UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 3 de Prácticas

Sobrecarga, visibilidad, paquetes y módulos

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2025-2026

Última modificación:
22 de septiembre de 2025

Sesión 3: Sobrecarga, visibilidad, paquetes y módulos

Índice

3.1. Sobrecarga de métodos 1
3.2. Visibilidad de atributos y métodos 3
3.2.1. Determinar la visibilidad de los atributos 3
3.2.2. Definir métodos getter y setter 3
3.3. Módulos y paquetes 6
3.3.1. Módulos en Python 6
3.3.2. Paquetes en Python 7
3.4. Relación de ejercicios 9

3.1 Sobrecarga de métodos

La **sobrecarga de métodos** en el contexto de la programación orientada a objetos consiste en disponer de diferentes métodos con el mismo nombre pero con distinto tipo y/o número de parámetros. Esto es útil cuando se desea que el comportamiento de un método varíe dependiendo de los argumentos que recibe, lo que ofrece flexibilidad en POO. Sin embargo, **en Python no existe la sobrecarga de métodos** como tal, ya que no se pueden definir varios métodos con el mismo nombre. Si intentamos hacerlo, el último método definido sobrescribirá los anteriores. No obstante, **Python nos permite simular este comportamiento mediante el uso de parámetros opcionales** en los métodos.

Un parámetro opcional es aquel que tiene un valor por defecto, por lo que no es necesario proporcionarlo cuando se llama al método, permitiendo así diferentes formas de invocarlo. Este enfoque es el que utilizaremos en la asignatura, ya que es una forma sencilla y eficiente de manejar la sobrecarga sin complicaciones adicionales. Por otro lado, aunque en Python también se pueden usar parámetros variables (*args) y parámetros con keywords (**kwargs) para simular sobrecarga, estos requieren un manejo más complejo, ya que es necesario conocer los datos suministrados o, en su defecto, implementar una gestión de errores exhaustiva. Por eso, nos enfocaremos únicamente en los parámetros opcionales para simplificar la gestión de la sobrecarga.

La sobrecarga también es aplicable a los constructores, los cuales permiten diferentes formas de inicializar un objeto. En Python, el método *__init__()* se utiliza como un **constructor explícito** que inicializa un objeto con los valores que se le pasan como parámetros. Si no se define un constructor explícito, Python utiliza un **constructor implícito** que no inicializa los atributos del objeto. En la práctica, podemos sobrecargar el constructor de una clase añadiendo parámetros opcionales con valor *None* por defecto, lo que permite crear objetos con diferentes configuraciones.

Para ilustrar mejor el concepto de sobrecarga, partiremos del código de ejemplo ilustrado en la Sesión 2 de prácticas, modificando el constructor para que algunos de los parámetros tengan un valor por defecto y, así, poder permitir llamadas al constructor con diferentes números de parámetros:

```python
class Persona:
    # Definir los atributos de instancia permitidos mediante __slots__
    __slots__ = ['_nombre', '_edad', '_altura', '_activo', '_hobbies']

    especie: str = "Humano"  # Atributo de clase de tipo string

    def __init__(self, nombre: str, edad: int, altura: float = None, activo: bool = None, hobbies: list[str] = None):
        self._nombre: str = nombre  # Atributo de instancia de tipo string
        self._edad: int = edad  # Atributo de instancia de tipo entero
        self._altura: float = altura  # Atributo de instancia de tipo real
        self._activo: bool = activo  # Atributo de instancia de tipo booleano

        # Manejo de hobbies
        if hobbies is None:
            self._hobbies: list = []  # Si no se pasan hobbies, asigna una lista vacía
        else:
            self._hobbies: list = hobbies  # Si se pasan hobbies, asigna la lista recibida

    # Método de instancia
    def obtener_info_sin_hobbies(self) -> str:
        return f"Nombre: {self._nombre}, Edad: {self._edad}, Altura: {self._altura}, Activo: {self._activo}"

    # Método de clase
    @classmethod
    def obtener_especie(cls) -> str:
        return f"La especie es {cls.especie}"

    # Método estático
    @staticmethod
    def es_mayor_edad(edad: int) -> bool:
        return edad >= 18

    # Método mágico __str__
    def __str__(self) -> str:
        if self._hobbies:
            hobbies_str = ", ".join(self._hobbies)  # Une los hobbies con coma y espacio
        else:
            hobbies_str = "No tiene hobbies"  # Mensaje si no hay hobbies

        return (f"Persona(nombre={self._nombre}, edad={self._edad}, altura={self._altura}, "
                f"activo={self._activo}, hobbies=[{hobbies_str}])")

    # Método mágico __gt__ para comparar la edad de dos personas
    def __gt__(self, otra_persona: 'Persona') -> bool:
        return self._edad > otra_persona._edad
```

```python
if __name__ == "__main__":
    # Crear objetos con diferentes parámetros en el constructor
    persona1: Persona = Persona("Carlos", 25)  # Solo nombre y edad
    persona2: Persona = Persona("Ana", 30, 1.65)  # Nombre, edad y altura
    persona3: Persona = Persona("Luis", 40, False)  # Nombre, edad, y activo
    persona4: Persona = Persona("Marta", 22, 1.70, True, ['leer', 'viajar'])  # Todos los parámetros

    # Mostrar la información de cada objeto
    print(persona1)  # Persona(nombre=Carlos, edad=25, altura=None, activo=None, hobbies=[No tiene hobbies])
    print(persona2)  # Persona(nombre=Ana, edad=30, altura=1.65, activo=None, hobbies=[No tiene hobbies])
    print(persona3)  # Persona(nombre=Luis, edad=40, altura=1.8, activo=False, hobbies=[No tiene hobbies])
    print(persona4)  # Persona(nombre=Marta, edad=22, altura=1.7, activo=True, hobbies=[leer, viajar])
```

Como podemos ver en el constructor, solo los parámetros nombre y edad son obligatorios, siendo el resto opcionales. Así, todos los opcionales son inicializados por defecto a *None* en caso de no suministrar un valor explícitamente, indicando que no hay un valor asignado. Imagina el caso de la *edad*, ¿podríamos realmente dar un valor por defecto a una persona si no nos facilita su edad? ¿Qué pasaría con la *altura*? Como es lógico, en el mundo real no podemos inventarnos la información, por lo que es preferible indicar que no tenemos un valor de inicialización (*None*).

Además, es interesante destacar la inicialización del atributo *hobbies*. En el caso de que no se haya pasado un listado como parámetro, por defecto se asignará None. Sin embargo, al ser una lista sí podríamos tomar que el valor de inicialización por defecto es la lista vacía. Por ello, comprobamos si el valor del parámetro es vacío y, en base a ello, inicializaremos la lista como vacía o asignaremos el valor suministrado al constructor. Esto realmente sería equivalente a poner *hobbies: list = []* como parámetro opcional. También se ha modificado el método mágico *__str__()* para incluir la lista de *hobbies* (no estaba representada en la Sesión 2 por simplicidad). Para ello, se comprueba si la lista es vacía o no (también se podría usar la función *len()*), y se emplea la función join () del tipo de datos String para construir la cadena de texto con todos los hobbies.

Atendiendo al *__main__()*, podemos ver cuatro formas diferentes de crear una persona (sobrecarga del constructor), teniendo siempre que especificar *nombre* y *edad*. Por ejemplo, *persona1* usa únicamente el constructor con los dos parámetros obligatorios (asignando al resto de atributos un su valor *None*). La *persona2* añade su altura y *persona3* especifica su actividad. Por último, *persona4* asigna a todos los atributos posibles.

3.2 Visibilidad de atributos y métodos

La **visibilidad** en POO es una característica fundamental que controla cómo los atributos y métodos de una clase son accesibles desde otras partes del programa. Esto es esencial para garantizar que los datos se mantengan **protegidos** y que los objetos interactúen de manera **segura y coherente**. En un diseño orientado a objetos, las clases deben interactuar enviando mensajes a través de la llamada a métodos, es decir, un objeto solicita a otro que realice una acción o le proporcione información. Esta interacción evita que los objetos modifiquen directamente los atributos de otros, lo que podría dar lugar a incoherencias o errores.

El **acceso a los atributos de una clase debe estar controlado** para evitar problemas como la asignación de valores no válidos o la modificación inesperada del estado interno del objeto. No se debe permitir que cualquier objeto pueda cambiar directamente el valor de un atributo, ya que esto podría comprometer la integridad del sistema. En su lugar, las clases deben exponer solo aquellos métodos que permitan acceder o modificar los datos de forma segura, **sin revelar detalles internos** de su implementación.

Para controlar este acceso, se utilizan modificadores de acceso, que determinan la visibilidad de los atributos y métodos de una clase:

- **Acceso público**: Un miembro público es accesible desde **cualquier parte del código**, tanto desde dentro de la clase como desde otras clases o módulos. Esto se utiliza cuando se desea que un método o atributo esté disponible de manera abierta.
- **Acceso protegido**: Un miembro protegido solo es accesible desde **la clase en la que se define y sus subclases** (en el caso de herencia de clases). Esto es útil cuando queremos restringir el acceso a los detalles internos de una clase, pero permitiendo que las clases derivadas sigan accediendo a ellos.
- **Acceso privado**: Un miembro privado solo es accesible **dentro de la propia clase**, sin posibilidad de que otras clases (incluso las que hereden de ella) puedan acceder a él. Esto es útil para encapsular completamente los datos que no deberían estar disponibles fuera de la clase.

En Python, no existen modificadores de acceso explícitos como en otros lenguajes. Es decir, no hay una forma efectiva nativa de proteger el acceso los miembros de una clase. Sin embargo, se siguen ciertas convenciones para indicar cómo debe tratarse un miembro:

- Un **guión bajo simple (_miembro)** antes del nombre de un atributo o método sugiere que este es protegido, aunque ten en cuenta que Python no impide realmente su acceso desde otras clases (no se lanza ningún error). Se utiliza como un acuerdo entre desarrolladores para señalar que no debería ser accedido directamente desde fuera de la clase.
- Un **doble guión bajo (__miembro)** antes del nombre indica que el atributo o método es privado. Python no permite el acceso externo a este atributo (*objeto.__atributo*) porque utiliza un mecanismo llamado *name mangling* que cambia internamente el nombre del atributo *__miembro* por la expresión *_Clase__miembro*. Sin embargo, esto solo dificulta el acceso, pues realmente podemos seguir accediendo a través de *objeto._Clase__miembro*. En Python no es común este nivel de visibilidad privado.

Por lo tanto, **la ausencia de uno o dos guiones bajos delante del nombre del miembro implica que la visibilidad debe considerarse como pública** y se podría acceder desde fuera de la clase.

3.2.1. Determinar la visibilidad de los atributos

Para comprender mejor estos aspectos de visibilidad, pasaremos a revisar la visibilidad de los atributos definidos actualmente en el ejemplo de código. Podemos ver que *especie* es un atributo público, lo que tiene sentido ya que este atributo es una característica compartida por todos los objetos de la clase y no es sensible ni propenso a cambios que puedan comprometer la integridad del sistema. Es una información general que podemos compartir a todos los niveles sin restricciones.

Por otro lado, el resto de los atributos actualmente definidos, como *_nombre*, *_edad*, *_altura*, *_activo*, y *_hobbies*, tienen visibilidad protegida. Este nivel de visibilidad nos ayuda a indicar que estos atributos no deben ser manipulados directamente desde fuera de la clase, pero son accesibles desde clases derivadas en caso de herencia de clases.

3.2.2. Definir métodos getter y setter

Es una buena práctica en POO utilizar **métodos getter y setter** para acceder a los atributos, independientemente del tipo de visibilidad que tengan. Los **métodos getter permiten consultar** el valor de un atributo de manera controlada, mientras que los **métodos setter permiten modificar** su valor, garantizando que cualquier cambio sea válido y coherente. Estos métodos no solo deben ser utilizados por clases externas, sino también dentro de la propia clase, para mantener la

consistencia del acceso y el control de errores. De este modo, se asegura que los atributos de una clase solo se modifiquen mediante métodos específicos que validen los valores, evitando que se asignen valores incorrectos o peligrosos.

Una vez tenemos nuestros atributos con la visibilidad adecuada, es el momento de determinar qué métodos *get()* y *set()* tienen sentido para cada uno de los atributos existentes. Estudiemos cada atributo, indicando para cada uno si requiere o no de métodos getter o setter. Para cada método se indica también su nivel de visibilidad:

- *especie*
  - Getter (público): Sí. Es posible que se quiera consultar la especie de una persona. Además, al ser un atributo público tiene sentido que haya un método getter asociado.
  - Setter: No. No tiene sentido cambiar la especie de una persona (¡al menos, no hasta dentro de unos cuantos millones de años!).
- *_nombre*
  - Getter (público): Sí. Es común querer acceder al nombre de una persona para mostrarlo o trabajar con él. Tener un método para obtener el nombre es útil y seguro.
  - Setter (público): Sí. El nombre de una persona es algo que podría cambiar. Por ejemplo, debido a un error o un cambio legal. Debería haber pues un método que permita cambiarlo de forma controlada.
- *_edad*:
  - Getter (público): Sí. El acceso a la edad es algo habitual en aplicaciones que manejan información personal.
  - Setter (protegido): Sí. La edad de una persona cambia con el tiempo. Sin embargo, quizás no sea útil que cualquier clase puede modificar la edad a su antojo. En su lugar, tendría sentido tener un método público *cumplir_años* que permita aumentar en uno el valor de la edad actual (e internamente la clase use el setter protegido) en lugar de asignar cualquier valor de edad.
- *_altura*:
  - Getter (público): Sí. La altura puede ser consultada para varios propósitos (por ejemplo, estadísticas o datos personales).
  - Setter (público): Sí. La altura de una persona podría cambiar o ser corregida. Un setter sería útil para validar que se asignen valores positivos y razonables.
- *_activo*:
  - Getter (público): Sí. Es útil saber si una persona está activa o no en ciertos contextos (por ejemplo, si está disponible para realizar ciertas tareas).
  - Setter (público): Sí. El estado de actividad puede cambiar, por lo que sería útil poder modificar este valor (por ejemplo, si una persona se va de vacaciones o está de baja médica).
- *_hobbies*:
  - Getter (público): Sí. Es útil poder acceder a los hobbies de una persona.
  - Setter: No. Los hobbies cambian con el tiempo. Sin embargo, en este caso podríamos asumir que es más conveniente disponer de funcionalidad diferenciada para agregar y eliminar un hobby, en dos métodos por separado en lugar de reemplazar la lista completa. En cualquier caso, esto dependerá mucho de las necesidades de la aplicación a realizar.

Como habrás observado, los métodos getter y setter pueden tener visibilidad pública porque su propósito principal es proporcionar un acceso controlado a los atributos públicos o protegidos desde fuera de la clase. No obstante, **la necesidad de cada método particular y de su visibilidad dependerá mucho del contexto de aplicación y de las necesidades de la aplicación**, por lo que hay un cierto margen de libertad para el programador en algunos casos. Veamos ahora cómo queda el código de la clase Persona tras añadir los getter/setter listados anteriormente:

```
class Persona:
    # Definir los atributos de instancia permitidos mediante __slots__
    __slots__ = ['_nombre', '_edad', '_altura', '_activo', '_hobbies']

    especie: str = "Humano"  # Atributo de clase de tipo string

    def __init__(self, nombre: str, edad: int, altura: float = None, activo: bool = None,
                 hobbies: list[str] = None):
        self._nombre: str = nombre  # Atributo de instancia de tipo string
        self._edad: int = edad  # Atributo de instancia de tipo entero
        self._altura: float = altura  # Atributo de instancia de tipo real
        self._activo: bool = activo  # Atributo de instancia de tipo booleano

        # Manejo de hobbies
        if hobbies is None:
            self._hobbies: list = []  # Si no se pasan hobbies, asigna una lista vacía
        else:
```

```
            self._hobbies: list = hobbies  # Si se pasan hobbies, asigna la lista recibida

    # Getter para el atributo de clase "especie"
    @classmethod
    def get_especie(cls) -> str:
        return cls.especie

    # Getter para el nombre
    def get_nombre(self) -> str:
        return self._nombre

    # Setter para el nombre
    def set_nombre(self, nuevo_nombre: str) -> None:
        self._nombre = nuevo_nombre

    # Getter para la edad
    def get_edad(self) -> int:
        return self._edad

    # Setter protegido para la edad con validación de valor positivo
    def _set_edad(self, nueva_edad: int) -> bool:
        if nueva_edad >= 0:
            self._edad = nueva_edad
            return True
        return False  # Edad negativa

    # Método público para cumplir años
    def cumplir_años(self) -> bool:
        return self._set_edad(self.get_edad() + 1)

    # Getter para la altura
    def get_altura(self) -> float:
        return self._altura

    # Setter para la altura con validación
    def set_altura(self, nueva_altura: float) -> bool:
        if nueva_altura > 0:
            self._altura = nueva_altura
            return True
        return False  # Altura inválida

    # Getter para el estado de actividad
    def get_activo(self) -> bool:
        return self._activo

    # Setter para el estado de actividad
    def set_activo(self, estado: bool) -> None:
        self._activo = estado

    # Getter para los hobbies
    def get_hobbies(self) -> list:
        return self._hobbies

    # Método para agregar un hobby
    def agregar_hobby(self, hobby: str) -> None:
        self._hobbies.append(hobby)

    # Método para eliminar un hobby
    def eliminar_hobby(self, hobby: str) -> bool:
        if hobby in self._hobbies:
            self._hobbies.remove(hobby)
            return True
        return False  # Hobby no encontrado

    # Método estático
    @staticmethod
    def es_mayor_edad(edad: int) -> bool:
        return edad >= 18

    # Método mágico __str__
    def __str__(self) -> str:
        if self._hobbies:
            hobbies_str = ", ".join(self._hobbies)  # Une los hobbies con coma y espacio
        else:
            hobbies_str = "No tiene hobbies"  # Mensaje si no hay hobbies

        return (f"Persona(nombre={self._nombre}, edad={self._edad}, altura={self._altura}, "
                f"activo={self._activo}, hobbies=[{hobbies_str}])")

    # Método mágico __gt__ para comparar la edad de dos personas
    def __gt__(self, otra_persona: 'Persona') -> bool:
        return self._edad > otra_persona._edad
```

En el código mostrado anteriormente hay algunos aspectos interesantes a destacar. En primer lugar, el método *obtener_especie* ha desaparecido, dando paso a *get_especie*: ambos implementan el mismo código. Sin embargo, por convención es preferible tener métodos getter/setter para tener una nomenclatura unificada en todas las clases. También

hemos eliminado el método *obtener_info_sin_hobbies()*, pues ya no es tan útil al tener el método mágico *__str__()*.

En aquellos métodos setter en los que se realiza control de errores sobre los parámetros, se devuelve un booleano. Así, al invocar, por ejemplo, al método *set_altura* con un valor de altura negativo, el método setter retornará False, indicando que no se ha podido modificar el estado del atributo. Otra alternativa podría haber sido retornar un string con un mensaje indicativo. No obstante, el usar un booleano es más práctico, pudiendo incluir la llamada en un bloque condicional para verificar si el método ha tenido el efecto deseado.

Es interesante destacar que en el código indicado sólo hemos implementado métodos para gestión de getter/setter principalmente. Sin embargo, además de estos habrá funciones para ofrecer funcionalidad asociada a las acciones que las personas pueden realizar. Ampliaremos la clase Persona en la siguiente sesión de prácticas para dotarla de mayor funcionalidad.

Finalmente, mostramos el contenido del *main* para ejemplificar el funcionamiento de la clase anterior:

```
if __name__ == "__main__":
    # Crear dos instancias de Persona
    persona1: Persona = Persona("Ana", 25, 1.68, True, ['leer', 'escribir'])
    persona2: Persona = Persona("Juan", 30, 1.75, True, ['deporte', 'viajar'])

    # Usar __str__ para mostrar la información de las personas
    print(persona1)  # Salida: Persona(nombre=Ana, edad=25, altura=1.68, activo=True)
    print(persona2)  # Salida: Persona(nombre=Juan, edad=30, altura=1.75, activo=True)

    print("Datos de persona 1: ")
    # Mostrar atributos de persona1 individualmente
    print("- Especie: ", Persona.get_especie())  # Salida: Humano
    print("- Nombre: ", persona1.get_nombre())  # Salida: Ana
    print("- Edad: ", persona1.get_edad())  # Salida: 25
    print("- Altura: ", persona1.get_altura())  # Salida: 1.68
    print("- Activo: ", persona1.get_activo())  # Salida: True
    print("- Hobbies: ", persona1.get_hobbies())  # Salida: ['leer', 'escribir']

    # Usar es_mayor_edad (método estático)
    print("Con 25 años es mayor de edad? ", Persona.es_mayor_edad(25))  # Salida: True (porque 25 >= 18)
    print("Con 17 años es mayor de edad? ", Persona.es_mayor_edad(17))  # Salida: False (porque 17 < 18)

    # Aumentar la edad
    persona1.cumplir_años()

    # Consultar edad tras incrementarla
    print("Edad: ", persona1.get_edad())  # Salida: 26

    # Intento de cambio de altura con valor inválido
    print("Cambio de altura: ", persona1.set_altura(-1))  # Salida: False

    # Usar __gt__ para comparar las edades de persona1 y persona2
    if persona1 > persona2:
        print(f"{persona1.get_nombre()} es mayor que {persona2.get_nombre()}")
    else:
        print(f"{persona2.get_nombre()} es mayor que {persona1.get_nombre()}")# Salida: Juan es mayor que Ana
```

3.3 Módulos y paquetes

En programación, la **modularidad** es uno de los principios clave para organizar y estructurar programas grandes. Modularizar significa dividir un programa en partes más pequeñas, llamadas módulos, que agrupan funciones, clases y variables relacionadas. Esta división favorece la reutilización del código, la organización y el mantenimiento, permitiendo una gestión más eficaz de proyectos complejos. A su vez, estos módulos pueden organizarse en paquetes, que son colecciones de módulos relacionados.

3.3.1. Módulos en Python

Un módulo en Python es simplemente un **archivo** que contiene código Python. Los módulos se usan para organizar el código en bloques más sencillos y reutilizables. Por ejemplo, en lugar de tener todas las funciones y clases en un solo archivo grande, puedes dividirlas en diferentes módulos según su funcionalidad. Cada módulo tiene su propio espacio de nombres, lo que significa que las funciones y variables de un módulo no interfieren con las de otro, salvo que se importen explícitamente.

El nombre de un módulo es el nombre del archivo sin la extensión *.py*. Dentro de un módulo, existe una variable especial llamada *__name__*, que contiene el nombre del módulo. Cuando ejecutas un módulo directamente, el valor de __name__ será __main__. Sin embargo, si el módulo es importado desde otro archivo, __name__ contendrá el nombre del módulo.

Para utilizar el código de un módulo en otro archivo, se utiliza la instrucción import. Existen varios tipos de importaciones que se pueden realizar en Python, lo que permite flexibilidad al momento de gestionar dependencias.

- **Importación de un módulo completo**: Cuando se importa un módulo completo, se puede acceder a sus funciones y variables utilizando el nombre del módulo seguido de un punto. Esto permite evitar conflictos entre diferentes módulos.

```
import math
print(math.sqrt(16))  # Accede a la función sqrt() del módulo math
```

- **Importación con alias**: Es posible importar un módulo o una función con un nombre alternativo, lo que puede ser útil cuando el nombre del módulo es largo o cuando queremos evitar conflictos de nombres.

```
import math as m
print(m.sqrt(16))  # Usamos el alias 'm' en lugar de 'math'
```

- **Importación selectiva**: En lugar de importar todo un módulo, se pueden importar solo partes específicas del mismo, como funciones o clases, utilizando la sintaxis `from ... import ...`

```
from math import sqrt
print(sqrt(16))  # No es necesario usar math.sqrt(), solo sqrt()
```

- **Importar todo el contenido de un módulo**: Se puede importar todo el contenido de un módulo utilizando el asterisco (*). Sin embargo, esta práctica no es recomendada ya que puede generar conflictos de nombres y hacer el código menos legible. En concreto, esta técnica hace que todas las funciones y variables del módulo estén disponibles en el espacio de nombres actual sin necesidad de usar el prefijo del módulo.

```
from math import *
print(sqrt(16))  # sqrt() está disponible directamente
```

3.3.2. Paquetes en Python

Un paquete es una **colección de módulos** organizados en una estructura de directorios. Los paquetes permiten estructurar el código de manera jerárquica, dividiendo los módulos en subpaquetes y submódulos, de una forma similar a cómo los sistemas operativos gestionan carpetas y archivos.

Para que Python reconozca una carpeta como un paquete, debe contener un archivo especial llamado __init__ .py. Este archivo puede estar vacío y su presencia indica a Python que esa carpeta debe tratarse como un paquete. Dentro de un paquete, los módulos se pueden organizar en subcarpetas, creando una jerarquía de módulos relacionados.

El propósito de los paquetes es facilitar la modularidad a gran escala y mejorar la organización de grandes proyectos, permitiendo que los módulos se agrupen de forma lógica. A través de la combinación de módulos y paquetes, es posible mantener el código de un proyecto grande de manera ordenada y fácil de mantener.

Una vez que conocemos cómo modularizar el código, vamos a ver una posible alternativa de división en módulos y paquetes para el código que ya tenemos:

```
proyecto_personas/        # Paquete raíz del proyecto
|-- __init__.py           # Indica que es un paquete
|-- tdas/                 # Subpaquete que agrupa las clases de datos o modelos
|   |-- __init__.py
|   |-- persona.py        # Módulo que contiene la clase Persona
|   |-- mascota.py        # Módulo que contiene la clase Mascota
|-- tools/                # Subpaquete para utilidades y funciones adicionales
|   |-- __init__.py
|   |-- validaciones.py   # Módulo para validaciones (como la validación de edad)
|   |-- utilidades.py     # Módulo para funciones genéricas o utilitarias
|   |-- gui/              # Subpaquete para la interfaz gráfica (GUI)
|       |-- __init__.py
|       |-- login.py      # Módulo para la funcionalidad de login
|       |-- home.py       # Módulo para la funcionalidad de la pantalla de inicio
|-- main.py               # Módulo principal que usa los otros módulos y paquetes
```

Así, tenemos un paquete *proyecto_personas* que contiene todo nuestro código y representa el paquete raíz del proyecto. Dentro, tenemos dos directorios (*tdas* y *tools*) que contendrán a su vez otros módulos, y el archivo *main.py* encargado de realizar las tareas de inicialización de la aplicación. En el paquete *tdas* tendremos las clases que creemos, en este caso tenemos como ejemplo los módulos *persona.py* y *mascota.py*, que contendrán respectivamente el código de las cases *Persona* y *Mascota*. Es importante destacar que dentro, podremos crear también nuevos directorios o paquetes para agrupar la información.

El directorio *tools* contiene dos módulos, uno para proporcionar funcionalidad dedicada a validar campos de entrada *validaciones.py*, y otro para ofrecer funciones genéricas de apoyo a las clases definidas y que pueden ser aplicables a varias clases (*utilidades.py)*. Además, el subpaquete *tools* tiene a su vez otro paquete dentro llamado *gui* que contendrá código necesario para la interfaz gráfica de la aplicación. En concreto, hemos definido como ejemplo dos módulos, *login.py* y *home.py*, y que contendría el código necesario representar una ventana de inicio de sesión y la página principal de la aplicación, respectivamente.

Esta estructura propuesta es un ejemplo de los muchos que podríamos pensar, y no debe servirnos como una estructura rígida a seguir. Dependiendo del problema a resolver y de los módulos a implementar, determinaremos de una u otra forma la modularización en módulos *.py* y paquetes. Esta creatividad forma parte del proceso de programar, y la iremos reforzando con el tiempo.

3.4 Relación de ejercicios

Esta sesión de prácticas parte del código implementado en el último ejercicio de la Sesión 2 donde tenemos implementadas completamente las clases de Polígono, Punto y Línea. A continuación se listan los atributos y métodos que deberían estar definidos en cada clase:

```
Clase Punto

Atributos:
    _coordenada_x: Coordenada X del punto.
    _coordenada_y: Coordenada Y del punto.

Métodos:
    obtener_coordenada_x()
    obtener_coordenada_y()
    obtener_cuadrante()
```

```
Clase Linea

Atributos:
    _punto_inicio: Punto inicial de la línea.
    _punto_fin: Punto final de la línea.

Métodos:
    obtener_punto_inicio()
    obtener_punto_fin()
    calcular_longitud()
    mostrar_puntos()
```

```
Clase Poligono

Atributos:
    numero_poligonos: Atributo de clase que cuenta el número de polígonos creados.
    _numero_lados: Número de lados del polígono.
    _color: Color del polígono.
    _forma: Forma del polígono (convexo, cóncavo, complejo).
    _relacion_lados: Relación entre los lados (regular, irregular).
    _lados: Lista de objetos Linea que representan los lados del polígono.

Métodos:
    obtener_numero_poligonos()
    obtener_numero_lados()
    obtener_color()
    obtener_forma()
    obtener_relacion_lados()
    establecer_numero_lados()
    calcular_perimetro()
    establecer_lados()
    __str__()
    __eq__()
    __add__()
    __lt__()
```

1. **Sobrecarga de métodos**. Trabajaremos con la clase *Polígono* para practicar la sobrecarga de constructores y métodos. Sigue los siguientes pasos:

   a) Actualmente, el constructor de la clase *Polígono* recibe dos parámetros obligatorios (*forma* y *relación_lados*) y dos parámetros opcionales (*numero_lados* y *color*). Transforma el constructor para que se puedan crear polígonos de las siguientes formas:

   - Especificado sólo el número de lados.
   - Especificando el número de lados y la forma.
   - Especificando el número de lados, la forma y la relación de lados.
   - Especificando el número de lados, la forma, la relación de lados y el color.

   b) Actualiza el método __str__ para que contemple las nuevas situaciones introducidas a nivel de constructor en la clase Polígono.

   c) Crea un método llamado **escalar_poligono** capaz de aumentar el número de lados de un polígono, devolviendo si ha sido posible el escalado. Podremos usarlo de dos formas diferentes:

   - La primera versión recibe un parámetro *factor*, que multiplica el número de lados del polígono por ese valor.
   - La segunda versión recibe el parámetro *ajuste_lados* que sumará (o restará) un número determinado de lados.

¿Podríamos apoyarnos en algún método ya existente en la clase *Polígono*? Si es así, ¿sería conveniente realizar alguna modificación?

d) Modifica el *main* para que implemente la siguiente funcionalidad:

- Crear dos polígonos de dos formas diferentes, especificando siempre el número de lados y otros parámetros alternativos.
- Escalar el primer polígono utilizando únicamente un factor de escalado.
- Escalar el segundo polígono usando el ajuste.
- Mostrar información sobre los atributos de ambos polígonos.

2. **Visibilidad de atributos**. En este ejercicio, continuaremos trabajando con la clase *Polígono* para profundizar en los conceptos de visibilidad de atributos y el uso de modificadores de acceso.

   a) Revisa la visibilidad actual de los atributos y determina cuáles deberían cambiar y cuáles mantener su visibilidad actual.

   b) Añade un atributo privado *contraseña_base_datos* que será exclusivamente accedido por la propia clase para dotar de protección extra.

3. **Métodos Getter/Setter para acceder a los atributos**. Para el tercer apartado, trabajaremos con getters y setters en el código de la clase Polígono. En particular, vamos a analizar si para cada atributo tiene sentido crear tanto un getter como un setter (ahora mismo con visibilidad pública), o si hay casos donde solo uno sería suficiente, o ninguno de ellos. Recuerda cambiar el nombre de los métodos *obtener()* y *establecer()* ya existentes a *get()* y *set()* por convención. Además, amplía el *main* para que incluya la siguiente funcionalidad:

   a) Tras crear y escalar ambos polígonos (ya implementado en el *main* en el apartado anterior), modifica el primer polígono para que tenga color AZUL, tenga forma convexa y sea un polígono regular. Haz uso de los métodos setter previamente definidos. ¿Podríamos modificar directamente al atributo *contraseña_base_datos*?

   b) Consulta los atributos modificados del primer polígono de forma adecuada haciendo uso de métodos getter. ¿Podemos leer directamente al atributo *contraseña_base_datos*?

   c) Muestra por pantalla el contenido de ambos objetos mediante el uso del método mágico __str__.

4. **Revisión de la visibilidad de los métodos**. En este apartado, ajustaremos la visibilidad de los métodos en la clase Polígono, considerando su funcionalidad y uso. **Considera ahora que sólo es posible modificar el número de lados a través del método *escalar_poligono()***:

   a) Evalúa si los métodos get/set deben ser públicos, protegidos o privados, teniendo en cuenta el acceso externo que se espera para cada atributo. Haz los cambios que consideres.

   b) Revisa la visibilidad de los métodos restantes de la clase *Polígono* (por ejemplo, los métodos de clase y otros métodos de instancia) y determina si deben ser públicos, protegidos o protegidos.

   c) Define un método para cambiar la contraseña de un polígono desde el programa principal. Se devolverá verdadero cuando se realice con éxito, falso en caso contrario. No se puede repetir la contraseña que haya en ese momento y la nueva contraseña debe tener más de 8 caracteres.

5. **Módulos y paquetes**. Define tu propia estructura de proyecto usando módulos y paquetes, de forma que modularicemos lo que hasta ahora tenemos sólo en un módulo. Algunas consideraciones importantes:

   a) En primer lugar, asegúrate que tienes un módulo .py para el main y un módulo por cada clase (o enumerado) de tu proyecto.

   b) Recuerda que tenemos las clases Polígono, Punto y Línea definidas de la sesión anterior y que debemos incluir en nuestra estructura de proyecto.

   c) Tras ello, ten en cuenta que además de la aplicación principal, se ofrece funcionalidad centrada en geometría para diferentes tipos de polígonos (podríamos tener diferentes tipos de polígonos como círculos, cuadrados o triángulos, entre otros muchos). Para definir nuestro proyecto, crea clases para **Círculo**, **Cuadrado** y **Triángulo** en módulos nuevos que estarán vacíos de código.

   d) Tras crear la estructura de paquetes y módulos, escribe código en el módulo *main.py* para que el programa principal pueda hacer uso de las clases implementadas. En particular, debe ejecutar las siguientes acciones:

   - Crea un polígono rojo de cuatro lados, convexo e irregular. Además, crea un segundo polígono de cinco lados con forma convexa (el resto de propiedades no se aportan para este polígono).
   - Accede a algunas de las propiedades de los polígonos creados mediante el uso de métodos get.
   - Modifica o establece algunas propiedades de los polígonos creados con métodos set.

6. **Mejora de las clases Punto y Línea**. Modificar las clases *Punto* y *Línea* para mejorar los siguientes aspectos, previamente definidos en el ejercicio 5 como módulos de nuestro proyecto:

- **Sobrecarga de métodos**: Revisar si sería necesario aplicar sobrecarga de métodos a los constructores y métodos de ambas clases.
- **Visibilidad de atributos**: Comprobar la visibilidad actual de los atributos de ambas clases y modificarlas de forma conveniente.
- **Métodos getter/setter**: Definir los métodos get/set adecuados para cada clase, sustituyendo aquellos métodos de consulta y modificación previamente existentes en la sesión 2.
- **Modificación del main**: Actualiza el main para crear, consultar y modificar puntos y líneas, además de polígonos.