }

10 // Constructor implícito. Estado por defecto.
11 public Car() { this(false, "red"); }
```

Profundizamos en ...

1 Introducción

2 Métodos y Variables

- Métodos
- Variables

3 Sobrecarga

- Python no admite la sobrecarga

4 Visibilidad

- Espacio de nombres: Modularidad
- Modificadores de acceso
- Representación UML

5 Resumen



Sobrecarga en Python = Redefinición

- En Python **no existe la sobrecarga** de funciones y será la última función la que **sobreescriba** la implementación de los anteriores.
- La **sobreescritura** de una función es redefinir la función.

```
>>> def f(p1):  
...     print(p1*10)  
...  
>>> f(1)  
10  
  
>>> def f(p1, p2):  
...     print(p1*p2)  
...  
>>> f(1) # Ya NO existe la función con un parámetro!!  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: f() missing 1 required positional argument: 'p2'
```

- Lo que veremos en este apartado sobre funciones también es aplicable a métodos/constructores en Python.

Parámetros Opcionales (valores por defecto)

- Dada una función con n -parámetros, se puede considerar que los k -últimos pueden ser opcionales.
- Los opcionales se determinan estableciendo un **valor literal por defecto**.

```
>>> def f(p0, p1=10, p2=100):  
...     print(p0, p1, p2)  
...  
>>> f(1)                                # Con parámetro obligatorio  
1 10 100  
>>> f(1, 50)                            # Con obligatorio y 1er opcional  
1 50 100  
>>> f(1, p2=-1)                         # Con obligatorio y 2o opcional  
1 10 -1  
>>> f(1, -10, -200)                    # Con obligatorio y opcionales  
1 -10 -200
```

Simulando la Sobrecarga con Parámetros Opcionales

- En Python no existe la sobrecarga, pero podemos conseguir un comportamiento similar usando valores por defecto
- 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, p3 = None):  
...     print(p1*p2) if p2 else print(p1*10)  
...  
>>> f(1)  
10  
>>> f(10, 100)  
1000
```

Parámetros Variables

- Se pueden definir funciones con un número variable de parámetros.
- Deberá considerar un parámetro que empezará con el signo `*` y se deberá colocar siempre **después** de los parámetros opcionales de la función.

```
>>> def f(p, *otros):  
...     print(p)  
...     for val in otros:  
...         print (f"\t{val}")  
...  
>>> f("El primero", 2, 3, "el cuarto")  
El primero  
    2  
    3  
    el cuarto
```

- Usar `*` significa **empaquetar argumentos** (packing arguments): todos los argumentos se empaquetan y se pasan como un solo parámetro

Simulando la Sobrecarga con Parámetros Variables

- Con el operador `*` también podemos obtener una aproximación a la sobrecarga de funciones con uno o varios parámetros.

```
>>> def f(*p):  
...     if len(p) == 1:  
...         print(p[0])  
...     elif len(p) == 2:  
...         print(p[0]+p[1])  
...  
>>> f(1)  
1  
>>> f(10, 100)  
110
```

- El problema de esta aproximación es que los parámetros no tienen nombre, pues solo importa el orden de aparición.
 - ¿No se podría definir una función con una lista variables de parámetros pero que tengan nombre?
- La respuesta es sí y se muestra en la siguiente diapositiva.

Funciones con diccionarios

- Se puede invocar a una función suministrando una lista variable de parámetros y que cada uno tenga su propia *keyword*.

```
>>> def fun(**kwargs):  
...     print(kwargs) # Muestra el diccionario  
...  
>>> fun(a=1, b=2, c=3, d=4)  
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

- En la invocación, Python construirá un **diccionario** de todos los argumentos de palabras clave y lo pondrá a disposición en el cuerpo de la función.
- El nombre `kwargs` se usan por convención, no forman parte de la especificación del lenguaje.
- `**kwargs` deben estar en último lugar en la lista de parámetros.

Simulando la Sobrecarga con Diccionarios

- Se puede acceder a elementos individuales de `kwargs` como en un diccionario normal.

```
>>> def f(**kwargs):  
...     for key in kwargs:  
...         print(f"key = {key}, valor = {kwargs[key]}")  
...  
>>> f(a = 1, b = "dos", c = 3.0)  
key = a, valor = 1  
key = b, valor = dos  
key = c, valor = 3.0
```

- Se puede entonces usar condicionales para actuar de la forma más adecuada en función de las claves suministradas.

```
>>> def f(**kwargs):  
...     if 'name' in kwargs:  
...         print("Nombre:" , kwargs['name'])  
...     if 'phone' in kwargs:  
...         print("Teléfono:" , kwargs['phone'])  
...  
>>> f(name = 'Luis', phone = '868931234')  
Nombre: Luis  
Teléfono: 868931234
```

Resumen

- Existen 4 tipos de parámetros en Python
- Posicionales **obligatorios**.
 - Siempre aparecerán los primeros.
- Posicionales **opcionales**.
 - Aparecerán después de los obligatorios asignándoles 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**.
- **Variables con keyword**.
 - Los argumentos variables se identifican con un solo parámetro
 - Aparecerá después de los variables sin keyword.
 - Empezará con ******.
 - El nombre usual es **kwargs**.

Ejemplo:

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

1 Introducción

2 Métodos y Variables

- Métodos
- Variables

3 Sobrecarga

- Python no admite la sobrecarga

4 Visibilidad

- Espacio de nombres: Modularidad
- Modificadores de acceso
- Representación UML

5 Resumen

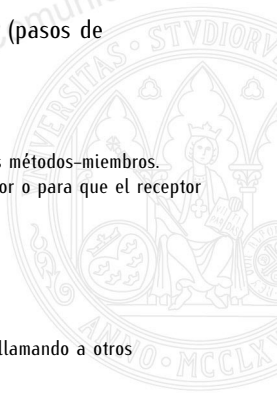


Paso de Mensajes

- Desde el punto de vista de la POO: (Visita <http://wiki.c2.com/?MessagePassing>)

Un mensaje es una llamada a un método en un objeto

- Un paso de mensaje en POO es “lo equivalente” a la invocación de una función en programación modular.
- Un programa en POO consta de una secuencia de invocaciones (pasos de mensajes) hasta resolver el problema.
- El **esquema básico del paso de mensajes** es:
 - Un objeto **envía** a otro **un mensaje** (solicita al otro una acción)
 - El envío de mensaje se realiza haciendo una llamada a uno de los métodos-miembros.
 - La acción puede ser para retornar/modificar un atributo del receptor o para que el receptor realice una rutina concreta.
 - 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 miembros del mismo objeto o de otros objetos.



¿Por qué ocultar miembros?

- Hay ocasiones en las que un objeto debe poner **restricciones** al acceso de sus atributos.

Ejemplo: No se puede permitir que un mensaje cambie directamente los valores de un atributo o que el mensaje solicite información “confidencial”

- Podría asignar valores a un atributo sin sentido. (p.e. una edad negativa)
 - Puede destruir objetos sin control. (p.e. maria.pareja = null)
 - No se deben retornar datos ocultos. (p.e. obtener pin de la tarjeta)
 - etc ...
- Un TDA tiene una interface pública (operaciones) y una interface privada. No tiene sentido acceder a la parte privada. Solo nos interesa **usar métodos de otra clase sin importarnos su funcionamiento.**

Ejemplo:

- La llave de contacto de un coche es el mecanismo que usamos para arrancar un coche. La implementación de cómo se arranca nos da igual (realmente es privado). Además, solo se puede actuar sobre el arranque con la llave de contacto.
- Resumen: Hay motivos para no acceder a ciertos miembros de una clase.
- Situaciones como las indicadas se resuelven con los **modificadores de visualización.**
- Este tema va del **encapsulamiento**, pero para ello hemos de tratar ante el problema de la **modularidad.**

Profundizamos en ...

1 Introducción

2 Métodos y Variables

- Métodos
- Variables

3 Sobrecarga

- Python no admite la sobrecarga

4 Visibilidad

- Espacio de nombres: Modularidad**
- Modificadores de acceso
- Representación UML

5 Resumen

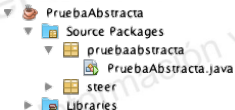


Espacio de Nombres

- Necesitamos entender qué es un **espacio de nombre**, **módulo** y un **paquete**.
- Un **namespace** es un **contenedor abstracto** por el que se agrupan a un conjunto de identificadores únicos.
 - Permite mantener un conjunto de nombres separado de otro
 - Evita conflictos de nombres
 - En dos *namespaces* se pueden tener declaradas clases con los mismo nombres.
 - Todos los identificadores que se definen en un programa son añadidos a un espacio de nombres.
- Un **módulo** es un **fichero** que consta de un conjunto de identificadores.
- Un **package** consta de una **colección de ficheros** (módulos) con nombres de declaraciones que pueden abarcar varios archivos.
 - Sirve para agrupar identificadores relacionados (p.e. clases, funciones, ...)
 - **Define un namespace** con los identificadores que contiene.
 - Así, p.e., dos paquetes pueden contener clases con los mismos nombres.
- Cada lenguaje usa los **namespaces** y los **paquetes** a conveniencia.
- Un programa se puede dividir en namespaces/paquetes.

Espacio de Nombres en Java, C# y C++

- En **Java**, los **paquetes** sirven para agrupar clases relacionadas y definen un espacio de nombres para las clases que contienen.
 - Cada paquete contiene varias clases (realmente es un directorio)
 - En general, cada clase se define en un fichero **.java**
 - Cada clase del paquete **namespace** empieza por **package namespace**
 - Las clases de otro paquete que quieran usar la clase **miclase** del paquete **namespace** deben indicarlo con **import namespace.miclase**.



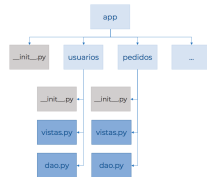
- En **C#** y **C++**,
 - Las clases del paquete **namespace** se ponen en una región declarativa

```
1 namespace NombreDelEspacio {  
2     declaración de las clases  
3 }
```

- A diferencia de Java, no se requiere de una estructura de directorios para los módulos y un fichero puede contener múltiples namespaces.
- Las clases que quieran usar la clase **miclase** del paquete **namespace** deben indicarlo con **using namespace.miclase** y **using namespace::miclase**.

Espacio de Nombres en Python - I

- En **Python** se distinguen tres espacios de nombres.
- **Espacio de nombres incorporado** (o *built-in namespace*)
 - En él se encuentran todos los identificadores que incorpora el lenguaje
 - Están disponibles en todo momento cuando se ejecuta Python
 - Se pueden mostrar con `dir(__builtins__)`
- **Espacio de nombres global**
 - Se construye uno por cada **módulo** (archivo .py) (principal e importados)
 - En diferentes módulos se pueden usar los mismos nombres y estos no interfieren entre sí.
 - Dentro de un módulo, se puede acceder al nombre del mismo a través de la variable global `__name__`.
Uso: `if __name__ == '__main__': \ main()`
 - `dir(modulo)` muestra los elementos definidos en un módulo.
- **Espacio de nombres local**
 - Se construye cada vez que se invoca a una función.
 - El espacio local se asocia a una función y contiene a todos los nombres definidos en ella.
- Un **paquete** en **Python** es un directorio formado por varios módulos (.py) y un fichero `__init__.py` que la mayoría de las veces está vacío.



Fuente: j2logo.com/python/tutorial

Ejemplo de uso de módulos

```
>>> import math
>>> print("PI=", math.pi)
PI= 3.141592653589793
```

```
>>> import math as m
>>> print("PI=", m.pi)
PI= 3.141592653589793
```

```
>>> from math import pi, cos
>>> print("cos(PI)=", cos(pi))
cos(PI)= -1.0
```

Ejemplo de uso de paquetes

```
# Usa un módulo del paquete
import Paquete.unmodulo
# Usa un módulo de un paquete interno
import Paquete.otropaquete.otromodulo

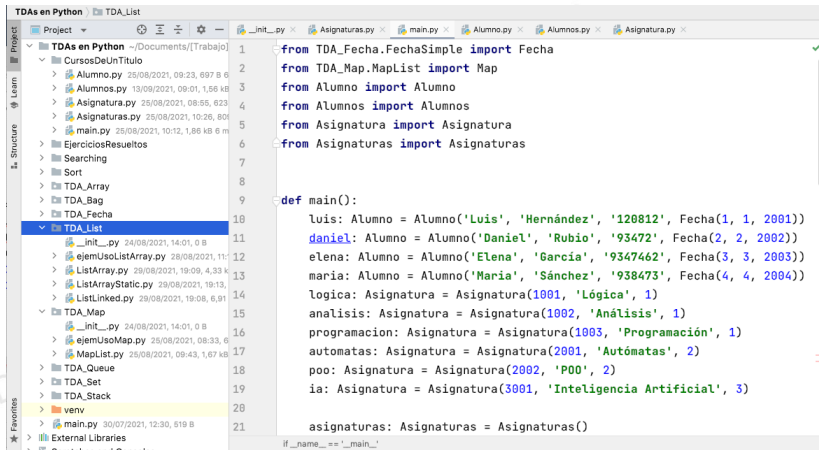
# Usa la función del módulo del paquete.
Paquete.unmodulo.funcA()
# Usa la función del módulo del paquete interno
Paquete.otropaquete.otromodulo.funcB()
```

```
# Usa una función del módulo de un paquete
from Paquete.unmodulo import funcA
# Usa una función del módulo de un paquete interno.
from Paquete.otropaquete.otromodulo import funcB

funcA()
funcB()
```

Espacio de Nombres en Python - III

Ejemplo con PyCharm



```
1 from TDA_Fecha.FechaSimple import Fecha
2 from TDA_Map.MapList import Map
3 from Alumno import Alumno
4 from Alumnos import Alumnos
5 from Asignatura import Asignatura
6 from Asignaturas import Asignaturas
7
8
9 def main():
10     luis: Alumno = Alumno('Luis', 'Hernández', '120812', Fecha(1, 1, 2001))
11     daniel: Alumno = Alumno('Daniel', 'Rubio', '93472', Fecha(2, 2, 2002))
12     elena: Alumno = Alumno('Elena', 'García', '9347462', Fecha(3, 3, 2003))
13     maria: Alumno = Alumno('Maria', 'Sánchez', '938473', Fecha(4, 4, 2004))
14     logica: Asignatura = Asignatura(1001, 'Lógica', 1)
15     analisis: Asignatura = Asignatura(1002, 'Análisis', 1)
16     programacion: Asignatura = Asignatura(1003, 'Programación', 1)
17     automatas: Asignatura = Asignatura(2001, 'Autómatas', 2)
18     poo: Asignatura = Asignatura(2002, 'POO', 2)
19     ia: Asignatura = Asignatura(3001, 'Inteligencia Artificial', 3)
20
21     asignaturas: Asignaturas = Asignaturas()
22
23 if __name__ == '__main__':
```

Modularidad

Fuente: <https://preparandoscjp.wordpress.com/2011/06/16/orientacion-a-objetos-viii-acoplamiento-y-cohesion/>

- En la pág. 39 se definió qué era un **módulo** y un **paquete/spacename**.
- Un módulo deber ser **Cohesivo**.
 - La **cohesión** es una medida para indicar que la clase tiene un propósito bien definido.
 - Cuanto más claro esté el propósito de la clase, mayor será la cohesión.
 - 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** es una medida que indica la interconexión/dependencia entre clases.
 - Acoplamiento fuerte implica que las clases relacionadas tienen que conocer detalles unas de otras.
 - Lo ideal es “independencia”: que una clase conozca lo mínimo esencial de otra clase.
 - 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.
 - El acoplamiento está muy relacionado con la jerarquización.

Profundizamos en ...

1 Introducción

2 Métodos y Variables

- Métodos
- Variables

3 Sobrecarga

- Python no admite la sobrecarga

4 Visibilidad

- Espacio de nombres: Modularidad
- Modificadores de acceso**
- Representación UML

5 Resumen



Modificadores de Acceso

- Un espacio de nombres determina el ámbito de los identificadores
- Pero para una buena **encapsulación** necesitaremos un proceso por el que se oculten los detalles del soporte de las características de una abstracción.
- El proceso consiste en **añadir modificadores** a los miembros de la clase.
- **Modificador privado**
 - Se indican con la palabra **private** antes del atributo, método o constructor.
 - El identificador será accesible solo por el módulo donde se declare.

Ejemplo: Si **var** es privada en **Clase** no se permitirá **Clase.var**

- **Modificador público**
 - Se suele indicar con la palabra **public** antes del atributo, método o constructor.
 - El identificador será accesible por todas los demás namespaces.

Ejemplo: Si **var** es pública en **Clase** sí se permitirá **Clase.var** desde **cualquier otra** clase de cualquier namespace.

- **Modificador interno**
 - Se indican con la palabra **internal** antes del atributo, método o constructor.
 - Alternativamente no se indica **ningún modificador**
 - El identificador será accesible tanto en el módulo donde se declare como en su namespace.

Ejemplo: Si **var** es interna en **Clase** solo se permitirá **Clase.var** desde **las clases del mismo namespace**.

Modificadores de Acceso en Python

- En **Python** no existen los modificadores de acceso o visibilidad.
- Esto quiere decir que para cualquier módulo importado **siempre se podrá acceder a cualquier identificado del módulo**.
- Ante la visibilidad total, **“la privacidad” se expresa en el nombre** del identificador.
- Es un convenio: usar **un guión bajo**, `_`, antes de un nombre indica que la variable, función, método o clase debe tratarse como “privada”.
 - Cualquier otro programador reconoce cuáles son las componentes “internas” del código aunque no lo sean.
- Si se utiliza el **dobles guión bajo** antes del nombre provocará que el intérprete modifique el nombre del miembro de la clase.
 - Cualquier nombre de la forma `__spam` se renombrará a `_NombreClase__spam`.
 - Esta forma evita conflicto con nombres definidos por otras subclases (ocultación).
 - No obstante `_NombreClase__spam` sigue siendo público.
- La recomendación estándar es usar un solo guión bajo para privadas e internas.
- Si no quieres liarte una recomendación es:
 - anteponer doble guión bajo para el modificador privado: `__var_privada`
 - anteponer un guión bajo para el modificador interno: `_var_interna`
 - no anteponer guiones para el modificador público: `var_publica`

Cómo usar los Modificadores de Acceso

- Los **métodos públicos** describen **qué** pueden hacer los objetos de esa clase.
 - Son los métodos que le interesa conocer a otro objeto.
Ejemplo: Nos interesa saber qué hay que hacer para encender un coche, para sacar dinero de un cajero, ...
- Los **métodos privados** describen **cómo** lo hacen.
 - Son los métodos de funcionamiento interno que no le interesa a otro objeto.
Ejemplo: Nos da igual cómo será el encendido del coche mientras arranque, también nos da igual lo que haga el cajero mientras nos dé el dinero solicitado, ...
- **Todo estado de un objeto tienen que ser privado**
 - Es el diseño correcto de POO
 - Sólo se puede cambiar el estado a través de una **interface pública** (métodos públicos).
 - La interface pública relacionada con el estado se llama Getter/Setter
 - **Todo atributo tendrá asociado un método get() y otro set()** (si tienen sentido)
 - Un método **get()** o **método de acceso** permite obtener el valor del atributo.
 - Pero no siempre tiene que existir el get de un atributo. P.e. No para el PIN del móvil
 - Un método **set()** o **método mutador** permite establecer el valor de un atributo.
 - El método set controlará que la asignación al atributo es coherente, con lo que se rechazará o alterará el argumento de entrada si fuera necesario.
 - En Python. La interface Getter/Setter **tiene que ser usada** por el resto de los métodos, aún cuando tengan acceso a los atributos.

Ejemplo en Pseudolenguaje

```
1 class Persona {
2     // Todo el estado de un objeto debe de ser privado
3     private final String DNI; // part-of
4     private int años;

6     // Constructor
7     public Persona(String dni) { this.dni ← dni; años ← 14; }

9     // get de DNI. Es Público. NO hay set: NO tiene sentido.
10    public String getDni() { return this.dni; }

12    // get de Años. Es Público. Todos pueden ver los años.
13    public int getAños() { return this.años; }

15    // set de Años. Es 'amigo'.
16    // Solo desde el paquete se puede cambiar los años.
17    void setAños(int años) {
18        if (this.años ≤ años && años ≤ 100) this.años ← años;
19    }

21    // Método público. Usa la interface Getter/Setter
22    public void cumpleAños() { this.setAños ( años+1 ); }
23 }
```

Ejemplo en Python

```
>>> class P:
...     def __init__(self, x):
...         self.set_x(x)
...     def get_x(self):
...         return self.__x
...     def set_x(self, x):
...         self.__x = x
...
>>> p = P(2)
>>> p.set_x(4)
>>> p.get_x()
4
```

- Usar Getter/Setter es lo correcto.
- Pero es más engorroso esto `p1.set_x(p2.get_x()+p3.get_x())` que esto `p1.x=p2.x+p3.x`
- El *Pythonic way* no usa Getter/Setter, sino el acceso directo con la notación punto.
- Pero eso es impropio para el uso correcto del encapsulamiento (ocultación).
- Python ofrece una solución a este problema. ¡La solución se llama **propiedades**!

Propiedades en Python - I

- El *Pythonic way* usa acceso directo con la notación punto sobre los atributos. Para acceder con **obj.att** y mutar **obj.att = valor**
- Una forma es con la siguiente función, manteniendo el atributo privado
`property(fget=None, fset=None, fdel=None, doc=None)`

```
>>> class Persona:
...     def __init__(self, dni):
...         self.__set_dni(dni)  # Método set es privado
...     def __str__(self):
...         return f'dni: {self.__dni}'
...     def __get_dni(self):
...         return self.__dni    # Atributo dni es privado
...     def __set_dni(self, dni):
...         self.__dni = dni     # Solo se podrá modificar vía set
...         # Permitimos que sea consultado pero no mutado
...         dni = property(fget=__get_dni, fset=None)
...
>>> p = Persona('12345P')
>>> print(p); print (p.dni); p.dni = '5432W'
dni: 12345P
12345P
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: property 'dni' of 'Persona' object has no setter
```

Propiedades en Python - II

- Otra forma es usando decoradores
- Para esta forma se debe permitir poder acceder al atributo.
- Los pasos son:
 - Un método `get_x` se llamará igual que el atributo y se decora con `@property`.
 - El método `set_x` se le llama igual que al atributo y se decora con `@x.setter`.
 - Tendremos dos métodos `x(self)` y `x(self, x)` con el mismo nombre y distinto número de parámetros. Permitido por los decoradores

```
>>> class Persona:
...     def __init__(self, nombre):
...         self.nombre = nombre # Método set será decorado
...     def __str__(self):
...         return f'nombre: {self.__nombre}'
...     @property
...     def nombre(self):
...         return self.__nombre # La decoración permitirá la
...                                # expresión p.nombre aunque
...                                # el atributo nombre sea privado
...     @nombre.setter
...     def nombre(self, nombre): # La decoración permitirá la
...         self.__nombre = nombre # expresión p.nombre = nombre
...                                # Solo se podrá modificar vía set
...
>>> p = Persona('L. Daniel'); p.nombre = 'Hernández'
>>> print(f'{p} {p.nombre}')
nombre: Hernández Hernández
```

Descriptores en Python

- En Python, un descriptor es un objeto que implementa uno o más de los métodos especiales `__get__`, `__set__` y `__delete__`.
- Los descriptores se utilizan para controlar el acceso a los atributos de un objeto y permiten personalizar el comportamiento de la lectura, escritura y eliminación de esos atributos.
- Cuando se accede al atributo del objeto o de la clase, si éste es un descriptor, Python invoca al método correspondiente `__get__`, `__set__` y `__delete__` del descriptor.

```
>>> class Descriptor:
...     def __get__(self, instance, owner):
...         return instance._attribute      # Obteniendo el valor del atributo
...     def __set__(self, instance, value):
...         instance._attribute = value     # Asignando un valor al atributo
...     def __delete__(self, instance):
...         del instance._attribute         # Eliminando el atributo
...
>>> class Persona:
...     nombre = Descriptor()              # Atributo de clase.
...     def __init__(self, nombre):
...         self.nombre = nombre          # Método set via descriptor. Attr de objeto
...
>>> p = Persona('L. Daniel')
>>> print(f'{p} {p.nombre}')
<__console__.Persona object at 0x100c351d0> L. Daniel
```

Ejercicio.

Un personaje se caracteriza por el dinero que posee.

- Se construye suministrando la cantidad de dinero inicial.
- Construye la interface Getter/Setter indicando qué métodos tienen sentido, sabiendo que un personaje solo es capaz de añadir/quitar una moneda cada vez.
- Construye el método para saber si un personaje tiene dinero
- Así como el método por el que otro personaje le dé una moneda de las que tiene.

Ejercicio.

Un móvil en el plano se caracteriza por su masa, posición (P), velocidad actual, máximas magnitudes de velocidad y aceleración. Si se le informa de un punto destino (Q), entonces se desplaza siguiendo el siguiente proceso:

- Calcula un nuevo vector aceleración: $\vec{a} = \vec{F}_u a_{max} / m$ con $\vec{F} = P\vec{Q}$
- Actualiza su vector velocidad: $\vec{v}_t = \vec{v}_{t-1} + \vec{a}\Delta_t$.
- Actualiza su posición: $\vec{P}_t = \vec{P}_{t-1} + \vec{v}_t\Delta_t$

Supondremos que $\Delta_t \approx 0$ es constante.

Profundizamos en ...

1 Introducción

2 Métodos y Variables

- Métodos
- Variables

3 Sobrecarga

- Python no admite la sobrecarga

4 Visibilidad

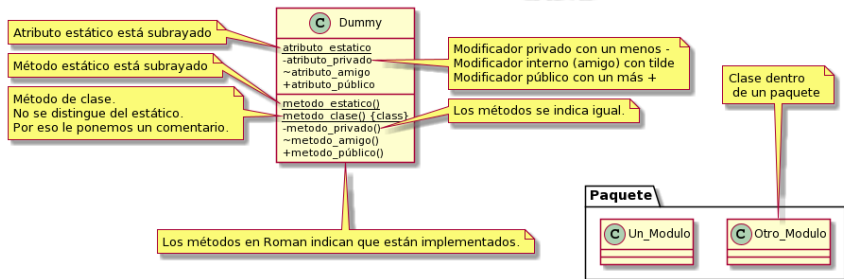
- Espacio de nombres: Modularidad
- Modificadores de acceso
- Representación UML**

5 Resumen



Representación UML

Modificadores de Acceso + Tipos de atributos



- Los atributos/métodos amigos, los que van con tilde, no necesitan marcarse.
- Según esta norma, los estáticos de la representación son internos.

1 Introducción

2 Métodos y Variables

- Métodos
- Variables

3 Sobrecarga

- Python no admite la sobrecarga

4 Visibilidad

- Espacio de nombres: Modularidad
- Modificadores de acceso
- Representación UML

5 Resumen



Principios de la POO

¿Los recuerdas?

- **Abstracción**². Proceso mental de extracción de las características esenciales de un concepto o proceso descartando los detalles (abstracción operacional, de tipo).
- **Encapsulación**³. Proceso por el que se ocultan los detalles del soporte de las características de una abstracción.
 - No se oculta la información, sino su soporte.
 - La información se accede por una interface.
- **Jerarquización**⁴. Estructurar por niveles (jerarquía) los elementos que intervienen en el proceso.
 - Jerarquía de clasificación (**Herencia**).
 - Jerarquía de composición (**Asociación**).
- **Polimorfismo**. La propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos.
- **Modularidad**⁵. Descomposición del sistema en conjunto de módulos poco acoplados (independientes) y cohesivos (con significado propio)

²Se trató en el tema anterior

³Se ha estudiado en este tema

⁴Lo trataremos en el tema siguiente

⁵Se ha estudiado en este tema

