UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 1 de Prácticas

Python y PyCharm

Material original por Luis Daniel Hernández Molinero (ldaniel@um.es)

Editado por Sergio López Bernal (slopez@um.es), Javier Pastor Galindo (javierpg@um.es), Jesús Damián Jiménez Re (jesusjr@um.es) y Gregorio Martínez Pérez (gregorio@um.es)

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2024-2025

Última modificación:
9 de septiembre de 2024

Sesión 1

Python y PyCharm

Índice

1.1 Introducción

Python es un lenguaje de programación de propósito general escrito sobre el lenguaje C. Se usa en una amplia variedad de disciplinas como biología, finanzas, química, análisis numérico, inteligencia artificial, etc. También se usa como lenguaje para crear scripts por parte de los administradores de sistemas informáticos.

En esta primera sesión práctica se indica cómo llevar a cabo la instalación de Python y el uso del entorno de desarrollo que se usará en las sesiones de prácticas. Finalmente, esta sesión ofrece un repaso general de cómo realizar programación imperativa en Python, mediante el uso de variables, colecciones, bloques condicionales, bucles y funciones.

1.2 Introducción y configuración del IDE

1.2.1. Instalación de Python

CPython es la implementación estándar de Python. `https://www.python.org/`. Aunque hay otras alternativas (revisar el Anexo I en la sección 1.5.1), en clase se usará CPython.

- CPython no traduce el código Python a C por sí mismo.
- Ejecuta un intérprete.
- Instalación básica $\approx$ 120Mb

- Cython es un módulo que compila a código en C o C++ desde Python
  - Permite incluir C en código Python
  - Permite incluir Python en código C y ser compilado.

Puedes instalar Python (CPython) sin problema en tu Sistema Operativo:

- **Windows**. Consulta https://www.python.org/downloads/windows/. Durante la instalación, verás una ventana de «Setup». Asegúrate de marcar las casillas «Add Python xx to PATH» o «Add Python to your environment variables». Para más detalles consultar https://tutorial.djangogirls.org/es/python_installation/
- **mac OS**. Descarga e instala (como en Windows) pero **ten mucho cuidado si tu SO es anterior al 2022**. Algunos macOS llevan Python como parte del sistema operativo. Trabajar directamente sobre Python del SO puede conllevar que éste deje de funcionar si no se sabe exactamente qué se está haciendo.
  Si tienes un Mac, abre un terminal y ejecuta `python -version`. Si como resultado obtienes `Python 2.xx` entonces debes instalar previamente Homebrew y después Python de Homebrew.
  1. Para instalar Homebrew, abrae un terminal y ejecuta:
     ```
     ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
     ```
  2. Inserta el directorio de Homebrew al inicio de tu variable de ambiente PATH. Puedes hacer esto agregando la siguiente línea al final de tu archivo `~/.profile` o `~/.bash_profile`
     ```
     export PATH=/usr/local/bin:/usr/local/sbin:$PATH
     ```
  3. Instalar Python 3: `brew install python3`

  Ahora tendrás dos versiones de Python. La del SO y la ubicada en `/usr/local/Cellar`.
  Ahora bien, si al ejecutar `python -version` obtienes un mensaje de error, entonces solo debes instalar la distribución de la página oficial: https://www.python.org/downloads/
- **GNU/Linux**
  ```
  Debian (Ubuntu inclusive): sudo apt install python3
  Fedora: sudo dnf install python3
  ```

1.2.2. Instalación de la Herramienta de Desarrollo

Existen varios Entornos de Desarrollo Integrado (Integrated Development Environment, IDE) y editores que ayudan a la programación. En el contexto de Python podemos destacar:

- **PyCharm**. Es uno de los IDE de Python más completos http://jetbrains.com/pycharm
- **PyDev**. Un buen IDE con soporte para CPython, Jython e Iron Python. http://pydev.org
- **Spyder**. IDE de código abierto y totalmente gratuito. Desarrollado principalmente para científicos e ingenieros. Se recomienda usar el que viene con Anaconda https://www.spyder-ide.org/
- **Visual Studio Code**. Es un editor de código fuente con multitud de plugins. https://code.visualstudio.com/
- **Sublime Text**. Es un editor que viene con muchas opciones. https://www.sublimetext.com/
- **Atom**. Otro editor. Es de lo más usados. https://atom.io/

Esta sesión describe los pasos para instalar y configurar **PyCharm**, que será el IDE a usar durante la asignatura.

Instalación de PyCharm

1. Instalar `PyCharm Community` desde https://www.jetbrains.com/es-es/pycharm/download/
2. Para instalar un diccionario español:
   - Descargar en cualquier directorio el diccionario https://github.com/EOM/Spell-Checking-PHPStorm-Spanish-dic-UTF8/blob/master/spanish-utf8.dic.
   - Desde PyCharm acceder en el menú de configuración a File > Settings > Editor > Natural Languages > Spelling y añadir el diccionario anterior. La ruta indicada corresponde a Windows. A partir de ahora mostraremos únicamente las opciones de Windows por simplicidad.
3. En Settings > Plugins instalarás complementos que te pueden ser útiles. En particular, algunos temas para el IDE:
   - Godot Theme (recomendado)
   - Theme Collection
   - Visual Studio Code Dark Plus Theme

   Pulsar Apply

4. En Settings > Editor > Natural Languages
   - Seleccionar +↓
   - Seleccionar Español

   Pulsar Apply

Para el desarrollo de las prácticas de este curso de Tecnología de la Programación es suficiente PyCharm Community, que es la que está instalada en los ordenadores de la universidad. Pero también se puede descargar la versión profesional en el mismo enlace.

Las diferencias entre una y otra versión se pueden ver la final de la página https://www.jetbrains.com/es-es/pycharm/features/. De la versión profesional os puede venir bien la "Capacidad para desarrollo remoto". Esta característica permite que todos los miembros de un grupo puedan desarrollar código a la vez, lo que es ideal para esta primera parte de las prácticas de la asignatura.

El problema está en que PyCharm Professional vale la friolera de unos cuantos cientos de dólares pero si eres estudiante te lo dan gratis. Solo tienes que seguir los pasos que se indican en https://www.jetbrains.com/es-es/community/education/#students. Si quieres, puedes solicitar la licencia rellenado los datos y pon como año de graduación 3 o 4 más del año actual. Recibirás un correo de "JetBrains account". Sigue las instrucciones del correo y recibirás un correo de "JetBrains sales" para activar tu paquete gratuito donde debes pulsar "click this link". Hecho todo esto, solo debes entrar en tu cuenta de JetBrains y descargar PyCharm Professional. Una vez instalado, inicia tu cuenta desde el propio programa para obtener el "code with me"[1].

Por simplicidad, en la asignatura se usará PyCharm Community.

1.2.3. Introducción a PyCharm

Esta sección muestra cómo crear en PyCharm un pequeño proyecto en Python para hacer nuestras primeras pruebas. Todo lo mostrado en esta sección podríamos realizarlo con un editor de texto simple (como el bloc de notas) y la línea de comandos. Sin embargo, veremos cómo el uso de un IDE nos facilita mucho las tareas de programación.

1. **Creación del directorio principal del proyecto, creación del directorio del proyecto fuente y activación del entorno virtual**

   - Seleccionar File > New Project
   - En la nueva ventana, en la opción Location pinchar sobre el icono 🗀
     - Seleccionar la carpeta donde crearás todos tus proyectos.
       En su caso selecciona New Folder
     - Cuando termine seleccionar OK

     Supondremos que la carpeta se llama 🗀 `Proyecto`.
   - En la misma ventana usa la opción Interpreter type: New environment
     - Selecciona el valor `Virtualenv`
     - En la opción Location
       - Pinchar sobre el icono 🗀
       - Selecciona New Folder y crea el directorio para el entorno virtual.
         Supongamos que se llama 🗀 `venv`
       - Seleccionar Open

       Los entornos virtuales (o virtualenv) nos permiten aislar configuraciones de Python independientes para cada proyecto (con librerías y versiones específicas). De no usarlos, tendríamos una única instancia de Python con un gran número de paquetes comunes a todos los proyectos. Esto también es muy útil en el caso de cometer errores a la hora de instalar librerías externas, evitando afectar a otros proyectos.
   - En la misma ventana, asegúrate que está activa la opción Create a welcome script.
   - Seleccionar Create.
   - En su caso, seleccionar Open Proyect con New Windows.

   Cuando finalices estos pasos se creará un nuevo proyecto donde verás el siguiente contenido:

   🗀 `Proyecto` ▸ `venv`
   🗀 `Proyecto` ▸ `main.py`

2. **Trabajar sobre el proyecto.**

[1] Estos pasos fueron facilitados por el alumno Pablo Roques Villa durante el curso 2021-2022

En 🗀 `Proyecto` deberá crear todo su código fuente. De todos los ficheros habrá uno que se llamará 🗀 `main.py` que será el módulo principal - lo construiste antes ¿recuerdas? Lo verás también porque en la barra superior aparece precisamente el nombre **main**▽

Para ejecutar el proyecto puedes usar estas alternativas.

- Pincha sobre ▶ en la barra superior
- Pincha sobre ▶ en el editor, en la linea donde aparezca `if __name__ == '__main__':`
- Selecciona Run > Run 'main'

3. **Desactivar el ambiente virtual.**
   La versión más sencilla de esta acción es cerrar el proyecto.

1.2.4. Documentar el Código del Proyecto

Documentar un programa es escribir las especificaciones de las abstraciones implementadas. Consiste en añadir información en el código fuente. Se realiza en forma de comentarios (docstrings) para explicar qué es lo que hace para que otros programadores que usen tu código entiendan su funcionamiento. Además, como ayuda a su entendimiento permite detectar errores y extender el programa para que tenga nuevas funcionalidades. Todo programa tiene errores (es cuestión de tiempo) y todo programa que tenga éxito será modificado en el futuro, por lo que la documentación es un paso fundamental.

Habrá situaciones en las que simplemente nos interese dejar comentarios en el código, sin generar documentación sobre ellos. Para ello, podemos usar tres comillas dobles al inicio y al final del bloque de código (o texto) a comentar, o bien hacer uso de la almohadilla para comentarios en una única línea. La # se usa para hacer comentarios entre líneas del código no trivial (no para comentar todo lo que se hace, pues puede dificultad la legibilidad del código).

En el caso de docstring, usaremos las tres comillas (para las definiciones de las clases y las especificaciones de los métodos junto con las aclaraciones sobres los mismos). Para documentar el proyecto con PyCharm con docstrings debe seguir los siguientes pasos:

1. Determinar cuál será el formato del Docstring.
   - PyCharm > Preferences > Tools > Python Integrated Tools
   - Seleccionar en Docstring el formato reStructuredText
2. Construir los docstring de cada función.
   El docstring es una especificación informal de la función. Consta de un pequeño resumen del objetivo de la función y una breve descripción de los parámetros de entrada y de salida. Para construirlos en PyCharm basta empezar una nueva línea con tres dobles comillas `"""` justo después de la signatura de la función, pulsar ⏎ y completar la información.
   Un ejemplo de docstring es el siguiente:

```
def check_fin(mensaje):
    """
    Realiza una pregunta de confirmación.
    Se dan dos opciones 1 y 2 en el mensaje de entrada.

    :param mensaje: El mensaje de la pregunta
    :return: True si se eligió 1, False en otro caso.
    """
    return '1' == input(f"{mensaje}\n1. sí\n2. no\n")
```

En el caso de querer indicar también el tipo de dato de los parámetros y retorno, deberás seguir los siguientes pasos:

1. Ir a la opción PyCharm > Preferences > Editor > General > Smart Keys > Python
2. Seleccionar Insert type placeholders in ...
3. Confirmar OK

Con esto es suficiente para que, durante el proceso de programación, puedas consultar la especificación de la función/método sin necesidad de ir a la parte del módulo donde escribiste el docstring. Si escribes una función documentada y colocas el cursor del ratón sobre el nombre de la función recién escrito verás su docstring.

En ocasiones nos interesa tener un documento completo con todas las especificaciones de todas las funciones para poder hacer un estudio más concienzudo de las funcionalidades. Aunque existen diferentes herramientas, tales como pydoctor, pdoc3 o Sphinx, nos centraremos en la primera ya que ofrece una forma sencilla de generar documentación del código Python creado. Para ello, se deberá seguir los siguientes pasos:

1. Instalar los nuevos módulos en el entorno virtual.
   Desde el IDE lo puedes hacer en File > Settings > Project: Nombre Proyecto > Python Interpreter. También está disponible en la vista principal del IDE, abajo a la derecha: Python 3.12 (Proyecto) > Manage Packages > Buscar e instalar.

- Asegúrate de que se tiene seleccionado el intérprete del entorno virtual, en Python Interpreter
- Seleccionar +, buscar pydoctor e instalar los paquetes.

2. Asegúrate de que tienen esta estructura de directorios:

   🗀 Proyecto, lugar donde está tus programas .py comentados con sus especificaciones.

   🗀 Proyecto ▸ docs, lugar donde estará toda la documentación referente a la resolución del problema (enunciados, ayudas, etc).

   🗀 Proyecto ▸ docs ▸ api, el lugar donde se generará la documentación del programa. Se asume que no contiene nada.

Una vez instalado pydoctor, debemos seguir los siguientes pasos:

1. Teclea en el terminal

```
$ cd Proyecto   # Asegúrate que estás en el directorio del proyecto
$ touch __init__.py  # Convertir el programa en módulo (importante)
$ pydoctor --make-html --html-output=docs/api --docformat=restructuredtext .
```

   Estas instrucciones accederán al directorio del proyecto, crearán un fichero .py de inicialización vacío, y lanzarán la herramienta pydoctor sobre el directorio actual (indicado con un punto al final de la instrucción), proporcionándole una serie de parámetros.

2. Visualizar la documentación.
   Con un navegador web abre el fichero 🗀 Proyecto ▸ docs ▸ api ▸ index.html

1.3 Programación imperativa en Python

Esta sección ofrece un resumen de los aspectos más relevantes del lenguaje de programación Python, siguiendo un enfoque de programación imperativa. Algunos aspectos más avanzados se pueden consultar en los anexos de esta sesión de prácticas.

1.3.1. Variables, Expresiones y Conversión de Tipo

Tipos de Datos. En programación imperativa se distinguen dos tipos de datos: elementales y compuestos.

- **Tipos de datos primitivos (o elementales)**
  - Formado por un único elemento.
  - Tipos: carácter, numérico, booleano, enumerado.
- **Tipos de datos compuestos**
  - Formado por una agrupación de elementos.
  - Tipos: string, array, registro, conjunto, lista, diccionario.

Variable y expresiones. Es importante enfatizar que declararemos las variables indicando de forma explícita su tipo de datos con el operador :. Tras ello, las asignaciones de valores se realizarán con =. Además, las variables usan la convención de nombres snake_case y cuando se quiere considerar que una variable es una constante su identificador tiene todos sus caracteres en mayúsculas. A continuación se muestra un ejemplo de representación para algunos de los tipos de datos más relevantes:

```
numero_entero: int = 2
numero_real: float = 10.01
caracter: str = '2'              # Se interpreta como un string
booleano: bool = True
string: str = " una cadena "     # Se interpreta como un string

# Asignación múltiple
numero_entero, numero_real, booleano= 2, 10.01, False

# Destrucción de variables
del(numero_entero)   # No es usual esta instrucción

# Las "constantes" se escriben en mayúsculas
PI: real = 3.1415      # La "constante" PI en Python. Ojo, puede cambiar su valor.
```

1.3.1.1. Tipos de datos primitivos

Se distinguen dos tipos de expresiones principales para trabajar con tipos de datos primitivos:

- Expresiones numéricas
  - Aritméticas: suma (+), resta y negación (-), multiplicación (*), división (/), división entera (//). módulo (%), exponente (**).
- Expresiones booleanas
  - Comparaciones: x == y, x != y, x > y, x >= y, x < y, x <= y
  - Operaciones: e1 and e2, e1 or e2, not e

Los resultados de las expresiones se guardan mediante la asignación. Las asignaciones no se consideran formalmente como expresiones, pero sí las construyen. P.e. x = x/10 ≡ x /= 10

Las asignaciones que usa Python son: =, +=, -=, *=, /=, %=, //=, **= (algunas son solo para expresiones numéricas).

Precedencia. El orden de precedencia de los operadores en Python se pueden consultar en la Figura 1.1.

Mutabilidad y Casting. Una variable es mutable si se puede cambiar el valor de la variable sin cambiar su referencia en memoria.

La instrucción `id`(var) muestra la referencia de la variable `var`. Esta instrucción nos permite comprobar la mutabilidad de una variable. Si se hacen dos asignaciones a una variable e `id`() cambia, la variable es inmutable. Números, booleanos y strings son inmutables.

El casting es el proceso por el que el valor de una variable se interpreta como otro tipo de dato. Por ejemplo, tenemos las funciones `int`(), `float`(), `str`() para realizar un casting explícito a entero, real y string, respectivamente.

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binario	Por la derecha	1
Identidad	+	Unario	—	2
Cambio de signo	-	Unario	—	2
Multiplicación	*	Binario	Por la izquierda	3
División	/	Binario	Por la izquierda	3
Módulo (o resto)	%	Binario	Por la izquierda	3
Suma	+	Binario	Por la izquierda	4
Resta	-	Binario	Por la izquierda	4
Igual que	==	Binario	—	5
Distinto de	!=	Binario	—	5
Menor que	<	Binario	—	5
Menor o igual que	<=	Binario	—	5
Mayor que	>	Binario	—	5
Mayor o Igual que	>=	Binario	—	5
Negación	**not**	Unario	—	6
Conjunción	**and**	Binario	Por la izquierda	7
Disyunción	**or**	Binario	Por la izquierda	8

Figura 1.1: Orden de Precedencia Fuente: `Introducción a la Programación con Python` (página 37)

1.3.1.2. Tipos de datos compuestos

En relación a los tipos (o estructuras) de datos compuestos, podemos clasificarlos en mutables e inmutables:

- Las **estructuras mutables** son aquellas cuyo estado puede cambiar una vez creadas.
- Las **estructuras inmutables** son aquellas cuyo estado no puede cambiar una vez creadas.

Python integra las siguientes estructuras de datos compuestas:

- Estructuras mutables. Se pueden cambiar sus términos una vez creada.
  - Listas: **secuencia** de elementos arbitrarios. Se escriben entre corchetes y se separan con comas.
    Ej: [1, "hola", 3].
  - Conjuntos: colección de elementos arbitrarios **únicos**. Se escriben entre llaves y se separan con comas.
    Ej: {1, "hola", 3}.
  - Diccionarios: conjuntos con objetos indexados. Cada elemento consta de un par clave: valor. La clave puede ser cualquier valor inmutable.
    Ej: ["a": 1, "b": "hola", "c": 3].
- Estructuras inmutables. No puede cambiar sus términos una vez creada.
  - Strings: **secuencia** de valores que representan códigos Unicode. Se escriben entre comillas.
    Ej: “hola”, ’adiós’
  - Tuplas: **secuencia** de elementos arbitrarios. Se escriben entre paréntesis y se separan con comas.
    Ej: (1, "hola", 3).
  - Rangos: **secuencias** que se construyen con range([start,] stop [[, step]]).
    Ej: range(1:10:2).
  - Conjuntos congelados (Frozensets). La versión inmutable de los conjuntos.
    frozenset({1, "hola", 3})

Las secuencias o tipos de datos secuenciales (o en secuencia) representan a colecciones finitas de datos referenciados por un índice. Alternativamente, una secuencia esta formada por una colección de objetos, o términos, indexados con

índices 0, 1, 2 ... Es importante destacar que todos aquellos tipos de datos que se basan en secuencias (tuplas, rangos, listas) tienen una serie de propiedades comunes que nos permiten interactuar con sus elementos:

Se puede acceder a sus elementos de distintas formas:

- s[i], el elemento i-esimo de s,
  Cuenta +1 desde el 0 si $i \geq 0$, o cuenta -1 desde la len(s)-i si $i < 0$.
- s[i:j], la rebanada (slice) de s desde i hasta j. Selecciona todos los elementos con índice t tal que $i \leq t < j$.
- Algunas secuencias también admiten la **"división ampliada"** con un tercer parámetro de "paso": [i:j:k] selecciona todos los elementos con índice t donde $t = i + n \times k$, $0 \leq n$ e $i \leq t < j$.

Funciones comunes

- len(s), que devuelve el número de elementos de una secuencia.
  Si len(s)==n, el conjunto de índices es $\{0, 1, \ldots, n-1\}$.
- min(s) / max(s): el item más pequeño/grande de s.
- sum(s): la suma de los elementos de s.
- sorted(s): una lista con los elementos ordenados de s.
- enumerate(s): un enumerado de la lista s.
- any(s): devuelve True si bool(x) es True para cualquiera de los elementos de la secuencia.
- all(s): devuelve True si bool(x) es True para todos los elementos de la secuencia.

Métodos comunes

- s.count(x): el número total de ocurrencias de x en s.
- s.index(x[, i[, j]])): el índice de x en s [después de i [y antes de j]]

Operaciones comunes en estos elementos son:

- x `in` s: Es True si `x` es un item de `s`.
- x `not in` s: Es True si `x` no es un item de `s`.
- `s + t`: concatena `s` y `t`.
- `n * s`: añade `s` un total de `n`-veces (en algunas secuencias).

A continuación se muestra en detalle cada uno de los tipos compuestos enumerados previamente.

Strings. Construcción: "", '', "cadena", 'cadena', str(), str("cad").

Un **string** es una cadena o secuencia de valores que representan códigos Unicode y se escriben entre comillas dobles. Python no tiene un tipo char y en su lugar cada carácter se representa como un objeto string con longitud 1.

Funciones asociadas a los caracteres son:

- ord(char): retorna el código decimal de un char.
- chr(codigo): retorna el char dado el código (número natural) con la función.

Son métodos de las cadenas:

- .lower()/.upper(): Retorna una copia de la cadena de caracteres con todas las letras en **minúsculas / mayúsculas**.
- .capitalize(): Retorna una copia de la cadena con el **primer carácter** en mayúsculas y el resto en minúsculas.
- .title(): Igual pero el primer carácter **de cada palabra** de la cadena.
- .casefold(): Retorna el texto de la cadena, normalizado a minúsculas. Los textos **normalizados** pueden usarse para realizar búsquedas textuales independientes de mayúsculas, minúsculas y caracteres idiomáticos.
- .count(sub[, start[, end]]): Retorna el número de **ocurrencias** no solapadas de la cadena sub en el rango [start, end].
- .find(sub[, start[, end]]): Retorna el menor índice de la cadena s donde se puede **encontrar** la cadena sub, considerando solo el intervalo s[start:end]
- .split(sep=None, maxsplit=-1): Retorna los distintos substring comprendidos entre dos separadores. El separador es el valor de sep. El parámetro maxsplit indica el máximo número de divisiones. Si no se indica ningún separador elimina todos los espacios y devuelve sólo las palabras que conforman la cadena.
- .join(lista): retorna un string formado por los elementos de la lista.

Hay muchas más funciones en `https://docs.python.org/es/3/library/stdtypes.html#text-sequence-type-str`.

Ejemplo 1.1

```
# Definir la cadena de texto con espacios
cadena: str = ' tengo espacios '

# Mostrar la cadena y su longitud
print(cadena, "len=", len(cadena))

# Dividir la cadena en una lista
cadena_split: list[str] = cadena.split()

# Unir una lista de palabras con la cadena original
cadena_join: str = cadena.join(['ahora', 'extras'])

# Definir las variables enteras
i: int = 2
j: int = 11
k: int = 4

# Mostrar diferentes slices de la cadena
print(cadena[i])  # Mostrar el elemento en la posición i=2
print(cadena[i:j])  # Mostrar los elementos entre i=2 hasta 10<j=11
print(cadena[i:j:k])  # Mostrar los elementos en saltos de k=4
print(cadena[-i])  # Mostrar el elemento en la posición i=-2 (equivalente a posición 15)
print(cadena[-j:-i])  # Mostrar los elementos entre j=-11 (posición 1) hasta i=-2 (sin incluir)
print(cadena[j:i:-k])  # Mostrar los elementos de forma inversa desde j=11 hasta i=2 con saltos de k=4
```

Listas. Construcción: [], [x, y, ...], list() o list(iterable). *Por comprehensión en Anexo III*

Una lista es una secuencia ordenada de elementos arbitrarios que es mutable.

El constructor list() construye una lista vacía. También se construye con un par de corchetes. Los demás constructores crean listas con elementos.

Ejemplo 1.2

```
# Definir la lista de cadenas
lista: list[str] = ['uno', 'dos', 'tres']

# Mostrar la lista y su longitud
print(lista, "len=", len(lista))
```

En el caso de tener una lista donde todos los elementos son del mismo tipo, lo indicaremos en la declaración explícita de la lista. Si contiene diferentes elementos, la declaración explícita indicará únicamente "list".

También se puede usar el operador * si deseas repetir una lista varias veces.

```
# Definir las listas de cadenas
l1: list[str] = ['uno', 'dos']
l2: list[str] = ['tres', 'cuatro']

# Concatenar las dos listas: ['uno', 'dos', 'tres', 'cuatro']
resultado_concatenacion: list[str] = l1 + l2

# Repetir la lista l1 tres veces: ['uno', 'dos', 'uno', 'dos', 'uno', 'dos']
resultado_repeticion: list[str] = 3 * l1
```

Dispone de las operaciones/funciones:

- s[i] = x. El elemento i de s es reemplazado por x.
- s[i:j] = t. La rebanada de valores de s que van de i a j es reemplazada por el contenido del iterador t.
- del s[i:j]. Equivalente a s[i:j] = [].
- s[i:j:k] = t. Los elementos de s[i:j:k] son reemplazados por los elementos de t.
- del s[i:j:k]. Borra los elementos de s[i:j:k] de la lista.

Y los siguientes métodos:

- .append(x) **Añade** un valor. Alternativametne lista = l1 + l2 concatena listas.
- .insert(i, x), **inserta** un ítem en una posición dada. Ej: [0, 1, 2].insert(1, 10) == [0, 10, 1, 2]
- .extend(iterable), **añade** cualquier objeto iterable a la lista.
- .pop([x]), **extrae y quita** el último elemento o el elemento indicado.

- .remove(x), **elimina** el primer elemento que sea igual a x.
- .index(x[,start[,end]]), **indica** el índice del elemento x en el rango dado.
- .sort(*, reverse=False), **ordena** los elementos de la lista in situ.

```
# Crea la lista de cadenas y números
lista: list = ['uno', 'dos', 'tres']
print(lista)

# Añade elementos
lista = lista + [4]
print(lista)

lista.append(5)
print(lista)

lista.insert(0, 'cero')
print(lista)

# Elimina elementos
lista.remove('dos')  # Por valor
print(lista)

del lista[0]  # Por índice
print(lista)

lista.pop()  # Extrae y quita el último elemento
print(lista)

lista.pop(1)  # Extrae y quita el elemento en la posición 1
print(lista)

# Encontrar elementos
indice_dos: int = lista.index('dos')  # Buscar el índice de 'dos'
print("Indice de 'dos':", indice_dos)

# Verificar si un elemento está en la lista
esta_dos: bool = 'dos' in lista
esta_4: bool = 4 in lista

print("'dos' está en la lista:", esta_dos)
print("4 está en la lista:", esta_4)
```

Ten en cuenta que los elementos de una lista son arbitrarios, por lo que puede contener elementos de distintos tipos:

```
# Crea una lista que contiene cadenas, enteros, y una lista anidada
lista: list = ['uno', 2, ['tres', 4]]

# Mostrar la lista completa
print(lista)

# Acceder al segundo elemento de la lista (un entero)
print(lista[1])

# Acceder al tercer elemento de la lista (una lista anidada)
print(lista[2])

# Acceder al primer elemento de la lista anidada
print(lista[2][0])
```

Por simplicidad, cuando queramos trabajar con una lista con varios tipos de elementos, la definiremos de tipo `list`. Si quisiésemos ser puristas, debería tener tipo `list[object]`, pues reflejará cualquier tipo de datos. No obstante, nos abstraeremos en la asignatura por simplicidad.

Tuplas. Construcción: (x, y, ...) o tuple(iterable). *Por comprehensión en Anexo III.*

La versión inmutable de las listas son las tuplas. Una tupla es una secuencia ordenada de elementos arbitrarios. Se declara con apertura de paréntesis, seguido de los elementos separados por comas y cierre de paréntesis. Una tupla vacía está formada por un par de paréntesis. Una vez creada no se puede modificar pero sí se puede consultar su contenido usando índices o el operador in.

Cuando realicemos una declaración explícita de una tupla, por simplicidad indicaremos únicamente "tuple", ya que si no tendríamos que especificar el tipo de cada uno de los elementos que componen la tupla.

Las tuplas son más rápidas que las listas y como solo se puede iterar entre ellos, es mejor usar una tupla que una lista si se va a trabajar con un conjunto estático de valores. Se pueden convertir tuplas en listas y viceversa.

- tuple() recibe como parámetro una lista y devuelve una tupla.
- list() recibe como parámetro una tupla y devuelve una lista.

Ejemplo 1.3

```
# Definir una tupla de cadenas
tupla: tuple = ('uno', 'dos', 'tres')
print(type(tupla))

# Definir una lista de cadenas
lista: list = ['uno', 'dos', 'tres']
print(type(lista))

# Convertir la lista en una tupla
t: tuple = tuple(lista)
print(type(t))

# Convertir la tupla en una lista
l: list = list(tupla)
print(type(l))

# Mostrar la tupla y su longitud
print(tupla, "len=", len(tupla))

# Verificar si el valor 2 está en la tupla
print(2 in tupla)

# Acceder al segundo elemento de la tupla
print(tupla[1])
```

Rangos. Construcción: range([start,] stop[, step])

El tipo range representa una secuencia inmutable de números y se usa usualmente para bucles que se deben de ejecutar un número de veces (bucles for). Un rango se construye con uno, dos o tres naturales:

- range(stop) construye un rango que empieza en 0 y finaliza en `stop`. Cada elemento de la secuencia se diferencia del anterior en una unidad.
- range(start, stop) construye un rango que empieza en `start` y finaliza en `stop`. Cada elemento de la secuencia se diferencia del anterior en una unidad.
- range(start, stop, step) construye un rango que empieza en `start`, finaliza en `stop` y cada elemento de la secuencia se diferencia del anterior en una `step`-unidades.

Ejemplo 1.4

```
tuple(range(10))
tuple(range(10, 100, 20))
tuple(range(100, 10, -20))
```

Nota *"Los rangos implementan todas las operaciones comunes de las secuencias, excepto la concatenación y la repetición. La ventaja de usar un objeto de tipo range en vez de uno de tipo list o tuple es que con range siempre se usa una cantidad fija (y pequeña) de memoria, independientemente del rango que represente (Ya que solamente necesita almacenar los valores para start, stop y step, y calcula los valores intermedios a medida que los va necesitando)".*

Conjuntos. Construcción: {}, {x, y, ...}, set() o set(iterable). *Por comprehensión en Anexo III.*

Estos tipos de datos representan a conjuntos no ordenado de elementos (objetos únicos). Sus términos no pueden ser indexados por ningún subíndice. La función incorporada len() devuelve el número de elementos en un conjunto. Se utilizan para pruebas rápidas de pertenencia, la eliminación de elementos duplicados de una secuencia y el cálculo de operaciones matemáticas como la intersección, unión, diferencia y diferencia simétrica. Se tienen dos tipos de conjuntos:

- Conjuntos (Set)
  Representan un conjunto mutable. Se declara con apertura de llaves, los elementos separados por comas y cierre de llaves. Usar solo la apertura y cierre de llaves sin elementos define un diccionario, no un conjunto. Para definir un conjunto vacío usa el constructor set().
- Conjuntos congelados (Frozensets)
  Representan un conjunto inmutable. Se crean con el constructor frozenset().

Ejemplo 1.5

```
# Definir un conjunto mutable de cadenas
conjunto_mutable: set = {"Perl", "Python", "Java"}
print(conjunto_mutable)

# Crear un conjunto mutable a partir de una tupla
conjunto_mutable_desde_tupla: set = set(("Perl", "Python", "Java"))
print(conjunto_mutable_desde_tupla)

# Convertir el conjunto mutable en un conjunto inmutable (frozenset)
conjunto_inmutable: frozenset = frozenset(conjunto_mutable)
print(conjunto_inmutable)
```

Nota *Los conjuntos no pueden estar formados por elementos mutables.*

```
set( {1: 2, 3: 4} )
set( [1   , "h", (0, (3, 4)) ] ); set((0, 1)) # ok
set(([0], [1])) # no ok. Los elementos son listas y éstas son mutables.
```

Algunos operadores que se tienen con conjuntos son:

- conjA - conjB realiza la diferencia del conjA con el conjunto conjB.
- conjA -= conjB calcula la diferencia del conjA con el conjunto conjB y el resultado se almacena en conjA.
- conjA | conjB calcula la unión del conjA con el conjunto conjB.
- conjA |= conjB calcula la unión del conjA con el conjunto conjB y el resultado se almacena en conjA.
- conjA & conjB calcula la intersección de los conjA y conjB.
- conjA &= conjB calcula la intersección del conjA con el conjunto conjB y el resultado se almacena en conjA.
- conjA <= conjB indica si el conjA es un subconjunto del conjunto conjB.
- conjA < conjB indica si el conjA es un subconjunto propio del conjunto conjB.
- conjA >= conjB indica si el conjA es un superconjunto del conjunto conjB.
- conjA > conjB indica si el conjA es un superconjunto propio del conjunto conjB.

Los métodos equivalentes a estos operadores son:

- .difference(): retorna la diferencia de dos conjuntos. conjA.difference(conjB) Alternativamente se puede usar el operador -.
- .difference_update(): retorna la diferencia de dos conjuntos. conjA.difference(conjB). Alternativamente se puede usar el operador -=.
- .union(): la unión de conjuntos. conjA.union(conjB). Alternativamente se puede usar el operador |.
- .update(): modifica el conjunto con la unión de conjuntos y el resultado lo almacena en el primer conjunto. conjA.update(conjB). Alternativamente se puede usar el operador |=.
- .intersection(): la intersección de conjuntos. conjA.intersection(conjB). Alternativamente se puede usar el operador &.
- .intersection_update(): modifica el conjunto con la intersección de conjuntos. Alternativamente se puede usar el operador &=.
- .issubset(): indica si un conjunto es subconjunto de otro. conjA.issubset(conjB). Alternativamente se puede usar el operador <=.
- .issuperset(): indica si un conjunto es un superconjunto de otro. conjA.issuperset(conjB). Alternativamente se puede usar el operador >=.

Otros métodos que se pueden realizar con los conjuntos son:

- .add(x): **añadir** un elemento x.
- .discard(x): **elimina** el elemento x.
- .remove(x): igual que .discard() **pero** si el elemento no existe aparecerá un mensaje de error.
- .clear(): **borrar** todos los elementos.
- .copy(): realizar una **copia** del conjunto.
- .isdisjoint(conj): indica si conj **es disjunto** con el conjunto actual.
- .pop(): **borra y retorna** un elemento arbitrario del conjunto.

Diccionarios Construcción: {}, {k1: v1, k2: v2, ...}, dict(), o dict(iterable). *Por comprehensión en Anexo III*.

Los mapeos representan a conjuntos finitos de objetos pero indexados. La notación de subíndice m[k] selecciona el elemento indexado por k en el mapeo m. Los índices no tienen que ser necesariamente números naturales. Para ello se utilizan tablas hash y políticas de resolución de colisiones.

De forma nativa Python implementa los **diccionarios**, que son su forma de llamar a los mapeos. Representan a una colección mutable y no ordenada de pares clave-valor. La clave puede ser cualquier valor inmutable (ese es el valor que indexa en el mapeo). La razón es que la implementación eficiente de los diccionario requiere una clave que permanezca constante.

Se dispone de la función len() que devuelve el número de elementos en un mapeo. Se pueden construir diccionarios vacíos con {} y dict(). Mediante asignación se define un diccionario usando llaves e indicando los pares (clave, valor):

```
dic = {
    <key>: <value>,
    <key>: <value>,
      .
      .
    <key>: <value>
}
```

Ejemplo:

```
dic = {
    'Madrid': 'España',
    'París': 'Francia',
    'Londres': 'Inglaterra'
}
```

Mediante el constructor se pasa como argumento una lista con tuplas (clave, valor):

```
dic = dict([
    (<key>, <value>),
    (<key>, <value>),
      .
      .
    (<key>, <value>)
])
```

Ejemplo:

```
dic = dict([
    ('Madrid': 'España'),
    ('París': 'Francia'),
    ('Londres': 'Inglaterra')
])
```

En el caso de que las claves sean string se puede simplificar esta segunda opción quitando la notación de tuplas y realizando la asignación `clave = valor`.

```
dic = dict(
    <key>=<value>,
    <key>=<value>,
      .
      .
    <key>=<value>
)
```

Ejemplo:

```
dic = dict(
    Madrid='España',
    París='Francia',
    Londres='Inglaterra'
)
```

Operadores con diccionarios son:

- d[key] = value: **Asigna** el valor value a d[key].
- len(d): retorna el número de entradas almacenadas; es decir, el número de pares (clave, valor).
- del d[k]: borra la clave k junto con su valor.
- key `[not] in` d: indica si la clave [no] está en d.

Métodos con diccionarios son:

- .setdefault(key,[v]): **Retorna** el valor de key, pero si key no existe le asigna el valor v. **Muy útil** para inicializaciones.
- copy(): realiza una copia del diccionario.
- .keys() y .values() retornan la lista de **claves** y **valores**, respectivamente.
- .get(k): retorna **el valor de la clave** dada.
- .items(): retorna una **lista** cuyos elementos son tuplas **(clave, valor)**.
- .popitem(): **Borra el último** par clave-valor añadido al diccionario y lo retorna como tupla.
- .pop(k): **Retorna** el valor de la clave dada **y elimina** del diccionario el par (key,pop(key)).
- .update(otroDict): **fusiona** las claves y valores de los diccionarios. Sobreescribe los valores que tengan la misma clave.
- .clear(): **Limpia** el diccionario de pares clave-valor.

```python
# Definir un diccionario con claves de tipos mixtos
d: dict = {1: 'value', 'key': 2}
print(d)

# Mostrar las claves del diccionario
print(d.keys())

# Mostrar los pares clave-valor del diccionario
print(d.items())

# Acceder a los valores usando las claves
print("d[1] =", d[1])
print("d['key'] =", d['key'])

# Crear diccionarios usando dict() a partir de listas de listas/tuplas
d1: dict = dict([[1, 2], [3, 4]])
print(d1)

d2: dict = dict([(3, 26), (4, 44)])
print(d2)

# Actualizar d1 con los pares clave-valor de d2
d1.update(d2)
print(d1)
```

Puede interesar defaultdict de collections. Ej: d = defaultdict(lambda: "No existe"). Si d[clave] no existe invocará a la función dada pero no lanzará un error.

1.3.2. Programación Estructurada Secuencial

Consta de una secuencia de órdenes directas. El conjunto de instrucciones **vienen dadas por el lenguaje** de programación.

- Sentencias de Asignación, consistentes en el paso de valores de una expresión o literal a una zona de la memoria.
- Lectura, `input()`, consistente en recibir desde un dispositivo de entrada algún dato (usualmente, el teclado).
- Escritura, `print()`, consiste en mandar a un dispositivo de salida algún valor (normalmente la pantalla). La función print() permite escribir el texto de diferentes formas, indicamos solo algunas:
  - **Print básico con varios argumentos**. Puede recibir múltiples argumentos y los separa automáticamente por un espacio.

    ```python
    print("Hola", "mundo", 123)
    # Salida: Hola mundo 123
    ```

  - **Uso de concatenación con el operador +**. Se pueden concatenar strings usando el operador +, pero debes asegurarte de que todos los elementos sean cadenas de texto.

    ```python
    nombre = "Sergio"
    print("Hola, " + nombre)
    # Salida: Hola, Sergio
    ```

  - **Uso de comas para separar elementos**. Al usar comas, la función automáticamente convierte los elementos en strings y los separa con un espacio.

    ```python
    edad = 25
    print("Tengo", edad, "años")
    # Salida: Tengo 25 años
    ```

  - **Uso de f-strings (cadenas formateadas)**. Introducido en Python 3.6, f-strings permiten insertar variables directamente en las cadenas utilizando llaves . Es la forma más moderna y recomendada para componer cadenas con texto y variables intercaladas.

    ```python
    nombre = "Alicia"
    edad = 25
    print(f"Hola, me llamo {nombre} y tengo {edad} años")
    # Salida: Hola, me llamo Alicia y tengo 25 años
    ```

- Generación de números aleatorios. Muy útil en programación. Debemos importar la librería random para que funcione. Destacamos algunas de las formas más comunes:
  - **Generar un número entero aleatorio dentro de un rango**. Usa random.randint(a, b) para generar un número entero aleatorio entre a y b, ambos incluidos.

    ```python
    import random
    numero_aleatorio: int = random.randint(1, 10)
    ```

```
print(numero_aleatorio)
# Salida: Un número entre 1 y 10, por ejemplo 7
```

- **Generar un número real aleatorio entre 0 y 1**. Usa random.random() para obtener un número flotante entre 0.0 y 1.0.

```
import random
numero_aleatorio: float = random.random()
print(numero_aleatorio)
# Salida: Un número flotante entre 0.0 y 1.0, por ejemplo 0.5678
```

- **Generar un número real en un rango específico**. Usa random.uniform(a, b) para generar un número flotante entre a y b.

```
import random
numero_aleatorio: float = random.uniform(1.5, 5.5)
print(numero_aleatorio)
# Salida: Un número flotante entre 1.5 y 5.5, por ejemplo 3.725
```

- Tamaño, `len`(), calcula el número de datos que tiene una secuencia (p.e. un string, una lista).
- Identificación, `id`(), retorna la referencia de una variable.
- Tipo, `type`(), indica el tipo de dato de un literal o de una variable.
- ...

En Python tenemos:

- **Sentencias de asignación:** `https://docs.python.org/3/reference/simple_stmts.html`
- **Built-in Functions:** `https://docs.python.org/3/library/functions.html`

1.3.3. Programación Estructurada Condicional

Es aquella que ejecuta ciertas órdenes si se cumple una condición booleana.
En Python se usa la sentencia compuesta con cláusulas if, elif, else.

- Condicional simple.

```
if condicion:
    estructura
```

```
if condicion: sentencia  # Inline
```

- Condicional doble.

```
if condicion:
    estructura_if
else:
    estructura_else
```

```
var = exp_si_true  if  condicion else exp_si_false # Inline
```

- Condicional anidado.

```
if condicion1 [op condicion2 [op condicion3] ... ]:
    estructura_if
elif condicion:
    estructura_else_if
else  # Casi obligado si se usa elif.
    estructura_else
```

1.3.4. Programación Estructurada Iterativa

Bucle while. La versión imperativa de esta estructura consta de los siguientes pasos:

1. Se parte de una variable, que se llamará variable de control y que se inicializará a cierto valor.
2. Entonces se comprueba una condición booleana donde interviene la variable de control.
3. Si la condición es cierta, entonces se ejecutarán nuevas estructuras.
4. Entres las estructuras habrá alguna secuencial que modifique la variable de control.
5. Se vuelve al paso 2.

El proceso se repite hasta que la variable de control tome un valor que hace que la condición booleana sea falsa.
En Python se usa la sentencia while:

```
control = valor_inicial
while expresion_booleana_con_la_var_de_control:
    estructuras
    modificar la variable de control
else: # opcional
   estructuras
```

El bloque else no se realizará si se ejecutara la sentencia break en el bloque while.

Bucle for. La Abstracción de Iteración permite recorrer elementos de un contenedor **sin tener en cuenta** su representación interna. De forma transparente al programador se recurre a un iterador que sabe cómo recorrer al contenedor. La abstracción tiene la forma:

```
Para cada elemento P de Contenedor
    acción sobre P
```

En Python algunos contenedores se llaman iterables.

- Todas las secuencias estudiadas previamente son objetos iterables.
- Los iterables responden a la abstracción de iteración.
- La abstracción se particulariza como:

```
for x in contenedor:
    acción con x
[else : acciones]
```

Si bien para el caso de diccionarios, se tienen estas opciones:

```
for k[,v] in diccionario:
    acción con k [,v]
[else : acciones]
```

```
for x in diccionario.values():
    acción con x
[else : acciones]
```

Instrucciones para controlar la iteración. Existen dos sentencias, break y continue, que pueden ser útiles a la hora de programar. Sin embargo, pueden llevar a malas prácticas en programación, alterando el flujo de ejecución del programa. En la asignatura no se permite su uso, salvo en situaciones muy concretas.

- break
  - Solo puede ocurrir sintácticamente en un bucle for o while.
  - Terminará el bucle adjunto más cercano y omitirá la cláusula opcional else.
- continue
  - Solo puede ocurrir sintácticamente en un bucle for o while.
  - Continúa con la siguiente iteración del bucle más cercano.
  - No ejecutará lo que aparezca después de continue.

Programación Procedimental

Una función es una **secuencia** de instrucciones identificada con un **nombre** que retorna un valor. En Python una función puede retornar varios valores y, en este caso, "empaqueta" todos los datos de retorno en una tupla.

```
def nombre_funcion ( lista de parámetros ) -> tipo de dato de retorno:
   estructuras de la función
   return valores
```

Un procedimiento es una función que no tiene la instrucción de retorno.

```
def nombre_procedimiento ( lista de parámetros ) [-> None]:
   estructuras del procedimiento
```

Si una función/método tiene n-parámetros podemos invocar a la función con n-argumentos de tal forma que el 1^{er} argumento se sustituya por el 1^{er} parámetro, el 2^{o} argumento por el 2^{o} parámetro, etc ... Son parámetros posicionales.

```
# Definir una función con 4 parámetros
def fun(a: int, b: int, c: int, d: int) -> None:
    print(a, b, c, d)

# Invocar la función con 4 parámetros posicionales
fun(1, 2, 3, 4)
```

Es importante destacar que podríamos eliminar el tipo de retorno None (nada) y funcionaría perfectamente.

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
```

Python permite invocar por palabras claves (keywords).

- Usar **keyword =** especifica el nombre del parámetro en la invocación.
- El orden de los parámetros pueden cambiarse.
- En la declaración de la función, los keywords siempre se pondrán al final.

```
# Para la función anterior
fun(b=2, d=4, a=1, c=3) # Invocamos con 4 keywords.
fun(1, 2, d=4, c=3) # Los keywords al final
fun(d=4, 1, 2, c=3) # Incorrecto
```

Cuando no se conoce el número de argumentos que se usarán, se usa el Packing Arguments (empaquetamiento de argumentos) con el operador *.

```
def fun(*args):   # Empaquetará todos los argumentos
    print(args)

fun(1, 2, 3, 4)   # Invocación desempaquetada
```

En el ejemplo, todos los argumentos se agrupan en una tupla.

También existe el Unpacking Arguments. Dada una función/método con varios parámetros podemos empaquetar los argumentos con *.

```
# Definir una función con 4 parámetros de tipo int
def fun(a: int, b: int, c: int, d: int) -> None:
    print(a, b, c, d)

# Crear una lista con 4 enteros
lista: list[int] = [1, 2, 3, 4]

# Invocar la función utilizando el desempaquetado de la lista
fun(*lista)  # Invocación empaquetada
```

Para acceder a uno de los argumentos empaquetados se usan índices:

```
def fun(*args):
    print(len(args), args[1]) # Muestra el cardinal y el 2o argumento

fun(1, 2, 3, 4)
```

Estaría mejor acceder a ellos con un nombre (keyword). De hecho, podemos empaquetar y usar keywords usando **diccionarios**, con **.

```
def fun(**kwargs):
    print(f"{len(kwargs)} elementos", kwargs) # Muestra el diccionario

fun(a=1, b=2, c=3, d=4)     # Invocación con keywords
diccionario = {'p1': 1, 'p2': 2, 'p3':3}  # Los veremos
fun(**diccionario)          # Invocación con empaquetado
```

Si se quieren usar los tres modos de pasar argumentos, el orden debe ser:

1. posicionales
2. empaquetados sin keyword
3. empaquetados con keyword

```
def fun(a: int, b: int, *args: int, **kwargs: int):
    print(a, b)  # Muestra los posicionales
    print(args)  # Muestra los empaquetados sin keyword
    print(kwargs)# Muestra los empaquetados con keyword

fun(1, 2, 3, 4, 5, p1=6, p2=7, p3=8)
```

1.4 Relación de ejercicios

1. **Tipos de datos primitivos y operadores**. Escribe un programa que pida al usuario dos números enteros. El programa debe realizar las siguientes operaciones: suma, resta, multiplicación, división y módulo entre ambos números. Luego realiza la siguiente operación $3 + 5 * 2^2/(4 - 1)$ y muestra el resultado por pantalla.
   Después de esto, compara ambos números utilizando las siguientes expresiones booleanas y muestra los resultados:
   - ¿El primer número es mayor que el segundo?
   - ¿El primer número es igual al segundo?
   - ¿El primer número es distinto al segundo?

   Asegúrate de que tu código contiene:
   - Al menos un comentario de una línea explicando brevemente lo que hace el programa.
   - Un comentario de varias líneas.

   Ejemplo de entrada del usuario:

   ```
   Ingresa el primer número: 4
   Ingresa el segundo número: 2
   ```

   Salida esperada:

   ```
   Suma: 6
   Resta: 2
   Multiplicación: 8
   División: 2.0
   Módulo: 0
   Resultado de 3 + 5 * 2 ** 2 / (4 - 1): 9.333333333333334
   ¿El primer número es mayor que el segundo? True
   ¿El primer número es igual al segundo? False
   ¿El primer número es distinto al segundo? True
   ```

2. **Casting entre tipos primitivos**. Crea un programa que reciba un valor numérico en formato string desde input(), lo convierta a entero y luego a flotante, y realice una operación matemática con él (por ejemplo, multiplicarlo por 1.5). Muestra el resultado de cada conversión y operación.
   Ejemplo de entrada del usuario:

   ```
   Ingresa un valor numérico: 2
   ```

   Salida esperada:

   ```
   Valor convertido a entero: 2
   Valor convertido a flotante: 2.0
   Resultado de multiplicar el valor por 1.5: 3.0
   ```

3. **Trabajando con strings (inmutables)**. Escribe un programa que reciba una frase del usuario y realice las siguientes operaciones:
   - Convertir toda la frase a mayúsculas usando upper().
   - Convertir toda la frase a minúsculas usando lower().
   - Aplicar capitalize() a la frase y mostrar el resultado.
   - Aplicar title() a la frase para capitalizar cada palabra.
   - Usar casefold() para comparar la frase original con una versión en minúsculas de la misma, verificando si son iguales.
   - Solicitar al usuario una letra y mostrar el código ASCII de esa letra utilizando ord(), y luego mostrar el carácter correspondiente a un código ASCII ingresado por el usuario utilizando chr().
   - Contar cuántas veces aparece una letra específica en la frase (la letra debe ser introducida por el usuario) utilizando count().

- Usar find() para buscar la posición de una palabra dentro de la frase.
- Dividir la frase en palabras usando split(), mostrar la palabra más larga, y luego unir las palabras de nuevo usando join() para reconstruir la frase.

Ejemplo de entrada del usuario:

```
Ingresa una frase: El sol brilla sobre el mar
Ingresa una letra para buscar su código ASCII: E
Ingresa un código ASCII para convertir a carácter: 97
Ingresa una letra para contar en la frase: l
Ingresa una palabra para buscar su posición en la frase: mar
```

Salida esperada:

```
Frase en mayúsculas: EL SOL BRILLA SOBRE EL MAR
Frase en minúsculas: el sol brilla sobre el mar
Frase con la primera letra en mayúscula: El sol brilla sobre el mar
Frase con cada palabra en mayúscula: El Sol Brilla Sobre El Mar
Comparación de la frase original con su versión en minúsculas: True
Código ASCII de la letra 'E': 69
Carácter correspondiente al código ASCII 97: a
La letra 'l' aparece 3 veces en la frase.
La palabra 'mar' se encuentra en la posición: 24
Palabra más larga en la frase: brilla
Frase reconstruida: El sol brilla sobre el mar
```

4. **Listas (mutables)**. Define una lista con cinco números de forma manual. Por ejemplo, con los valores [10, 20, 30, 40, 50]. Luego, realiza las siguientes operaciones en el siguiente orden:
   - Consultar el valor del tercer elemento de la lista.
   - Obtener una porción (slice) de la lista que contenga los elementos desde la posición 1 a la 3 (sin incluir la posición 3).
   - Agregar un nuevo número al final de la lista (por ejemplo, el 60).
   - Eliminar el segundo número de la lista.
   - Generar un número aleatorio y añadirlo como primera posición de la lista.
   - Ordenar la lista de mayor a menor.

   Ejemplo de salida:

```
Lista original: [10, 20, 30, 40, 50]
Valor del tercer elemento: 30
Slice de la lista (elementos 1 a 3): [20, 30]
Lista después de agregar un número: [10, 20, 30, 40, 50, 60]
Lista después de eliminar el segundo elemento: [10, 30, 40, 50, 60]
Lista ordenada de mayor a menor: [60, 50, 40, 30, 10]
Lista después de añadir un número aleatorio en la primera posición:
[3, 60, 50, 40, 30, 10]
Lista final después de todas las operaciones: [3, 60, 50, 40, 30, 10]
```

5. **Tuplas (inmutables)**. Define manualmente una tupla que contenga los nombres de cuatro ciudades. Luego, realiza las siguientes operaciones:
   - Accede al segundo valor de la tupla y muéstralo por pantalla.
   - Convierte la tupla en una lista utilizando `list`() para poder ser modificada.
   - Cambia el valor de la tercera ciudad por otro nombre de ciudad de tu elección. Tras ello, utilizando `print`(), `type`() y `len`(), muestra el contenido actualizado de la lista, el tipo de la estructura y su longitud después de la modificación, en una única línea.

- Convierte la lista modificada de nuevo en una tupla utilizando `tuple()`. Nota importante: especifica la nueva variable de tipo tupla únicamente con el tipo explícito `tuple`, pues la lista que convirtamos podría tener datos de diferentes tipos (enteros, string, etc).
- Muestra por pantalla el contenido de la tupla, su tipo y longitud.

Ejemplo de salida:

```
Tupla original: ('Madrid', 'Barcelona', 'Valencia', 'Sevilla')
La segunda ciudad es: Barcelona
Lista convertida desde la tupla:
['Madrid', 'Barcelona', 'Valencia', 'Sevilla']
Lista actualizada:
['Madrid', 'Barcelona', 'Malaga', 'Sevilla'], Tipo: <class 'list'>, Longitud: 4
Tupla después de la reconversión:
('Madrid', 'Barcelona', 'Malaga', 'Sevilla'), Tipo: <class 'tuple'>, Longitud: 4
```

6. **Conjuntos (mutables)**. Escribe un programa que defina dos conjuntos de números enteros de forma manual y los muestre por pantalla. Por ejemplo, el primer conjunto podría ser {1, 2, 3, 4, 5} y el segundo {4, 5, 6, 7, 8}. Luego, realiza las siguientes operaciones, mostrando el contenido de ambos conjuntos tras cada operación:
   - Muestra la unión de ambos conjuntos.
   - Muestra la intersección de ambos conjuntos.
   - Muestra los elementos que están en el primer conjunto pero no en el segundo.
   - Añade un nuevo número al primer conjunto.
   - Elimina un número del segundo conjunto.

   Ejemplo de salida:

```
Conjunto 1: {1, 2, 3, 4, 5}
Conjunto 2: {4, 5, 6, 7, 8}
Unión de los conjuntos: {1, 2, 3, 4, 5, 6, 7, 8}
Intersección de los conjuntos: {4, 5}
Elementos en el conjunto 1 pero no en el conjunto 2: {1, 2, 3}
Conjunto 1 después de añadir el número 9: {1, 2, 3, 4, 5, 9}
Conjunto 2 después de eliminar el número 6: {4, 5, 7, 8}
```

7. **Crea un diccionario donde las claves sean los nombres de cuatro estudiantes y los valores sean sus respectivas notas en un examen. Luego, realiza las siguientes operaciones, mostrando el diccionario actualizado tras cada operación:**
   - Muestra el diccionario original.
   - Muestra el listado de estudiantes y el número total de estudiantes en el diccionario.
   - Muestra el listado de las notas almacenadas, sin mostrar los nombres de los estudiantes.
   - Añade un nuevo estudiante y su nota al diccionario.
   - Modifica la nota de uno de los estudiantes.
   - Elimina a un estudiante del diccionario.
   - Muestra el promedio de las notas de los estudiantes restantes.

   Ejemplo de salida:

```
Diccionario original: {'Ana': 8.5, 'Luis': 7.3, 'María': 9.1, 'Pedro': 6.8}
Listado de estudiantes: ['Ana', 'Luis', 'María', 'Pedro']
Número total de estudiantes: 4
Listado de notas: [8.5, 7.3, 9.1, 6.8]
Diccionario después de añadir a Carlos:
{'Ana': 8.5, 'Luis': 7.3, 'María': 9.1, 'Pedro': 6.8, 'Carlos': 8.0}
Diccionario después de modificar la nota de Ana:
{'Ana': 9.0, 'Luis': 7.3, 'María': 9.1, 'Pedro': 6.8, 'Carlos': 8.0}
Diccionario después de eliminar a Pedro:
```

```
{'Ana': 9.0, 'Luis': 7.3, 'María': 9.1, 'Carlos': 8.0}
Promedio de las notas: 8.35
```

8. **Condicionales**. Escribe un programa que determine si un cliente de una tienda es elegible para un descuento especial. El programa debe solicitar al cliente dos datos: 1) La cantidad de dinero que ha gastado en la tienda durante el último mes (en euros), y 2) Si tiene una membresía premium (si/no). Nota: representar internamente la membresía como un booleano. El programa debe aplicar las siguientes reglas para determinar si el cliente recibe un descuento:
   - Si el cliente ha gastado más de 100€ y tiene una membresía premium, se le otorga un descuento del 20 %.
   - Si el cliente ha gastado más de 100€ o tiene una membresía premium, se le otorga un descuento del 10 %.
   - Si no cumple ninguna de las condiciones anteriores, no se le otorga ningún descuento.

   El programa debe mostrar el porcentaje de descuento que recibe el cliente.

   Ejemplo de salida esperada:

   ```
   Introduce el gasto del último mes: 120
   ¿Tienes membresía premium (si/no)? no
   Descuento aplicado: 20%
   ```

9. **Bucle while**. Escribe un programa que solicite al usuario un número entero positivo. El programa debe sumar los números desde 1 hasta ese número, pero se detendrá si la suma acumulada supera un valor límite, que también debe ser proporcionado por el usuario. Utiliza un bucle `while` con una expresión booleana en la condición para controlar el proceso.
   Reglas: el bucle debe continuar mientras el número actual sea menor o igual al número proporcionado y la suma acumulada sea menor o igual al límite establecido.
   Ejemplo de salida esperada:

   ```
   Introduce un número entero positivo: 5
   Introduce un límite para la suma: 10
   Suma acumulada: 1
   Suma acumulada: 3
   Suma acumulada: 6
   Suma acumulada: 10
   La suma se detiene porque alcanzó o superó el límite.
   ```

10. **Bucle for con range()**. Escribe un programa que solicite al usuario un número entero positivo que representará el tamaño de una palabra. El programa debe componer un string de dicho tamaño utilizando la función `range`(). En cada iteración del bucle `for`, se generará un número aleatorio del 0 al 9, se mostrará la posición actual, y se concatenará el número al string resultante. Finalmente, el programa deberá mostrar el string generado.
    Ejemplo de salida:

    ```
    Introduce un número entero positivo: 5
    Posición 0: 4
    Posición 1: 9
    Posición 2: 3
    Posición 3: 7
    Posición 4: 2
    String generado: 49372
    ```

11. **Bucle for con listas**. Escribe un programa que genere una lista de tamaño aleatorio entre 3 y 7. Luego, usando un bucle `for`, llena la lista con números aleatorios entre 0 y 9 (ambos inclusive). Finalmente, mostraremos el contenido de la lista de tres formas diferentes:
    - Hacer `print`() directamente sobre la variable de tipo lista creada.
    - Usar un bucle `for` con `range`() para mostrar el contenido indicando el índice de la posición y su valor. Nota: considerar el uso de `len`().

- Usar un bucle `for` haciendo uso del operador `in` de la siguiente forma: `for` elemento `in` lista .

Ejemplo de salida:

```
Tamaño de la lista: 5
Lista generada: [2, 8, 5, 3, 7]
Elemento en la posición 0: 2
Elemento en la posición 1: 8
Elemento en la posición 2: 5
Elemento en la posición 3: 3
Elemento en la posición 4: 7
```

12. **Funciones con parámetros obligatorios**. Escribe una función que reciba dos números enteros como parámetros y retorne el mayor de ellos. El programa debe generar aleatoriamente cinco pares de números, y para cada par, utilizar la función para encontrar y mostrar el mayor número. ***¿Sabrías documentar la función con Docstring?***
    Ejemplo de salida:

    ```
    El mayor número entre 82 y 93 es: 93
    El mayor número entre 28 y 62 es: 62
    El mayor número entre 78 y 30 es: 78
    El mayor número entre 23 y 69 es: 69
    El mayor número entre 25 y 32 es: 32
    ```

13. **Funciones con parámetros opcionales**. Crea una función que reciba el nombre de una persona y su edad, siendo la edad un parámetro opcional (por defecto 18). La función debe imprimir un mensaje del tipo: "Hola [nombre], tienes [edad] años". Si no se proporciona la edad, debe imprimir el valor por defecto.
    Ejemplos de salidas posibles:

    ```
    Hola Ana, tienes 18 años.
    Hola Carlos, tienes 25 años.
    ```

14. **Funciones con argumentos empaquetados**. Escribe una función que reciba una cantidad variable de números (*args) y devuelva el promedio de todos ellos. Además, la función debe aceptar argumentos con nombre (**kwargs) para mostrar mensajes personalizados al usuario (por ejemplo, un mensaje de bienvenida y otro de despedida).
    Ejemplo de salida:

    ```
    ¡Hola! Bienvenido a la calculadora de promedios.
    Gracias por usar el programa.
    El promedio es: 30.0
    ```

15. **Funciones con argumentos de todo tipo**. Define una función que reciba como argumento obligatorio el nombre de una persona, un parámetro opcional que represente el saldo (por defecto 0), y una serie de transacciones utilizando *args para sumar al saldo. Por ejemplo, las transacciones a realizar se pueden expresar como [100, -50, 200, -30], donde números positivos indicarían depósitos a cuenta (aumentar el valor de la cuenta), y números negativos retiradas de efectivo (valores negativos). Además, recibe **kwargs con información adicional sobre la persona (como dirección o número de cuenta). La función debe devolver dos valores: el saldo final después de aplicar las transacciones y un string con un mensaje ilustrativo detallando los aspectos adicionales de la persona.
    Ejemplo de salida

    ```
    Saldo final: 320
    Cliente: Juan Pérez
    Direccion: Calle Falsa 123
    Numero_cuenta: 123456789
    ```

1.5 Anexos

1.5.1. Anexo I: Métodos alternativos de instalación de Python

Como alternativa a CPython (la distribución estándar) puedes instalar Python usando alguna suit especializada que lo incluya.

Quizás la más conocida sea Anaconda Python. Desarrollada por Anaconda. `https://www.anaconda.com/`

- Pensada para desarrolladores/empresas de Python que necesiten respaldo.
- Es una distribución pensada para ciencia de los datos y aprendizaje automático que incluye Python, IPython y R entre otros.
- Orientado el trabajo comercial y científico.
- La más completa para ciencias de los datos.
- La edición individual es gratis. Otras superan los $10mil.
- Instalación básica $\approx$ 4.8Gb. CPython+Editores+IDE+...
- Gestiona paquetes Python y de terceros.

Existe una versión libre y versiones comerciales. Al adaptarse desde estudiantes a profesiones, la distribución es usada por millones de usuarios. Destaca su sistema de gestión de paquetes llamado conda.

Otra suite alternativa es ActivePython. Desarrollado por Activestate. `https://www.activestate.com/`. Tiene la misma filosofía de Anaconda.

Otras alternativas curiosas son:

- PyPy. Alternativa a CPython. `https://www.pypy.org/`
  - Compilador JIT[2] de RPython.
  - 4.2 veces más rápido que CPython.
  - Usa el "núcleo" de CPython.
- IronPython. Alternativa a CPython. `https://ironpython.net/`
  - Implementado en C#.
  - Integra Python en .NET[3].
  - Ejecuta Python en Microsoft DLR (entorno de ejecución)
- Jython. Alternativa a CPython. `https://www.jython.org/`
  - Implementado en Java.
  - En Java se pueden añadir librerías de Jython.
  - En Jython se puede añadir Java.
  - Ejecuta Python en JVM.
- Hay más: WinPython, MicroPython, Pycopy, RustPython, MesaPy, ...

1.5.2. Anexo II: Gestión de Algunos Errores en PyCharm

1.5.2.1. Resolver el problema de Paquetes en PyCharm

A algunos de vosotros os ocurre que algún módulo (fichero .py) no reconoce algún otro módulo del proyecto. Para solucionar esto sigue los siguientes pasos.

Memoriza la teoría. En las transparencias se indica que se debe tener un fichero vacío __init__.py para poder importar todos los módulos de una carpeta. Es decir, hay que construir un paquete con sus módulos. La construcción de paquetes desde PyCharm se hace seleccionando File > New > Python Package. Simplemente crea un carpeta con el fichero __init__.py. Observa que todas las carpetas menos venv están en el mismo color. Si además tienen un punto sobre la carpeta, entonces esa carpeta es un paquete.

En principio, solo con lo anterior deberías poder importar los módulos de los paquetes. Pero si te siguiera saliendo el mensaje `Unresolved Reference XXXX` entonces sigue los siguientes pasos:

- Marca el paquete como una fuente raíz.
  Pon el ratón sobre la carpeta y pulsa botón derecho del ratón. Selecciona Mark Directory as > Sources Root. La carpeta cambiará de color.

[2] Compilación en tiempo de ejecución (JIT) mejora el rendimiento compilando a bytecode y traducir el bytecode a código máquina nativo en tiempo de ejecución.

[3] Plataforma de aplicaciones que permite la creación y ejecución de servicios web y aplicaciones de Internet.

- Añadir la nueva al `PYTHONPATH`.
  Selecciona Settings » Build, Execution, Deployment » Console » Python Console y asegúrate de marcar las dos opciones de `Add content` y `Add source`.
- Limpia la caché y reinicia PyCharm. Selecciona File » Invalidate Cache » Restart.

Ya no deberías tener más problemas de reconocimiento de los módulos de los paquetes.

IMPORTANTE:

Algunos, para solucionar el problema podrían haber seleccionado seleccionado Install and Import package en vez de hacer la importación como se indica aquí. Si tras realizar esa acción PyCharm ha dejado de quejarse, **tienes un problema**. Habrás instalado unos paquetes extras en `venv` y ahora tu programa entiende que debe importar esos paquetes en vez de los que tú has desarrollado. Obviamente tendrás que quitar esos paquetes extras que no os sirven para nada en la resolución del ejercicio.

Posiblemente algunos de ellos son: `Point`, `Agent`, `Vector`, `Vector2`, `state`, ... La forma de comprobar que has instalado unos paquetes que NO deberías de haber instalado es seleccionar Preferences » Project » Python Interpeter para ver el listado de los paquetes que has instalado. Si hay alguno cuyo nombre coincide con algunos de los nombre que te indico o con el nombre de algún módulo/clase que tú hayas construido tendrás que eliminar el paquete porque muy probablemente tu programa estará usando ese paquete en vez del tuyo.

1.5.3. Anexo III: Aspectos avanzados de Python

Este anexo sirve como una ampliación de lo tratado en la sección 1.3 y servirá como documentación de consulta del lenguaje en casos puntuales en los que el estudiante lo requiera.

1.5.3.1. Construcción de Contenedores por comprehensión

Un contenedor por comprensión es el que se construye basado en la notación matemática de creación de conjuntos (builder notation[4]). Por ejemplo, podemos construir: $S = \{ \underbrace{2 \cdot x}_{\text{Expresión salida}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{Conj. entrada}} , \underbrace{x^2 > 3}_{\text{Predicado}} \}$

Definir conjuntos basándonos en propiedades también se conoce como comprensión de conjuntos, abstracción de conjuntos o definición por intención de un conjunto.
En Python la construcción por comprehensión[5] se expresa como

```
salida = <expresion(x) for x in iterable [if condicion]>
```

donde:

- < > representa a `[ ]` si se quiere construir **listas**, representa a `( )` si queremos construir **tuplas** y representa a `{ }` para construir **conjuntos**.
  En el siguiente ejemplo se construye un conjunto porque se usan llaves:

```
{2*x for x in range(5) if x*x > 3}
```

- `for x in secuencia` se puede sustituir por cualquier conjunto de sentencias que definan el valor de `x`. Por ejemplo, un bucle anidado donde aparezca `x`.

```
[2*x for v in range(2) for x in range(5) if x*x > 3]
```

  Notar que el condicional sirve para filtrar los valores de x. Solo a aquellos valores de la secuencia que cumplan la condición se les aplicará la expresión.
- expresion(x) es cualquier expresión sobre x. Algunos ejemplos son:
  - No alterar el valor de x; es decir, expresion(x)=x.
  - Aplicar alguna operación o función. P.e. expresion(x)=x**2.
  - Usar algún método. P.e. expresion(x)=x.upper().
  - Usar expresiones ternarias. P.e. expresion(x)=x `if` x %2==0 `else` 1000.

[4] https://en.wikipedia.org/wiki/Set-builder_notation
[5] https://en.wikipedia.org/wiki/List_comprehension

1.5.3.2. Funciones Lambda o Anónimas.

El λ-cálculo es un sistema formal matemático desarrollado en los años 1930s diseñado para trabajar con la noción de función, aplicación de funciones y recursión. Proporciona una semántica simple para la computación y la primera simplificación es que el λ-cálculo **trata las funciones de forma anónima**. La escritura anónima de $\text{square\_sum}(x, y) = x^2 + y^2$ es $(x, y) \mapsto x^2 + y^2$.

En Python las expresiones anónimas se llaman Funciones Lambda y tiene la siguiente expresión:

`lambda` argumentos: expresión

En la definición deberás tener en cuenta:

- Puedes usar cualquier número de argumentos.
- Solo habrá una única expresión.

```
el_doble = lambda x: x * 2
print(el_doble(10))
suma = lambda x, y: x + y
print(suma(4, 5))
acotado = lambda x: 4 <= x <= 8
print(acotado(5))
```

Son útiles junto con funciones clausura como `map`(), `filter`(), reduce(), ... para trabajar con colecciones. Se verán más adelante.

UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 2 de Prácticas

Python Orientado a Objetos

Material original por Luis Daniel Hernández Molinero (ldaniel@um.es)

Editado por Alejandro Buitrago López (alejandro.buitragol@um.es), Sergio López Bernal (slopez@um.es), Javier Pastor Galindo (javierpg@um.es), Jesús Damián Jiménez Re (jesusjr@um.es) y Gregorio Martínez Pérez (gregorio@um.es)

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2024-2025

Última modificación:
16 de septiembre de 2024

Sesión 2: Python Orientado a Objetos

Índice

1.1 Aspectos esenciales de Programación Orientada a Objetos en Python

La Programación Orientada a Objetos es una extensión de la Programación Estructurada y Modular (encargada de dividir programas complejos en módulos que se integran unos con otros). Los dos primeros conceptos más básicos son:

- Clase. Es una plantilla que encapsula atributos y métodos.
  - Atributo: es una variable que puede ser de cualquier tipo, incluidas otras clases.
  - Método: es el concepto de función en programación modular pero que se define dentro de una clase.
- Objeto. Un caso particular de la clase (o plantilla). También se llama instancia de la clase.
  Los atributos o variables con valores concretos representan las propiedades de un objeto (su estado) y los métodos definen su comportamiento (lo que es capaz de hacer u operaciones que realiza).

1.1.1. Componentes principales de una clase

Para definir una clase se necesitan definir los siguientes aspectos:

- Los atributos: son las variables que definirán los estados de los objetos. Pueden diferenciarse entre atributos de instancia y atributos de clase.
- Los métodos: representan el comportamiento de los objetos, asociados con acciones que puede realizar un objeto. Pueden tener diferentes objetivos: consultar el estado de los atributos de un objeto, modificar dichos atributos, o realizar operaciones haciendo uso usando los atributos. Además, los métodos pueden ser de tres tipos: de instancia, de clase, o estáticos.
- El constructor: se puede entender como un método especial que permite establecer el estado inicial de los objetos cuando se construyen por primera vez.

En las siguientes secciones se profundiza sobre cada uno de estos componentes, ofreciendo ejemplos para comprender mejor su notación y funcionamiento.

1.1.2. Atributos

La mejor forma de ilustrar la definición de los dos tipos de atributos existentes en Python es mediante un ejemplo de código, que iremos incrementando en cada una de las secciones siguientes:

```
class Persona:
    # Definir los atributos 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, activo: bool, hobbies: list):
        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
        self._hobbies: list = hobbies  # Atributo de instancia de tipo lista (compuesto)
```

El código anterior incluye una serie de atributos de instancia de diferentes tipos de datos, tanto primitivos como compuestos, así como el uso de su constructor __init__ capaz de darles un valor de inicialización. Como podemos ver, todos los nombres de atributos de instancia anteponen un guión bajo (_) para indicar que son privados (veremos más detalles en la siguiente sesión).

Por otro lado, se define un atributo de clase, especie, que define información de la clase en su conjunto. Esto es, todas las personas tienen en común que son humanas, independientemente de las características individuales que definan a cada persona (atributos de instancia), como son su nombre o su altura.

Es interesante el uso de __slots__ para especificar el listado de atributos de instancia que definen una clase dada. Así, será imposible definir nuevos atributos desde fuera de la clase.

1.1.3. Métodos

Como se comentaba anteriormente, disponemos de tres tipos de métodos diferentes en Python: de instancia, de clase y estáticos. Además, tenemos otra categoría denominada métodos mágicos que veremos a continuación. Para comprender los métodos, ampliemos la clase Persona definida anteriormente para que tenga tres métodos:

```
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, activo: bool, hobbies: list):
        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
        self._hobbies: list = hobbies  # Atributo de instancia de tipo lista (compuesto)

    # Método de instancia
    def obtener_info(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
```

En primer lugar, el método de instancia obtener_info devuelve un string representando, de forma textual, una descripción del estado actual de la persona. Retorna información sobre todos los atributos de instancia salvo el listado de hobbies (por simplicidad en el ejemplo). Como podemos ver, este método recibe como único parámetro self, que es obligatorio para todo método de instancia, y representa al propio objeto que invocará a la función. Si el método requiriese más parámetros, se colocarían a la derecha de self.

En relación al método de clase, retornará una representación textual de la especie asociada a todas las personas. Es importante destacar que en los métodos de clase se debe indicar como primer parámetro cls, que representa a la propia clase sobre la que se invocará el método. Además, debemos colocar el decorador @classmethod sobre la definición de la función.

En cuanto al método estático, es capaz de determinar si, dado un valor de edad, se es mayor de edad o no. En este caso, se deberá usar el decorador @staticmethod. Estos métodos son independientes de cualquier variable de clase o de instancia

y, de hecho, no podrán hacer uso de ellos. Son de utilidad para crear métodos auxiliares que implementan funcionalidad de apoyo a la clase.

Finalmente, un método mágico en Python es un método especial que tiene una sintaxis definida por el lenguaje y que permite a los programadores personalizar el comportamiento de las clases de manera implícita. Estos métodos son útiles para integrar los objetos personalizados en el ecosistema del lenguaje, mejorando la legibilidad y la funcionalidad del código. Un método mágico en Python siempre comienza y termina con dos guiones bajos (__). Además, Python invoca estos métodos automáticamente en circunstancias particulares.

Veamos un listado de los métodos mágicos más comúnmente usados:

- **__init__(self, ...)**: Se ejecuta al crear una instancia de una clase; es el constructor que inicializa el estado del objeto.
- **__str__(self)**: Define la representación legible para humanos de un objeto, utilizada por `print()` y `str()`.
- **__len__(self)**: Define el comportamiento de `len()` aplicado a una instancia de la clase.
- **__add__(self, other)**, **__sub__(self, other)**, **__mul__(self, other)**, **__truediv__(self, other)**, **__floordiv__(self, other)**, **__mod__(self, other)**, **__pow__(self, other)**: Sobrecargan operadores aritméticos (+, -, *, /, //, %, **).
- **__eq__(self, other)**, **__ne__(self, other)**, **__lt__(self, other)**, **__le__(self, other)**, **__gt__(self, other)**, **__ge__(self, other)**: Sobrecargan los operadores de comparación (==, !=, <, <=, >, >=).

Para comprender mejor su funcionamiento, extenderemos el ejemplo para incorporar dos métodos mágicos: __str__ y __gt__.

```
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, activo: bool, hobbies: list):
        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
        self._hobbies: list = hobbies  # Atributo de instancia de tipo lista (compuesto)

    # Método de instancia
    def obtener_info(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:
        return f"Persona(nombre={self._nombre}, edad={self._edad}, altura={self._altura}, activo={self._activo})"

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

El primero mostrará una representación textual, similar al método obtener_info que ya teníamos definido. Así, definiendo el método mágico no sería necesario mantener la función antigua, aunque la dejamos como ejemplo de método de instancia. Por su parte, __gt__ comprobará si la persona actual (objeto que invoca al método) tiene una edad superior a otra persona que se le pase por parámetro.

Es importante destacar que a __gt__ se le proporciona un parámetro de la clase Persona. En este caso, el tipo del parámetro se debe especificar entre comillas. Esto se utiliza para indicar que el tipo es la misma clase que se está definiendo, pero que todavía no está disponible en ese momento (todavía no está completamente definida pues estamos en proceso de ello). Esto no es algo exclusivo de los métodos mágicos, si no de cualquier definición de parámetros en métodos con POO.

Al margen de los métodos mágicos más comunes, es importante destacar dos métodos mágicos que no deben usarse:

- **__new__(cls, ...)**: Es el método creador. No debe usarse nunca, ya que su uso lo gestiona internamente Python al crear variables de una determinada clase.
- **__del__(self)**: Se invoca cuando un objeto va a ser destruido (destructor). No es recomendable su uso, ya que Python cuenta con un mecanismo de recolección de basura que se encarga de eliminar automáticamente las variables en desuso. Puede ser útil en situaciones muy concretas, como liberar recursos externos (archivos, conexiones de red, etc.) cuando un objeto es destruido.

Por otro lado, Python proporciona muchos otros métodos mágicos adicionales, aunque no son relevantes en el contexto de la asignatura. Por ejemplo, proporciona métodos mágicos para la gestión de colecciones, definiendo cómo se debe retornar elementos de una colección (**__getitem__(self, key)**, **__setitem__(self, key, value)**, **__delitem__(self, key)**), cómo iterar sobre ellos (**__iter__(self)**, **__next__(self)**), o cómo consultar si contienen un elemento (**__contains__(self, item)**), entre otros. Pueden consultarse en los siguientes enlaces:

- A Guide to Python's Magic Methods: https://rszalski.github.io/magicmethods/
- Lista de métodos mágicos asociados a cada operación: https://docs.python.org/3/library/operator.html?highlight=operations

1.1.4. Creación y uso de objetos

Esta sección describe cómo hacer uso de objetos en el __main__, donde se ilustra la definición e inicialización de objetos de la clase Persona y se invocan los métodos previamente definidos.

```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, activo: bool, hobbies: list):
        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
        self._hobbies: list = hobbies  # Atributo de instancia de tipo lista (compuesto)

    # Método de instancia
    def obtener_info(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:
        return f"Persona(nombre={self._nombre}, edad={self._edad}, altura={self._altura}, activo={self._activo})"

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

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 obtener_info (método de instancia)
    print(persona1.obtener_info())  # Salida: Nombre: Ana, Edad: 25, Altura: 1.68, Activo: True
    print(persona2.obtener_info())  # Salida: Nombre: Juan, Edad: 30, Altura: 1.75, Activo: True

    # 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)

    # Usar obtener_especie (método de clase)
    print(Persona.obtener_especie())  # Salida: La especie es Humano

    # Usar es_mayor_edad (método estático)
    print(Persona.es_mayor_edad(25))  # Salida: True (porque 25 >= 18)
    print(Persona.es_mayor_edad(17))  # Salida: False (porque 17 < 18)

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

Así, creamos dos objetos de la clase Persona para, posteriormente, obtener su información haciendo uso de los métodos obtener_info y __str__ (este último se invocará implícitamente al usar `print`(). Posteriormente, se usa la función estática para comprobar si dos edades diferentes corresponden o no a la mayoría de edad. Finalmente, usamos el operador $>$ entre objetos (implícitamente realiza una llamada a __gt__) para comparar la edad de dos personas.

Es importante destacar que en el caso de los métodos de clase y estático hemos invocado a las funciones sobre el nombre de la clase en lugar de sobre el objeto persona. En el caso del método de clase podríamos haberla invocado sobre el objeto persona sin problemas, mientras que para el método estático es obligatorio hacerlo sobre el nombre de la clase.

1.2 Relación de ejercicios

Aunque podríamos realizar todos los ejercicios de forma incremental en un mismo fichero, es recomendable tener un fichero .py por cada ejercicio realizado, de forma independiente (tendremos una solución completa por cada ejercicio). Así, cuando vayamos a comenzar con un nuevo ejercicio, copiaremos y pegaremos el contenido del ejercicio anterior en un nuevo fichero y comenzaremos a trabajar en el nuevo enunciado.

1. **Creación de clases**. Crea una clase en Python que represente un **Polígono**, dentro de un fichero llamado clases_geometria.py. En este primer ejercicio, el constructor (método __init__) **no recibirá ningún parámetro de inicialización** (salvo el self). La clase debe contener los siguientes atributos de instancia:

   - **numero_lados**: de tipo entero, y que representa el número de lados que tiene el polígono. Se establecerá a 3.
   - **color**: de tipo string, y que representa el color del polígono. Se establece a "BLANCO".
   - **forma**: este atributo será de tipo enumerado (de string) llamado TipoForma (recuerda que para ello se crea una clase *class TipoForma(Enum)*). El enumerado TipoForma tendrá tres opciones posibles (CONVEXO, CONCAVO o COMPLEJO) cuyos valore será 1, 2 o 3, respectivamente. El atributo se establecerá a COMPLEJO en el momento de la construcción del objeto.
   - **relación_lados**: este atributo será de tipo enumerado (de string) llamado TipoRelacionLado (para ello, definir *class TipoRelacionLado(Enum)*). Sólo lo podría tener uno de los siguientes valores: REGULAR o IRREGULAR (como en el caso anterior, cada opción tendrá un entero asociado). El atributo se establecerá a REGULAR.

   Crea un objeto de dicha clase, llamado **mi_poligono**, y muestra por pantalla sus cuatro atributos desde el main, accediendo directamente a los atributos de la instancia (*RECUERDA: ¡esto último no es una buena práctica, por eso PyCharm lanza advertencias!*).

2. **Inicialización de la clase**. Modificar el constructor (método __init__) de la clase para que reciba como parámetros obligatorios los valores de **forma** y **relación de lados**. El número de lados seguirá siendo siempre 3. El siguiente parámetro es opcional, y si no se establece, cogerá el siguiente valor por defecto:

   - **color**: Por defecto, si no se pasa el parámetro en el constructor, deberá cogerse el color "BLANCO".

   Modifica la creación del objeto **mi_poligono** adecuadamente para que la instancia se cree como un polígono CONVEXO y REGULAR, pero sin establecer ningún color.

3. **Métodos adicionales**. Añadir los siguientes métodos de instancia a la clase Polígono:

   - Un método llamado **establece_numero_lados** que permita **modificar el número de lados del polígono**. El método recibirá un parámetro con los lados, que debería ser mayor o igual que 3. Si es menor que 3, el método devolverá un mensaje de error (string) diciendo que no es posible construir un polígono con menos de 3 lados.
   - Un método que permita obtener el número de lados del polígono.
   - Un método que permita obtener el color del polígono.
   - Un método que permita obtener la forma del polígono.
   - Un método que permita obtener la relación entre los lados del polígono.
   - Implementar el método mágico __str__ en la clase Polígono, que permita mostrar por pantalla el estado del objeto. Es decir, el valor de todos sus atributos en el momento de su invocación. Esta función debe utilizar los cuatro métodos anteriores para evitar acceder directamente a los atributos de la clase.

   Tras la creación del objeto **mi_poligono**, establece ahora que el polígono definido tiene 4 lados a través del primer método creado. Evita ahora el acceso directo a los atributos haciendo uso de los métodos de consulta de atributos implementados. (*Observa cómo ahora el programa principal accede a los atributos a través de métodos y no por la notación punto, así que PyCharm deja de lanzar avisos.*). Por último, muestra el objeto directamente haciendo uso de print(mi_poligono). ¿Qué sucede y por qué?

4. **Creación de atributos y métodos de clase**. Vamos a crear un atributo en la clase que permita obtener el número de objetos Polígono que existen actualmente creados. Para ello, añadir el **atributo de clase** llamado **numero_poligonos** que se inicialice a 0. Hay que realizar las modificaciones que consideres necesarias, para que cada vez que se cree

un objeto Polígono, se aumente el número existente. *¿Cómo se haría?*
Tras visualizar los datos del objeto **mi_poligono**, mostrar por pantalla el número de objetos Polígono actualmente creados. Para ello, puedes crear un **método de clase** encargado de devolver el número de polígonos creados hasta ahora.
Por último, en este ejercicio debemos sustituir el método **muestra_poligono** por el método mágico __str__ para que devuelva la información del estado (string). Tras ello, probar que funciona correctamente, haciendo uso de *print()* para mostrar directamente el objeto **mi_poligono**. *¿Ves la diferencia con respecto al print() que se proponía en el ejercicio anterior, cuanto no habíamos definido el método mágico __str__?*

5. **Creación de la clase Punto**. Crear una clase dentro del fichero que represente un **Punto**, definida por los siguientes atributos:

   - **coordenada_x**, con valor de tipo float.
   - **coordenada_y**, con valor de tipo float.

   El método __init__ debe permitir establecer ambas coordenadas como parámetro. Adicionalmente, habría que crear los siguientes métodos:

   - **obtiene_coordenada_x**. Devuelve la coordenada x del punto.
   - **obtiene_coordenada_y**. Devuelve la coordenada y del punto.
   - **obtiene_cuadrante**. Devuelve un entero indicando el cuadrante en el que se encuentra el punto:
     - Devuelve 1 si está en el primer cuadrante (x e y positivos).
     - Devuelve 2 si está en el segundo cuadrante (x negativo e y positivo).
     - Devuelve 3 si está en el tercer cuadrante (x e y negativos).
     - Devuelve 4 si está en el cuarto cuadrante (x positivo e y negativo).
     - Devuelve 0 si el punto está en uno de los ejes X o Y.

   Crear los cuatro siguientes objetos / instancias de la clase Punto, y mostrar el cuadrante al que pertenece cada uno:

   - Un punto A con las coordenadas (-2,2)
   - Un punto B con las coordenadas (2,2)
   - Un punto C con las coordenadas (-2,-2)
   - Un punto D con las coordenadas (2,-2)

6. **Creación de la clase Línea**. Crear una clase dentro del fichero que represente una Línea. La clase debe contener dos atributos:

   - **punto_inicio**, que es de la clase Punto, y representa el punto de inicio de la línea.
   - **punto_fin**, que es de la clase Punto, y representa el punto de fin de la línea.

   El método __init__ debe permitir establecer los dos puntos que definen la línea. Adicionalmente, habría que crear los métodos siguientes:

   - **obtiene_punto_inicio**. Obtiene el punto de inicio de la línea.
   - **obtiene_punto_fin**. Obtiene el punto de fin de la línea.
   - **calcula_longitud**. Obtiene la longitud de la línea. Para calcularla, se puede utilizar la fórmula de distancia de Manhattan: d = $|x2 - x1| + |y2 - y1|$
   - **muestra_puntos**. Este método muestra por pantalla las coordenadas del punto de inicio de la línea y del punto de fin.

   Crea ahora 4 objetos de la clase línea, que conformarán en sí un cuadrado. Mostrar por pantalla las coordenadas de los puntos de inicio y fin y la longitud de cada una de ellas.

   - **linea_1**, con punto de inicio el punto A y punto de fin el punto B.
   - **linea_2**, con punto de inicio el punto B y punto de fin el punto D.
   - **linea_3**, con punto de inicio el punto D y punto de fin el punto C.
   - **linea_4**, con punto de inicio el punto C y punto de fin el punto A.

7. **Ampliar la clase Polígono**. Vamos ahora a ampliar la definición y los métodos con los que cuenta nuestra clase Polígono. En primer lugar, la clase debería contar con los siguientes nuevos atributos:

- **lados**, una lista de elementos de tipo Linea, y que contendrá cada una de las líneas que sirve de lado del polígono.

También necesitaremos añadir los siguientes nuevos métodos:

- **calcula_perimetro**, que obtiene el perímetro.
- **establece_lados**, que recibirá como parámetro una lista de Líneas que representan los lados del polígono.

Finalmente, añadirle al objeto **mi_poligono** una **lista con las cuatro líneas** creadas en el ejercicio 6, y mostrar por pantalla el **perímetro** del cuadrado.
Mostrar por pantalla el perímetro del cuadrado.

8. **Ampliación de la clase Polígono con métodos mágicos**. En este ejercicio, vamos a seguir trabajando con la clase Polígono, añadiendo algunos métodos mágicos adicionales para extender su funcionalidad:

   - **__eq__(self, otro)**. Comprueba si dos polígonos tienen el mismo número de lados.
   - **__add__(self, otro)**. Suma los perímetros de dos polígonos y devuelve el resultado.
   - **__lt__(self, otro)**. Compara si un polígono tiene un perímetro menor que otro.
   - **__len__(self)**. Devuelve el número de lados del polígono.

   Tras implementar los métodos mágicos, haz uso de ellos en la función __main__ para comprobar que se han implementado correctamente.
   Finalmente, muestra por pantalla el número de polígonos creados actualmente. Debería indicar que tenemos 2.

UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 3 de Prácticas

Sobrecarga, visibilidad, paquetes y módulos

Autores: Alejandro Buitrago López (alejandro.buitragol@um.es), Sergio López Bernal (slopez@um.es), Javier Pastor Galindo (javierpg@um.es), Jesús Damián Jiménez Re (jesusjr@um.es) y Gregorio Martínez Pérez (gregorio@um.es)

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2024-2025

Última modificación:
23 de septiembre de 2024

Sesión 3: Sobrecarga, visibilidad, paquetes y módulos

Índice

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
import random

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

    # Constructor con sobrecarga mediante parámetros opcionales
    def __init__(self, nombre: str, edad: int, altura: float = None, activo: bool = None,
                 hobbies: list = 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 4 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.

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).
- *_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:

```
import random

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

    # Constructor con sobrecarga mediante parámetros opcionales
    def __init__(self, nombre: str, edad: int, altura: float = None, activo: bool = None,
                 hobbies: list = 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

    # Setter para el atributo de clase "especie"
    @classmethod
    def set_especie(cls, nueva_especie: str) -> None:
        cls.especie = nueva_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 de instancia
    def obtener_info_sin_hobbies(self) -> str:
        return f"Nombre: {self._nombre}, Edad: {self._edad}, Altura: {self._altura}, Activo: {self._activo}"

    # 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.

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.

Finalmente, 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.

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   # Indica que es un subpaquete
|       |-- 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.

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 un parámetro opcional (*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) Crea un método llamado **escala_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?

   c) 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 *obtiene()* y *establece()* 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 *escala_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) 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.

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

   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:
   ***NOTA**: Las clases Punto y Línea deberían estar ya implementadas de manera autónoma sobre la Sesión 2, así como la ampliación de la clase Polígono.*

   - **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.
   - **Módulos y paquetes**: Incluye el contenido de las clases *Punto* y *Línea* dentro de la estructura definida en el ejercicio 4.
   - **Modificación del main**: Actualiza el main para crear, consultar y modificar puntos y líneas, además de polígonos.

   Además, amplía el contenido de la clase *Polígono* dentro del proyecto para incluir la funcionalidad que se definió en los ejercicios 7 y 8 de la sesión 2.

UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 4 de Prácticas

Relación entre clases

Autores: Alejandro Buitrago López (alejandro.buitragol@um.es), Sergio López Bernal (slopez@um.es), Javier Pastor Galindo (javierpg@um.es), Jesús Damián Jiménez Re (jesusjr@um.es) y Gregorio Martínez Pérez (gregorio@um.es)

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2024-2025

Última modificación:
30 de septiembre de 2024

Sesión 4: Relación entre clases

Índice

4.1 Tipos de relaciones en POO

Cuando trabajamos con múltiples clases en un programa, es común que establezcan vínculos entre ellas. Estos vínculos se denominan **relaciones** y pueden variar en su **nivel de dependencia**, desde relaciones simples donde un objeto utiliza a otro de manera puntual, hasta relaciones complejas que afectan al ciclo de vida de los objetos. A continuación, se describen los principales tipos de relaciones en programación orientada a objetos.

4.1.1. Relación de Uso

Una **relación de uso** ocurre cuando una clase utiliza los servicios o funcionalidades de otra de forma **esporádica**. Esto significa que el vínculo entre ambas clases **no es constante ni duradero en el tiempo**, sino que ocurre puntualmente cuando una clase necesita interactuar con la otra. El caso más típico es cuando una clase A invoca un método de una clase B para ejecutar una acción, sin que A almacene una referencia a B. Es decir, la clase B es utilizada temporalmente para cumplir una función específica. Un ejemplo cotidiano de este tipo de relación es el caso de una persona que utiliza un cajero automático para realizar una transacción, sin necesidad de ser cliente del banco. En el código, este tipo de relación se manifiesta al pasar un objeto de la clase B como parámetro a un método de la clase A.

Para comprender mejor todos los tipos de relaciones, retomaremos el código de la sesión de prácticas anterior, que hacía uso de la clase `Persona`. Sin embargo, para trabajar mejor con relaciones vamos a necesitar una clase adicional: `Libro`. La definición de un libro, de forma muy simplificada, podría venir determinada por su título, su autor, y su número de páginas. Es importante destacar que no ofrecemos métodos set para el título y el autor, pues no tiene sentido modificar esta información una vez creado. Sin embargo, si podríamos necesitar cambiar el número de páginas, por ejemplo para un libro que está escribiéndose (por ejemplo, pensemos en un diario personal).

```python
# Módulo libro.py
class Libro:
    def __init__(self, titulo: str, autor: str, paginas: int):
        self._titulo: str = titulo
        self._autor: str = autor
        self._paginas: int = paginas

    # Método getter para el título
    def get_titulo(self) -> str:
        return self._titulo

    # Método getter para el autor
    def get_autor(self) -> str:
        return self._autor

    # Métodos getter y setter para las páginas
    def get_paginas(self) -> int:
        return self._paginas

    def set_paginas(self, nuevas_paginas: int) -> bool:
        if nuevas_paginas > 0:
            self._paginas = nuevas_paginas
            return True
        return False
```

Por su parte, la clase `Persona` establece una **relación de uso** con la clase Libro haciendo uso del método *hojear_libro*. Como podemos observar, este método simplemente muestra un mensaje por pantalla en base a la información del libro que se ha pasado por parámetro, simulando un uso puntual de un objeto con el que no mantenemos ninguna vinculación como tal (no lo poseemos, no lo hemos leído, etc). Este es un ejemplo sencillo, aunque clarificador, de utilización de relación de uso: la clase Persona no tiene una relación a nivel de atributos con la clase Libro y se usa directamente el libro recibido como parámetro.

```python
# Módulo persona.py
class Persona:
    especie: str = "Humano"

    def __init__(self, nombre: str, edad: int, altura: float = None, activo: bool = None):
        self._nombre: str = nombre
        self._edad: int = edad
        self._altura: float = altura
        self._activo: bool = activo
        self._hobbies: list = []

        if hobbies is None:
            self._hobbies: list = []
        else:
            self._hobbies: list = hobbies

    # (...) Métodos get/set no mostrados por simplicidad

    # Método para hojear un libro (relación de uso)
    def hojear_libro(self, libro: Libro) -> None:
        print(f"{self.get_nombre()} está ojeando {libro}")
```

Finalmente, es importante destacar que en estos ejemplos asumiremos que todas las clases están en un mismo fichero *.py* por simplicidad y, por tanto, no se ha incluido ningún `import` de módulos en el código. Sin embargo, gracias a la sesión de prácticas previa, sabemos que el código de calidad debe estar correctamente organizado en **paquetes y módulos**.

4.1.2. Relación de Asociación

Una relación de **asociación** se establece cuando una clase mantiene una **referencia a otra clase**, es decir, una instancia de una clase A tiene un atributo que es una instancia de la clase B. A diferencia de la relación de uso, donde la interacción entre las clases es temporal, en una relación de asociación el vínculo es **más estable en el tiempo**. Sin embargo, este vínculo **puede cambiar**, ya que un objeto de la clase A puede asociarse con diferentes objetos de la clase B a lo largo de su ciclo de vida.

Un aspecto importante de la asociación es que **la existencia de un objeto de la clase A no depende de la existencia del objeto de la clase B**. Si la relación termina, el objeto de la clase A puede continuar existiendo, aunque la referencia al objeto de la clase B sea eliminada o quede en un estado nulo. Un ejemplo común es la relación entre un **empleado** y un **proyecto**. El empleado está asignado a un proyecto y realiza tareas en él, pero este vínculo puede cambiar si el proyecto finaliza o el empleado se reasigna a otro proyecto. El empleado sigue existiendo en la organización independientemente de la existencia de los proyectos con los que esté asociado.

4.1.2.1. Relación Unidireccional

Llevemos estos conceptos al ejemplo de las clases `Libro` y `Persona`. En primer lugar, vamos a definir una **relación de asociación unidireccional**, en la que una `Persona` ha leído un listado de libros. Como podemos ver, la clase `Libro` no sufre ningún cambio, ya que es `Persona` la que establece la relación hacia `Libro` de forma unidireccional.

```
# Módulo persona.py
class Persona:
    especie: str = "Humano"

    def __init__(self, nombre: str, edad: int, altura: float = None, activo: bool = None,
                 hobbies: list = None):
        self._nombre: str = nombre
        self._edad: int = edad
        self._altura: float = altura
        self._activo: bool = activo

        if hobbies is None:
            self._hobbies: list = []
        else:
            self._hobbies: list = hobbies

        self._libros_leidos: list[Libro] = []

    # (...) Métodos get/set no mostrados por simplicidad

    # Método para agregar un libro a la lista de libros leídos
    def leer_libro(self, libro: Libro) -> None:
        self.get_libros_leidos().append(libro)

    # Método para listar los libros leídos por la persona
    def listar_libros_leidos(self) -> None:
        print(f"Libros leídos por {self.get_nombre()}:")
        for libro in self.get_libros_leidos():
            print("> ", libro)
```

Así, para establecer esta relación unidireccional definiremos el listado de libros leídos como lista vacía ya que inicialmente una persona no ha leído libros. Además, definimos dos métodos adicionales: `leer_libro` y `listar_libros_leidos`. No tiene sentido en este contexto un método denominado `eliminar_libro_leido`, pues una vez que hemos leído un libro ya no podemos volver atrás en el tiempo y evitar leerlo (aunque a veces nos gustaría poder hacerlo).

4.1.2.2. Relación Bidireccional

Pasemos ahora a estudiar cómo podríamos definir una **relación de asociación bidireccional** entre las clases `Persona` y `Libro`. En este caso, ambas clases deben incluir un atributo que haga referencia a la clase opuesta. En concreto, la clase `Persona` contiene el **listado de libros leído** (ya definido en el apartado anterior), mientras que la clase `Libro` almacena el **listado de lectores** que han completado ese libro.

Por lo tanto, ahora se modifica la clase `Libro` para añadir un nuevo atributo, `listado_lectores`, que representa el listado de personas que han leído un libro concreto. En el constructor, dicha lista es vacía pues aún no habrá sido leído por nadie. Además, es interesante definir métodos de consulta y modificación: `get_listado_lectores` y `agregar_lector`:

```
# Módulo libro.py
class Libro:
    def __init__(self, titulo: str, autor: str, paginas: int):
        self._titulo: str = titulo
        self._autor: str = autor
        self._paginas: int = paginas
        self._listado_lectores: list[Persona] = []  # Lista de personas que han leído el libro

    # (...) Métodos get/set no mostrados por simplicidad

    # Métodos getter y setter para los lectores del libro
    def get_listado_lectores(self) -> list['Persona']:
        return self._listado_lectores

    def agregar_lector(self, persona: Persona) -> None:
        self.get_listado_lectores().append(persona)
```

Por otro lado, es necesario modificar el método `leer_libro` en la clase Persona para actualizar el listado de lectores del libro antes de ser añadido a la lista de libros leídos de la persona. De esta forma establecemos la relación bidireccional cuando vamos añadiendo libros leídos manualmente a una persona que ya existe.

```python
# Módulo persona.py
class Persona:
    # (...) Código omitido

    # Método para agregar un libro a la lista de libros leídos
    def leer_libro(self, libro: Libro) -> None:
        libro.agregar_lector(self)  # Asociar la persona con este libro
        self.get_libros_leidos().append(libro)
```

En base a este código, y una vez hemos creado un objeto de la clase Persona, se tendrá que invocar al método *leer_libro* cada vez que la persona haya completado un libro. Así, este método se encargará primero de marcar el libro como leído por dicha persona (pasando el self como parámetro) y, finalmente, añadir este libro a la lista de libros leídos de la persona.

4.1.3. Relación de Agregación

La relación de agregación es un **tipo especial de asociación** en la que una clase contiene o **tiene un objeto de otra clase**, pero **ambas pueden existir de forma independiente**. Esto se denomina relación ***has-a***. En la agregación, un objeto no depende del ciclo de vida del otro objeto: si el objeto contenedor (A) se destruye, los objetos que contiene (B) pueden seguir existiendo. Un ejemplo clásico es el de una **factura** y un **cliente**. La factura puede estar asociada a un cliente, pero el cliente seguirá existiendo aunque la factura sea eliminada. También, un cliente puede estar relacionado con varias facturas, pero las facturas no necesariamente tienen que destruirse si el cliente deja de existir. Otro ejemplo de agregación es la relación entre una **persona** y su **coche**. La persona puede tener uno o varios coches, pero si la persona ya no está, los coches pueden seguir existiendo sin ella. En este caso, el coche es parte de la persona en el sentido de que está vinculado a ella, pero ambos objetos pueden existir de manera independiente.

Retomemos el ejemplo de `Persona` y `Libro`. En este contexto, una posible relación de agregación consiste en que **una persona posee un listado de libros en propiedad, pero cada libro solo tiene únicamente un propietario**. A pesar de la relación, ambos objetos pueden existir independientemente:

- Si una persona es eliminada (deja de existir), los libros que poseía siguen existiendo y podrían transferirse a otro propietario o quedar sin dueño.
- Un libro tiene un único propietario, pero el libro puede seguir existiendo si se cambia su propietario o si la persona deja de existir.

Veamos cómo queda el código tras añadir esta nueva **relación de agregación bidireccional** entre clases:

```python
# Módulo libro.py
class Libro:
    def __init__(self, titulo: str, autor: str, paginas: int, propietario: 'Persona' = None):
        self._titulo: str = titulo
        self._autor: str = autor
        self._paginas: int = paginas
        self._listado_lectores: list['Persona'] = []  # Lista de personas que han leído el libro
        self._propietario: 'Persona' = propietario  # Relación de agregación: un libro tiene un propietario

    # (...) Métodos get/set no mostrados por simplicidad

    # Métodos getter y setter para el propietario del libro
    def get_propietario(self) -> 'Persona':
        return self._propietario

    def _set_propietario(self, propietario: 'Persona'):
        self._propietario = propietario

    def comprar_libro(self, propietario: 'Persona'):
        self._set_propietario(propietario)
```

Para realizar esta relación de agregación es necesario incluir un nuevo atributo, `propietario`, añadiéndose en el constructor. Además, hemos añadido un método get para obtener el propietario de un libro, y la funcionalidad de establecer el propietario. Sin embargo, podemos ver cómo el método get se ha definido como protegido. Así, en su lugar se expone el método *comprar_libro* por el que protegemos el método set. Por su parte, veamos los cambios a realizar en la clase Persona:

```python
# Módulo persona.py
class Persona:
    especie: str = "Humano"

    # Constructor con sobrecarga mediante parámetros opcionales
    def __init__(self, nombre: str, edad: int, altura: float = None, activo: bool = None,
                 hobbies: list = None:
        self._nombre: str = nombre
        self._edad: int = edad
        self._altura: float = altura
        self._activo: bool = activo
```

```
        if hobbies is None:
            self._hobbies: list = []
        else:
            self._hobbies: list = hobbies

        self._libros_leidos: list[Libro] = []

        # Relación de agregación: la persona tiene libros en propiedad
        self._libros_propiedad: list[Libro] = []

    # (...) Métodos get/set no mostrados por simplicidad

    # Método para comprar un libro (lo agrego a mi listado de libros en propiedad)
    def comprar_libro(self, libro: Libro) -> None:
        # Asociar el libro con esta persona como propietario (delegación de métodos)
        libro.comprar_libro(self)
        self._libros_propiedad.append(libro)

    # Método para listar los libros en propiedad de la persona
    def listar_libros_propiedad(self) -> None:
        print(f"Libros en propiedad de {self.get_nombre()}:")
        for libro in self.get_libros_propiedad():
            print("> ", libro)
```

En la clase Persona es necesario definir el atributo `libros_propiedad`, definido como un listado de libros que la persona posee. Además, definimos dos métodos: *comprar_libro* y *listar_libros_propiedad*. Es importante destacar dos aspectos relevantes del método encargado de comprar. En primer lugar, este método se encarga de establecer la relación bidireccional, pues invoca a la clase Libro para actualizar su propietario. Por otro lado, el método *comprar_libro* de la clase Persona realiza una invocación al método *comprar_libro* de la clase Libro, usando así delegación de métodos.

Es importante remarcar que una persona puede tener libros en propiedad que todavía no ha leído, libros que ha leído pero no tiene en propiedad, o libros que posee y que además ha leído.

4.1.4. Relación de Composición

La **relación de composición** es un tipo más fuerte de asociación, donde un objeto es una **parte esencial** de otro. A diferencia de la agregación, en la composición **los objetos dependen completamente del ciclo de vida del objeto contenedor**. Si el objeto que **contiene** se destruye, entonces todas sus **partes** también se destruyen. Esta relación se denomina ***part-of***. Otra característica clave de la composición es que el **objeto contenedor** es responsable de la creación y destrucción de los objetos **contenidos**. En la mayoría de los casos, el objeto contenido se inicializa y destruye junto con el contenedor. Un ejemplo típico de composición es la relación entre una **casa** y sus **habitaciones**. Las habitaciones no tienen sentido fuera del contexto de la casa; si la casa deja de existir, las habitaciones también dejan de hacerlo.

Vamos a implementar composición en el contexto de `Persona` y `Libro`. Para ello, podemos considerar un **diario** como un libro que solo puede ser utilizado por una persona concreta. Así, si la persona se elimina, el diario debería también eliminarse.

```
# Módulo persona.py
class Persona:
    especie: str = "Humano"

    # Constructor con sobrecarga mediante parámetros opcionales
    def __init__(self, nombre: str, edad: int, altura: float = None, activo: bool = None,
                 hobbies: list = None:
        self._nombre: str = nombre
        self._edad: int = edad
        self._altura: float = altura
        self._activo: bool = activo

        if hobbies is None:
            self._hobbies: list = []
        else:
            self._hobbies: list = hobbies

        self._libros_leidos: list[Libro] = []

        # Relación de agregación: la persona tiene libros en propiedad
        self._libros_propiedad: list[Libro] = []

        # Relación de composición
        self._diario: Libro = Libro(titulo="Diario", autor=self._nombre, paginas=0, propietario=self)

    # (...) Métodos get/set no mostrados por simplicidad
```

Como podemos ver, el atributo *diario* se crea en el constructor, definiendo un Libro en el que el título es "Diario", el nombre del autor es el nombre de la persona que lo crea, el número de páginas es cero (todavía está vacío), y el propietario es la propia persona que lo crea (se pasa como parámetro al constructor de Libro el self de la propia persona). Así,

podremos usar el diario como cualquier otro libro, con la diferencia de que si se elimina la persona, el diario también se debería eliminar (cosa que no pasaría con el resto de libros restantes). No implementaremos más funcionalidad sobre el diario por simplicidad.

4.1.5. Relación de Herencia

La **herencia** es un mecanismo fundamental en la programación orientada a objetos, donde una clase **hereda** atributos y métodos de otra, estableciendo una relación **es-un** (*is-a*). Esto permite que una clase hija utilice o modifique las propiedades y comportamientos de una clase padre o base. A través de la herencia, es posible **reutilizar código** de manera eficiente, evitando la duplicación y promoviendo la creación de jerarquías entre clases. En la siguiente sesión exploraremos en detalle cómo implementar herencia en Python.

4.2 Delegación

La **delegación** es un principio clave en el diseño de software, donde una clase **cederá** la ejecución de ciertos métodos o acciones en instancias de otras clases. Este enfoque es útil cuando se desea **distribuir la responsabilidad de ciertas funcionalidades entre diferentes clases**, lo que ayuda a mantener el **código más modular, claro y fácil de mantener**. La delegación permite que una clase principal no se sobrecargue con demasiada lógica, sino que reparta tareas específicas a otras clases especializadas. Un ejemplo común es cuando una clase *Rectángulo* tiene un método *trasladar*, que delega la tarea de ajustar las coordenadas en una instancia de la clase *Punto*, encargada de gestionar las posiciones x e y. Esta forma de dividir las responsabilidades mejora la reutilización de código y su legibilidad.

Ya hemos visto un ejemplo anteriormente, donde Persona y Libro definen un método con el mismo nombre: *comprar_libro*, donde el método de Persona delega funcionalidad en el de Libro. Además, podríamos aplicar delegación en otros aspectos del código:

1. **Delegación en la gestión de libros**: La clase Persona puede delegar las acciones relacionadas con los libros leídos o en propiedad a la clase `Libro`. Por ejemplo, si un libro es leído por una persona, la gestión de la lista de lectores o la asignación de un propietario puede delegarse en la instancia de Libro. De esta manera, la clase Persona no se encargará de los detalles internos de la gestión del libro, promoviendo una separación de responsabilidades.
2. **Delegación en la visualización de datos**: Cuando la clase `Persona` quiera mostrar sus libros leídos, puede delegar esta tarea a la clase `Libro`. Esto permite que cada clase sea responsable de gestionar y presentar su propia información, manteniendo el código más organizado y fácil de mantener.

Por simplicidad no implementaremos estas ideas de delegación. Sirven únicamente como ejemplo para comprender cómo podríamos aplicar estos principios en el código que implementemos.

4.3 Clonación de Objetos

La clonación de objetos es un proceso importante en POO, ya que permite crear copias de instancias sin alterar el objeto original. Existen dos tipos principales de clonación, que elegiremos en función del contexto particular:

1. **Copia superficial**: Esta técnica crea una nueva instancia de la clase, copiando los valores de los atributos del objeto original. Sin embargo, en el caso de los **atributos mutables**, como listas, diccionarios u otros objetos, **solo se copia la referencia** al mismo objeto. Esto implica que si se modifica el contenido mutable en la copia, también se verá reflejado en el objeto original, lo que puede dar lugar a efectos no deseados si no se tiene cuidado. En Python, la copia superficial puede realizar con el método `copy.copy()` del módulo `copy`.
2. **Copia profunda**: En contraste con la copia superficial, una copia profunda no solo crea una nueva instancia del objeto, sino que **también realiza una copia de todos los objetos referenciados dentro de los atributos mutables**. Esto significa que los objetos anidados también se clonan, lo que **previene la compartición de referencias entre el objeto original y la copia**. Este tipo de clonación es **más costosa** en términos de rendimiento, pero garantiza que la copia sea completamente independiente del objeto original. Se puede realizar con el método `copy.deepcopy()` del módulo `copy`.

Además de los métodos ofrecidos por el módulo `copy`, la clonación en Python también se puede personalizar en cada clase mediante los métodos mágicos `__copy__()` y `__deepcopy__()`, permitiendo un mayor control sobre el proceso de clonación para las clases definidas por el usuario.

Trabajaremos sobre la clase `Persona` para realizar, en primer lugar, una **copia superficial** de un objeto. Sin tener que realizar cambios en la clase `Persona`, podemos hacer uso del siguiente código en el *main* para realizar una copia superficial:

```python
# Módulo main.py
if __name__ == "__main__":
    # (...) Código omitido

    # Crear una copia superficial de persona1
    persona_copia = copy.copy(persona1)

    # Modificar el objeto original (lista de libros en propiedad)
    print("\nAntes de modificar persona1:")
    persona1.listar_libros_propiedad()
    persona_copia.listar_libros_propiedad()

    print("\nModificando la lista de libros en propiedad de persona1...")
    libro_nuevo = Libro("El hobbit", "J.R.R. Tolkien", 310)
    persona1.comprar_libro(libro_nuevo)

    # Mostrar los datos del original y de la copia después de la modificación
    print("\nDatos de persona1 tras la modificación:")
    persona1.listar_libros_propiedad()

    print("\nDatos de persona_copia tras la modificación en persona1 (copia superficial):")
    persona_copia.listar_libros_propiedad()
```

Un ejemplo de salida esperada podría sería la siguiente:

```
(...)

Modificando la lista de libros en propiedad de persona1...

Datos de persona1 tras la modificación:
Libros en propiedad de Sergio:
>  'El señor de los anillos' de J.R.R. Tolkien, 1178 páginas.
>  '1984' de George Orwell, 328 páginas.
>  'El hobbit' de J.R.R. Tolkien, 310 páginas.

Datos de persona_copia tras la modificación en persona1 (copia superficial):
Libros en propiedad de Sergio:
>  'El señor de los anillos' de J.R.R. Tolkien, 1178 páginas.
>  '1984' de George Orwell, 328 páginas.
>  'El hobbit' de J.R.R. Tolkien, 310 páginas.
```

Como podemos observar, el listado de libros en propiedad cambia en ambos objetos tras modificar la copia, debido a que las listas en Python son mutables y, por tanto, el objeto copiado almacena para esta lista una referencia (**aliasing**) a la lista del objeto original. Si queremos redefinir el comportamiento de la copia superficial de forma manual tendremos que definir el método mágico __copy__ en la clase Persona, de la siguiente forma:

```python
# Módulo persona.py
class Persona:
    # (...) Código omitido

    # Método mágico __copy__ para la copia superficial
    def __copy__(self):
        # Crear una nueva instancia de Persona
        nueva_persona = Persona(
            self._nombre,
            self._edad,
            self._altura,
            self._activo,
            self._hobbies  # Aquí no se crea una copia nueva de la lista, se usa la misma referencia
        )

        nueva_persona._diario = copy.copy(self._diario)  # Invocación a copy por ser un objeto
        nueva_persona._libros_leidos = self._libros_leidos  # Se copia la referencia
        nueva_persona._libros_propiedad = self._libros_propiedad

        return nueva_persona
```

En este código vemos que creamos un objeto *nueva_persona* a partir de los atributos de la propia persona (self). Además, debemos copiar aquellos atributos que no se usan en el constructor, como son el diario y las dos listas. En el caso de las listas de libros leídos y en propiedad, bastará con asignar el atributo de la clase actual. Sin embargo, para el diario, como es un objeto, debemos delegar en el método mágico *copy* de la clase Libro. En caso de que no esté definido (como es el caso actual), aplicará copia superficial por defecto. Finalmente, es esencial destacar que este código tendrá el mismo efecto que no incluirlo, pues implícitamente Python ya implementa la copia superficial cuando hacemos uso del método `copy()`.

En cuanto a la **copia profunda**, podemos hacer uso del método `deepcopy()` de forma equivalente a como lo hicimos con `copy()`. Si queremos implementar nuestra propia versión de copia profunda, lo podremos hacer de la siguiente forma:

```python
# Módulo persona.py
class Persona:
    # (...) Código omitido

    # Método mágico __deepcopy__ para la copia profunda
        def __deepcopy__(self, memo):
        nueva_persona = Persona(
            self._nombre,  # El nombre es inmutable (string), por lo que puede copiarse superficialmente
            self._edad,  # La edad es un entero (inmutable), también puede copiarse superficialmente
            self._altura,  # Altura es un float (inmutable)
            self._activo,  # Activo es un booleano (inmutable)
            copy.deepcopy(self._hobbies, memo),  # Se hace una copia profunda de la lista de hobbies
        )

        nueva_persona._diario = copy.deepcopy(self._diario, memo)
        nueva_persona._libros_leidos = copy.deepcopy(self._libros_leidos, memo)
        nueva_persona._libros_propiedad = copy.deepcopy(self._libros_propiedad, memo)

        return nueva_persona
```

En primer lugar, para aquellos atributos que se pasan en el constructor y que son inmutables, simplemente pasaremos el atributo al constructor. Sin embargo, como *hobbies* es una lista y, por tanto, mutable, se debe invocar a su vez a *deepcopy* sobre este atributo. Lo mismo sucede para aquellos atributos de la clase Persona que no se usan en el constructor, requiriendo hacer uso de *deepcopy* para todos ellos al ser mutables (dos listas y un objeto de la clase Libro)

Es importante destacar que el método mágico `__deepcopy__` requiere de un parámetro denominado memo en su funcionamiento interno. Sin embargo, nosotros cuando invoquemos al método lo haremos pasándole como parámetro únicamente el objeto que queremos copiar, de forma similar a *copy()*, abstrayéndonos de este aspecto. Esta copia profunda hará que los atributos inmutables generen una copia idéntica y, por tanto, no se produzca aliasing entre objetos.

4.4 Relación de ejercicios

Esta sesión de prácticas parte del código implementado en el último ejercicio de la Sesión 3 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 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).
    _relacion_lados: Relación entre los lados (regular,ew irregular).
    _lados: Lista de objetos Linea que representan los lados del polígono.
    __contraseña_base_datos: Contraseña interna del polígono.

Métodos:
    get_numero_poligonos()
    set_numero_poligonos()
    get_numero_lados()
    set_numero_lados()
    get_color()
    set_color()
    get_forma()
    set_forma()
    get_relacion_lados()
    set_relacion_lados()
    get_lados()
    set_lados()
    get_contraseña()
    __set_contraseña()
    cambia_contraseña()
    calcular_perimetro()
    escala_poligono()
    __str__()
    __eq__()
    __len__()
    __add__()
```

```
Clase Linea

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

Métodos:
    get_punto_inicio()
    set_punto_inicio()
    get_punto_fin()
    set_punto_fin()
    calcula_longitud()
    muestra_puntos()
```

```
Clase Punto

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

Métodos:
    get_coordenada_x()
    set_coordenada_x()
    get_coordenada_y()
    set_coordenada_y()
    get_cuadrante()
```

1. **Relación de uso**.
   a) Implementa el método **calcula_distancia_puntos** que calcule la distancia de Manhattan entre dos puntos, *¿en qué clase debes hacerlo?*
   b) ¿Podríamos aprovechar este nuevo método para reutilizarlo en funcionalidad ya existente? ¿Podríamos hacer uso de delegación de métodos entre clases en este ejercicio? Si la respuesta a alguna de estas preguntas es sí, implementa los cambios necesarios.
2. **Relación de asociación**. En este ejercicio, se trabajará con la idea de polígonos vecinos. Un polígono se considera vecino de otro si existe al menos un vértice de uno de los polígonos que se encuentra a una distancia menor o igual a un valor dado (distancia máxima) de cualquier vértice del otro polígono.
   a) Define el atributo **distancia_maxima_vecinos** que representa el valor de la distancia a partir de la cual dos polígonos ya no se considerarían vecinos. El valor de la distancia máxima es un **valor común a todos los polígonos existentes**. Por simplicidad, le asignaremos un valor de 10.
   b) Define el atributo **poligonos_vecinos** que representa un listado de polígonos vecinos asociados al polígono actual (inicialmente vacío).
   c) Implementa el método **es_vecino** capaz de comparar si otro polígono es vecino del polígono actual.
   d) Implementa el método **agrega_vecino** para añadir un polígono a la lista de polígonos, siempre que se cumplan las condiciones para serlo.
3. **Relación de agregación**. Actualmente tenemos las clases Punto, Línea y Polígono definidas en el proyecto:
   - ¿Existe alguna relación de agregación entre ellas?
   - ¿Se te ocurren ejemplos de otras clases con las que se podrían agregar?
4. **Relación de composición**. Amplía la clase Poligono para que contenga un **conjunto** de vértices.
   a) Añade el método **get_puntos** a la clase Linea para que devuelva los dos puntos que componen la línea.
   b) Define el atributo **vertices** que representa la colección de vértices como un conjunto de puntos, creado a partir de las líneas que definen a un polígono.
   c) ¿Debemos modificar métodos ya existentes en la clase Poligono?

   Nota: Este ejercicio asume que existe coherencia en las líneas que forman un polígono. Sin embargo, como no se implementan mecanismos de coherencia, podría pasar que las líneas establecidas no generen realmente un polígono en el código actual.
5. **Clonación**. Este ejercicio hace uso de los métodos **copy** y **deepcopy** proporcionados por el módulo *copy* de Python (sin implementar métodos mágicos) sobre un objeto de la clase Poligono. En tu función *main* realiza:
   a) **Copia superficial**: Clona de manera superficial un polígono. Modifica los atributos **color** y **lados** del polígono clonado. ¿Se modifica el color en ambos objetos? ¿Y los lados? ¿Por qué?
   b) **Copia profunda**: Clona de manera profunda un polígono. Modifica los atributos **color** y **lados** del polígono clonado. ¿Se modifica el color en ambos objetos? ¿Y los lados? ¿Por qué?

Nota: para hacer este ejercicio crea un método en la clase Polígono llamado **elimina_lado_aleatorio**. Este método seleccionará uno de los lados del polígono de forma aleatoria y lo eliminará. ¿Sería necesario actualizar el conjunto de vértices?

6. **Relación de asociación bidireccional**. Implementa los siguientes pasos:

   a) Haz las modificaciones necesarias para que un punto almacene la línea a la que pertenece.
   b) Haz las modificaciones necesarias para que una línea almacene el polígono al que pertenece.

UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 5 de Prácticas

Herencia

Autores: Alejandro Buitrago López (alejandro.buitragol@um.es), Sergio López Bernal (slopez@um.es), Javier Pastor Galindo (javierpg@um.es), Jesús Damián Jiménez Re (jesusjr@um.es) y Gregorio Martínez Pérez (gregorio@um.es)

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2024-2025

Última modificación:
7 de octubre de 2024

Sesión 5: Herencia

Índice

5.1 Herencia en Python

La **herencia** es uno de los pilares fundamentales de la Programación Orientada a Objetos (POO) en Python. Permite que una clase (denominada **clase hija** o **subclase**) herede comportamientos y características de otra clase (la **clase padre** o **superclase**), permitiendo la reutilización de código y la extensión de funcionalidad sin necesidad de reescribir lo que ya existe en la clase padre.

Herencia de Métodos

Cuando heredamos métodos de una clase padre, tenemos **tres formas principales** de gestionar cómo se comportan esos métodos en la clase hija:

- **Heredar y usar el método tal cual**:
  - La subclase hereda un método definido en la clase padre y lo usa sin hacer modificaciones. El método funciona exactamente como en la superclase, sin necesidad de redefinirlo.
  - Esto es útil cuando el comportamiento del método es adecuado para ambas clases, padre e hija, sin cambios.

```
class Padre:
    def metodo(self):
        print("Método en clase Padre")

class Hija(Padre):
    pass  # No se modifica el método, se hereda tal cual

hija = Hija()
hija.metodo()  # Salida: "Método en clase Padre"
```

- **Extender el método del padre**:
  - La subclase puede añadir funcionalidad extra al método heredado. En este caso, se usa **super()** para llamar al método del padre, ejecutando su lógica, y luego se añade nueva funcionalidad específica de la subclase.
  - Esto es útil cuando se quiere mantener parte del comportamiento original del método del padre, pero también se necesita algo adicional en la subclase.

```
class Padre:
    def metodo(self):
        print("Método en clase Padre")

class Hija(Padre):
    def metodo(self):
        super().metodo()  # Llama al método de la clase Padre
        print("Método extendido en clase Hija")

hija = Hija()
hija.metodo()
# Salida:
# "Método en clase Padre"
# "Método extendido en clase Hija"
```

- **Sobreescribir el método**:
  - La subclase puede cambiar por completo el comportamiento del método heredado. Para ello, simplemente redefine el método con la misma declaración, pero con una nueva implementación que no tiene ninguna relación con la del método original.
  - Esto se utiliza cuando el comportamiento del método del padre no es adecuado o necesario para la subclase.

```
class Padre:
    def metodo(self):
        print("Método en clase Padre")

class Hija(Padre):
    def metodo(self):
        print("Método completamente nuevo en clase Hija")

hija = Hija()
hija.metodo()  # Salida: "Método completamente nuevo en clase Hija"
```

Herencia de Atributos

En cuanto a los atributos de una clase, también existen **tres formas principales** de gestionarlos al aplicar herencia:

- **Heredar y usar los atributos tal cual**:
  - Los atributos definidos en la clase padre son heredados por la subclase y se utilizan sin modificación.
  - Esto es útil cuando las subclases necesitan comportarse de manera similar a la clase padre y los atributos no requieren cambios.

```
class Padre:
    def __init__(self, atributo: str):
        self.atributo = atributo

class Hija(Padre):
    def __init__(self, atributo: str):
        super().__init__(atributo)

hija = Hija("Atributo en clase Hija")
print(hija.atributo)  # Salida: "Atributo en clase Hija"
```

- **Extender los atributos del padre**:
  - Además de los atributos heredados de la clase padre, podemos definir nuevos atributos en la clase hija.
  - Muy común, pues las clases hijas suelen ser una especialización de la clase padre y, por tanto, tienden a definir propiedades adicionales.

```
class Padre:
    def __init__(self, atributo: str):
        self.atributo = atributo

class Hija(Padre):
    def __init__(self, atributo: str, nuevo: str):
        super().__init__(atributo)
        self.nuevo = nuevo

hija = Hija("Atributo en clase Hija", "Atributo nuevo")
print(hija.atributo)  # Salida: "Atributo en clase Hija"
print(hija.nuevo)     # Salida: "Atributo nuevo"
```

- **Ocultar el atributo del padre**:
  - La subclase puede ocultar un atributo heredado del padre redefiniéndolo con el mismo nombre. En este caso, el atributo de la clase hija **sobrescribe** el de la clase padre, de modo que cualquier referencia a dicho atributo usará la versión de la subclase.
  - Este enfoque se usa cuando el mismo atributo debe tener un significado diferente o un valor específico en la subclase.

```
class Padre:
    def __init__(self):
        self.atributo = "Atributo en clase Padre"

class Hija(Padre):
    def __init__(self):
        self.atributo = "Atributo en clase Hija"  # Oculta el atributo de la clase Padre

hija = Hija()
print(hija.atributo)  # Salida: "Atributo en clase Hija"
```

5.2 Ejemplo de herencia: clases Persona y Libro

Partiendo de la base del código ya presentado en la sesión 4 de prácticas, vamos a extender nuestro proyecto para disponer de dos tipos de libros: libros infantiles y libros científicos. Recordemos que hasta ahora tenemos la clase `Libro`:

```
class Libro:
    def __init__(self, titulo: str, autor: str, paginas: int, propietario: 'Persona' = None):
        self._titulo: str = titulo
        self._autor: str = autor
        self._paginas: int = paginas
        self._listado_lectores: list['Persona'] = []  # Lista de personas que han leído el libro
        self._propietario: 'Persona' = propietario  # Relación de agregación: un libro tiene un propietario

    # Método getter para el título del libro
    def get_titulo(self) -> str:
        return self._titulo

    # Método getter para el autor del libro
    def get_autor(self) -> str:
        return self._autor

    # Método getter/setter para el número de páginas del libro
    def get_paginas(self) -> int:
        return self._paginas

    # Método setter para el número de páginas del libro (caso del diario)
    def set_paginas(self, nuevas_paginas: int) -> bool:
        if nuevas_paginas > 0:
            self._paginas = nuevas_paginas
            return True
        return False

    # Método getter/setter para los lectores del libro
    def get_listado_lectores(self) -> list['Persona']:
        return self._listado_lectores

    def agregar_lector(self, persona: 'Persona') -> None:
        self.get_listado_lectores().append(persona)

    # Métodos getter y setter para el propietario del libro
    def get_propietario(self) -> 'Persona':
        return self._propietario

    def _set_propietario(self, propietario: 'Persona'):
        self._propietario = propietario

    def comprar_libro(self, propietario: 'Persona'):
        self._set_propietario(propietario)

    # Método mágico __str__ para obtener la información del libro
    def __str__(self) -> str:
        if len(self.get_listado_lectores()) > 0:
            lectores = ", ".join([persona.get_nombre() for persona in self.get_listado_lectores()])
        else:
            lectores = "Nadie lo ha leído"

        if self.get_propietario():
            propietario = self.get_propietario().get_nombre()
        else:
            propietario = "Sin propietario"

        return (f"'{self.get_titulo()}' de {self.get_autor()}, {self.get_paginas()} páginas. "
                f"Leído por: {lectores}")
```

Los libros infantiles se caracterizan por estar dirigidos a niños, por lo que añaden nuevas propiedades sobre los libros comunes: tienen una edad recomendada de lectura, pueden tener o no ilustraciones, y pueden ser o no interactivos. Por su parte, los libros científicos se caracterizan por estar enmarcados en el contexto de un campo de estudio, tienen un determinado nivel de dificultad para comprenderlos y, finalmente, disponen de un listado de referencias a otros libros en los que se apoyan a nivel bibliográfico.

En base a ello, presentamos inicialmente el código de la clase LibroInfantil:

```
class LibroInfantil(Libro):

    def __init__(self, titulo: str, autor: str, paginas: int, edad_recomendada: int, ilustraciones: bool,
                 interactividad: bool, propietario: 'Persona' = None):
        super().__init__(titulo, autor, paginas, propietario) # Llamada al constructor de la clase padre

        # Nuevos atributos de la clase hija
        self._edad_recomendada = edad_recomendada # Edad recomendada de lectura
        self._ilustraciones = ilustraciones # El libro tiene ilustraciones
        self._interactividad = interactividad # El libro es interactivo

    # (...) Métodos getter/setter obviados por simplicidad: métodos nuevos sobre la clase padre
```

```
    # Ejemplo de extensión de funcionalidad heredada
    def __str__(self) -> str:
        # Obtiene la descripción básica del libro (superclase)
        descripcion_basica = super().__str__()

        # Detalles extra por ser libro infantil
        detalles_infantiles = (f"Edad recomendada: {self._edad_recomendada}, "
                               f"Ilustraciones: {'Sí' if self._ilustraciones else 'No'}, "
                               f"Interactividad: {'Sí' if self._interactividad else 'No'}")
        return f"{descripcion_basica} | {detalles_infantiles}"

    '''
    Ejemplo de extensión de funcionalidad:
        - Se añade funcionalidad al método de la clase hija: comprobación de la edad
        - Se hace uso de super().agregar_lector() para llamar al método de la clase padre
    '''
    def agregar_lector(self, persona: 'Persona') -> None:
        if persona.get_edad() < self._edad_recomendada:
            print(f"{persona.get_nombre()} es demasiado joven para leer este libro infantil.")
        else:
            # Si la edad es adecuada, llamamos al método de la clase padre
            super().agregar_lector(persona)
            print(f"{persona.get_nombre()} ha sido añadido como lector del libro '{self._titulo}'.")
```

En relación a los atributos, se definen simplemente tres nuevos atributos que extienden la definición dada por la clase Libro, no realizando en ningún caso ocultación de los atributos de la clase padre. En cuanto a la herencia de métodos podemos ver cómo LibroInfantil hereda todos los métodos definidos en la clase Libro, tales como los getter/setter y el método *comprar_libro*. En estos casos, se realizará un uso directo de los métodos definidos en la clase Libro, no teniendo que realizar ningún cambio en la clase LibroInfantil. Además, la clase LibroInfantil definiría nuevos métodos getter/setter sobre los atributos adicionales, que no se dejan indicados en el código por brevedad.

Sin embargo, esta clase presenta dos casos de extensión de funcionalidad sobre métodos definidos en la superclase. En primer lugar, el método __str__ define su propia implementación teniendo como base la de la clase Libro. Para ello, hace uso de *super()* sobre el método __str__ de la clase padre y, una vez ha obtenido la descripción básica, pasa a cumplimentar la información específica de los libros infantiles. Una vez terminada, este método ahora se encarga de mostrar la información tanto básica como la específica infantil.

En segundo lugar, se extiende la funcionalidad del método *agregar_lector* para añadir una restricción de edad. Recordemos que un libro infantil tiene una edad de lectura recomendada, por lo que no debería poder leerse si la persona es menor a dicha recomendación. En el caso de que el lector sea mayor de esa edad, sí podremos agregar el libro, para lo que nos apoyamos del método *agregar_lector* de la superclase, otra vez haciendo uso de *super()*. Finalmente, en esta clase no hay ningún método que sobrescriba por completo la funcionalidad heredada.

Pasemos ahora a revisar el contenido de la clase LibroCientifico:

```
class LibroCientifico(Libro):
    def __init__(self, titulo: str, autor: str, paginas: int, campo_estudio: str, nivel_dificultad: str,
                 propietario: 'Persona' = None):
        super().__init__(titulo, autor, paginas, propietario)

        # Nuevos atributos de la clase hija
        self._campo_estudio: str = campo_estudio
        self._nivel_dificultad: str = nivel_dificultad
        self._referencias: list[Libro] = []

    # (...) Métodos getter/setter obviados por simplicidad: métodos nuevos sobre la clase padre

    # Ejemplo de extensión de funcionalidad heredada
        def __str__(self) -> str:
        descripcion_basica = super().__str__()  # Llamada al método __str__ de la clase padre

        # Construcción de los detalles específicos del libro científico
        detalles_cientificos = (f"Campo de estudio: {self._campo_estudio}, "
                                f"Nivel de dificultad: {self._nivel_dificultad}")

        # Añadir las referencias si existen
        if self._referencias:
            referencias_str = ", ".join([referencia.get_titulo() for referencia in self._referencias])
            detalles_cientificos += f", Referencias: {referencias_str}"
        else:
            detalles_cientificos += ", No hay referencias"

        return f"{descripcion_basica} | {detalles_cientificos}"

    # Ejemplo de nueva funcionalidad añadida en la clase hija
    def agregar_referencia(self, referencia: Libro) -> None:
        self._referencias.append(referencia)

    '''
    Sobrescritura completa del método:
        - Se añade funcionalidad para verificar si el propietario tiene experiencia
```

```
        previa con libros científicos
        - Se cambia el tipo de retorno, de None a bool
    '''
    def comprar_libro(self, propietario: 'Persona') -> bool:
        # Verificar si la persona ha leído libros científicos con anterioridad.
        # Uso de isinstance() para verificar el tipo de libro
        libros_cientificos_leidos = []
        for libro in propietario.get_libros_leidos():
            if type(libro) is LibroCientifico:
                libros_cientificos_leidos.append(libro)

        if len(libros_cientificos_leidos) >= 2:  # Tiene experiencia previa leyendo libros científicos
            self._set_propietario(propietario)
            return True
        else:
            return False
```

En este caso también se definen tres nuevos atributos adicionales a los que proporciona la clase padre, sin realizar ocultación de ninguno de ellos. En relación a los métodos, se realiza extensión de funcionalidad en __str__ de forma análoga a lo que se ha explicado previamente en LibroInfantil. Además, se añade sobre la clase padre métodos getter/setter sobre los nuevos atributos. Como ejemplo se muestra el método *agregar_referencia*, que permite añadir nuevas referencias bibliográficas a un libro científico.

Finalmente, el comportamiento del método *comprar_libro* se sobrescribe por completo. En esta nueva versión, decidimos que solo se podrá comprar un libro científico si el lector tiene experiencia previa en la lectura de este tipo de libros (en el mundo real esto no sucedería, pero nos sirve para ilustrar este tipo de métodos). Así, se recupera el listado de libros leídos de la persona y, si hay al menos dos libros científicos, podrá comprar el libro. Es interesante destacar el uso del método type(), que permite comprobar en tiempo de ejecución de qué clase es un objeto dado.

Finalmente, se muestra un ejemplo de *main* en el que se invoca a la funcionalidad indicada previamente:

```
# Crear instancias de Persona
persona1 = Persona("Paco", 9)
persona2 = Persona("Sergio", 25)  # Persona adulta

# Crear un libro base y probar su funcionalidad
libro1 = Libro("El Hobbit", "J.R.R. Tolkien", 300)
persona2.leer_libro(libro1)
print("Libro1: ", libro1)

# Crear un libro infantil y probar su funcionalidad
libro_infantil = LibroInfantil("El Principito", "Antoine de Saint-Exupéry", 96, 10, True, True)
print("Libro infantil: ", libro_infantil)

# Agregar lector al libro infantil
libro_infantil.agregar_lector(persona1)  # Edad inadecuada (9 años)
libro_infantil.agregar_lector(persona2)  # Edad adecuada (25 años)

# Crear libros científicos y probar su funcionalidad
libro_cientifico1 = LibroCientifico("Física Cuántica", "Einstein", 500, "Física", "Avanzado")
libro_cientifico2 = LibroCientifico("Relatividad General", "Einstein", 300, "Física", "Avanzado")
print("Libro científico 1: ", libro_cientifico1)
print("Libro científico 2: ", libro_cientifico2)

# Probar la compra de los libros científicos según la experiencia de la persona
persona2.leer_libro(libro_cientifico1)  # Sergio lee un libro científico
resultado_compra1 = libro_cientifico1.comprar_libro(persona2)  # No tiene suficiente experiencia
print(f"Resultado de la compra del libro científico 1: {resultado_compra1}")  # False

persona2.leer_libro(libro_cientifico2)  # Sergio lee otro libro científico
resultado_compra2 = libro_cientifico1.comprar_libro(persona2)  # Ahora puede comprar el libro
print(f"Resultado de la compra del libro científico 1 después de leer más: {resultado_compra2}")  # True

# Ejemplo de agregar una referencia solo en el libro científico
libro_cientifico1.agregar_referencia(libro_cientifico2)

listado_referencias = libro_cientifico1.get_referencias()

print(f"Referencias en el libro científico 1: ")
for referencia in libro_cientifico1.get_referencias():
    print("> ", referencia)

# Usar type() para comparar tipos exactos
print("libro_infantil es de tipo LibroInfantil? ", type(libro_infantil) is LibroInfantil)  # True
print("libro_cientifico1 es de tipo LibroCientifico? ", type(libro_cientifico1) is LibroCientifico)  # True
# False, no es exactamente de tipo Libro
print("libro_cientifico1 es de tipo Libro? ", type(libro_cientifico1) is Libro)

# Usar isinstance() para verificar si un objeto es de una clase o una subclase
print("libro_infantil es instancia de LibroInfantil? ", isinstance(libro_infantil, Libro)) # True
print("libro_cientifico1 es instancia de Libro? ", isinstance(libro_cientifico1, Libro))  # True
print("libro_cientifico1 es instancia de LibroInfantil? ",
    isinstance(libro_cientifico1, LibroInfantil)) # False
```

```
# Usar issubclass() para verificar si una clase es una subclase de otra
print("LibroCientifico es subclase de Libro? ", issubclass(LibroCientifico, Libro))  # True
print("LibroInfantil es subclase de Libro? ", issubclass(LibroInfantil, Libro)) # True
print("LibroInfantil es subclase de LibroCientifico? ", issubclass(LibroInfantil, LibroCientifico))  # False
```

Así, podemos ver que el código crea dos personas y les añade libros, estableciendo las relaciones entre clases de forma adecuada, tal y como vimos en la sesión de prácticas anterior. Así, se define un libro infantil que es leído por ambas personas. Sin embargo, el método *agregar_lector* incluye en el caso de LibroInfantil una restricción en base a la edad de la persona. Por otro lado, en relación a los libros científicos, podemos ver cómo para comprar un libro científico la persona debe haber leído antes al menos dos libros. Finalmente, vemos un ejemplo de uso de *type()*, *isinstance()* y *issubclass()*.

5.3 Relación de ejercicios

Esta sesión de prácticas parte del código implementado en el último ejercicio de la Sesión 4 donde tenemos implementadas completamente las clases de Polígono, Punto y Línea, incluyendo relaciones entre ellas. A continuación se listan los atributos y métodos que deberían estar definidos en cada clase.

```
Clase Poligono

Atributos:
    numero_poligonos: Atributo de clase que cuenta el número de polígonos creados.
    distancia_maxima_vecinos: Distancia máxima para determinar vencidad de polígonos.
    _numero_lados: Número de lados del polígono.
    _color: Color del polígono.
    _forma: Forma del polígono (convexo, cóncavo).
    _relacion_lados: Relación entre los lados (regular,ew irregular).
    _lados: Lista de objetos Linea que representan los lados del polígono.
    __contraseña_base_datos: Contraseña interna del polígono.
    _poligonos_vecinos: Listado de polígonos vecinos a un polígono.
    _vertices: Listado de vértices asociados a un polígono.

Métodos:
    get_numero_poligonos()
    set_numero_poligonos()
    get_numero_lados()
    set_numero_lados()
    get_color()
    set_color()
    get_forma()
    set_forma()
    get_relacion_lados()
    set_relacion_lados()
    get_lados()
    set_lados()
    elimina_lado_aleatorio()
    __get_contraseña()
    __set_contraseña()
    cambia_contraseña()
    calcula_perimetro()
    escala_poligono()
    es_vecino()
    agrega_vecino()
    __extrae_vertices()
    __str__()
    __eq__()
    __len__()
    __add__()
    __lt__()
```

```
Clase Linea

Atributos:
    _punto_inicio: Punto inicial de la línea.
    _punto_fin: Punto final de la línea.
    _poligono: Polígono al que pertenece.

Métodos:
    get_punto_inicio()
    set_punto_inicio()
    get_punto_fin()
    set_punto_fin()
    get_poligono()
    set_poligono()
    calcula_longitud()
    muestra_puntos()
    get_puntos()
```

```
Clase Punto

Atributos:
```

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

Métodos:
    get_coordenada_x()
    set_coordenada_x()
    get_coordenada_y()
    set_coordenada_y()
    get_linea()
    set_linea()
    get_cuadrante()
    calcula_distancia_puntos()
```

Partiendo de la definición de la clase Poligono, vamos a implementar tres nuevos tipos de polígonos: cuadrados, triángulos y círculos. Para ello, veamos cuáles son sus características principales.

1. **Clase Triángulo**.
   - **Atributos**: Un triángulo se caracteriza por ser un polígono de tres lados. Además, un triángulo puede ser de un determinado tipo: escaleno, isósceles o equilátero. Esta propiedad la denominaremos **tipo_triangulo**. *Nota: recuerda que podemos heredar los atributos del padre, extenderlos u ocultarlos*.
   - **Métodos**:
     a) Para el cálculo del perímetro, un triángulo debe verificar que tiene exactamente tres lados. En caso contrario, no se podrá hacer uso de esta funcionalidad.
     b) Un triángulo debe permitir calcular su área de forma conveniente. Para ello, puedes hacer uso de la fórmula de Herón: $A = \sqrt{s(s-a)(s-b)(s-c)}$, donde a, b y c representan las longitudes de cada uno de los tres lados del triángulo, y s representa el semiperímetro, calculado como $s = \frac{a+b+c}{2}$.
     c) No se puede permitir eliminar un lado aleatorio de un triángulo, pues dejaría de ser un polígono. Haz los cambios necesarios para tener en cuenta esta restricción.
     d) Revisa los métodos mágicos de esta clase y modifícalos en caso de que sea necesario. Si hay que modificarlos, considera si se trata de una extensión de funcionalidad o de una reescritura completa.
   - **Preguntas**:
     a) ¿Se aprovechan todos los atributos de la clase Polígono en la clase Triángulo?
     b) ¿La clase Triangulo se beneficia de la definición de atributos de la clase Poligono? ¿Tendría sentido entonces definir la relación de herencia?
     c) ¿Tenemos métodos añadidos sobre los definidos en la clase padre? ¿Hay métodos en la superclase que no tengan sentido en la subclase y, por tanto, tengamos que sobreescribirlos?
2. **Clase Círculo**.
   - **Atributos**: Aunque un círculo no es realmente un polígono, en esta sesión lo consideraremos como tal debido a fines didácticos. Así, un círculo tiene como propiedades su **radio** y su **centro**.
   - **Métodos**:
     a) Un círculo debe permitir calcular su área y su perímetro de forma conveniente.
     b) Adapta el método *es_vecinos()* para que los cálculos se realicen en base a la distancia al centro del círculo (entre centros si se trata de dos círculos, o del centro del círculo a cualquier otro vértice del polígono en caso contrario).
     c) El escalado de un círculo consiste en modificar su radio en base a los valores de factor y ajuste, en lugar de su número de lados.
     d) Realiza los cambios necesarios en los métodos para que el código sea compatible con las restricciones con la clase Poligono analizadas el apartado de atributos.
     e) Revisa los métodos mágicos de esta clase y modifícalos en caso de que sea necesario. Si hay que modificarlos, considera si se trata de una extensión de funcionalidad o de una reescritura completa.
   - **Preguntas**:
     a) ¿Se aprovechan todos los atributos de la clase Polígono en la clase Circulo?
     b) ¿La clase Circulo se beneficia de la definición de atributos de la clase Poligono? ¿Tendría sentido entonces definir la relación de herencia?
     c) ¿Tenemos métodos añadidos sobre los definidos en la clase padre? ¿Hay métodos en la superclase que no tengan sentido en la subclase y, por tanto, tengamos que sobreescribirlos?
3. **Clase Cuadrado**.
   - **Atributos**: Un cuadrado se caracteriza por ser un polígono de cuatro lados. Además, definiremos como

propiedad de un cuadrado la longitud de su diagonal, denominada **longitud_diagonal**, obtenida a partir de la fórmula $longitud\_lado * sqrt(2)$.

- **Métodos**:
  a) Para el cálculo del perímetro, un cuadrado debe verificar que tiene exactamente cuatro lados. En caso contrario, no se podrá hacer uso de esta funcionalidad.
  b) Un cuadrado debe permitir calcular su área de forma conveniente.
  c) Revisa los métodos mágicos de esta clase y modifícalos en caso de que sea necesario. Si hay que modificarlos, considera si se trata de una extensión de funcionalidad o de una reescritura completa.
- **Preguntas**:
  a) ¿Se aprovechan todos los atributos de la clase Polígono en la clase Cuadrado?
  b) ¿La clase Cuadrado se beneficia de la definición de atributos de la clase Poligono? ¿Tendría sentido entonces definir la relación de herencia?
  c) ¿Tenemos métodos añadidos sobre los definidos en la clase padre? ¿Hay métodos en la superclase que no tengan sentido en la subclase y, por tanto, tengamos que sobreescribirlos?

4. **Definición de nuevas subclases**. Define las clases TrianguloEscaleno, TrianguloIsosceles y TrianguloEquilatero para que podamos instanciar objetos de esos tipos, gestionando la herencia y haciendo las comprobaciones pertinentes para que estas clases sean coherentes. *¿Se extienden nuevos atributos? ¿Se ocultan atributos existentes? ¿Hay sobreescritura completa de métodos o sólo extensiones de los mismos?*

UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 6 de Prácticas

Clases Abstractas

Autores: Alejandro Buitrago López (alejandro.buitragol@um.es), Sergio López Bernal (slopez@um.es), Javier Pastor Galindo (javierpg@um.es), Jesús Damián Jiménez Re (jesusjr@um.es) y Gregorio Martínez Pérez (gregorio@um.es)

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2024-2025

Última modificación:
14 de octubre de 2024

Sesión 6: Clases Abstractas

Índice

6.1 Definición de una clase abstracta

En Python, las clases abstractas proporcionan una forma de definir una estructura o comportamiento común que otras clases pueden heredar. Una clase abstracta no se puede instanciar directamente; en su lugar, se utiliza como una plantilla que garantiza que las subclases implementen ciertos métodos necesarios. Las clases abstractas en Python se manejan a través del módulo `abc` (Abstract Base Classes).

Para definir una clase abstracta, dicha clase debe heredar de `ABC` y marcar los métodos que deben ser implementados por las subclases con el decorador `@abstractmethod`, de forma similar a como se gestionan los métodos estáticos y de clase. De esta manera, las subclases están obligadas a proporcionar una implementación de estos métodos abstractos. Esto garantiza que las subclases sigan el comportamiento esperado.

Para ilustrar este concepto, consideremos el caso de una clase abstracta llamada `Vehiculo`. Un `Vehiculo` tiene atributos comunes como `marca` y `modelo`, pero no tiene sentido instanciar un vehículo sin saber qué tipo de vehículo es, por ejemplo, un coche o una moto. Por lo tanto, podemos definir una clase abstracta `Vehiculo`, donde algunos métodos, como `arrancar()`, no tienen una implementación predeterminada y deben ser implementados por las subclases, como `Coche` y `Moto`.

Aquí tienes un ejemplo básico de la definición de la clase abstracta `Vehiculo` con métodos abstractos:

```
from abc import ABC, abstractmethod

# Definición de la clase abstracta Vehiculo
class Vehiculo(ABC):

    @abstractmethod
    def arrancar(self) -> None:
        pass

    @abstractmethod
    def detener(self) -> None:
        pass
```

En este ejemplo, hemos definido la clase `Vehiculo` con dos métodos abstractos: `arrancar()` y `detener()`. Estos métodos no tienen una implementación concreta, ya que cada subclase (como `Coche` o `Moto`) deberá proporcionar su propia implementación.

6.2 Constructor en una clase abstracta

Las clases abstractas pueden incluir constructores que permitan inicializar atributos comunes en las subclases. Esto es útil cuando hay propiedades compartidas entre todas las subclases que necesitan ser inicializadas de manera uniforme. En nuestro ejemplo, tanto los coches como las motos tienen una `marca` y un `modelo`, por lo que podemos inicializar estos atributos en la clase abstracta `Vehiculo`. Sin embargo, es esencial destacar que **usar el constructor de una clase abstracta no crea objetos de esa clase**. Se usa simplemente para la **inicialización de atributos comunes a sus subclases.**

Veamos cómo añadir un constructor a la clase `Vehiculo`:

```python
from abc import ABC, abstractmethod

class Vehiculo(ABC):

    def __init__(self, marca: str, modelo: str) -> None:
        self._marca = marca
        self._modelo = modelo

    # (...) Obviamos métodos get/set

    @abstractmethod
    def arrancar(self) -> None:
        pass

    @abstractmethod
    def detener(self) -> None:
        pass
```

Aquí hemos añadido un constructor a la clase abstracta `Vehiculo`, que inicializa los atributos `marca` y `modelo`. Aunque no podemos instanciar directamente un `Vehiculo`, las subclases que heredan de `Vehiculo` podrán llamar a este constructor para inicializar estos atributos comunes.

Ahora definamos las subclases `Coche` y `Moto`, que heredan de `Vehiculo` y proporcionan sus propias implementaciones de los métodos abstractos. Además, estas clases definen sus propios atributos que definen características propias a cada tipo de vehículo, como el número de puertas en el caso de un coche, o el tipo en el caso de una moto.

```python
class Coche(Vehiculo):

    def __init__(self, marca: str, modelo: str, numero_puertas: int) -> None:
        super().__init__(marca, modelo)
        self._numero_puertas = numero_puertas

    # (...) Obviamos métodos get/set

    def arrancar(self) -> None:
        print(f"El coche {self.get_marca()} {self.get_modelo()} con {self.get_numero_puertas()} puertas
            está arrancando.")

    def detener(self) -> None:
        print(f"El coche {self.get_marca()} {self.get_modelo()} con {self.get_numero_puertas()} puertas
            se ha detenido.")


class Moto(Vehiculo):

    def __init__(self, marca: str, modelo: str, tipo: str) -> None:
        super().__init__(marca, modelo)
        self.tipo = tipo

    # (...) Obviamos métodos get/set

    def arrancar(self) -> None:
        print(f"La moto {self.get_marca()} {self.get_modelo()} de tipo {self.get_tipo()} está arrancando.")

    def detener(self) -> None:
        print(f"La moto {self.get_marca()} {self.get_modelo()} de tipo {self.get_tipo()} se ha detenido.")
```

Ahora las subclases `Coche` y `Moto` hacen uso del constructor de `Vehiculo` y proporcionar sus propias implementaciones de `arrancar()` y `detener()`.

6.3 Uso de métodos no abstractos

En algunas situaciones puede ser útil definir un método no abstracto en la clase abstracta. Esto es útil cuando todas las subclases deben tener una funcionalidad básica que puede ser reutilizada, pero que también puede ser sobrescrita si es necesario. Por tanto, podemos ver cómo no es obligatorio que todos los métodos de una clase abstracta sean abstractos. Así, una clase será abstracta si tiene al menos un método abstracto.

En el caso de `Vehiculo`, podríamos añadir un método `descripcion()` que proporcione una descripción básica del vehículo. Las subclases pueden heredar esta funcionalidad o sobrescribirla para proporcionar más detalles.

```python
class Vehiculo(ABC):
    # (...)

    def descripcion(self) -> str:
        return f"Vehículo marca {self.marca}, modelo {self.modelo}"

class Coche(Vehiculo):
    # (...)

    def descripcion(self) -> str:
```

```
        return super().descripcion() + f", con {self.get_numero_puertas()} puertas."

class Moto(Vehiculo):
    # (...)

    def descripcion(self) -> str:
        return super().descripcion() + f", tipo {self.get_tipo()}."
```

En este caso, hemos definido el método `descripcion()` con una implementación por defecto que proporciona una descripción básica del vehículo. Las subclases, como `Coche` o `Moto`, pueden heredar este método directamente, extenderlo o sobrescribirlo.

6.4 Ejemplo de clases abstractas: clases Persona y Libro

Esta sección continúa el proyecto de las sesiones de prácticas anteriores para ilustrar el uso de clases abstractas. En concreto, asumiremos que en nuestro proyecto concreto no tiene sentido crear objetos genéricos de la clase `Libro` y, por tanto, instanciaremos objetos de tipos de libros específicos (libros científicos, libros infantiles, etc). Además, el segundo motivo que justifica el definir esta clase como abstracta es la necesidad de definir métodos cuya funcionalidad sea determinada por cada tipo de libro de forma independiente. Veamos el código modificado de la clase `Libro`:

```
from abc import ABC, abstractmethod

class Libro(ABC):
    def __init__(self, titulo: str, autor: str, paginas: int, propietario: 'Persona' = None):
        self._titulo: str = titulo
        self._autor: str = autor
        self._paginas: int = paginas
        self._listado_lectores: list['Persona'] = []  # Lista de personas que han leído el libro
        self._propietario: 'Persona' = propietario  # Relación de agregación: un libro tiene un propietario

    # (...) Obviamos métodos get/set

    @abstractmethod
    def comprar_libro(self, propietario: 'Persona') -> bool:
        pass

    @abstractmethod
    def recomendar_libro(self, persona: 'Persona') -> bool:
        pass

    # (...) Método mágico __str__
```

Tras indicar que `Libro` hereda de `ABC`, es necesario identificar qué métodos van a ser abstractos. En este ejemplo, consideremos que el proceso de compra de un libro depende completamente de cada tipo de libro. Así, en el caso de `LibroCientifico`, había unos requisitos específicos para poder realizar su compra, en base al número de libros científicos leídos previamente. En el caso de `LibroInfantil`, simplemente consistía en cambiar el propietario del libro.

Además, definimos un método abstracto adicional denominado `recomendar_libro`, encargado de determinar si un tipo de libro es apto o no para recomendación a una persona concreta, en base a su perfil particular. De igual forma, consideramos que estas recomendaciones deben ser realizadas de forma específica por cada tipo de libro. En el caso del método **__str__**, mantenemos el mismo comportamiento de la sesión anterior: cada tipo de libro usa el método mágico de la clase padre con `super()`, extendiendo su funcionalidad para mostrar información específica de ese tipo de libro.

Finalmente, es importante destacar que, en este ejemplo, podríamos perfectamente no haber definido la clase `Libro` como abstracta, permitiendo que hubiese libros genéricos sin una categoría clara. En muchos casos, la decisión de si una clase debe ser o no abstracta vendrá motivada por el problema a resolver.

Mostramos a continuación los cambios realizados en la clase `LibroCientifico`:

```
class LibroCientifico(Libro):
    def __init__(self, titulo: str, autor: str, paginas: int, campo_estudio: str,
                 nivel_dificultad: str, propietario: 'Persona' = None):
        super().__init__(titulo, autor, paginas, propietario)

        # Nuevos atributos de la clase hija
        self._campo_estudio: str = campo_estudio
        self._nivel_dificultad: str = nivel_dificultad
        self._referencias: list[Libro] = []

    # (...) Los métodos get/set y __str__ permanecen igual

    # Implementación del método abstracto
    def comprar_libro(self, propietario: 'Persona') -> bool:
        # Verificar si la persona ha leído libros científicos con anterioridad.
        # Uso de isinstance() para verificar el tipo de libro
        libros_cientificos_leidos = []
        for libro in propietario.get_libros_leidos():
            if isinstance(libro, LibroCientifico):
```

```
                libros_cientificos_leidos.append(libro)

        if len(libros_cientificos_leidos) >> 2: # Tiene experiencia previa leyendo libros científicos
            self._set_propietario(propietario)
            return True
        else:
            return False

    # Implementación del método abstracto
    def recomendar_libro(self, persona: 'Persona') -> bool:
        if self.get_campo_estudio() in persona.get_hobbies():
            return True
        return False
```

Como podemos ver, no se realizan cambios en el constructor respecto a la sesión anterior. Usaremos la clase padre abstracta como plantilla para definir atributos, haciendo uso de `super()`. En cuanto al método `comprar_libro()`, el código no requiere cambios respecto al que ya teníamos usando herencia. Es importante destacar que la clase `Libro` contiene métodos que sirven de soporte a los métodos abstractos implementados por las subclases, como el método `_set_propiterio()`. En el caso del método `recomendar_libro()`, consideramos que solo tiene sentido recomendar un libro científico siempre que el campo de estudio del libro se encuentre entre los hobbies de la persona.

Pasemos a ver el código modificado de la clase `LibroInfantil`:

```
class LibroInfantil(Libro):

    def __init__(self, titulo: str, autor: str, paginas: int, edad_recomendada: int, ilustraciones: bool,
                 interactividad: bool, propietario: 'Persona' = None):
        super().__init__(titulo, autor, paginas, propietario) # Llamada al constructor de la clase padre

        # Nuevos atributos de la clase hija
        self._edad_recomendada = edad_recomendada # Edad recomendada de lectura
        self._ilustraciones = ilustraciones # El libro tiene ilustraciones
        self._interactividad = interactividad # El libro es interactivo

    # (...) Los métodos get/set y __str__ permanecen igual

    # Implementación del método abstracto
    def comprar_libro(self, propietario: 'Persona') -> bool:
        self._set_propietario(propietario)
        return True

    # Implementación del método abstracto
    def recomendar_libro(self, persona: 'Persona') -> bool:
        if persona.get_edad() < self.get_edad_recomendada():
            return False
        return True
```

En este caso, la clase debe dar implementación a los dos métodos abstractos heredados. En el caso de `comprar_libro()`, simplemente implica asignar al libro un propietario. Sin embargo, si en el futuro la funcionalidad de este método debiese cambiar, podría adaptarse a nuevos requisitos sin problemas. En el caso de la recomendación de libros, consideramos que se debe recomendar siempre que la edad de la persona sea mayor o igual a la edad recomendada del libro.

Finalmente, si aplicamos estos cambios veremos que el `main` nos da un error, indicándonos que el diario que tiene `Persona` como atributo no puede ser de tipo `Libro`, ya que es abstracta. Para ello, definimos un nuevo tipo de libro: LibroDiario:

```
from datetime import datetime

class LibroDiario(Libro):
    def __init__(self, titulo: str, autor: str, paginas: int, propietario: 'Persona' = None):
        super().__init__(titulo, autor, paginas, propietario)
        self.fecha_creacion = datetime.now()
        self.completado: bool = False
        self.numero_entradas: int = 0

    # (...) Métodos get/set obviados por simplicidad

    def __str__(self) -> str:
        descripcion_basica = super().__str__()
        detalles_diario = (f"Fecha de creación: {self.fecha_creacion}, "
                           f"Completado: {'Sí' if self.completado else 'No'}, "
                           f"Número de entradas: {self.numero_entradas}")
        return f"{descripcion_basica} | {detalles_diario}"

    # Implementación del método abstracto
    def comprar_libro(self, propietario: 'Persona') -> bool:
        return True

    # Implementación del método abstracto
    def recomendar_libro(self, persona: 'Persona') -> bool:
        return True
```

Como podemos ver en el código, definimos algunos atributos adicionales a los definidos en la clase `Libro`, como son su fecha de creación, si se ha completado o no, y el número de entradas que contiene. Estas propiedades son ejemplos y seguramente se nos ocurren otras tantas diferentes para extender la clase. Sin embargo, es importante fijarse en cómo hemos implementado los dos métodos abstractos. En este caso de ejemplo, hemos determinado que ambos métodos devuelvan `True`. Esto es, que siempre recomendaremos comprar y recomendar un diario. Así, simplemente basta con modificar la clase `Persona` para que su atributo diario pase a ser de este nuevo tipo de libro, sin tener que cambiar los parámetros con los que se llamaba al constructor.

6.5 Relación de ejercicios

Esta sesión de prácticas parte del código implementado en el último ejercicio de la Sesión 5. En esta versión deberíamos tener una jerarquía de clases similar a la mostrada en la Figura 6.1, así como atributos y métodos similares.

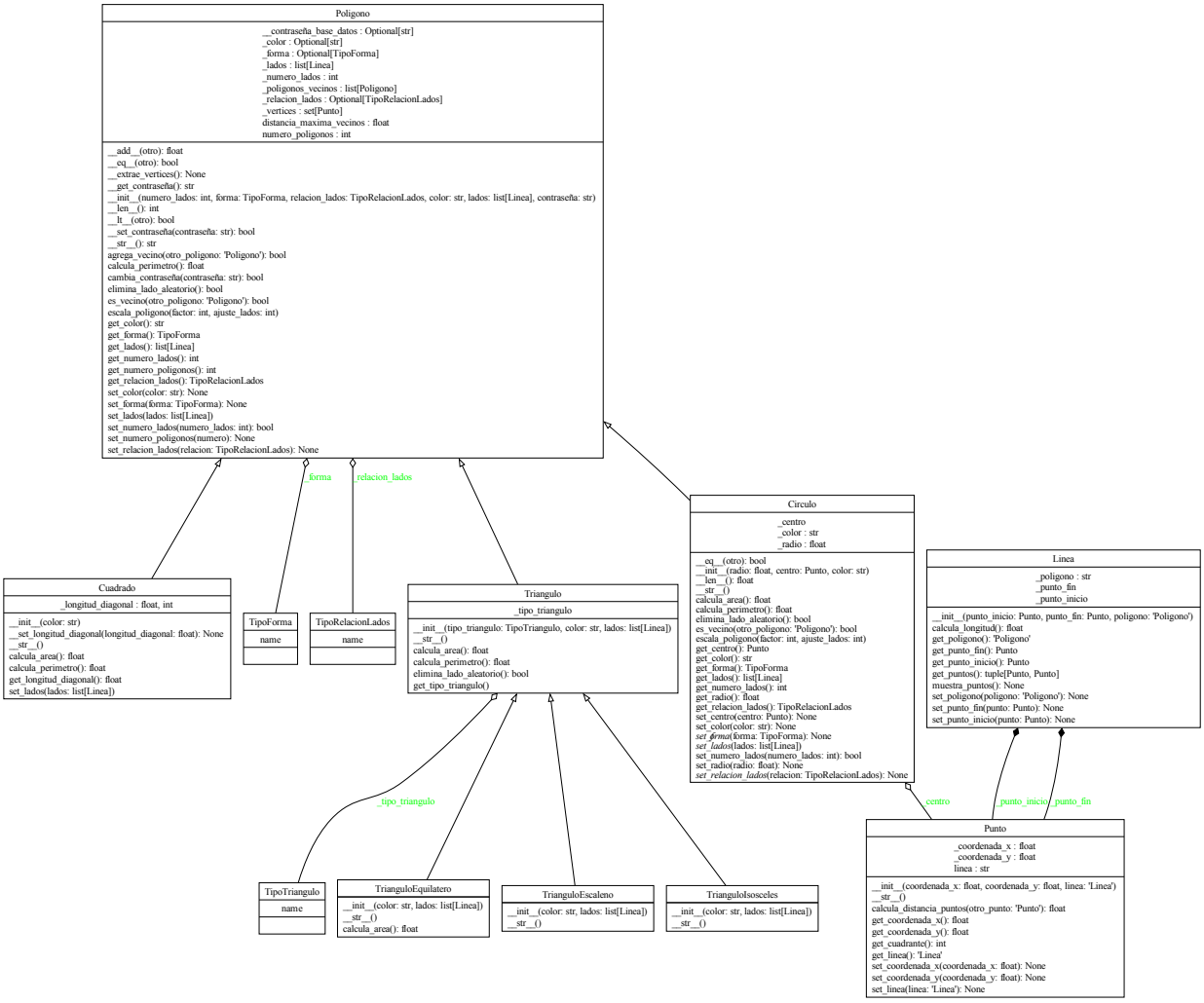


Figura 6.1: Diagrama UML representando el estado del proyecto tras la sesión 5.

Como Círculo no tiene sentido que sea un polígono (lo definimos como un polígono con fines didácticos en la sesión anterior), eliminaremos el Círculo como clase hija. Así, modificaremos el método *es_vecino()* para que únicamente compruebe la distancia entre centros con otro círculo. Todos los métodos innecesarios que se heredaron de Polígono tendremos que borrarlos, para dejar un código limpio.

1. **Clase abstracta Poligono**. A partir de ahora asumimos que no es necesario crear objetos de tipo `Poligono`, pues tenemos cuadrados, triángulos y círculos con los que trabajaremos. Para ello, completa los siguientes apartados:

    - Convierte la clase `Poligono` a clase abstracta.

- Convierte el método `calcula_area()` a método abstracto, de tal forma que cada tipo particular de polígono (triángulos y cuadrados) implementen su propia versión personalizada.
- Haz las modificaciones oportunas en la función `main` para probar que los objetos creados de las subclases funcionan correctamente, y verifica que no podemos crear objetos de la clase `Poligono`.
- ¿Sería conveniente convertir el método `calcula_perimetro()` a método abstracto?

2. **Clase abstracta Triangulo**. De igual forma a lo que pasaba anteriormente con `Poligono`, consideramos que no es conveniente en nuestro proyecto usar triángulos genéricos, pues los tres tipos de triángulos actuales (equiláteros, isósceles y escalenos) representan la totalidad de triángulos posibles.
   - Convierte la clase `Triangulo` a abstracta.
   - Implementa el método abstracto `calcula_altura()`, que es dependiente del tipo de triángulo.
     - Triángulo equilátero: $h = \frac{\sqrt{3}}{2} \times L$, donde L es la longitud de uno de los lados.
     - Triángulo isósceles: $h = \sqrt{a^2 - \left(\frac{b}{2}\right)^2}$, Donde a es la longitud de uno de los lados iguales y b es la longitud de la base.
     - Triángulo escaleno: $h = \frac{2 \times A}{b}$, donde A es el área calculada en base a la fórmula de Herón.
   - Haz los cambios pertinentes en el `main`.
3. **Clase FiguraCurva**. Este ejercicio define una nueva clase abstracta llamada `FiguraCurva`, que servirá como clase padre para las subclases `Circulo` (ya implementada en la sesión anterior) y `Elipse`, ambas figuras geométricas curvas.
   - Implementa la clase `FiguraCurva`, sabiendo que toda figura curva cuenta con centro de tipo `Punto` y un color. Define la visibilidad y los métodos get/set adecuados. Además, toda figura curva permite calcular su área y su perímetro. ¿El cálculo del área y del perímetro deben ser métodos abstractos?
   - Adapta la clase `Circulo` para que sea una subclase de FiguraCurva.
   - Implementa la clase `Elipse`, sabiendo que una elipse tiene dos atributos principales: el semieje mayor y el semieje menor. Los semiejes son segmentos que describen las dimensiones principales de la figura, dividiéndolas en dos mitades iguales. Por ejemplo, el semieje mayor es el segmento más largo de la elipse que atraviese su centro y conecta dos puntos opuestos en el borde de la elipse. Representaremos los semiejes como valores reales de distancia.

UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 7 de Prácticas

Interfaces

Autores: Alejandro Buitrago López (alejandro.buitragol@um.es), Sergio López Bernal (slopez@um.es), Javier Pastor Galindo (javierpg@um.es), Jesús Damián Jiménez Re (jesusjr@um.es) y Gregorio Martínez Pérez (gregorio@um.es)

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2024-2025

Última modificación:
22 de octubre de 2024

Sesión 7: Interfaces

Índice

7.1 Interfaces formales

En POO, las interfaces formales son un mecanismo para definir contratos que las clases deben seguir. Una interfaz establece un conjunto de métodos que cualquier clase que implemente la interfaz debe definir. Esto asegura que las clases que implementan una interfaz particular puedan ser usadas de manera intercambiable, independientemente de su implementación interna.

7.1.1. Herencia desde `ABC`

En lenguajes como Python, las interfaces formales se pueden implementar utilizando la clase base abstracta `ABC` (Abstract Base Class), disponible en el módulo `abc`. La clase `ABC` permite definir métodos abstractos, los cuales deben ser implementados por las clases derivadas. En este contexto, una interfaz se define como una clase que hereda de `ABC` y contiene únicamente métodos abstractos.

Recuerda que un método abstracto es aquel que se declara pero no se implementa en la interfaz. Las clases que heredan de una interfaz deben proporcionar una implementación concreta de todos los métodos abstractos para poder ser instanciadas.

```
from abc import ABC, abstractmethod

class MiInterfaz(ABC):

    @abstractmethod
    def metodo1(self):
        pass

    @abstractmethod
    def metodo2(self):
        pass
```

7.1.2. Ventajas de las interfaces formales

Las interfaces formales ofrecen varias ventajas en el diseño de software:

- **Polimorfismo**: Las interfaces permiten que diferentes clases implementen el mismo conjunto de métodos, permitiendo a los desarrolladores trabajar con objetos de diferentes tipos de manera uniforme.
- **Desacoplamiento**: Al trabajar con interfaces en lugar de implementaciones concretas, los sistemas son más modulares y fáciles de modificar o extender.
- **Claridad**: Las interfaces actúan como contratos que especifican qué operaciones deben soportar las clases, lo que facilita la comprensión del diseño del sistema.

7.1.3. Implementación de la interfaz

Las clases que implementan una interfaz formal deben proporcionar una implementación concreta de todos los métodos abstractos. Si una clase no lo hace, seguirá siendo abstracta y no podrá ser instanciada.

```
class Implementacion(MiInterfaz):

    def metodo1(self):
        print("Implementación del método 1")

    def metodo2(self):
        print("Implementación del método 2")
```

En este ejemplo, la clase `Implementacion` proporciona una implementación de los dos métodos abstractos definidos en `MiInterfaz`. Si no se implementaran todos los métodos abstractos, la clase no podría instanciarse y seguiría siendo una clase abstracta.

7.2 Relación de ejercicios

Esta sesión de prácticas parte del código implementado en el último ejercicio de la Sesión 6. En esta versión deberíamos tener una jerarquía de clases similar a la mostrada en la Figura 7.1, así como atributos y métodos similares.

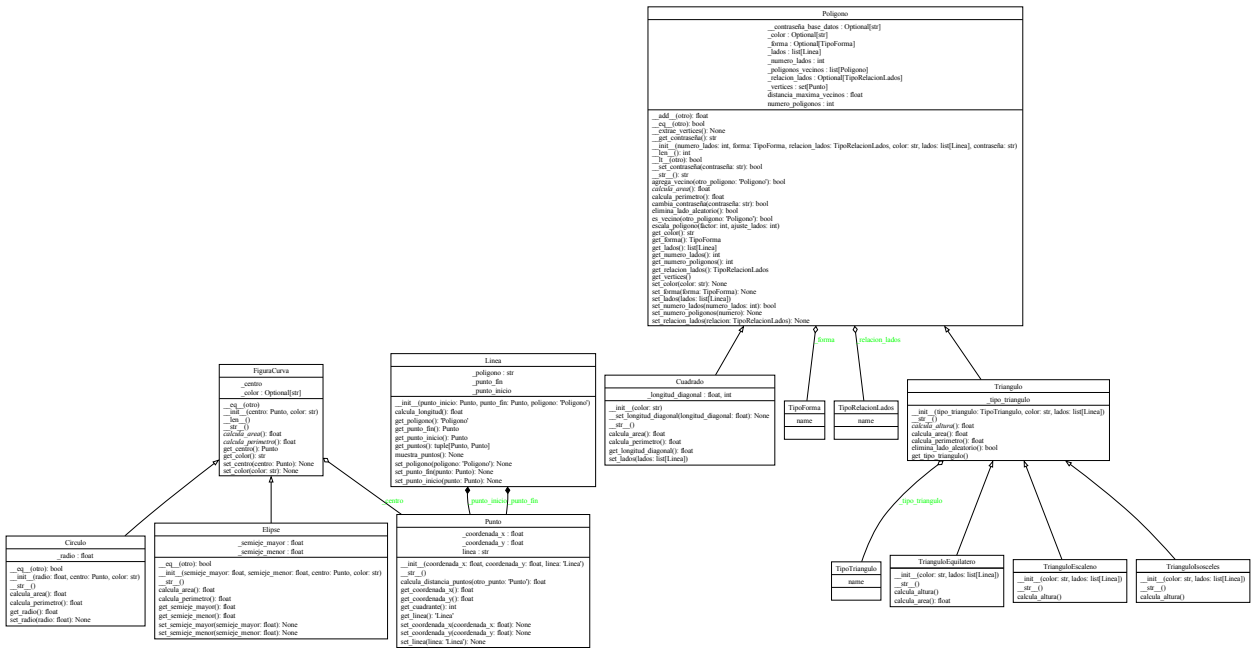


Figura 7.1: Diagrama UML representando el estado del proyecto tras la sesión 6.

1. **Gestión de entrada/salida**. Para simular la gestión de imágenes e información de las clases a través de ficheros, realizaremos lo siguiente:
   - **Clase GestionPersistencia**: Definimos los métodos que simulan la gestión de E/S:
     - Método que guarda la información básica (atributos con tipo de dato elemental) de un objeto dado, en formato diccionario, para guardarlo en un archivo JSON de una ruta dada.
       Provisionalmente, puesto que no hemos explicado E/S, se imprimirá "Guardando el JSON {dic} en {ruta}."
     - Método para cargar la información básica (atributos con tipo de dato elemental) desde un archivo JSON en una ruta dada y actualizar el objeto dado.
       Provisionalmente, como no sabemos leer un archivo, se actualizará el objeto exactamente con el mismo estado que tenga, imprimiendo "Cargando el JSON {dic} desde {ruta}."
     - Método para guardar la información básica (atributos con tipo de dato elemental) de un objeto dado en un archivo tabular CSV con ruta dada. El objeto debe poder transformarse en un diccionario cuyas claves representen las columnas del CSV.
       Se simulará imprimiendo "Guardando el CSV {dic} en {ruta}."
     - Método para cargar la información básica (atributos con tipo de dato elemental) desde un archivo CSV en una ruta dada y actualizar el objeto dado. El objeto debe poder actualizarse a partir de los datos leídos desde el archivo tabular, donde en cada columna tenemos un atributo.
       Para este ejercicio, en lugar de leer realmente un archivo, se actualizará el objeto con su estado actual, imprimiendo "Leyendo el CSV {dic} desde {ruta}."
     - Método para guardar la imagen de un objeto dado en la ruta especificada. Se precisa acceder a la imagen del objeto (de tipo `Image`, definido el módulo `PIL`) para que se pueda persistir.
       Actualmente podemos imprimir "Guardando imagen en {ruta}."
     - Método para cargar una imagen desde la ruta especificada y asignársela a un objeto dado. La imagen es de tipo `Image`.
       Provisionalmente no leemos desde fichero, sino que recuperamos la imagen del objeto y la volvemos a establecer, imprimiendo "Leyendo la imagen desde {ruta}."
   - **Interfaz IPersistente**: ¿Qué funcionalidades precisamos de aquellos objetos que queremos persistir a través

de la clase anterior? Define la interfaz `IPersistente` con todos aquellos métodos que necesitamos garantizar en aquellas clases que quieran ser persistentes para gestionar CSV, JSON e imágenes.

- Asegúrate que la clase `Polígono` y `FiguraCurva` implementan la interfaz `IPersistente`.
- Instancia la clase `GestionPersistencia` en el módulo principal `main.py` para simular la gestión de entrada/salida de un objeto de tipo `TrianguloEscaleno`. En particular, haciendo uso de los métodos ofertados por la clase `GestionPersistencia` para guardar y cargar CSV, JSON e imágenes.

UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 8 de Prácticas

Excepciones y Entrada/Salida

Autores: Alejandro Buitrago López (alejandro.buitragol@um.es), Sergio López Bernal (slopez@um.es), Javier Pastor Galindo (javierpg@um.es), Jesús Damián Jiménez Re (jesusjr@um.es) y Gregorio Martínez Pérez (gregorio@um.es)

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2024-2025

Última modificación:
1 de noviembre de 2024

Sesión 8: Excepciones y Entrada/Salida

Índice

8.1 Excepciones en Python

Python proporciona un mecanismo robusto para manejar errores en tiempo de ejecución mediante el uso de excepciones. Estas permiten capturar y gestionar situaciones imprevistas en un programa de forma controlada. En las siguientes secciones veremos cómo podemos capturar excepciones, lanzarlas en caso de saber que se producirá un error e, incluso, definir nuestras propias excepciones personalizadas.

8.1.1. Excepciones comunes en Python

Esta sección presenta las excepciones más comunes en Python, divididas por categorías. Es importante destacar que algunas de ellas las gestiona y lanza el propio intérprete de Python, por lo que nosotros como programadores no las gestionaremos en nuestros programas. Piensa, por ejemplo, en un error de sintaxis o nombres de variables, es un error que nos lanza el intérprete de Python al ejecutar nuestro código con errores. Sin embargo, sí nos viene bien conocer qué significan para poder corregir rápido los errores de nuestro código.

1. **Errores de sintaxis y nombres**
   - `SyntaxError`: Se produce cuando hay un error en la sintaxis del código. Por ejemplo, si no ponemos los dos puntos tras la condición de un *if*, si se nos olvida poner el nombre de una función al definirla, o si no cerramos las comillas al escribir un string.

```
# Ejemplo de SyntaxError
if 10 > 5
    print("Esto generará un SyntaxError.")

# Salida esperada en la consola:
File "example.py", line 2
    if 10 > 5
             ^
SyntaxError: invalid syntax
```

- `NameError`: Se lanza cuando se intenta utilizar una variable o función que no ha sido definida.

```
# Ejemplo de NameError
print(valor)

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    print(valor)
NameError: name 'valor' is not defined
```

- `IndentationError`: Ocurre cuando la indentación (tabulación) del código no es correcta.

```
# Ejemplo de IndentationError
def funcion():
print("Esto generará un IndentationError")

# Salida esperada en la consola:
File "example.py", line 3
    print("Esto generará un IndentationError")
    ^
IndentationError: expected an indented block
```

- `UnboundLocalError`: Se lanza cuando una variable local es referenciada antes de haber sido asignada.

```
# Ejemplo de UnboundLocalError
def funcion():
    print(variable)
    variable = 5

funcion()

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in funcion
    print(variable)
UnboundLocalError: local variable 'variable' referenced before assignment
```

2. **Errores relacionados con los tipos de datos**
   - `TypeError`: Ocurre cuando se intenta realizar una operación con un tipo de dato incorrecto.

```
# Ejemplo de TypeError
resultado = "texto" + 5

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    resultado = "texto" + 5
TypeError: can only concatenate str (not "int") to str
```

   - `ValueError`: Se lanza cuando una operación recibe un tipo correcto, pero el valor no es apropiado.

```
# Ejemplo de ValueError
numero = int("texto")

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    numero = int("texto")
ValueError: invalid literal for int() with base 10: 'texto'
```

3. **Errores relacionados con el manejo de archivos y E/S**
   - `FileNotFoundError`: Se lanza cuando se intenta abrir un archivo que no existe.

```
# Ejemplo de FileNotFoundError
with open('archivo_inexistente.txt', 'r') as f:
    contenido = f.read()

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    with open('archivo_inexistente.txt', 'r') as f:
FileNotFoundError: [Errno 2] No such file or directory: 'archivo_inexistente.txt'
```

   - `IOError`: Se produce en operaciones de entrada/salida fallidas. Por ejemplo, si no se tiene permiso para acceder a un fichero.

```
# Ejemplo de IOError
with open('/ruta_no_permitida/archivo.txt', 'r') as f:
    contenido = f.read()
```

```
# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    with open('/ruta_no_permitida/archivo.txt', 'r') as f:
IOError: [Errno 13] Permission denied: '/ruta_no_permitida/archivo.txt'
```

- `EOFError`: Ocurre cuando se alcanza el final de un archivo sin obtener los datos esperados.

```
# Ejemplo de EOFError
while True:
    entrada = input("Introduce algo: ")

# Salida esperada en la consola (presionando Ctrl+D para finalizar):
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    entrada = input("Introduce algo: ")
EOFError: EOF when reading a line
```

- `IsADirectoryError`: Se lanza cuando se intenta realizar una operación de archivo en un directorio.

```
# Ejemplo de IsADirectoryError
with open('/ruta_a_un_directorio/', 'r') as f:
    contenido = f.read()

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    with open('/ruta_a_un_directorio/', 'r') as f:
IsADirectoryError: [Errno 21] Is a directory: '/ruta_a_un_directorio/'
```

4. **Errores de números y operaciones matemáticas**

- `ZeroDivisionError`: Se lanza cuando se intenta dividir un número por cero.

```
# Ejemplo de ZeroDivisionError
resultado = 10 / 0

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    resultado = 10 / 0
ZeroDivisionError: division by zero
```

- `OverflowError`: Ocurre cuando el resultado de una operación aritmética es demasiado grande.

```
# Ejemplo de OverflowError
import math
resultado = math.exp(1000)

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    resultado = math.exp(1000)
OverflowError: math range error
```

- `FloatingPointError`: Se lanza cuando ocurre un error en una operación de coma flotante (números reales).

```
# Ejemplo de FloatingPointError
resultado = 1e308 * 1e308  # Número demasiado grande

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 6, in <module>
    resultado = 1e308 * 1e308
FloatingPointError: (34, 'Result too large')
```

5. **Errores relacionados con índices y claves**

- `IndexError`: Se produce cuando se intenta acceder a un índice fuera de rango en una secuencia.

```
# Ejemplo de IndexError
lista = [1, 2, 3]
elemento = lista[5]

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 3, in <module>
    elemento = lista[5]
IndexError: list index out of range
```

- `KeyError`: Se lanza cuando se intenta acceder a una clave inexistente en un diccionario.

```
# Ejemplo de KeyError
diccionario = {"nombre": "Juan", "edad": 30}
valor = diccionario["direccion"]

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    valor = diccionario["direccion"]
KeyError: 'direccion'
```

6. **Errores de iteración y objetos**
   - `StopIteration`: Ocurre cuando se termina de iterar sobre un iterador y no hay más elementos.

```
# Ejemplo de StopIteration
lista = iter([1, 2, 3])
while True:
    print(next(lista))

# Salida esperada en la consola:
1
2
3
Traceback (most recent call last):
  File "example.py", line 4, in <module>
    print(next(lista))
StopIteration
```

   - `AttributeError`: Se lanza cuando se intenta acceder a un atributo que no existe en un objeto.

```
# Ejemplo de AttributeError
lista = [1, 2, 3]
lista.longitud()

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 3, in <module>
    lista.longitud()
AttributeError: 'list' object has no attribute 'longitud'
```

   - `NotImplementedError`: Se utiliza para indicar que un método o función aún no ha sido implementado.

```
# Ejemplo de NotImplementedError
class Base:
    def metodo(self):
        raise NotImplementedError("Este método no está implementado")

objeto = Base()
objeto.metodo()

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 6, in <module>
    objeto.metodo()
NotImplementedError: Este método no está implementado
```

   - `RuntimeError`: Un error genérico que ocurre durante la ejecución de un programa.

```
# Ejemplo de RuntimeError (agregar al mismo conjunto mientras se itera implícitamente)
conjunto = {1, 2, 3}
conjunto.add(frozenset(conjunto))

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 3, in <module>
    conjunto.add(frozenset(conjunto))
RuntimeError: Set changed size during iteration
```

7. **Errores relacionados con la memoria**
   - `MemoryError`: Ocurre cuando no hay suficiente memoria disponible para realizar una operación.

```
# Ejemplo de MemoryError
# Intentar crear una lista muy grande
lista = [1] * (10**10)

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    lista = [1] * (10**10)
MemoryError
```

- `RecursionError`: Se lanza cuando se excede el límite máximo de recursividad.

```
# Ejemplo de RecursionError
def recursividad():
    return recursividad()

recursividad()

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 4, in <module>
    recursividad()
  File "example.py", line 2, in recursividad
    return recursividad()
  [Previous line repeated 995 more times]
RecursionError: maximum recursion depth exceeded
```

8. **Errores relacionados con la importación de módulos**
   - `ImportError`: Se produce cuando no se puede importar un módulo.

```
# Ejemplo de ImportError
from math import no_existe

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    from math import no_existe
ImportError: cannot import name 'no_existe' from 'math'
```

   - `ModuleNotFoundError`: Subclase de `ImportError`, se lanza cuando no se encuentra el módulo.

```
# Ejemplo de ModuleNotFoundError
import modulo_que_no_existe

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    import modulo_que_no_existe
ModuleNotFoundError: No module named 'modulo_que_no_existe'
```

9. **Excepciones del sistema**
   - `KeyboardInterrupt`: Se lanza cuando el programa es interrumpido manualmente por el usuario.

```
# Ejemplo de KeyboardInterrupt
while True:
    pass

# Salida esperada en la consola (al presionar Ctrl+C):
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    pass
KeyboardInterrupt
```

   - `SystemExit`: Se utiliza para solicitar la salida del programa.

```
# Ejemplo de SystemExit
import sys
sys.exit("Saliendo del programa")

# Salida esperada en la consola:
Saliendo del programa
Traceback (most recent call last):
  File "example.py", line 3, in <module>
    sys.exit("Saliendo del programa")
SystemExit: Saliendo del programa
```

10. **Otras excepciones comunes**
    - `AssertionError`: Ocurre cuando una afirmación `assert` falla.

```
# Ejemplo de AssertionError
assert 1 == 2, "1 no es igual a 2"

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    assert 1 == 2, "1 no es igual a 2"
AssertionError: 1 no es igual a 2
```

   - `PermissionError`: Se lanza cuando una operación no tiene los permisos necesarios.

```
# Ejemplo de PermissionError
with open('/archivo_protegido.txt', 'w') as f:
    f.write("Prueba de escritura")

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    with open('/archivo_protegido.txt', 'w') as f:
PermissionError: [Errno 13] Permission denied: '/archivo_protegido.txt'
```

- `TimeoutError`: Ocurre cuando una operación expira por exceder el tiempo límite establecido.

```
# Ejemplo de TimeoutError

'''
(...) Código para solicitar el contenido de una página web:
    - Crear la conexión
    - Establecer un timeout (tiempo máximo) para establecer la conexión
    - Establecer la conexión.
        Ejemplo solicitud a dirección IP y puerto concretos: s.connect(('10.255.255.1', 80))
'''

# Salida esperada en la consola:
Traceback (most recent call last):
  File "example.py", line 8, in <module>
    s.connect(('10.255.255.1', 80))
socket.timeout: timed out
```

8.1.2. Capturar excepciones

Para capturar errores en Python se utiliza la sentencia `try-except`, que permite ejecutar un bloque de código e interceptar posibles errores en dicho código. Esto permite gestionar dicho error en lugar de que el programa se interrumpa directamente. La sintaxis básica en el control de excepciones se muestra a continuación:

```
try:
    # Bloque de código que puede generar un error
    resultado = 10 / 0
except ZeroDivisionError:
    # Se captura el error (excepción) específico: ZeroDivisionError
    print("Error: No se puede dividir entre cero.")
```

También se pueden usar bloques adicionales como `else` y `finally` para controlar el flujo del programa:

- **`else`**: Se ejecuta solo si no se genera ninguna excepción en el bloque `try`.
- **`finally`**: Se ejecuta siempre, ocurra o no una excepción. Es útil para liberar recursos, como cerrar archivos.

```
try:
    resultado = 10 / 2
except ZeroDivisionError:
    print("Error: No se puede dividir entre cero.")
else:
    print("La operación fue exitosa.")
finally:
    print("Este bloque se ejecuta siempre.")
```

Como podemos ver en el código anterior, dividir 10 entre 2 no generará una excepción de tipo `ZeroDivisionError`, por lo que no entraremos en el `except`. En su lugar, entraremos en el `else` y, tras ello, en el `finally`. Es importante destacar que estos dos bloques son opcionales y, por tanto, dependerá del programa concreto el si definir uno, el otro o los dos.

Manejo de múltiples excepciones: Es posible capturar diferentes tipos de excepciones tras un único bloque `try`, lo que permite tratar cada error de forma específica. Esto es útil cuando conocemos de antemano que una o varias instrucciones generarán unos errores determinados. En el siguiente ejemplo podemos ver que, tras leer un número introducido por el usuario, el valor podría generar dos situaciones de error. Estos errores pueden ser previstos por el programador, como son el recibir un valor no numérico, o recibir un cero.

```
try:
    valor = int(input("Introduce un número: "))
    resultado = 10 / valor
except ValueError:
    print("Error: El valor introducido no es un número.")
except ZeroDivisionError:
    print("Error: No se puede dividir entre cero.")
```

Finalmente, hay situaciones en las que podría generarse una excepción que no tenemos prevista. Una posible opción sería capturar `Exception`, que es el tipo de excepción padre de todas las demás. Podemos ver un ejemplo a continuación:

```
try:
    print("Ejecución de un bloque de código con errores.")
except Exception:
    print("Error: Tratamiento de cualquier tipo de error.")
```

Aunque usar Exception puede resultar muy tentador (gestiono todas las excepciones de forma global sin preocuparme de los tipos de excepciones concretas), por norma general no suele ser muy recomendable, por varios motivos:

- **Poca especificidad**: Capturar todas las excepciones impide distinguir entre diferentes tipos de errores. Es posible que captures una excepción que debería haber sido gestionada de manera diferente, lo que puede ocultar errores graves.
- **Dificultad en la depuración**: Al capturar todas las excepciones, puede ser más difícil identificar la causa raíz del problema en un programa, ya que se está generalizando el manejo de errores.
- **Peligro de ignorar excepciones importantes**: Algunas excepciones pueden ser críticas para el funcionamiento del sistema y pueden ser capturadas inadvertidamente si no se tiene cuidado.

8.1.3. Lanzar excepciones

Además de capturar excepciones, Python permite lanzar manualmente excepciones con las sentencias `raise` y `assert`. Esto es útil para realizar validaciones dentro del código y asegurar que se cumplan ciertas condiciones.

```
def dividir(a: int, b: int) -> float:
    if b == 0:
        raise ZeroDivisionError("El divisor no puede ser cero.")
    return a / b

try:
    dividir(10, 0)
except ZeroDivisionError as e:
    print(f"Excepción capturada: {e}")
```

El código anterior hace uso de `raise` para lanzar una excepción en caso de que el divisor proporcionado a la función sea cero. Esto es, podemos controlar manualmente aquellas situaciones que sabemos que van a generar un error para lanzar nosotros la excepción.

Por su parte, `assert` evalúa una expresión y lanza un `AssertionError` si la condición es falsa. Es útil para validar precondiciones en el código. Como podemos ver en el código siguiente, podremos tener funciones con condiciones que se deben cumplir obligatoriamente para que la función pueda trabajar. Por ejemplo, que los valores de los parámetros sean mayor que cero, que no sean None, que la longitud de una lista proporcionada sea mayor que cero, etc. Esto se denomina precondición con programación. Usaremos `assert` típicamente al inicio de las funciones para comprobar las precondiciones y, en caso de no cumplirse, lanzar la excepción.

```
def verificar_positivo(numero: int) -> None:
    assert numero > 0, "El número debe ser positivo"

verificar_positivo(-5)  # Lanza un AssertionError
```

8.1.4. Definir excepciones personalizadas

Python permite definir nuestras propias excepciones creando subclases de `Exception`. Esto es útil cuando se quiere capturar errores específicos de la lógica del programa.

```
# Clase que define la excepción
class ValorFueraDeRangoError(Exception):
    def __init__(self, valor: int, mensaje: str = "Valor fuera del rango permitido"):
        super().__init__(self._mensaje)
        self._valor = valor
        self._mensaje = mensaje

    def __str__(self):
        return f"{self.valor} -> {self.mensaje}"

# Método definido en una clase cualquiera
def verificar_rango(valor: int) -> None:
    if valor < 10 or valor > 100:
        raise ValorFueraDeRangoError(valor)

# Fragmento de código en una clase y método cualquiera
try:
    verificar_rango(150)
except ValorFueraDeRangoError as e:
    print(e) # Salida: "Valor fuera del rango permitido"
```

En el código anterior hemos definido la clase `ValorFueraDeRangoError` que hereda de `Exception`. En el constructor hace uso de *super()* para invocar al constructor de su clase padre, definiendo dos atributos propios, un valor y un mensaje personalizado. Así, imaginemos que tenemos un método que comprueba el valor de un parámetro recibido, de tal forma que dea garantizar que el valor esté comprendido entre 10 y 100. En caso de que se de esa situación de error, se lanzará nuestra excepción personalizada, que podremos gestionar con *try-except* de forma habitual como cualquier otra excepción proporcionada de base por Python.

8.2 Entrada y salida de datos en Python

El manejo de entrada y salida de datos en Python comprende la interacción con una gran variedad de sistemas externos a nuestro programa, tales como el sistema operativo, periféricos (como el teclado o el ratón), la consola, servidores, archivos, URLs, etc.

8.2.1. Manejo de ficheros

Python ofrece un conjunto de herramientas para leer, escribir y manipular ficheros de texto y binarios, además de trabajar con formatos específicos como JSON, CSV o pickle, entre otros muchos. En esta sección, exploraremos las funcionalidades clave para trabajar con ficheros y su contenido.

El manejo de ficheros en Python sigue un patrón común: primero se abre el archivo, se realiza la operación necesaria (lectura o escritura) y finalmente se cierra el archivo para liberar recursos. A continuación, se muestra el esquema general para abrir y cerrar un archivo en Python.

```
# Abrir un fichero
f = open('archivo.txt', 'r')

# Leer del fichero
contenido = f.read()

# Cerrar el fichero
f.close()
```

En este ejemplo, se abre un archivo llamado `archivo.txt` en modo lectura (`'r'`) y su contenido se almacena en la variable `contenido`, haciendo uso de la función `read()`. Finalmente, el archivo se cierra con `f.close()`.

Es importante destacar que los nombres de los archivos y los modos pueden indicarse con comillas dobles o simples, indistintamente. Además, en lugar de haber indicado `archivo.txt`, podríamos haber indicado una ruta de directorios hasta el archivo a abrir. Por ejemplo, asumamos que en el mismo directorio en el que está el archivo a ejecutar (supongamos *main.py*), tenemos una carpeta llamada ficheros y, dentro de ella, será donde se encuentre nuestro archivo de texto. Así, el código tendría que ser el siguiente:

```
# Abrir un fichero
f = open('directorio/archivo.txt', 'r')

# Leer del fichero
contenido = f.read()

# Cerrar el fichero
f.close()
```

Los ficheros pueden abrirse en diferentes modos:

- **r**: Lectura. El archivo debe existir.
- **w**: Escritura. Crea un archivo si no existe o lo sobreescribe si ya existe.
- **a**: Añadir. Escribe al final del archivo sin borrar su contenido). En caso de que no exista el fichero, lo crea.
- **b**: Binario. Se utiliza para leer o escribir archivos binarios, como imágenes.

A continuación, se muestra un ejemplo de cómo escribir en un archivo y luego leer su contenido:

```
# Escribir en un fichero
f = open('archivo.txt', 'w')
f.write('Hola, mundo\n')
f.write('Segunda línea\n')
f.close()

# Leer del fichero
f = open('archivo.txt', 'r')
contenido = f.read()
print(contenido)
f.close()
```

En este ejemplo, como usamos la opción de escritura (w), si el fichero ya existía previamente lo sobreescribe; si no, lo crea por primera vez. Tras ello, se escriben dos líneas al fichero (fijaos en el salto de línea al final de cada escritura), y se

cierra el fichero. Una vez hemos escrito, procedemos a abrir el fichero, leer su contenido, mostrarlo por pantalla y cerrarlo. Es esencial cerrar los ficheros tras terminar de usarlos, y es un error muy común olvidarnos de hacerlo.

8.2.1.1. Uso de `with` para el manejo de ficheros

Python proporciona una manera más segura y conveniente de manejar ficheros con la sentencia `with`. Esta se asegura de que el archivo se cierre automáticamente después de que se haya terminado de trabajar con él, incluso si ocurre una excepción. Esto evita incluso tener que utilizar `f.close()`. Veamos un ejemplo:

```
# Abrir y manejar el fichero con 'with'
with open('archivo.txt', 'r') as f:
    contenido = f.read()
    print(contenido)
# No es necesario cerrar el fichero, 'with' lo gestiona automáticamente.
```

Es importante destacar que todo el código que incluyamos dentro del bloque `with` en este ejemplo tendrá el fichero abierto y disponible para su uso. Tras este bloque, el fichero no estará abierto y no podremos acceder a su contenido para leer o escribir, a no ser que volvamos a acceder a él de forma explícita.

8.2.1.2. Trabajando con formatos específicos

Python también ofrece soporte para trabajar con formatos de datos comunes como JSON, CSV y pickle. A continuación, se explican cada uno de estos formatos y se muestran ejemplos de su manejo.

Ficheros JSON JSON (*JavaScript Object Notation*) es un formato ligero de intercambio de datos, muy utilizado en aplicaciones web. Su estructura se basa en pares clave-valor, similar a los diccionarios en Python, lo que lo hace fácil de interpretar y manipular en distintos lenguajes de programación. Un archivo JSON puede almacenar objetos anidados, listas y otros tipos de datos simples como cadenas, números, `true`, `false` y `null`. A continuación se muestra un ejemplo de un archivo JSON que contiene información de datos personales para una persona:

```
{
    "nombre": "Juan",
    "edad": 30,
    "ciudad": "Madrid",
    "hobbies": ["leer", "viajar", "cocinar"],
    "trabajo": {
        "empresa": "Tech Solutions",
        "puesto": "Desarrollador"
    }
}
```

Este archivo JSON representa un objeto que contiene varias claves como `nombre`, `edad`, `ciudad`, así como una lista de `hobbies` y un objeto anidado en `trabajo` que incluye los datos de la empresa y el puesto de trabajo. Como podemos ver, las claves y los valores están separadas por dos puntos, y cada bloque viene delimitado entre llaves. Podemos ver, además, que se separan los campos del objeto (asociación clave-valor) unos de otros usando una coma.

También es posible trabajar con un listado de objetos JSON dentro de un listado. Por ejemplo, podemos representar una lista de contactos, donde cada contacto es un objeto JSON con su respectiva información:

```
[
    {
        "nombre": "Ana",
        "telefono": "123456789",
        "email": "ana@example.com"
    },
    {
        "nombre": "Luis",
        "telefono": "987654321",
        "email": "luis@example.com"
    },
    {
        "nombre": "Pedro",
        "telefono": "555666777",
```

```
        "email": "pedro@example.com"
    }
]
```

En este ejemplo, cada objeto dentro de la lista representa un contacto con sus claves `nombre`, `telefono` y `email`. Este tipo de estructura es útil cuando se desea almacenar múltiples objetos con la misma estructura en un solo archivo JSON. De forma similar a los campos de un objeto, los objetos se separan unos de otros por una coma.

Python proporciona la librería `json` para trabajar con este formato. A continuación se muestra cómo leer y escribir en archivos JSON utilizando Python.

```python
import json

# Definir un diccionario
datos = {
    "nombre": "Juan",
    "edad": 30,
    "ciudad": "Madrid",
    "hobbies": ["leer", "viajar", "cocinar"],
    "trabajo": {
        "empresa": "Tech Solutions",
        "puesto": "Desarrollador"
    }
}

# Escribir el diccionario en un fichero JSON
with open('datos.json', 'w') as f:
    json.dump(datos, f, indent=4)

# Leer el fichero JSON y cargarlo como un diccionario
with open('datos.json', 'r') as f:
    datos_leidos = json.load(f)
    print(datos_leidos)
```

Como podemos ver, el formato JSON es idéntico al que definiríamos en Python para crear un diccionario. Una vez tenemos nuestro diccionario `datos`, lo guardamos en el archivo `datos.json`, abriéndolo en modo escritura. Para volcar el contenido del diccionario al archivo recién abierto, hacemos uso del método `dump()` de la librería, que recibe tres argumentos: el diccionario, el fichero que acabamos de abrir, y un valor que indica el número de espacios que usaremos para formatear el texto (una tabulación equivale a cuatro caracteres de espacio, por lo que siempre usaremos un valor de 4).

Si queremos leer el contenido escrito al fichero, bastará con abrir el fichero en modo lectura y hacer uso de la función `load()`. Es interesante destacar que no hemos especificado el tipo de datos de la variable **datos_leidos**. Esto se debe a que, en función del contenido del archivo JSON, internamente Python representará la variable de una u otra forma: si es un único objeto de JSON nos devolverá un diccionario, si es un listado de objetos nos creará una lista, etc. En este caso seremos prácticos y no especificaremos el tipo de datos de la variable. En lugar de hacer un `print()`, podría interesarnos acceder a uno de los campos concretos recuperados del JSON. Por ejemplo, al nombre de la persona. Para acceder a un campo concreto, haremos uso de `datos_leidos["nombre"]`. Siempre es recomendable comprobar que la clave existe en el diccionario antes de acceder al valor para evitar lanzar un `KeyError`, de la siguiente forma: `if “nombre” in datos_leidos`.

Se puede obtener más información sobre el uso de la librería en su documentación oficial: Documentación de JSON

Ficheros CSV El formato CSV (*Comma-Separated Values*) es ampliamente utilizado para almacenar y transferir datos tabulares, como hojas de cálculo. Un archivo CSV contiene datos separados por comas (o a veces por otros delimitadores como punto y coma), donde cada línea del archivo representa una fila de la tabla, y cada valor separado por comas corresponde a una columna.

Además, los archivos CSV son muy usados para análisis de datos e inteligencia artificial, donde librerías como `Pandas` permiten trabajar con `Dataframes`, estructuras de datos que facilitan la búsqueda y el procesamiento de información contenidas en un CSV. A continuación se muestra un ejemplo de un archivo CSV que almacena datos de contacto:

```
Nombre,Edad,Ciudad
Ana,25,Barcelona
Luis,34,Valencia
Pedro,28,Madrid
```

En este archivo, la primera fila representa los encabezados (o nombres) de las columnas (`Nombre`, `Edad`, `Ciudad`), y cada fila subsiguiente representa una persona, con los valores separados por comas. El siguiente archivo CSV también sería válido, usando como delimitador el punto y coma. En algunos casos suele ser preferible el punto y coma, pues en algunas codificaciones los números decimales usan coma en lugar de punto.

```
Nombre;Edad;Ciudad
Ana;25;Barcelona
Luis;34;Valencia
Pedro;28;Madrid
```

Python proporciona la librería `csv` para facilitar la lectura y escritura de archivos CSV. A continuación, se muestra cómo trabajar con este formato.

```python
import csv

# Escribir datos en un fichero CSV
with open('datos.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(['Nombre', 'Edad', 'Ciudad'])
    writer.writerow(['Ana', 25, 'Barcelona'])
    writer.writerow(['Luis', 34, 'Valencia'])
    writer.writerow(['Pedro', 28, 'Madrid'])

# Leer el fichero CSV
with open('datos.csv', 'r') as f:
    reader = csv.reader(f)
    for fila in reader:
        print(fila)
```

Primero, utilizamos la función `csv.writer()` para escribir filas en un archivo CSV. Cada fila se pasa como una lista, donde cada elemento corresponde a una columna. Para escribir en un archivo CSV, se utiliza `writerow()` para escribir una fila de datos. El parámetro `newline=''` en la apertura del archivo en modo escritura es importante, ya que previene la adición de líneas en blanco entre las filas en algunos sistemas operativos, especialmente en Windows. Luego, para leer el archivo CSV, utilizamos `csv.reader()`, que obtiene el contenido completo del archivo separado por líneas. Así, tendremos que hacer un for para acceder línea a línea al contenido del fichero.

Para trabajar con archivos CSV que contienen encabezados, Python ofrece una estructura útil utilizando `csv.DictReader()`, que convierte cada fila en un diccionario donde las claves son los encabezados de la primera fila:

```python
import csv

# Leer un fichero CSV con encabezados
with open('datos.csv', 'r') as f:
    reader = csv.DictReader(f)

    for fila in reader:
        # Acceder a cada columna por su nombre
        nombre = fila['Nombre']
        edad = fila['Edad']
        ciudad = fila['Ciudad']

        # Imprimir cada columna por separado
        print(f"Nombre: {nombre}, Edad: {edad}, Ciudad: {ciudad}")
```

En este caso, el resultado sería un diccionario por cada fila, con las claves siendo los nombres de las columnas (`Nombre`, `Edad`, `Ciudad`) y los valores los datos correspondientes de esa fila.

Para más información, consultad la documentación oficial de la librería CSV en Python: Documentación de CSV

Ficheros pickle El módulo `pickle` en Python se utiliza para serializar y deserializar objetos Python. Es decir, convertir objetos en un formato que pueda ser almacenado en un archivo (serializar) y luego recuperar esos objetos en su forma original (deserializar). La serialización con `pickle` convierte los objetos en una secuencia de bytes que puede ser escrita en archivos binarios. Esto es útil para guardar estructuras de datos complejas como listas, diccionarios o incluso instancias de clases. En el contexto de la inteligencia artificial, es muy común almacenar los modelos entrenados en este formato. Estos archivos tienen la extensión `.pkl`.

A diferencia de JSON, que se limita a tipos de datos básicos (como cadenas, listas, diccionarios, etc.), `pickle` es capaz de serializar objetos más complejos, como instancias de clases o estructuras de datos anidadas. Veamos un ejemplo:

```python
import pickle

# Definir un diccionario de ejemplo
datos = {'nombre': 'Ana', 'edad': 25, 'ciudad': 'Barcelona'}

# Serializar la lista en un fichero binario
with open('datos.pkl', 'wb') as f:
    pickle.dump(datos, f)

# Deserializar la lista desde el fichero binario
with open('datos.pkl', 'rb') as f:
    datos_cargados = pickle.load(f)
    print(datos_cargados)
```

En el código anterior, primero definimos un diccionario llamado `datos` que contiene información personal. Utilizamos `pickle.dump()` para escribir el diccionario en un archivo binario. El archivo se abre en modo escritura binaria (`'wb'`). Luego, leemos el archivo con `pickle.load()` para deserializar el objeto y recuperar el diccionario tal como estaba.

Cuando el archivo binario se ha deserializado y el objeto ha sido cargado en memoria, se puede acceder a sus componentes como lo haríamos con cualquier objeto nativo de Python. A continuación se muestra cómo acceder a cada uno de los campos del diccionario después de deserializarlo:

```
# Acceder a los valores del diccionario
nombre = datos_cargados['nombre']
edad = datos_cargados['edad']
ciudad = datos_cargados['ciudad']

# Mostrar los valores individualmente
print(f"Nombre: {nombre}, Edad: {edad}, Ciudad: {ciudad}")
```

En este caso, accedemos a las claves `'nombre'`, `'edad'` y `'ciudad'` del diccionario y mostramos los valores de forma individual. Esto es similar a cómo se acceden a los elementos en un diccionario normal.

Finalmente, es importante tener en cuenta que `pickle` es una herramienta poderosa, pero tiene algunos inconvenientes:

- **Seguridad**: Nunca deserialices datos de una fuente no confiable, ya que `pickle.load()` puede ejecutar código arbitrario.
- **Compatibilidad**: Los archivos `pickle` no son necesariamente portables entre versiones de Python o plataformas.
- **Tamaño**: Los archivos `pickle` pueden ser más grandes en comparación con archivos JSON, ya que incluyen información extra sobre la estructura del objeto. Sin embargo, son mucho más flexibles.

Podemos obtener más información sobre Pickle en su documentación oficial: Documentación de Pickle

8.2.2. Manejo de librerías para E/S

Existen varias librerías adicionales en Python que proporcionan herramientas para manejar archivos y directorios locales, así como para acceder a recursos remotos. Dos de las más comunes son `os` y `urllib`.

8.2.2.1. Librerías para ficheros locales: `os`

La librería `os` permite interactuar con el sistema operativo, incluyendo operaciones sobre archivos y directorios. Veamos un ejemplo sencillo:

```
import os

# Crear un directorio
os.mkdir('nuevo_directorio')

# Cambiar el directorio actual
os.chdir('nuevo_directorio')

# Mostrar el directorio de trabajo actual
print(os.getcwd())

# Volver al directorio anterior y eliminar el nuevo directorio
os.chdir('..')
os.rmdir('nuevo_directorio')
```

En este ejemplo, creamos un nuevo directorio, cambiamos al nuevo directorio, mostramos el directorio actual y luego volvemos atrás y eliminamos el directorio creado. Esto es útil cuando, desde nuestro código, tenemos que interactuar con carpetas y ficheros, por ejemplo para leer o escribir datos relevantes para nuestra aplicación.

La documentación oficial documenta todos los métodos que ofrece la librería, disponible aquí: Documentación de OS

8.2.2.2. Acceso a ficheros remotos con `urllib`

La librería `urllib` proporciona herramientas para acceder a recursos remotos (típicamente en Internet) a través de URLs. Es especialmente útil para descargar datos o interactuar con otras aplicaciones que proporcionan datos. Veamos un ejemplo de uso de esta librería:

```
from urllib import request

# Leer el contenido de una URL real (ejemplo del Proyecto Gutenberg)
# Aquí estamos leyendo el texto del libro "Drácula" de Bram Stoker
url = 'https://www.gutenberg.org/files/345/345-0.txt'

# Abrir la URL y leer el contenido
with request.urlopen(url) as response:
    # Leemos el contenido completo de la URL
```

```
    contenido = response.read()

    # Decodificar el contenido (está en bytes) a texto usando utf-8
    texto = contenido.decode('utf-8')

    # Mostrar una parte del contenido (por ejemplo, los primeros 500 caracteres)
    print(texto[:500])
```

El código comienza importando la librería `request` de `urllib`, lo que permite realizar solicitudes a recursos web. Luego, se define la URL de un libro disponible en formato de texto plano en el Proyecto Gutenberg (en este caso, *Drácula* de Bram Stoker), que es una URL pública que devuelve un archivo de texto. Utilizando `request.urlopen(url)`, se abre la URL para leer su contenido, lo que devuelve un objeto de respuesta que contiene los datos. A continuación, se lee el contenido de la respuesta utilizando el método `.read()`, que devuelve el contenido en formato de bytes. Dado que el contenido descargado está en formato binario, se usa `.decode('utf-8')` para convertirlo en una cadena de texto utilizando la codificación UTF-8, que es una de las más extendidas actualmente. Finalmente, se imprime una parte del contenido (los primeros 500 caracteres) para verificar que se ha descargado correctamente el archivo.

La documentación de esta librería, así como ejemplos adicionales, pueden consultarse en el siguiente enlace: Documentación de urllib

8.2.3. Generación y manipulación de imágenes

El código utiliza la librería `Pillow`, que es la versión moderna de la librería `PIL` (*Python Imaging Library*), para crear y manipular imágenes en Python. Permite abrir, modificar y guardar imágenes en varios formatos. Para poder hacer uso de ella debemos instalarla, pues no viene incorporada en Python. Para ello, ejecutaremos en la consola de Pycharm el comando *pip install pillow*. Esto hará que se instale la librería en el *virtualenv* del proyecto. Otra opción sería gestionar la instalación desde Pycharm de forma interactiva, de forma similar a como se hizo con `pydoctor` en la Sesión 1 de prácticas.

```
from PIL import Image, ImageDraw

# Crear una imagen nueva en blanco
imagen = Image.new('RGB', (200, 200), color = 'white')

# Dibujar un rectángulo en la imagen
dibujar = ImageDraw.Draw(imagen)
dibujar.rectangle([50, 50, 150, 150], outline="blue", width=5)

# Guardar la imagen
imagen.save('imagen.png')

# Abrir la imagen generada
imagen.show()
```

Primero, se importan los módulos `Image` y `ImageDraw`, que permiten crear imágenes y dibujar en ellas, respectivamente. Se crea una imagen nueva utilizando `Image.new()` con un tamaño de 200x200 píxeles, en modo `RGB` (color) y con un fondo blanco. A continuación, se instancia un objeto `ImageDraw.Draw()` que permite realizar operaciones de dibujo sobre la imagen. En este ejemplo, se dibuja un rectángulo utilizando el método `rectangle()`, donde los parámetros definen la posición y el tamaño del rectángulo, así como su color de borde y el grosor de la línea. Finalmente, la imagen se guarda en formato `PNG` utilizando `image.save()` y se muestra en pantalla con el método `image.show()`, que abre la imagen generada en el visor de imágenes predeterminado del sistema operativo.

El código anterior representa un ejemplo muy simple del uso de la librería. Sin embargo, permite muchas más funcionalidades. Entre las más relevantes podemos destacar la creación y manipulación de formatos de imagen, dibujar formas geométricas, o aplicar filtros y transformaciones. Para más información sobre la librería, se puede consultar su documentación en este enlace: Documentación de Pillow

Veamos un ejemplo en el que convertimos una imagen PNG a blanco y negro y la recortamos:

```
from PIL import Image

# Cargar la imagen PNG
imagen = Image.open('imagen_original.png')

# Convertir la imagen a blanco y negro (escala de grises)
imagen_bn = imagen.convert('L')

# Recortar la imagen (coordenadas de la caja: izquierda, superior, derecha, inferior)
# Por ejemplo, cortamos un cuadrado de 100x100 píxeles desde la esquina superior izquierda
imagen_recortada = imagen_bn.crop((0, 0, 400, 400))

# Guardar la imagen procesada en un nuevo archivo
imagen_recortada.save('imagen_bn_recortada.png')

# Mostrar la imagen procesada
imagen_recortada.show()
```

La Figura 8.1 muestra cómo quedaría una imagen de entrada a color (izquierda) tras pasarla a blanco y negro y recortarla (derecha).

(a) Imagen original (b) Imagen recortada en blanco y negro

Figura 8.1: Comparación entre la imagen original y la recortada en blanco y negro.

8.3 Relación de ejercicios

1. **Gestión de excepciones en clases de geometría**. El objetivo de este ejercicio es implementar una gestión de excepciones adecuada sobre las clases actualmente definidas, identificando posibles situaciones en las que se podrían generar errores y lanzar excepciones adecuadas para su manejo. Implementa el siguiente manejo de excepciones:
   - **Clase Punto**:
     - Actualización de las coordenadas X e Y: controla que los valores pasados sean numéricos (instancia de un tipo de datos correcto) y las coordenadas sean válidas. Entenderemos como coordenadas válidas aquellas comprendidas entre los valores -10.000 y 10.000. Haz uso de `assert`.
   - **Clase Linea**:
     - Cálculo de la longitud: en primer lugar, los puntos de inicio y fin deben estar definidos. Para ello, se usará `assert` para garantizar esta precondición. En segundo lugar, si los puntos no son idénticos (error a nivel del valor de los puntos), se lanzará una excepción.
   - **Clase Poligono**:
     - Actualización del número de lados: el valor suministrado es un entero y, además, se debe generar una excepción cuando el número de lados sea menor que 3.
     - Actualización del listado de lados: el valor suministrado deberá ser de tipo lista y todos los lados deberán ser de tipo Linea. Además, en caso de que la longitud de la lista no sea la misma que la del número de lados, se generará una excepción.
     - Eliminar lado aleatorio: tiene como precondición que el polígono tenga al menos un lado, pues si no no podremos eliminar ninguno. Por otro lado, en caso de que el número de lados no sea mayor que 3, se lanzará una excepción.
     - Cálculo del perímetro: en caso de que el polígono no tenga al menos tres lados, se lanzará una excepción.
     - Comprobación de polígonos vecinos: de partida, el otro polígono suministrado no podrá ser `None` y deberá tener lados definidos.
2. **Continuación de excepciones en clases de geometría**. Retoma el ejercicio anterior para definir precondiciones y excepciones sobre todas las clases restantes centradas en geometría en nuestro proyecto (todos los tipos de polígonos y figuras curvas).
3. **Gestión de excepciones en el main**: Añadir bloques try-except en el código del `main` para capturar y gestionar las excepciones que podrían lanzarse durante la ejecución del código anterior, mostrando un mensaje representativo al capturarlas
4. **Gestión de entrada/salida: representación textual**. Partiremos de la implementación realizada en la sesión de

prácticas previa para las clases `GestionPersistencia` e `IPersistente`. Sobre dicho código, debemos:

- Definir un nuevo método en la clase `GestionPersistencia` que permita guardar una representación textual de un objeto que ofrezca capacidad de persistirse. En concreto, la representación textual del objeto se guardará en un archivo con extensión `.txt`
- Realizar los cambios oportunos en la clase `IPersistente`, si aplica.

5. **Gestión de entrada/salida: archivos JSON**. Haz los cambios necesarios para que ahora GestionPersistencia sea capaz de guardar información en archivos JSON y recuperar su contenido.
6. **Gestión de entrada/salida: imágenes**. Haz los cambios necesarios para que ahora GestionPersistencia sea capaz de almacenar y recuperar la imagen asociada a una figura geométrica.
7. **Gestión de entrada/salida: archivos JSON con tipos complejos**. Extiende el código para que, además de los tipos elementales, también se guarden el resto de atributos. Realiza los cambios necesarios en el código.

UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 9 de Prácticas

Interfaz Gráfica

Autores: Alejandro Buitrago López (alejandro.buitragol@um.es), Sergio López Bernal (slopez@um.es), Javier Pastor Galindo (javierpg@um.es), Jesús Damián Jiménez Re (jesusjr@um.es) y Gregorio Martínez Pérez (gregorio@um.es)

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2024-2025

Última modificación:
4 de noviembre de 2024

Sesión 9: Interfaz Gráfica

Índice

9.1 Introducción a las Interfaces Gráficas de Usuario

Una interfaz gráfica de usuario (en inglés GUI, de *Graphical User Interface*), es la parte del software que realiza la función de **interactuar con el usuario** de una manera sencilla y visual. Normalmente están compuestas tanto por imágenes como por objetos gráficos y permiten representar la información y acciones que se encuentran en la interfaz. El objetivo es crear un entorno visual que sea fácil de utilizar y que facilite la **comunicación** entre el usuario y el sistema operativo.

Aunque hoy día estamos realmente acostumbrados a estas aplicaciones, no hace tantos años las interfaces de usuario se basaban en **líneas de comando** (véase Figura 9.1), a través de sistemas operativos en los que los usuarios daban instrucciones a algún programa informático por medio de una línea de texto simple.

9.1.1. Elementos de una GUI

Las Interfaces Gráficas de Usuario deberían ser **consistentes**, **intuitivas** y fácilmente **entendibles** por el usuario, sin que éste requiera de habilidades informáticas avanzadas ni conocimiento de programación previos (aunque son desarrolladas a través de programación). Así, deben proporcionar a los usuarios la **habilidad de usar un programa** sin tener que preocuparse de los comandos para su ejecución.

Para conseguirlo, existe una gran variedad de **componentes** que hacen de la **navegación por la interfaz** mucho más amena. En este apartado hacemos un breve repaso de algunos de estos componentes; pero existen muchos más, y además

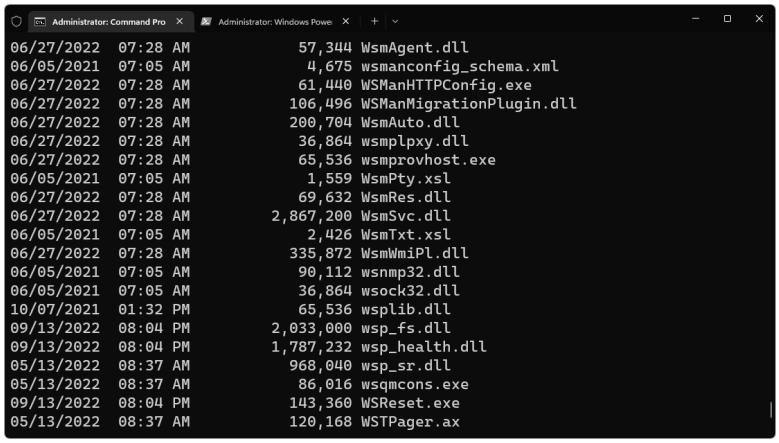


Figura 9.1: Ventana de comandos de Windows.

son dependientes del **lenguaje de programación**, de las **librerías** utilizadas, del sistema operativo, y de varios factores adicionales:

9.1.1.1. Ventana

Son **áreas visuales**, normalmente de forma rectangular, que contienen la interfaz, mostrando la **salida** y permitiendo la **entrada de datos** para los procesos que se ejecutan en el programa informático. Suelen contar con un conjunto de propiedades y características que el usuario puede personalizar a su gusto en función de las necesidades de cada momento. Por ejemplo, la mayoría de las ventanas pueden ser **redimensionadas** (maximizadas, minimizadas, . . .), movidas, ocultadas, restauradas y cerradas a voluntad.

A su vez, como se puede observar en la Figura 9.2, podríamos encontrar diferentes tipos de ventanas dentro un programa:

- **Ventanas de aplicación**: las más comunes, son el tipo estándar de ventanas, que contienen documentos o datos de la aplicación.
- **Ventanas de utilidad**: aparecen superpuestas al resto de ventanas y ofrecen herramientas o información sobre una aplicación.
- **Cuadros de diálogo**: ventanas emergentes que informan de alguna acción al usuario o que le solicitan confirmar o aceptar cierta información.
- **Inspectores**: ventas que muestran propiedades de un elemento y que aparecen siempre por encima de otras ventanas de la misma aplicación.

Una ventana está compuesta por varios elementos que facilitan el acceso a distintas opciones y funcionalidades de la aplicación con la que estamos trabajando. Estos elementos, conocidos como **widgets** (explicados en detalle en la Sección 9.1.2), pueden adoptar diversas formas, tales como barras, botones, y etiquetas (ver Figura 9.3). A continuación, repasaremos algunas de las barras más comunes que suelen aparecer en una ventana:

- **Barra de título**: elemento en la parte superior de una ventana, donde generalmente aparece el nombre de la aplicación que se está utilizando o el nombre del fichero que tengamos abierto en ese momento.
- **Barra de menús**: en Windows esta barra ocupa típicamente la parte superior de la ventana y a través de ella podemos acceder a determinadas opciones de la aplicación con la que estamos trabajando (Archivo, Edición, Vista. . .). Estas opciones se visualizan como menús desplegables por los que el usuario puede navegar y acceder a la opción que desee.
- **Barra de herramientas**: conjunto de iconos que permite acceder rápidamente a herramientas o funciones específicas de una aplicación. Ofrece “atajos” a las funciones más usadas de la barra de menús. En Windows, suele ubicarse en la parte superior de la ventana, justo debajo de la barra de menús. En general, las operaciones disponibles en la barra de herramientas también se encuentran en la barra de menús.
- **Barra de pestañas**: hay aplicaciones que permiten tener varias ventanas agrupadas dentro de una misma ventana. Esto es frecuente en los navegadores de Internet, que permiten tener abiertas varias ventanas simultáneamente, aunque solo tengamos una activa. A través de esta barra podemos seleccionar qué pestaña tenemos activa en cada momento.
- **Barra de progreso**: elemento de una ventana que permite mostrar de forma gráfica el estado de avance de una tarea o proceso que se esté ejecutando en una aplicación.

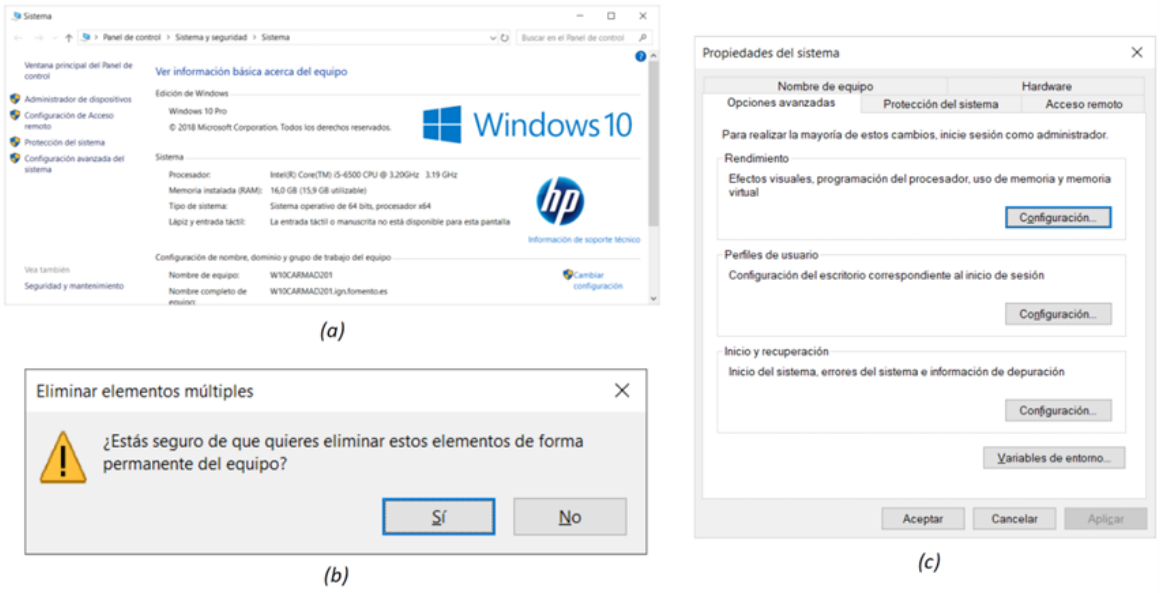


Figura 9.2: Ejemplos de (a) ventana de aplicación, (b) cuadro de diálogo, y (c) ventana de utilidad.

- **Barra de desplazamiento**: barra horizontal o vertical, con dos extremos con flechas que apuntan en sentidos contrarios que permiten desplazar el contenido del cuadro hacia un lado u otro. Las barras suelen activarse cuando el recuadro de la ventana no es lo suficientemente grande como para visualizar todo su contenido. Estas barras las solemos encontrar en páginas web y en visores de texto. Es difícil visualizar en una sola ventana toda la información que contiene una web o un archivo de texto y por eso es frecuente navegar en ellos hacia arriba y hacia abajo con las opciones de esta barra.
- **Barra de estado**: generalmente se ubica en la parte inferior de las ventanas y muestra información acerca de la aplicación con la que se está trabajando.

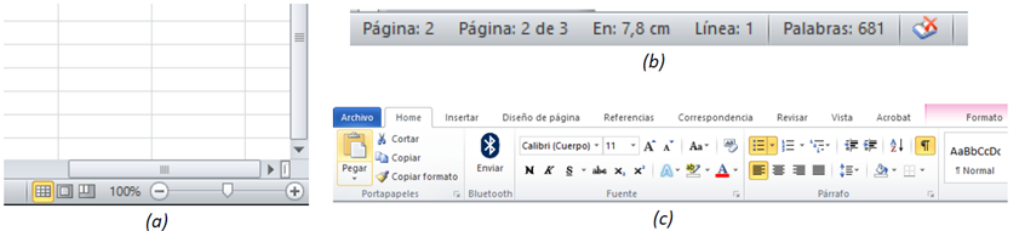


Figura 9.3: Ejemplos de (a) barras de desplazamiento, (b) barra de estado y (c) barra de menús y herramientas.

9.1.1.2. Botones

Los **botones** son uno de los elementos más comunes en cualquier interfaz. Aunque la tipología es muy variada, por norma general los botones se suelen representar con un **rectángulo** que cuenta con una leyenda o icono en su interior, generalmente con efecto de relieve. A pesar de la variada casuística de botones, todos tienen un **objetivo común**: buscan la **interacción del usuario** con el software.

Algunos botones están diseñados para ser presionados solo una vez y ejecutar un comando, mientras que otros pueden usarse para recibir **retroalimentación instantánea** y pueden requerir que el usuario haga clic más de una vez para recibir el resultado deseado. Otros botones están diseñados para **activar y desactivar** el comportamiento como una casilla de verificación. A continuación, se describen brevemente los **principales tipos de botón** que podemos encontrar en una GUI (ver Figura 9.4):

- **De pulsación**: responde a un clic del usuario y es utilizado para iniciar o confirmar una acción. Algunos ejemplos de estos botones son los de “Enviar” o “Cancelar”.
- **De lista**: los botones de selección en una lista permiten el acceso rápido a una lista desplegable de elementos. La lista solo es visible cuando se acciona el botón que tiene asociado.

- **De marcado o de verificación (checkbox)**: los botones de comprobación se utilizan frecuentemente como botones de estado. Proporcionan información del tipo "Sí" o "No"; el argumento es del tipo booleano (true o false). En los checkbox, de haber más de una opción, pueden activarse varias.
- **De opción (radio button o checkbox group)**: los botones de verificación se pueden agrupar para formar un botón de radio. Se trata de una agrupación de botones checkbox en la que siempre hay un único botón activo.

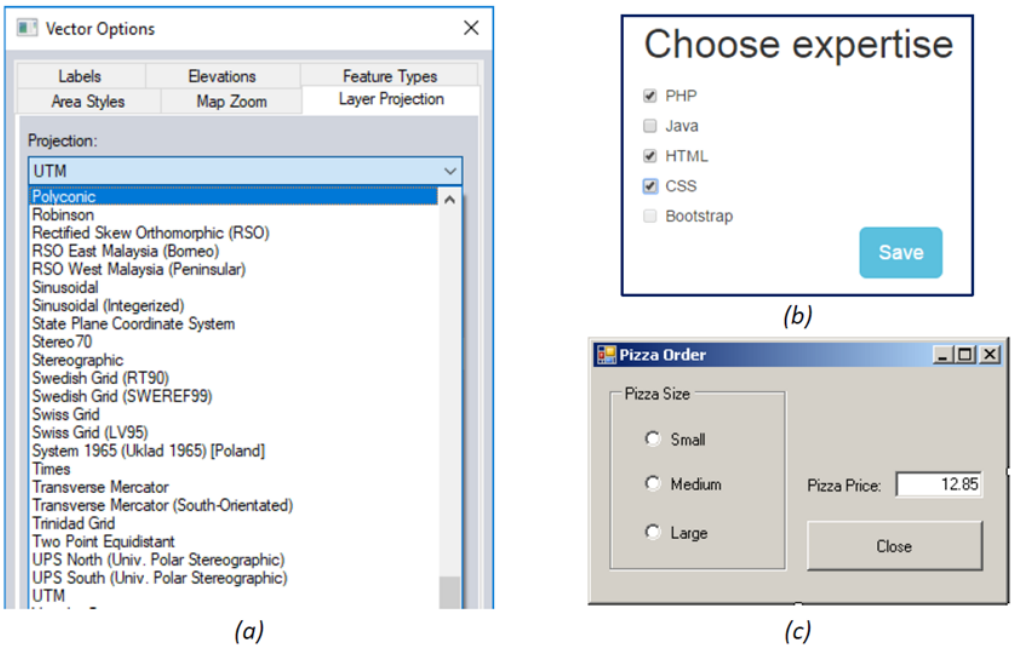


Figura 9.4: Ejemplos de (a) botones de lista, (b) botones de verificación (checkbox) y (c) opción (radio).

9.1.1.3. Otros elementos de una interfaz

Existen una amplia tipología de elementos que nos podemos encontrar en una interfaz, además de las ventanas y los botones que hemos visto anteriormente. A continuación, se describen algunos de los componentes más habituales (ver Figura 9.5):

- **Etiquetas / labels**: las etiquetas son widgets estáticos que proporcionan una forma de mostrar un texto en una ventana. Su objetivo es el de mostrar información que pueda ser de utilidad para el usuario de la aplicación, no buscan interacción usuario-aplicación.
- **Listas**: se trata de una impresión en una ventana de una serie de elementos. A diferencia de los botones de lista, las listas son visibles todo el tiempo, lo que facilita al usuario la manipulación simultánea de muchos elementos.
- **Campos de texto**: son widgets cuyo objetivo es que el usuario interactúe con la aplicación. Estos widgets permiten la entrada directa de datos por teclado. Los campos de texto se pueden crear vacíos o rellenos con un texto predefinido. En ocasiones se pueden autocompletar con botones de lista (p.ej. para seleccionar el directorio en el que queremos almacenar cierta información).
- **Áreas de texto**: las áreas de texto permiten al usuario incorporar texto multilínea dentro de una ventana habilitada para este fin. Estos objetos se utilizan para introducir elementos de texto que ocupan más de una línea.
- **Canvas**: son lienzos o zonas de dibujo que pueden albergar imágenes o capturar eventos de exposición, de ratón u otros eventos.
- **Barras de desplazamiento (Scrollbar)**: permiten trabajar con rangos de valores o áreas. Estas barras se suelen acompañar de una caja de texto que muestre los valores asociados a los desplazamientos de la scrollbar.

9.1.2. Concepto de widget

Un **widget** es cualquier elemento gráfico que forma parte de la interfaz de usuario de una aplicación. Los widgets son los **componentes básicos** con los que los usuarios interactúan, y en realidad pueden ser cualquiera de los componentes explicados en las secciones anteriores, como botones, cuadros de texto, etiquetas, menús, etc. Cada widget tiene su propio conjunto de **propiedades y métodos** para controlar su apariencia y comportamiento.

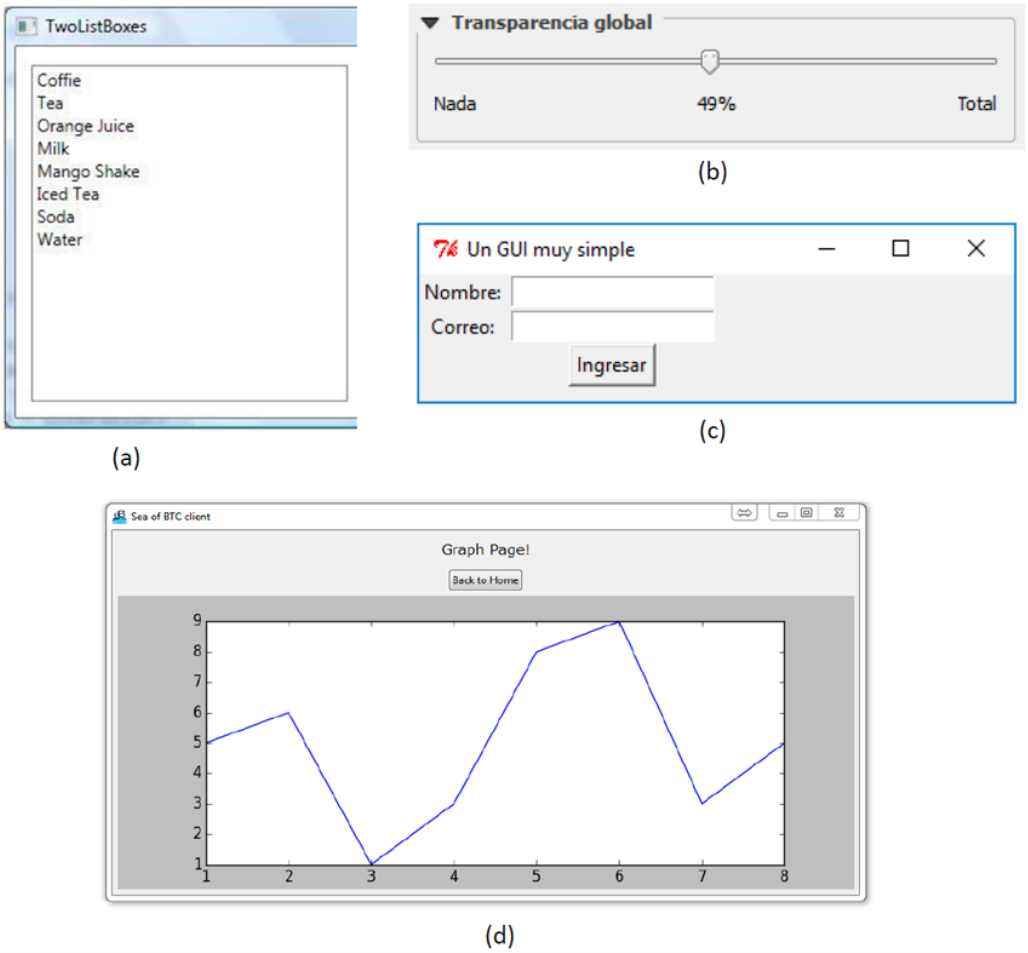


Figura 9.5: Ejemplos de (a) listas, (b) barras de desplazamiento, (c) campos de entrada de texto y (d) canvas con un gráficos.

En particular, en **Tkinter**, que veremos a continuación en la Sección 9.2, nos referiremos a un widget como cualquiera de los elementos existentes dentro de la GUI. Aunque no vamos a profundizar mucho en ello, es interesante conocer que en esta librería existe un módulo llamado *Geometry Manager* que es el encargado de indicar exactamente dónde se colocará cada widget, capaz de trabajar con lo que llama widgets *master* (maestro) y *slave* (esclavo). Se entiende por un widget maestro a una ventana o marco que contiene incrustados widgets esclavos.

9.2 Librería Tkinter

Antes de llegar a entender qué es Tkinter, tenemos que saber que *Tcl* es un lenguaje multiplataforma de scripting, muy utilizado en sistemas operativos como Linux pero que también puede ser utilizado en Windows. Por otro lado, *Tk* es una GUI multiplataforma que permite la creación de interfaces gráficas justamente para *Tcl*. Pues bien, podríamos decir que Tkinter actúa como un "puente" entre Python y Tcl/Tk, permitiéndole crear interfaces gráficas utilizando la sintaxis y funcionalidad de Python.

Hoy día Tkinter es considerado un estándar para la interfaz gráfica de usuario (GUI) para Python [1] y viene preinstalado por defecto con las versiones para Microsoft Windows y en la mayoría de las distribuciones con GNU/Linux. Se puede encontrar más información aquí: `https://docs.python.org/es/3.10/library/tkinter.html`

9.2.1. Instalación e importación del módulo

La librería TKinter debería estar preinstalada en Python. No obstante, podríamos comprobar en *PyCharm* que está instalada si, al escribir lo siguiente al principio del cualquier módulo, no se produce ningún error:

[1] Aunque existen otras librerías de GUI para Python, como son wxPython, PyQt y PyGTK, pero en la asignatura nos centraremos en la más ampliamente utilizada que es Tkinter.

```
from tkinter import *
```

La anterior es una manera de importarla en nuestros módulos, pero también podría importarse de cualquier de las dos siguientes maneras:

```
import tkinter
import tkinter as tk    # Simplifica la escritura posteriormente
```

9.2.2. Creación de una primera ventana

Continuando con el proyecto de las sesiones prácticas anteriores (Libros y Personas), durante esta guía vamos a ir creando una pequeña interfaz gráfica que nos permita crear un objeto de una de las dos clases de libros que tenemos definidas hasta ahora (LibroCientifico y LibroInfantil). Omitiremos por simplificar los libros de tipo diario.

Dentro de nuestro módulo *main.py* podríamos crear una primera ventana gráfica de la siguiente manera:

```
root = tk.Tk()
root.title ("Alta de libro")
# root.resizable(True, False)
root.geometry ("800x600")
root.mainloop()
```

1. La primera línea crea un objeto *root* que representará la ventana. Se utiliza la palabra *root* por convención entre los programadores de Python, pero no es necesario que se llame así.
2. Invocando al método *title()* lo que hacemos es ponerle un título a la ventana.
3. El método *resizable()* tiene dos parámetros booleanos, que indican si la ventana permite cambiar sus dimensiones por el usuario tanto en vertical como horizontal, respectivamente.
4. El método *geometry()* lo que establece es el tamaño **inicial** en píxeles de la ventana (horizontal x vertical). Tiene dos parámetros opcionales que permiten definir el valor en el eje X y el eje Y donde se situará la ventana. Existe un método *maxsize()* que establece el tamaño máximo de la ventana, en el caso de permitirse modificar su tamaño por el usuario.
5. Finalmente, la ventana se mostrará cuando invoquemos el método *mainloop()*. Es importante destacar que si ejecutamos el código anterior desde un programa principal en Python (por ejemplo, el main.py), el programa quedará parado tras crear la ventana, hasta que ésta sea cerrada, pulsando el botón "X".

9.2.3. Etiquetas en la ventana: Widget label

Las etiquetas (*labels* en inglés) son uno de los mecanismos que permiten introducir textos informativos dentro de las ventanas. Para conseguirlo en TKinter, lo que debemos hacer es:

1. Crear variables de tipo Label (dándoles un nombre), a través de la función *Label*, que recibe como primer parámetro el nombre de la ventana (normalmente `root`), y como segundo parámetro el texto a mostrar (por ejemplo `text="Esta es una ventana"`).
2. Posicionar estas etiquetas en la ventana, a través de una de estas tres funciones:
   - **pack**: Posiciona los widgets en la ventana según el orden que indiquemos en el código, pero no nos permite determinar la posición de cada elemento dentro de la ventana. Va colocando cada texto uno debajo del otro.
   - **grid**: Permite gestionar la posición de los widgets en la ventana, indicándole la celda en la que se sitúa haciendo uso para ello del índice de fila y de columna correspondiente. Por tanto, antes de definir los widgets y sus posiciones, debemos crear la cuadrícula de la ventana y configurar el número y formato de filas y columnas de la ventana con la función *rowconfigure* y *columnconfigure*. (Las filas y las columnas empiezan en 0).
   - **place**: Posiciona los widgets estableciendo las coordenadas (x,y) del widget dentro de la ventana. Permite la máxima flexibilidad, pero lo cierto es que es el más difícil de utilizar porque debemos calcular constantemente los píxeles para posicionar los widgets en la ventana. No lo vamos a utilizar en estas prácticas.

Estas tres formas de posicionar las etiquetas serían perfectamente válidas para aplicarlos a otros widgets de la GUI que crearemos posteriormente. Veamos a continuación una manera de posicionar el texto a través del método *pack*. Lo describimos aquí a título informativo, pero **NO** vamos a utilizarlo para nuestro ejemplo de alta de Libros:

```
root = tk.Tk()
root.title ("Alta de libro")
etiq_tipo_libro = tk.Label(root, text="Seleccione el tipo de libro: ")
etiq_tipo_libro.pack()
root.geometry ("800x600")
root.mainloop()
```

Como se observa, la colocación de etiquetas (y widgets en general) a través del método pack no permite mucha flexibilidad. Para tener más control, vamos a ver como podríamos utilizar el método *grid*, y para ello configuraremos nuestra ventana con la estructura materializada en la Tabla 9.1.

	Columna 0	**Columna 1**	**Columna 2**
Fila 0	(0,0)	(0,1)	(0,2)
Fila 1	(1,0)	(1,1)	(1,2)
Fila 2	(2,0)	(2,1)	(2,2)
Fila 3	(3,0)	(3,1)	(3,2)
Fila 4	(4,0)	(4,1)	(4,2)
Fila 5	(5,0)	(5,1)	(5,2)
Fila 6	(6,0)	(6,1)	(6,2)
Fila 7	(7,0)	(7,1)	(7,2)

Tabla 9.1: Estructura de la ventana haciendo uso del método *grid*.

El planteamiento será crear una ventana para el alta de los libros, que contendrá los siguientes elementos (los iremos completando a lo largo de las secciones siguientes):

- (0,0) = Texto: Tipo de libro
- (0,1) = Radio button: O Infantil
- (0,2) = Radio button: O Científico
- Desde (1,0)-(1,1) hasta (3,0)-(3,1): Irán las etiquetas y los campos de texto que contienen los atributos básicos de un libro.
- Desde (4,0)-(4,1) hasta (6,0): Irán las etiquetas y los campos de texto que contienen los atributos específicos de un libro infantil (son 3) o bien de un libro científico (son 2).
- (7,0): Botón guardar.
- (7,1): Botón cancelar.
- (7,2): Botón de información.

Modifiquemos el código anterior para utilizar el Grid:

```
root = tk.Tk()
root.title("Alta del libro")
root.columnconfigure(3, weight=1)
root.rowconfigure(8, weight=1)
etiq_tipo_libro = tk.Label(root, text="Seleccione el tipo de libro: ")
etiq_tipo_libro.grid(row=0, column=0, padx= 20, pady=20)
root.geometry("800x600")
root.mainloop()
```

La función *grid* permite muchos parámetros, como el espaciado horizontal y vertical de la celda (*padx* y *pady*), el espaciado interno horizontal y vertical (*ipadx* e *ipady*); si queremos que el widget ocupe más de una columna y/o fila y cuantas más (*columnspan* y *rowspan*), así como permitir que un widget sea expansible y que esté anclado a ciertas posiciones de la ventana (*sticky*).

Todo esto va más allá de estas prácticas, pero se puede consultar la siguiente documentación: https://www.pythonguis.com/tutorials/create-ui-with-tkinter-grid-layout-manager/

9.2.4. Creación de una clase para la interfaz gráfica

Al igual que tenemos clases que representan nuestros modelos de la realidad, como Libro, LibroCientífico, LibroInfantil y Persona, lo normal al crear una interfaz gráfica es crear nuestra (o nuestras) propia clase que representa la interfaz gráfica o GUI. Vamos por ello a generar la misma ventana que llevamos hasta ahora, pero creando nuestra propia clase, que podríamos crear dentro del paquete y módulo: `app.gui.alta_libro_gui`:

```
import tkinter as tk

class AltaLibroGUI:
    def __init__ (self, master):

        # Atributo de instancia que mantiene la ventana en TK
        self._master = master

        # Configura la ventana de Alta de libros
        self._master.title("Alta de libro")
        self._master.columnconfigure(3, weight=1)
        self._master.rowconfigure(8, weight=1)
```

```
        self._master.geometry("800x600")

        # Convertimos las etiquetas en atributos de instancia
        self._etiq_tipo_libro = tk.Label(self._master,
                                         text="Seleccione el tipo de libro: ")


    # Muestra la ventana
    def muestra_ventana(self):
        # Posicionamos las etiquetas
        self._etiq_tipo_libro.grid(row=0, column=0, padx=20, pady=20)
        self._master.mainloop()
```

En el programa principal, crearíamos un objeto de esta clase de la siguiente manera:

```
# Interfaz gráfica:
root = tk.Tk()
alta_libros_gui = AltaLibroGUI(root)
alta_libros_gui.muestra_ventana()
```

Es importante destacar que la ventana gráfica creada es idéntica a la mostrada en el código de la sección anterior en la que todo se programaba en el *main.py*, quedando como resultado la ventana mostrada en la Figura 9.6.

Figura 9.6: Implementación parcial de una ventana con un único label.

9.2.5. Botones en Tkinter

Los botones son uno de los widgets que más se suelen utilizar en cualquier interfaz gráfica. En Tkinter se utilizan creando objetos *Button*, y pueden contener texto, imágenes o llamar a otras funciones de Python. Al crear un Button se deben pasar los siguientes argumentos:

- La ventana a la que está asociado (obligatorio)
- El texto que contendrá (opcional, pero muy recomendable).
- Función de Python que se ejecutará al hacer clic en dicho botón (opcional).

En el siguiente código vamos a añadir tres botones a nuestra ventana de alta de libros:

- Uno de ellos nos permitirá cerrar la ventana sin hacer nada (Cancelar).
- Otro permitirá abrir una ventana modal con información sobre los tipos de libros disponibles.
- El último, creará un objeto del tipo de libro adecuado, y cerrará la ventana.

```
import tkinter as tk

class AltaLibroGUI:
    def __init__ (self, master):

        # Atributo de instancia que mantiene la ventana en TK
```

```
        self._master = master

        # Configura la ventana de Alta de libros
        self._master.title("Alta de libro")
        self._master.columnconfigure(3, weight=1)
        self._master.rowconfigure(8, weight=1)
        self._master.geometry("800x600")

        # Etiqueta para pedir el tipo de libro
        self._etiq_tipo_libro = tk.Label(self._master, text="Seleccione el tipo de libro: ")

        # Botones
        self._boton_info = tk.Button(self._master, text="Ayuda: Tipos de libros")
        self._boton_cancelar = tk.Button(self._master, text='Cancelar')
        self._boton_guardar = tk.Button(self._master, text='Guardar')

    # Muestra la ventana
    def muestra_ventana(self):
        # Posicionamos las etiquetas
        self._etiq_tipo_libro.grid(row=0, column=0, padx=20, pady=20)

        # Posicionamos los botones
        self._boton_guardar.grid(row=7, column=0, padx=20, pady=20)
        self._boton_cancelar.grid(row=7, column=1, padx=20, pady=20)
        self._boton_info.grid(row=7, column=2, padx=20, pady=20)

        # Mostrar la pantalla
        self._master.mainloop()
```

La Figura 9.7 muestra el resultado de la ventana con el añadido de los tres botones definidos en el código anterior.

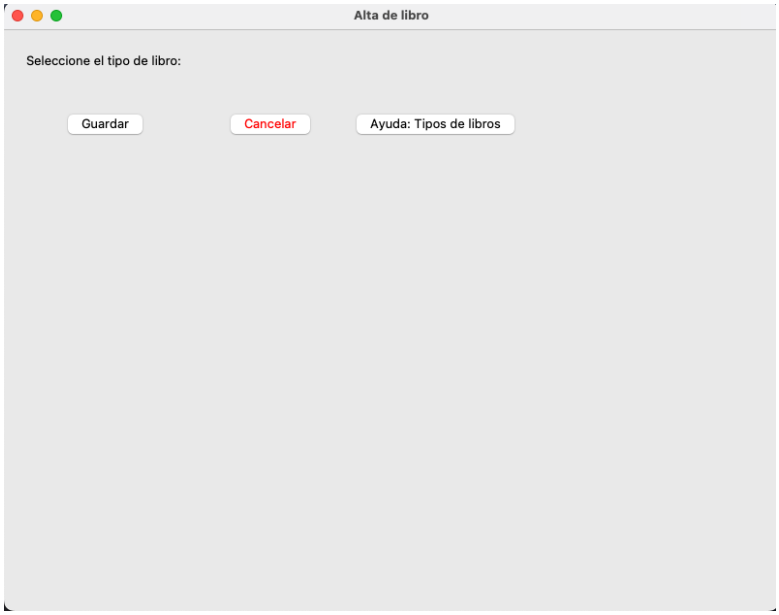


Figura 9.7: Ejemplo de ventana con un label y tres botones.

9.2.5.1. Añadir a un botón llamadas a funciones predefinidas

Por sí mismos, los botones no realizan ninguna acción. Lo interesante es enlazarlos a funciones de tal forma que, al pulsar el botón, se ejecute una operación. Para ello, el argumento *command* permite enlazar la función que queremos ejecutar al pulsar el botón.

En la siguiente sección veremos que podemos definir nuestras propias funciones; pero también existen funciones que Python tiene predefinidas y que podemos ejecutar sin necesidad de definir previamente. Por ejemplo, una de ellas permite cerrar una ventana, que es justo lo que necesitamos que suceda cuando pulsamos el botón *Cancelar*:

```
(...)
self._boton_cancelar = tk.Button(self._master, text='Cancelar',
command=self._master.destroy, fg="red")
(...)
```

Existen una gran variedad de funciones predefinidas que podríamos utilizar, pero exceden el ámbito de esta asignatura.

9.2.5.2. Añadir a un botón llamadas a funciones propias

Ahora vamos a crear un método propio dentro de la clase `AltaLibroGUI`, y lo vamos a vincular a uno de los botones que tenemos creado para que se ejecute al pulsarlo (concretamente al botón `boton_info`). De momento la implementación del método solo imprimirá un mensaje en la consola, pero en la siguiente sección veremos cómo podemos crear ventanas emergentes para mostrar información.

```
class AltaLibroGUI:
    def __init__ (self, master):

         [...]
        # Botones
        self._boton_info = tk.Button(self._master,
    text="Ayuda: Tipos de libros",
    command=self.muestra_info_libros)
         [...]

    # Muestra INFO de tipos de libro
    def muestra_info_libros(self):
        print("Aquí crearemos una ventana emergente con info")
```

9.2.6. Ventanas emergentes

Las ventanas emergentes permiten notificar al usuario de una cierta actividad o información que se ha realizado en la aplicación. En esta sesión, las utilizaremos para mostrar la información sobre los tipos de libro que se pueden dar de alta.

En Tkinter, se utiliza el módulo `tkinter.messagebox` para crear ventanas emergentes (también llamadas en ocasiones mensajes modales). El método básico es utilizar `messagebox.Message(master=None, **options)`, pero existen más posibilidades:

- **Mensajes de información:**
  - `messagebox.showinfo()`
- **Mensajes de aviso:**
  - `messagebox.showwarning()`
  - `messagebox.showerror()`
- **Mensajes de preguntas:**
  - `messagebox.askquestion()`
  - `messagebox.askokcancel()`
  - `messagebox.askretrycancel()`
  - `messagebox.askyesno()`
  - `messagebox.askyesnocancel()`

Modifiquemos ahora nuestro método `muestra_info_libros` para que se muestre información en una ventana emergente:

```
# Muestra INFO de tipos de libro
def muestra_info_libros(self):
    mensaje_info:str = "En esta pantalla podrá dar de alta un nuevo libro en el sistema. \n"
    mensaje_info += "Tendrá que elegir entre un Libro Infantil y un Libro científico.\n"
    mensaje_info += "Los libros infantiles tienen una edad recomendada y pueden ser interactivos. \n"
    mensaje_info += "Los libros científicos tienen un campo de estudio y un nivel de dificultad. \n"
    tk.messagebox.showinfo("Información sobre libros", mensaje_info)
```

Para que este código funcione, hace falta importar el paquete `messagebox` al principio de la clase:

```
from tkinter import messagebox
```

La Figura 9.8 muestra el contenido de la ventana informativa creada. Es importante destacar que la estética de las ventanas dependerá mucho del sistema operativo en el que se ejecuten.

9.2.7. Cuadros de entrada de información

Uno de los widgets que más se utilizan son los cuadros de entrada de texto, que permiten la interacción con el usuario dentro de lo que se denomina un formulario. Los formularios permiten introducir, a través de estos campos de texto, información al usuario mediante teclado.

En Tkinter estos widgets se generan con la función *Entry* y *Text*. La principal diferencia entre ambas funciones es que *Entry* permite la entrada de una única línea de texto mientras que la función *Text* acepta una entrada multilínea.

Para poder utilizarlos, primero debemos comprender el concepto de **Variable de Control**.

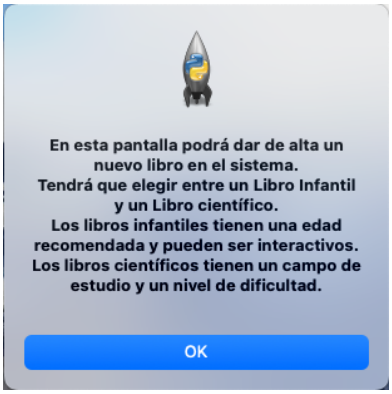


Figura 9.8: Ejemplo de ventana informativa.

9.2.7.1. Variables de Control

Las variables de control son objetos que se asocian a los widgets, y almacenan valores del widget para que estén disponibles en otras partes del programa. De esta manera, cuando cambian sus valores, queda reflejado de forma automática en el widget. Existen cuatro tipos:

```
entero = IntVar()    # Para trabajar con enteros
real = DoubleVar()    # Para usar números de tipo flotante
cadena = StringVar()    # Para usar cadenas de caracteres
booleano = BooleanVar() # Para trabajar con booleanos.
```

Respecto a los métodos, estas variables cuentan con los siguientes:

- `set(valor)`: asigna un valor a la variable de control.
- `get()`: devuelve el valor de la variable de control.
- `trace(tipo, función)`: se emplea para detectar cuándo la variable es leída, cambia su valor o es borrada. Siendo:
  - **tipo**: el tipo de suceso a comprobar: `'r'` para lectura de variable; `'w'` para escritura de variable; `'u'` para borrado de variable.
  - **función**: indica la función que será llamada cuando ocurra el suceso.

Siguiendo con nuestro ejemplo vamos a crear en nuestra clase las siguientes variables de control, que asociaremos a nuestros campos de texto de tipo Entry después:

```
class AltaLibroGUI:
    def __init__ (self, master):

        # Atributos de cualquier libro (Variables de control)
        self._titulo = tk.StringVar()
        self._autor = tk.StringVar()
        self._paginas = tk.IntVar()

        # Atributos de libros infantiles (Variables de control)
        self._edad_recomendada = tk.IntVar()
        self._ilustraciones = tk.BooleanVar()
        self._interactividad = tk.BooleanVar()

        # Atributos de libros científicos (Variables de control)
        self._campo_estudio = tk.StringVar()
        self._nivel_dificultad = tk.StringVar()
        [...]
```

9.2.7.2. Campos Entry

Los campos Entry tienen los siguientes parámetros mínimos que habrá que rellenar:

- Nombre de la ventana a la que se quiere asociar este elemento.
- Nombre de la variable a la que se va a asociar el texto que se introduzca por teclado. Este texto se puede recuperar y utilizar en otra función dentro de la misma ventana.

```
class AltaLibroGUI:
    def __init__ (self, master):

      [...]
        # Campos de texto comunes a todos los clientes y sus etiquetas
```

```
        self._etiq_titulo = tk.Label(self._master, text="Título: ")
        self._entry_titulo = tk.Entry(self._master, textvariable=self._titulo)

        self._etiq_autor = tk.Label(self._master, text="Autor: ")
        self._entry_autor = tk.Entry(self._master, textvariable=self._autor)

        self._etiq_paginas = tk.Label(self._master, text="Num Páginas: ")
        self._entry_paginas = tk.Entry(self._master, textvariable=self._paginas)

        # Campos de texto específicos de libros infantiles
        self._etiq_edad_recomendada = tk.Label(self._master, text="Edad recomendada: ")
        self._entry_edad_recomendada = tk.Entry(self._master, textvariable=self._edad_recomendada)

        self._etiq_ilustraciones = tk.Label(self._master, text="¿Contiene ilustraciones?")
        self._entry_ilustraciones = tk.Entry(self._master, textvariable=self._ilustraciones)

        self._etiq_interactividad = tk.Label(self._master, text="¿Es interactivo?")
        self._entry_interactividad = tk.Entry(self._master, textvariable=self._interactividad)

        # Campos específicos de libros científicos
        self._etiq_campo_estudio = tk.Label(self._master, text="Campo de estudio: ")
        self._entry_campo_estudio = tk.Entry(self._master, textvariable=self._campo_estudio)

        self._etiq_nivel_dificultad = tk.Label(self._master, text="Nivel de dificultad: ")
        self._entry_nivel_dificultad = tk.Entry(self._master, textvariable=self._nivel_dificultad)

        [...]
```

9.2.7.3. Campos Text

Este tipo de widgets de texto permiten editar un texto de varias líneas y personalizar la forma en que debe mostrarse, como cambiar su color y fuente. La diferencia principal con los campos *Entry*, es que el texto introducido en el widget *Text* no se almacena en ninguna variable. **En esta sesión, no vamos a utilizarlos**.

9.2.8. Botones de radio (Radiobutton)

Los botones de radio (`Radiobutton`) permiten elegir una de las opciones que aparecen en una lista. Sólo es posible elegir un valor de esta lista (si fueran varios los valores posibles a elegir, se utilizaría otro tipo de botón que se llama `checkbox`).

Los argumentos de entrada para estos botones son:

- Nombre de la ventana a la que se asocia el widget.
- Texto que va a acompañar al botón (`text="Texto que acompaña al botón"`)
- Un valor numérico que se vincula a este botón de radio (`value="Valor que se da a este botón de radio"`)
- Nombre de la variable al que se asocia dicho valor numérico (`variable = "nombre de la variable"`). La variable debe estar declarada previamente.

En el siguiente código, vamos a añadir un par de botones de radio que permitan elegir el tipo de libro que queremos dar de alta, y haremos que cuando se pulse cualquier de ellos, se lance un método que permitirá mostrar los atributos específicos de uno u otro tipo de libro, en base al botón pulsado.

```
class AltaLibroGUI:
    def __init__ (self, master):

        [...]

        # Atributo que contiene si es un Libro Infantil o Científico.
        # 1=Infantil. 2=Científico
        self._tipo_libro = tk.IntVar()  # Variable de control del radiobutton

        [...]

        # Radiobutton que permite elegir el tipo de libro
        self._etiq_tipo_libro = tk.Label(self._master,
                                text="Seleccione el tipo de libro: ")

        self._rboton_tipo_libro_infantil = tk.Radiobutton(self._master,
                                text="Libro infantil", value=1,
                                variable=self._tipo_libro,
                                command=self._muestra_atributos_tipo_libro)

        self._rboton_tipo_libro_cientifico = tk.Radiobutton(self._master,
                                text="Libro científico", value=2,
                                variable=self._tipo_libro,
                                command=self._muestra_atributos_tipo_libro)
        [...]
```

```
    def _muestra_atributos_tipo_libro(self):
        if self._tipo_libro.get() == 1:    # Libro infantil
            # Ocultamos los atributos de libro científico
            self._etiq_campo_estudio.grid_forget()
            self._entry_campo_estudio.grid_forget()

            self._etiq_nivel_dificultad.grid_forget()
            self._entry_nivel_dificultad.grid_forget()

            # Mostramos los atributos de libro infantil
            self._etiq_edad_recomendada.grid(row=4, column=0, padx=5, pady=5)
            self._entry_edad_recomendada.grid(row=4, column=1, padx=5, pady=5)

            self._etiq_ilustraciones.grid(row=5, column=0, padx=5, pady=5)
            self._entry_ilustraciones.grid(row=5, column=1, padx=5, pady=5)

            self._etiq_interactividad.grid(row=6, column=0, padx=5, pady=5)
            self._entry_interactividad.grid(row=6, column=1, padx=5, pady=5)
            return

        if self._tipo_libro.get() == 2:
            # Ocultamos los atributos de libro infantil
            self._etiq_edad_recomendada.grid_forget()
            self._entry_edad_recomendada.grid_forget()

            self._etiq_ilustraciones.grid_forget()
            self._entry_ilustraciones.grid_forget()

            self._etiq_interactividad.grid_forget()
            self._entry_interactividad.grid_forget()

             # Mostramos los atributos de los libros cientificos
            self._etiq_campo_estudio.grid(row=4, column=0, padx=5, pady=5)
            self._entry_campo_estudio.grid(row=4, column=1, padx=5, pady=5)

            self._etiq_nivel_dificultad.grid(row=5, column=0, padx=5, pady=5)
            self._entry_nivel_dificultad.grid(row=5, column=1, padx=5, pady=5)
            return
```

9.2.9. Botones de checkbox

Los botones *checkbox* permiten realizar selecciones principalmente cuando son valores booleanos.
Los argumentos de entrada para estos botones son:

- Nombre de la ventana a la que se asocia el widget.
- Texto que va a acompañar al botón (`text="Texto que acompaña al botón"`)
- Nombre de la variable `BooleanVar` asociada al *checkbox* (`variable = "nombre de la variable"`). La variable debe estar declarada previamente.

En el siguiente código, vamos a transformar los dos campos que son booleanos en los libros infantiles para utilizar este tipo de botones: Si es `interactivo` y si contiene `ilustraciones`:

```
class AltaLibroGUI:
    def __init__ (self, master):
        [...]
        self._etiq_ilustraciones = tk.Label(self._master, text="¿Contiene ilustraciones?")
        self._entry_ilustraciones = tk.Checkbutton(self._master, text="", variable=self._ilustraciones)

        self._etiq_interactividad = tk.Label(self._master, text="¿Es interactivo?")
        self._entry_interactividad = tk.Checkbutton(self._master, text="", variable=self._interactividad)
        [...]
```

Tras todos estos pasos, ya hemos completado la parte visual de la GUI. En la figura Figura 9.9 Podemos ver el estado de la interfaz nada más iniciarse el programa, mostrando únicamente los campos comunes a todos los tipos de libros. Cuando pulsamos en la opción de libro infantil (ver Figura 9.10), se añaden los atributos propios de esta clase de libros, mientras que si pulsamos en la opción de libro científico los atributos de libro infantil se ocultan y se muestran únicamente los atributos de los libros científicos (ver Figura 9.11).

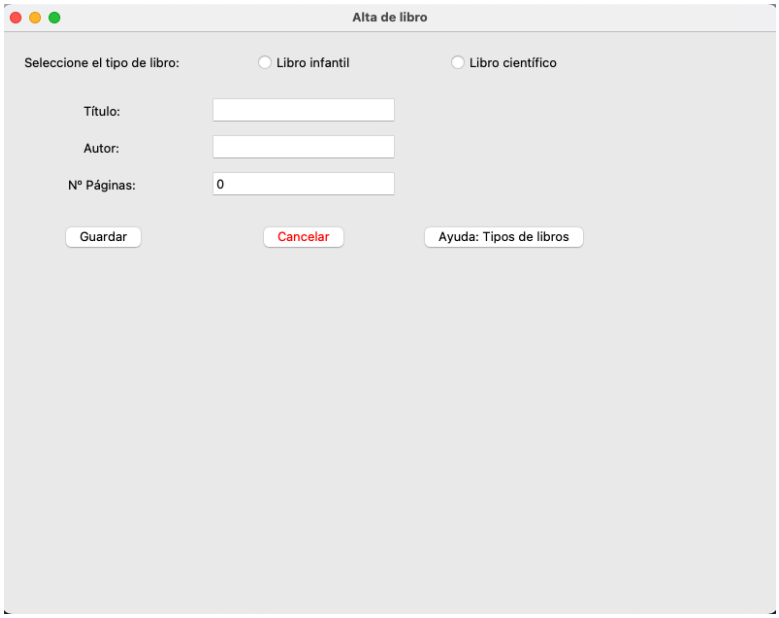


Figura 9.9: Campos generales para todos los tipos de libros.

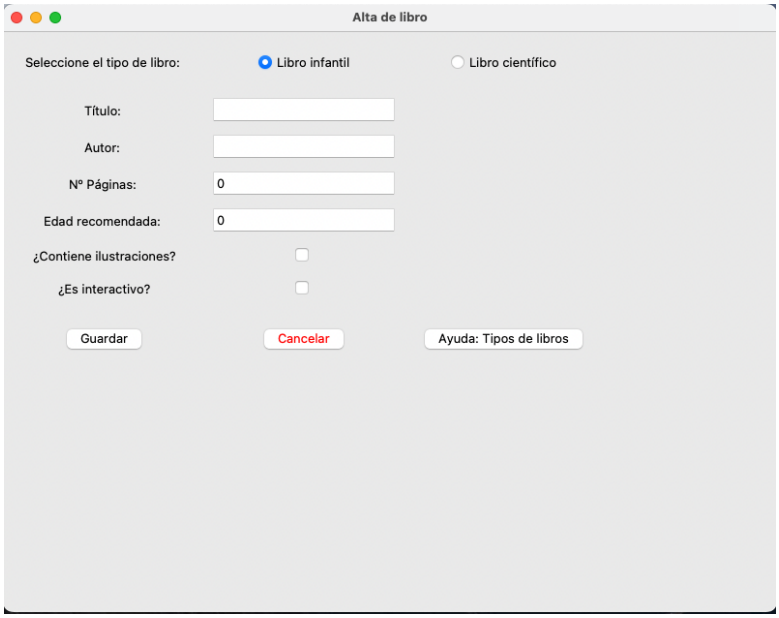


Figura 9.10: Campos mostrados al elegir un libro infantil.

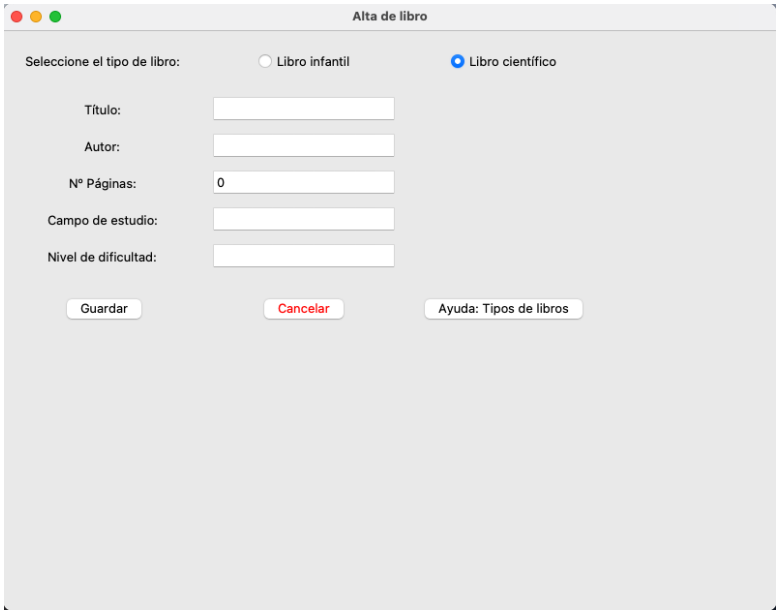


Figura 9.11: Campos mostrados al elegir un libro científico.

9.2.10. Implementación para crear el libro

Nos quedaba darle la funcionalidad necesaria al botón guardar. Para ello, vamos a realizar estos pasos adicionales:

1. Crear una variable de instancia booleana llamada `_guardado_libro` que indique si el libro ha sido guardado.
2. Crear un método de instancia llamado `_guardar_libro()` que cambiará el valor de esa variable a `True` y destruirá la ventana.
3. Implementar un método público de instancia llamado `obtiene_libro()`, que devolverá un objeto de tipo `Libro`; si se ha creado un `LibroInfantil`, será de este tipo, y en caso de que se cree un `LibroCientifico`, será de este otro tipo.

```
class AltaLibroGUI:
    def __init__ (self, master):

        # Atributo que indica si se han guardado los datos del libro:
        self._guardado_libro: bool = False

        [...]
        # Botones
        self._boton_guardar = tk.Button(self._master,
                                        text='Guardar',
                                        command=self._guardar_libro)
        [...]

    def _guardar_libro(self):
        self._guardado_libro=True
        self._master.destroy()

    def libro_guardado(self)->bool:
        return self._guardado_libro

    def obtiene_libro(self)->Libro:

        if self._tipo_libro.get()==1:    # Libro infantil
            libro_nuevo = LibroInfantil(self._titulo.get(),
                                        self._autor.get(),
                                        self._paginas.get(),
                                        self._edad_recomendada.get(),
                                        self._ilustraciones.get(),
                                        self._interactividad.get())
            return libro_nuevo


        elif self._tipo_libro.get()==2:  # Libro cientifico
            libro_nuevo = LibroCientifico(self._titulo.get(),
                                          self._autor.get(),
                                          self._paginas.get(),
                                          self._campo_estudio.get(),
                                          self._nivel_dificultad.get())
```

```
            return libro_nuevo
```

En el módulo `main.py`, el código para la creación de la interfaz gráfica y obtener desde ella el nuevo libro, podría ser algo como lo siguiente:

```
from app.gui.alta_libro_gui import AltaLibroGUI
from app.tdas.libros.libroCientifico import LibroCientifico
from app.tdas.libros.libroInfantil import LibroInfantil
from app.tdas.libros.libro import Libro
import tkinter as tk


# Interfaz gráfica:
# libro_creado_gui: Libro = Libro()
root = tk.Tk()
alta_libros_gui = AltaLibroGUI(root)
alta_libros_gui.muestra_ventana()
if alta_libros_gui.libro_guardado():
    nuevo_libro = alta_libros_gui.obtiene_libro()
    print ("SE HA DADO DE ALTA UN NUEVO LIBRO: \n")
    print (nuevo_libro)
```

9.3 Relación de ejercicios

1. **Interfaz gráfica para triángulos**. Crear una clase que represente una interfaz gráfica (GUI) que se lance desde vuestro módulo main.py, y abra una ventana que permita crear instancias de alta usando los atributos básicos de los triángulos equiláteros, escalenos e isósceles. Entendemos por atributos básicos los mismos que se utilizaron para los ejercicios de entrada / salida.
2. **Interfaz gráfica para círculos y elipses**. De manera análoga al ejercicio anterior, crear una interfaz gráfica que permita dar de alta los atributos básicos de los círculos y elipses.
3. **Grabar en JSON desde la GUI**. Añadir a la interfaz gráfica para triángulos, un botón que permita guardar directamente la instancia que se está editando en un fichero JSON en el sistema de ficheros.
4. **Obtener datos desde fichero en la GUI**. Añadir a la interfaz gráfica para triángulos, un botón que permita leer un fichero JSON, que contiene los datos básicos de ese triángulo, y lo muestre en pantalla.

UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 10 de Prácticas

TDA sin relación: Correo

Autores: Alejandro Buitrago López (alejandro.buitragol@um.es), Sergio López Bernal (slopez@um.es), Javier Pastor Galindo (javierpg@um.es), Jesús Damián Jiménez Re (jesusjr@um.es) y Gregorio Martínez Pérez (gregorio@um.es)

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2024-2025

Última modificación:
11 de noviembre de 2024

Sesión 10: TDA sin relación. Correo

Índice

9.1 TDA Correo

El objetivo en esta práctica es elaborar un TDA que represente un correo electrónico básico. Trabajaréis en los aspectos de especificación, representación e implementación de un TDA, para comprender cómo interactuar con los datos de un correo de manera abstracta, sin preocuparnos por los detalles internos considerados para su implementación.

A continuación se detalla una especificación textual del TDA y los tres aspectos principales a cubrir, ofreciendo como ejemplo la resolución asociada al TDA Fecha. Estas instrucciones serán comunes a las siguientes sesiones de prácticas, donde se describirán TDAs de diferentes tipos que los alumnos deberán resolver y entregar como parte del **Proyecto 2 de la asignatura**.

9.2 Descripción textual del TDA Correo

Un correo electrónico es una elemento digital que consta de varios elementos: un remitente, es decir, la dirección de correo electrónico de quien envía el mensaje; un destinatario, que es la dirección de correo a la que se envía el mensaje (en esta práctica solo consideraremos un único destinatario); un breve asunto, que resume el contenido del mensaje; el cuerpo del mensaje, que incluye el contenido principal que se desea comunicar; una fecha y hora de envío o recepción; un estado de lectura, que indica si el correo ha sido leído o no; y un estado de borrador, que indica si el correo aún no ha sido enviado. Además, un correo electrónico contendrá un identificador único que permitirá diferenciarlo de otros correos.

Además de estas características, el correo debe poder realizar una serie de operaciones para interactuar con sus datos de manera controlada. Entre estas operaciones se incluyen cambiar el estado de lectura para marcarlo como leído o no leído. Además, un correo debe permitir actualizar los datos del destinatario, remitente, asunto y cuerpo del correo, pero únicamente cuando el correo sea un borrador. Una vez enviado el correo ya no podrá editarse. También debe ser posible buscar una palabra específica dentro del correo, independientemente de si es en el asunto o en el cuerpo del mensaje, y determinar si dicha palabra se encuentra escrita en el correo.

A partir de esta descripción se debe diseñar el TDA Correo para que tenga las operaciones y la representación acorde a la descripción anterior.

9.2.1. Especificación del TDA Correo

En relación a la especificación, documentaremos los tres apartados principales que la componen: 1) nombre y descripción, 2) especificación de los datos, y 3) especificación de las operaciones. Para ello, seguiremos una estructura similar a la indicada en el siguiente ejemplo, donde ilustramos la especificación del **TDA Fecha**:

- **Descripción**: Representa fechas válidas de acuerdo al calendario Gregoriano.
- **Usa**: enteros (**N**), booleanos (**B**).
- **Operaciones**:
  - **Fecha**(*d*: Entero, *m*: Entero, *a*: Entero) : Fecha

- **Descripción**: Crea una instancia de Fecha con el día, mes y año especificados, validando que sea una fecha válida en el calendario Gregoriano.
- **Precondiciones**: Los valores de día, mes y año deben formar una fecha válida: $a \neq 0$, $1 \leq m \leq 12$ y $1 \leq d \leq 31$
- **Retorna**: Fecha (una nueva instancia de Fecha).
- **Excepciones**: Fecha es no válida.

- **anterior**(*f*: Fecha) : Fecha
  - **Descripción**: Devuelve la fecha anterior a la fecha especificada.
  - **Precondiciones**: La fecha *f* debe ser válida.
  - **Retorna**: Fecha (la fecha anterior a *f*).
  - **Excepciones**: Fecha es nula o no válida.
- **siguiente**(*f*: Fecha) : Fecha
  - **Descripción**: Devuelve la fecha siguiente a la fecha especificada.
  - **Precondiciones**: La fecha *f* debe ser válida.
  - **Retorna**: Fecha (la fecha posterior a *f*).
  - **Excepciones**: Fecha es nula o no válida.
- **dia**(*f*: Fecha) : Entero
  - **Descripción**: Devuelve el día del mes de la fecha especificada.
  - **Precondiciones**: La fecha *f* debe ser válida.
  - **Retorna**: Entero (el día del mes de la fecha *f*).
  - **Excepciones**: Fecha es nula o no válida.
- **mes**(*f*: Fecha) : Entero
  - **Descripción**: Devuelve el mes de la fecha especificada.
  - **Precondiciones**: La fecha *f* debe ser válida.
  - **Retorna**: Entero (el mes de la fecha *f*).
  - **Excepciones**: Fecha es nula o no válida.
- **año**(*f*: Fecha) : Entero
  - **Descripción**: Devuelve el año de la fecha especificada.
  - **Precondiciones**: La fecha *f* debe ser válida.
  - **Retorna**: Entero (el año de la fecha *f*).
  - **Excepciones**: Fecha es nula o no válida.
- **es_bisiesto**(*f*: Fecha) : Booleano
  - **Descripción**: Indica si el año de la fecha especificada es un año bisiesto.
  - **Precondiciones**: La fecha *f* debe ser válida.
  - **Retorna**: Booleano (verdadero si el año es bisiesto, falso en caso contrario).
  - **Excepciones**: Fecha es nula o no válida.
- **comparar**(*f1*: Fecha, *f2*: Fecha) : Entero
  - **Descripción**: Compara dos fechas. Retorna un valor negativo si *f1* es anterior a *f2*, cero si son iguales, y un valor positivo si *f1* es posterior a *f2*.
  - **Precondiciones**: Las fechas *f1* y *f2* deben ser válidas.
  - **Retorna**: Entero (un valor de comparación entre *f1* y *f2*).
  - **Excepciones**: Fecha es nula o no válida.

Observa que la especificación es independiente del lenguaje de programación y, por tanto, no se usan tipos de datos concretos.

9.2.2. Representación del TDA Correo

Tras definir la especificación del TDA, es necesario indicar su representación, que constará de tres apartados esenciales: 1) una estructura que indique los campos que representa el TDA, 2) La función de abstracción, y 3) El invariante de la representación. Veamos, como ejemplo, cómo sería la representación del TDA Fecha:

- **Estructura**:

```
struct rep {
    entero dia
    entero mes
    entero año
}
```

- **Función de abstracción**:
  Abst(r)=Fecha(r.dia,r.mes,r.año)

- **Invariante de la representación**:

$$I(r) = \begin{cases} 1 \leq r.\text{mes} \leq 12 \\ r.\text{mes} \in \{1,3,5,7,8,10,12\} \Rightarrow 1 \leq r.\text{dia} \leq 31 \\ r.\text{mes} \in \{4,6,9,11\} \Rightarrow 1 \leq r.\text{dia} \leq 30 \\ r.\text{mes} = 2 \Rightarrow \begin{cases} 1 \leq r.\text{dia} \leq 29, & \text{si bisiesto}(r.\text{año}) \\ 1 \leq r.\text{dia} \leq 28, & \text{si no es bisiesto}(r.\text{año}) \end{cases} \end{cases}$$

Nota importante: en la entrega del **Proyecto 2**, solo se deberá entregar la estructura definida, pero no la función de abstracción o el invariante de la representación.

9.2.3. Implementación del TDA Correo

Se debe implementar el TDA propuesto haciendo uso del lenguaje de programación Python, aplicando los principios de encapsulamiento y ocultación. Además, se debe garantizar que los métodos respeten las restricciones impuestas en la especificación del TDA. Para ello, programa en Python lo siguiente:

- **Implementación del TDA Correo** (parte privada): Código fuente de la clase o clases usadas para implementar la solución del TDA.
- **Módulo main** (parte pública): Este main debe contener un ejemplo de llamada a los métodos del TDA, ilustrando el funcionamiento del TDA y, por tanto, cómo un cliente final haría uso de las operaciones proporcionadas por dicho TDA.

UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 11 de Prácticas

TDA sin orden: Gestor de correo

Autores: Alejandro Buitrago López (alejandro.buitragol@um.es), Sergio López Bernal (slopez@um.es), Javier Pastor Galindo (javierpg@um.es), Jesús Damián Jiménez Re (jesusjr@um.es) y Gregorio Martínez Pérez (gregorio@um.es)

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2024-2025

Última modificación:
18 de noviembre de 2024

Sesión 11: TDA sin orden. Gestor de correo

Índice

9.1 TDA Gestor de Correo

El objetivo en esta práctica es elaborar un TDA que represente un gestor de correo básico. Trabajaréis en los aspectos de especificación, representación e implementación del TDA, para comprender cómo gestionar correos electrónicos de manera abstracta, sin preocuparnos por los detalles internos considerados para su implementación.

A continuación se detalla una especificación textual del TDA. Es importante destacar que para la realización de la especificación y representación del TDA Gestor de Correo podremos basarnos en las consideraciones indicadas en el guion de prácticas de la sesión anterior. Finalmente, es importante destacar que la realización del TDA Gestor de Correo forma parte de la entrega del **Proyecto 2 de la asignatura**.

Importante: En caso de definir TDAs auxiliares que sirvan como apoyo al gestor de correo, será imprescindible **definir su representación**.

9.2 Descripción textual del TDA Gestor de Correo

Un gestor de correo es una entidad digital que permite gestionar de forma eficiente los correos electrónicos de un usuario, brindando funcionalidades que van desde la redacción hasta la organización, envío y recepción de correos. Cada gestor de correo está asociado a un único usuario, cuya identidad se maneja mediante un nombre de usuario (dirección de correo electrónico) y una contraseña.

El gestor cuenta con varias bandejas para clasificar y organizar los correos según su estado. Estas bandejas son:

- **Bandeja de entrada**: almacena los correos recibidos por el usuario.
- **Bandeja de salida**: contiene los correos enviados.
- **Bandeja de borradores**: almacena los correos en estado de borrador, permitiendo que el usuario los edite hasta que decida enviarlos.
- **Bandeja de spam**: contiene los correos identificados como no deseados o sospechosos, en función de ciertos criterios de spam configurados.

Cada bandeja permite almacenar múltiples correos electrónicos, teniendo en cuenta que el orden de cómo se almacenan estos correos no es relevante en el contexto de este TDA. Así, el gestor de correo debe poder acceder a la colección de correos asociada a cada bandeja, así como el número de correos contenida en ella en un momento dado.

El gestor de correos contendrá, además, criterios de spam, que consisten en una colección de palabras clave que, al detectarse en el asunto o cuerpo de un correo entrante, provocan que este se añada automáticamente en la bandeja de spam, sin pasar por la bandeja de entrada. Esto permite filtrar de manera automática los correos no deseados en el momento de su recepción.

Las principales funcionalidades que ofrece un gestor de correo para interactuar con la información del usuario y los correos electrónicos son:

- **Cambiar contraseña**: el usuario podrá cambiar libremente la contraseña de su cuenta de correo.
- **Gestión de criterios de spam**: el usuario podrá añadir y eliminar criterios de spam, que servirán para clasificar de forma automática si un correo entrante debe ir a la bandeja de entrada o a la de spam. Esos criterios corresponden a un listado de palabras sin relación de orden.

- **Redactar un correo**: el gestor permite redactar correos, que se almacenan siempre inicialmente en la bandeja de borradores, pudiendo ser modificados libremente por el usuario. El gestor permitirá cambiar la dirección de correo electrónico del destinatario, el contenido del asunto, o el cuerpo del correo. Sin embargo, estas modificaciones solo podrán realizarse sobre aquellos correos en la bandeja de borradores.
- **Enviar y recibir un correo**: Una vez que el usuario del gestor de correo (lo denotaremos en este guión como *gestor emisor* por brevedad) decide que un correo de la bandeja de borradores no necesita más modificaciones, podrá enviar dicho correo al gestor de correo de otro usuario (gestor receptor). El envío de un correo consta de los siguientes pasos:
  1. Acciones en el gestor emisor: el correo seleccionado en el gestor de correo emisor, a partir de su identificador único de correo, pasará de la bandeja de borradores a la bandeja de enviados, cambiando el estado del correo, de borrador a enviado.
  2. Acciones en el gestor receptor: el gestor emisor, al enviar el correo, deberá invocar a la funcionalidad de recepción de correos en el gestor receptor, quien añadirá el correo recibido al buzón de correo correspondiente. Así, si se detecta que el asunto o el mensaje del correo recibido contiene entre sus palabras al menos uno de los criterios de spam definidos en el gestor receptor, este correo electrónico se almacenará en la bandeja de spam. En caso contrario, se almacenará en la bandeja de entrada.
- **Cambiar el estado de lectura de un correo**: el usuario puede marcar un correo como leído o como no leído, pero únicamente en las bandejas de entrada y spam.
- **Eliminar un correo**: el gestor permite eliminar un correo electrónico de una bandeja de correo determinada, en base a su identificador único de correo. Se pueden eliminar correos de todas las bandejas.
- **Búsqueda de correos**: el gestor de correo permite obtener, en formato textual, la colección de correos que contienen una palabra o frase. Esta búsqueda se realizará en la totalidad del contenido del correo, esto es, tanto en su asunto como en el cuerpo del mensaje. Así, el gestor permitirá, de forma diferenciada, realizar la búsqueda por palabras clave tanto de forma individual por cada bandeja de correo, como sobre el total de las cuatro bandejas.

En base a todo lo anterior, provee la especificación, representación e implementación del TDA Gestor de Correo. De cara a la implementación, recuerda seguir las buenas prácticas de POO, en relación al uso de clases, delegación, etc. Además, **las colecciones que usemos en la implementación de este TDA deberán estar basadas en una estructura enlazada**.

9.3 Cómo representar colecciones de datos

Por último, presentamos un resumen de los aspectos más destacables en relación a cómo podemos trabajar de forma abstracta con colecciones de datos, en caso de que un TDA requiera almacenar una colección de elementos como parte de sus propiedades. Estos aspectos han sido ampliamente discutidos en teoría, por lo que se recomienda repasar el temario teórico antes de realizar la sesión.

- **Estructuras enlazadas**:
  - Consisten en un **conjunto de nodos enlazados mediante referencias a otros nodos que nosotros debemos implementar**.
  - Estos nodos serán representadas como **instancias individuales** que mantendrán referencia a otros elementos con los que guardan un cierto tipo de relación.
  - En este tipo de representación **no podemos hacer uso de tipos de datos ya definidos en Python**, como `list`, `set`, o `map`, entre otros, sino que es la propia estructura de nodos enlazada la que nos permite crear manualmente la colección de elementos a relacionar.
- **Estructuras contiguas**:
  - Representan **espacio en memoria reservado de forma contigua, en el que podemos almacenar elementos**.
  - Por simplicidad, en este caso **sí podremos hacer uso del tipo de datos `list` de Python** como medio para almacenar los elementos secuencialmente. Sin embargo, **no se permite el uso de los métodos que proporciona Python para trabajar con listas**. Por ejemplo, *append*, *insert*, *remove*, etc
  - En base a lo anterior, **trabajaremos las listas de manera manual**, de tal forma que la inserción supondrá crear una lista con un tamaño superior y gestionar de forma adecuada el contenido, la eliminación supondrá crear una nueva lista con un tamaño menor, y así sucesivamente. Esto es, de forma similar a cómo gestionábamos los arrays en Processing en la asignatura de *Fundamentos de programación*.

UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 12 de Prácticas

TDA con orden: DataFrame

Autores: Alejandro Buitrago López (alejandro.buitragol@um.es), Sergio López Bernal (slopez@um.es), Javier Pastor Galindo (javierpg@um.es), Jesús Damián Jiménez Re (jesusjr@um.es) y Gregorio Martínez Pérez (gregorio@um.es)

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2024-2025

Última modificación:
2 de diciembre de 2024

Sesión 12: TDA con orden. DataFrame

Índice

9.1 TDA DataFrame

El objetivo en esta práctica es elaborar un TDA que represente un DataFrame. Trabajaréis en los aspectos de especificación, representación e implementación del TDA, para comprender cómo gestionar información contenida en esta estructura de manera abstracta, sin preocuparnos por los detalles internos considerados para su implementación.

A continuación se detalla una especificación textual del TDA. Es importante destacar que para la realización de la especificación y representación del TDA DataFrame podremos basarnos en las consideraciones indicadas en el guion de prácticas de la sesión 10 de prácticas. Finalmente, es importante destacar que la realización del TDA DataFrame forma parte de la entrega del **Proyecto 2 de la asignatura**.

Importante: En caso de definir TDAs auxiliares que sirvan como apoyo al DataFrame, será imprescindible **definir su representación**.

9.2 Descripción textual del TDA DataFrame

Un DataFrame es una estructura de datos tabular, ampliamente utilizada para gestionar y analizar datos en diferentes contextos, como bases de datos, procesamiento de información y análisis estadístico. Además, los DataFrames son ampliamente usados para gestionar la información usada por modelos de inteligencia artificial para aprender. Esta estructura se organiza en filas y columnas, donde cada columna tiene un nombre único, y cada fila puede contener datos de diferentes tipos.

Un DataFrame contiene varias características clave que lo hacen versátil para manipulación de datos:

- **Columnas**: Cada columna tiene un nombre único, que permite referenciarla directamente para consultarla o modificarla. Esto es, acceder a la columna a partir de su nombre. Todas las columnas deben tener la misma longitud, correspondiente al número de filas del DataFrame.
- **Filas**: Las filas contienen los datos del DataFrame y son organizadas por posición. Dicha posición corresponde a un índice secuencial que comienza siempre en cero. Cada fila puede ser accedida o manipulada mediante su índice correspondiente.
- **Valores nulos**: Las celdas del DataFrame pueden contener valores nulos (`None`), correspondiente a información faltante, algo muy común en entornos reales. Así, un DataFrame ofrece funcionalidades para trabajar eficientemente con estos valores.

La Figura 9.1 muestra una representación visual de un DataFrame. En este ejemplo, los valores nulos en las celdas se representan como `NaN` (Not a Number). Sin embargo, a nivel de implementación nosotros los consideraremos como `None`.

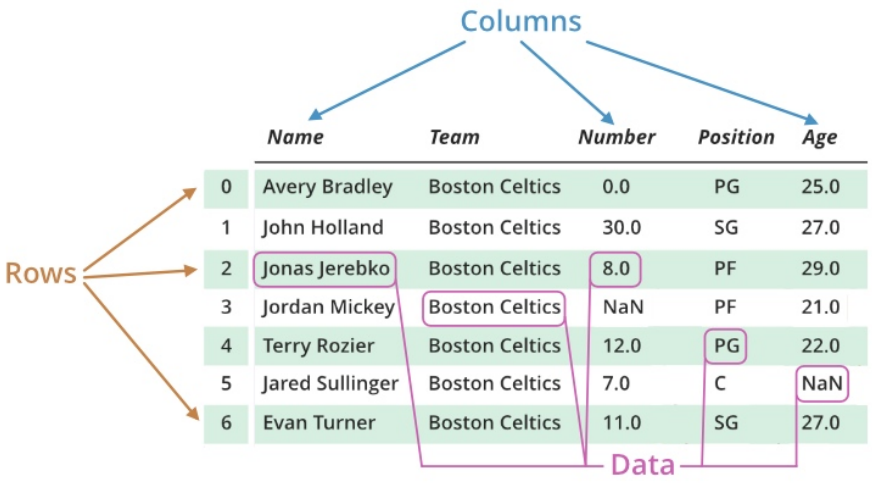


Figura 9.1: Ejemplo de contenido de un DataFrame, en el que se puede ver una división de los datos en filas y columnas.

Las principales funcionalidades que ofrece el TDA DataFrame son las siguientes:

- **Inserción y eliminación de contenido**:
  - Insertar una nueva columna, pudiendo proporcionar una colección de valores específicos. En caso de no suministrarlos, se almacenarán valores nulos, correspondiendo a una ausencia de datos para esas celdas.
  - Insertar una fila, indicando la posición correspondiente sobre la que insertar la nueva fila.
  - Eliminar una columna a partir de un nombre de columna.
  - Eliminar una fila indicando la posición a eliminar.
- **Exploración y análisis**:
  - Verificar si una columna existe por su nombre.
  - Obtener las primeras (`head`) o las últimas (`tail`) `n` filas del DataFrame.
  - Determinar la forma (`shape`) del DataFrame (número de filas y columnas).
  - Calcular el valor máximo de una columna.
  - Calcular el valor medio de una columna.
- **Análisis de datos**:
  - Identificar y visualizar el conjunto de filas que contienen, al menos, una celda con valor nulo. Deberá mostrarse también el índice de la fila para poder localizar fácilmente dónde hay una ausencia de datos.
  - Reemplazar todos los valores nulos del DataFrame por un valor dado. Esto permite sustituir la ausencia de datos por un valor por defecto que sí puede ser interpretable para el cliente del TDA.
- **Visualización del DataFrame en formato textual**:
  - Visualizar un rango de filas, comprendidas entre dos índices.
  - Mostrar el contenido de una fila específica, en base a su índice.
  - Mostrar los valores de una columna específica, referenciada por su nombre. Los elementos se mostrarán como un listado (todos en una única línea de la terminal).

    Para la visualización de filas, sea una única o un conjunto dentro de un rango, mostraremos en primer lugar los nombres de las columnas en una primera línea. Tras ello, mostraremos todas las filas que sea necesario visualizar, una debajo de la anterior. Esto es, una visualización tabular como la ejemplificada en la Figura 9.1.

En base a todo lo anterior, provee la especificación, representación e implementación del TDA DataFrame. De cara a la implementación, recuerda seguir las buenas prácticas de POO. Además, **las colecciones que usemos en la implementación de este TDA deberán estar basadas en una representación contigua**.

9.3 Notas importantes

Se recomienda el uso del entorno de depuración (debugger) de PyCharm para ir siguiendo paso a paso las actualizaciones que hace nuestro programa sobre el DataFrame y así poder identificar rápidamente los errores que se haya cometido en la implementación.

9.4 Cómo representar colecciones de datos

Por último, presentamos un resumen de los aspectos más destacables en relación a cómo podemos trabajar de forma abstracta con colecciones de datos, en caso de que un TDA requiera almacenar una colección de elementos como parte de sus propiedades. Estos aspectos han sido ampliamente discutidos en teoría, por lo que se recomienda repasar el temario teórico antes de realizar la sesión.

- **Estructuras enlazadas**:
  - Consisten en un **conjunto de nodos enlazados mediante referencias a otros nodos que nosotros debemos implementar**.
  - Estos nodos serán representadas como **instancias individuales** que mantendrán referencia a otros elementos con los que guardan un cierto tipo de relación.
  - En este tipo de representación **no podemos hacer uso de tipos de datos ya definidos en Python**, como `list`, `set`, o `map`, entre otros, sino que es la propia estructura de nodos enlazada la que nos permite crear manualmente la colección de elementos a relacionar.
- **Estructuras contiguas**:
  - Representan **espacio en memoria reservado de forma contigua, en el que podemos almacenar elementos**.
  - Por simplicidad, en este caso **sí podremos hacer uso del tipo de datos `list` de Python** como medio para almacenar los elementos secuencialmente. Sin embargo, **no se permite el uso de los métodos que proporciona Python para trabajar con listas**. Por ejemplo, *append*, *insert*, *remove*, etc
  - En base a lo anterior, **trabajaremos las listas de manera manual**, de tal forma que la inserción supondrá crear una lista con un tamaño superior y gestionar de forma adecuada el contenido, la eliminación supondrá crear una nueva lista con un tamaño menor, y así sucesivamente. Esto es, de forma similar a cómo gestionábamos los arrays en Processing en la asignatura de *Fundamentos de programación*.

Proyecto 1

Programación Orientada a Objetos en Python

Convocatoria de enero de 2025

Curso 2024-25

Contenido

1. Introducción

En este documento se describe el **Proyecto 1** (Proyecto software de Programación Orientada a Objetos en Python) de la asignatura **Tecnología de la Programación** de segundo curso del Grado en Matemáticas para la convocatoria de enero del curso 2024-2025.

Todo lo explicado en la parte de POO de la asignatura es susceptible de ser usado para resolver este enunciado, donde el diseño seguirá las buenas prácticas de programación, así como el criterio de la pareja.

Este documento incluye también las normas de realización, entrega y defensa, así como los criterios generales de evaluación que se utilizarán para corregir el trabajo entregado.

2. Enunciado del proyecto

2.1. UMUTickets

En este proyecto vamos a implementar en Python una versión sencilla de la aplicación **UMUTickets**, que permitirá que i) ciertas entidades organicen y oferten diferentes tipos de eventos, tales como deportivos o de ocio, y ii) los clientes podrán reservar y comprar entradas para los mismos.

Todos los elementos del sistema se modelarán con clases, que estarán relacionados entre sí mediante relaciones de uso, asociación, agregación, composición, herencia e interfaces.

En las siguientes subsecciones se comentan los detalles de las funcionalidades mínimas. Las clases pueden tener los atributos que se consideren necesarios para que la lógica del programa funcione correctamente.

2.2. Tipos de clientes y condición de socio

UMUTickets contará con un conjunto de **clientes**. Cada cliente tendrá un identificador de cliente que debería ser único.

Los clientes podrán ser **particulares** o **empresariales**.

- De los clientes **particulares** será necesario almacenar la **persona** asociada (nombre, apellidos, dirección postal y DNI). Adicionalmente, se deberá almacenar información sobre los datos de una tarjeta de crédito para poder realizar las reservas y las posteriores compras.
- De los clientes **empresariales**, es necesario guardar información sobre la **empresa** asociada (razón social, su CIF, su teléfono y su dirección postal) y la **persona** de contacto principal (nombre, apellidos, dirección postal y DNI), así como una tarjeta de crédito de manera análoga a los clientes particulares.

Un cliente puede ser socio de UMUTickets, lo que le ofrece ciertas ventajas:

- Dispondrán de un número de socio (que debería ser único y diferente del número de cliente).
- Dispondrán de una tarjeta de socio con un saldo económico asociado. El sistema deberá ofrecer la funcionalidad necesaria para realizar recargas de cantidades de

5, 10, 20 o 50 euros en el saldo de la tarjeta de socio, y que se harán desde la tarjeta de crédito asociada al cliente.

Los socios podrán renovar su condición de socio de manera mensual, trimestral o anual, y dependiendo de la periodicidad, tiene un coste diferente:

- Las renovaciones mensuales tienen un coste de 5 euros al mes.
- Las renovaciones trimestrales tienen un coste de 20 euros al trimestre.
- Las renovaciones anuales tienen un coste de 50 euros anuales.

En cualquier caso, las renovaciones se realizarán siempre con cargo a su tarjeta de socio, por lo que para realizarla debe comprobarse que existe suficiente saldo. Para cada renovación realizada por un socio se debería registrar la fecha de inicio de la renovación y la fecha de expiración prevista.

Además de poder hacerse socio, un cliente puede gestionar reservas y comprar entradas para los diferentes tipos de eventos ofertados.

2.3. Tipos de eventos y organización de eventos

UMUTickets gestionará **eventos.** Todos los tipos de eventos comparten un conjunto de características comunes:

- Un identificador único del evento.
- Un nombre.
- Una descripción del evento.
- Algunos eventos tendrán una fecha única del evento mientras que otros podrán tener un periodo de duración (una fecha de inicio y una fecha de fin).
- Dirección del evento.
- Una URL/link con la ubicación exacta del evento.
- Número de entradas disponibles.
- Precio original de la entrada al evento (sin incluir el I.V.A.)
- Organizador (que será una empresa organizadora o UMUTickets).

Sin embargo, UMUTickets no debe permitir crear eventos generales, pues sólo existen eventos específicos que calculan el precio final de sus reservas/entradas en función de criterios diferentes (explicados en el siguiente apartado 2.4). Inicialmente están previstos los siguientes tres **tipos de eventos** (aunque se añadirán más en el futuro), y que contendrán la siguiente información:

- **Deportivos**. Incluirán el deporte del que se trata, que en principio solo podrá ser de Fútbol, Tenis o Baloncesto.
- **Espectáculo audiovisual**. Incluirá el nombre del artista o grupo que realiza el espectáculo, así como una edad mínima recomendada para el mismo.
- **Feria empresarial**. Este evento solo estará dirigido a clientes que son empresas, por lo que deberá de tenerse en cuenta que solo este tipo de clientes podrán realizar reservas o comprar entradas. Además, en este tipo de evento el número máximo de entradas que el evento puede ofertar es de 100.

Los eventos podrán ser organizados por i) el elemento central **UMUTickets** o bien ii) por **empresas organizadoras**. Este tipo de empresas que son organizadoras de eventos,

además de la razón social, CIF, teléfono y dirección postal, tendrán un listado de sus eventos organizados, así como un identificador de organizador que es único en el sistema.

En particular, la funcionalidad de gestionar eventos permite **crear eventos, modificarlos y anularlos**. UMUTicket tiene privilegios para gestionar cualquier evento existente en el sistema, mientras que una empresa organizadora sólo puede gestionar sus propios eventos. En un futuro, se espera que más elementos puedan gestionar eventos a su manera.

2.4. Reservas y ventas de entradas

La gestión de las reservas y venta de entradas se realizará a través de UMTickets. El sistema ofrecerá todos los métodos necesarios para realizar la reserva y anulación, así como la venta de las entradas (con o sin reserva previa).

En particular, los clientes de UMUTickets pueden realizar **reservas** en los eventos ofrecidos, siempre y cuando sea anterior a 7 días previos a la fecha de inicio del evento y quede disponibilidad de entradas. La reserva tendrá una fecha y el precio de la entrada reservada, que dependerá de si el cliente es socio o no y del tipo de evento acorde a las siguientes condiciones:

- Todas las reservas, por el hecho de hacerse con una semana de antelación, tienen de base un 10% de descuento sobre el precio de la entrada al evento.
- En los **eventos que son ferias empresariales**, los clientes (empresariales) que son socios tendrán un descuento del 30% extra.
- Respecto a los **eventos deportivos y espectáculos audiovisuales**, los clientes que son socios tendrán un descuento extra del 20%. Sin embargo, el precio final de la reserva se incrementará si quedan pocas entradas:
    - **Eventos deportivos**: cuando quede un 25% del número máximo de entradas disponibles, el precio de la reserva resultante aumenta un 10% automáticamente.
    - **Espectáculos audiovisuales**: cuando quede un 10 % del número máximo de entradas disponibles, el precio de la reserva aumenta un 30% automáticamente.

Al realizar una reserva, se puede pagar directamente la entrada o bien más adelante. En cualquiera de los casos, deberá respetarse el precio de venta reflejado en la reserva. También existe la posibilidad de que un cliente **anule una reserva**, pero siempre y cuando no se haya realizado el pago en firme de la entrada. Además, como los dueños de UMUTickets no tienen claro si los eventos de feria empresarial son rentables para el año que viene, cada anulación de un evento de este tipo debería de anotarse, de cara a tener el **número total de anulaciones** de eventos de feria empresarial, y tomar decisiones para el año próximo.

Por otro lado, se puede realizar una compra directamente de una entrada sin reserva (si hay disponibilidad y se realiza antes del inicio del evento). El precio de venta no tendrá el descuento de la reserva, pero cumplirá las condiciones para socios y no socios, así como las reglas de incremento. Lógicamente, la venta de una entrada solo puede realizarse hasta el día de inicio del evento.

Cuando se vende una entrada (bien reservada o de manera directa), se registrará que se ha realizado la venta, la fecha de la venta, el número de entrada oficial, y el cliente al que se le ha realizado.

2.5. Gestión y estadísticas de UMUTickets

De cara al control interno, el sistema ofrecerá métodos que permitan:

1. Obtener el **beneficio** que ha obtenido UMUTickets de un evento particular, teniendo en cuenta que:
    - En los eventos deportivos, por cada entrada vendida a clientes no socios, obtiene un beneficio de un 10% del coste de dicha entrada; por cada entrada vendida a clientes socios, obtiene un beneficio de un 5% del coste de dicha entrada.
    - En los eventos de espectáculo audiovisual, por cada entrada vendida a clientes no socios, obtiene un beneficio de un 15 % del coste de la entrada; por cada entrada vendida a clientes socios, obtiene un beneficio de un 10 % del coste de la entrada.
    - En las ferias empresariales, por cada entrada vendida a clientes empresariales no socios, se obtiene un 10% del coste de esta, mientras que la vendida a un socio genera un 5% de beneficio con respecto a su coste.
2. Obtener el **número de socios actuales**, y los **ingresos totales** que está recibiendo UMUTickets con esa cantidad de socios.
3. Obtener el **porcentaje medio de anulaciones** sobre las reservas realizadas en las ferias empresariales.

2.6. Interfaz gráfica del sistema y simulación

En este proyecto, y como versión inicial, será necesario crear una interfaz gráfica para el sistema, que permita la siguiente funcionalidad:

- Ofrecer una ventana que permita dar de alta un cliente visualmente (el identificador del cliente, su nombre, apellidos, dirección postal y DNI) y lo guarde en un fichero que tendrá estructura de JSON. En este fichero se irán concatenando todos los usuarios que se den de alta. NO se pide que se realice el alta como socio; solo la información de cliente.
- Tenga un botón llamado “Cargar clientes” que lea el fichero anterior, y los cargue en el sistema.
- Tenga un botón llamado “Simulación” que realice las siguientes acciones, y muestre el resultado en pantalla, en una ventana emergente de la interfaz gráfica:
    1. Mostrar información sobre cada uno de los clientes de UMUTickets.
    2. Alta de un cliente adicional que no sea socio.
    3. Alta de dos cliente adicionales que sean socios, uno de ellos una empresa.
    4. Creación de una empresa organizadora.
    5. El alta de un evento deportivo y un espectáculo audiovisual directamente por los gestores de la central.
    6. El alta de un evento de tipo feria, a través de la empresa organizadora creada anteriormente.
    7. Realizar la reserva del cliente no socio a los espectáculos deportivo y audiovisual.

8. Realizar la reserva del cliente socio particular a los espectáculos deportivo y audiovisual.
9. Realizar la reserva del cliente socio empresarial al evento de feria.
10. Simular la compra de al menos 3 eventos previamente reservados.
11. Mostrar los intentos de anulación de una reserva para la que todavía no se haya realizado la compra, y de otra en la que se haya realizado la compra.
12. Mostrar información sobre la anulación de uno de los eventos anteriores.
13. Mostrar información sobre los beneficios obtenidos por la central de reserva de cada uno de los eventos creados.
14. Mostrar información del número de socios actuales y los ingresos obtenidos en la central por sus cuotas.

3. Normas de entrega del proyecto

La práctica deberá de presentarse por el Aula Virtual (AV), vía Tareas, en las fechas establecidas y que serán comunicadas por los profesores de la asignatura.

Todo deberá entregarse en **un único fichero comprimido en formato ZIP**. El fichero comprimido contendrá la siguiente estructura de directorios (no uses acentos o espacios en los nombres de los directorios/ficheros):

```
Proyecto
|
|--Docs
   |--Informe.pdf
   |--Diagrama.pdf
   |--API
      |--*.html
|--DB
   |--clientes.json
|--Source
   |--main.py
   |--Paquete1
      |--Subpaquete1.1
      |--etc
   |--Paquete2
   |--etc
```

Las subcarpetas contendrán el siguiente contenido:

1. La carpeta **Docs** contiene la documentación de la práctica, que incluye **un fichero en formato PDF** con el informe escrito por vosotros, una subcarpeta con la **documentación** generada automáticamente en formato .html por la herramienta **Pydoctor** (ver apartado 4), y un **diagrama de clases UML** generado a partir del proyecto Python **en formato PDF** (ver apartado 4).
2. La carpeta **Source** contiene el proyecto PyCharm completo, que contiene solo los archivos fuente (SIN incluir la carpeta **venv** del entorno virtual de Python).
3. La carpeta **DB** contiene un fichero JSON con un listado de varios clientes dados de alta en el sistema, sirviendo de base de datos para la ejecución.

4. Documentación del proyecto

En la carpeta de la documentación (**Docs**), se entregará:

1. Un documento en formato PDF llamado "**Informe.pdf**" que contendrá los nombres de los autores y los siguientes apartados. El objetivo de este documento es justificar las decisiones de diseño y/o implementaciones adoptadas.
    - **Estructura del proyecto**. Indicar brevemente cómo se ha estructurado el programa con paquetes justificando la agrupación de las clases relacionadas.
    - **Clases**. Listado de las clases indicando el propósito de cada una los tipos de relaciones que hay entre ellas. Por tanto, en herencia se debe indicar el nombre de las clases hijas más representativas. En el caso de que se trate de una clase abstracta, justifica por qué debe ser de este tipo. Indicar los aspectos de visibilidad y delegación que consideres más relevantes.
    - **Interfaces**. Listado de las interfaces indicando el propósito de cada una. A continuación, indica el nombre de las clases más representativas que la implementen.
    - **Gestión de entrada/salida.** Indicar cómo hemos resuelto el almacenamiento y recuperación de información en el proyecto. Mostrad un ejemplo del formato elegido para representar a varios clientes.
    - **Interfaz gráfica**. Explicar cómo se ha planteado las ventanas gráficas del proyecto, incluyendo capturas de pantalla de estas en las que se vean en funcionamiento.
    - **Programa principal.** Describir brevemente el fichero Python main.py que permiten ejecutar el programa, indicando los principales objetos creados y sus intercambios de mensajes.
2. Un diagrama de clases UML en formato PDF llamado **"Diagrama.pdf"** obtenido de forma automatizada a partir del código (ver Anexo I).
3. Subcarpeta (**Docs/API**) que estará formada por la documentación generada por la librería de Python Pydoctor (en formato .html), tal como está explicado en la Sesión 1 de prácticas. Con carácter general:
    - Cada módulo comenzará con un docstring explicando la funcionalidad y justificación de este.
    - Cada clase contendrá un docstring explicando el propósito de dicha clase.
    - El método de inicialización tendrá un docstring indicando los atributos propios y, en su caso, los atributos heredados más importantes. Cada método (de instancia o de clase) explicará en su docstring qué es lo que hace y, si fuera necesario, las particularidades y ejemplos de uso de este.
    - Los métodos Getter y propiedades no requieren documentación.
    - Los métodos Setter documentarán las restricciones de uso que tengan.
    - Los métodos de instancia y estáticos tendrán un docstring indicando una breve descripción, el significado de sus parámetros y una descripción de los datos devueltos.

5. Evaluación del proyecto

Este proyecto constituye el 50% de la nota de prácticas de la asignatura.

El **plagio (total o parcial) en cualquiera de las partes presentadas para su evaluación conllevará una valoración de 0 puntos en la nota final de la convocatoria correspondiente para todos los estudiantes implicados**. Los profesores usarán diversas herramientas de carácter informático para la detección de plagio sobre cualquier entrega realizada.

5.1. Requisitos mínimos

Para aprobar el proyecto, se deben cumplir los siguientes requisitos mínimos. De no cumplir alguno de ellos, **el proyecto se puede suspender automáticamente**:

- **Requisito 1**. Entregar en tiempo, a través de la tarea pertinente, un fichero .zip con todo el proyecto, incluyendo documentación y código, de acuerdo con las normas de entrega indicadas en el apartado 3.
- **Requisito 2**. La documentación debe cumplir las normas indicadas en el apartado 4.
- **Requisito 3**. El programa debe responder a las funcionalidades que se piden en el enunciado de este documento, apartado 2.
- **Requisito 4**. El programa principal se ejecuta sin errores y lo programado funciona correctamente.

5.2. Criterios de evaluación y calificación

Los aspectos que se tendrán en cuenta durante la evaluación y calificación del trabajo presentado son los siguientes. Tanto el código como la documentación son de obligada entrega para poder evaluar la práctica.

- **Código** (80% de la nota):
    - Funcionamiento correcto de los programas presentados (no pueden tener errores en tiempo de ejecución).
    - Originalidad de la solución al problema planteado.
    - Usar correctamente la metodología de la POO en todos los aspectos que han sido mostrados en las sesiones teórico-prácticas, destacando:
        - Empleo de la declaración de tipos explícita y uso correcto de los tipos de datos.
        - Eficiencia y correctitud de las funciones, métodos y algoritmos.
        - Definir correctamente las clases (métodos y atributos) y la visibilidad de cada miembro.
        - Uso correcto de sobrecarga de métodos.
        - Definir las relaciones entre las clases (uso, asociación, agregación, composición y herencia). Uso correcto de delegación de responsabilidades entre clases.
        - Empleo adecuado de clases abstractas e interfaces.
        - Crear una buena estructuración en paquetes y módulos de la solución construida.
    - Uso adecuado de excepciones para gestionar errores en el código.
    - Uso correcto de entrada/salida en base a los requisitos del problema.

    - Diseño y funcionamiento correcto de la interfaz gráfica.
- **Documentación** (20% de la nota):
    - El informe debe contener todos los apartados mínimos solicitados.
    - Explicación, presentación y concreción del informe de prácticas.
    - Completitud y calidad de la documentación HTML generada automáticamente (descripciones representativas, tipos de datos indicados, valores devueltos, etc).

5.3. Entrevista

Si los profesores lo consideran necesario, podríais ser citados para realizar una entrevista.

Durante la entrevista se preguntarán cómo están realizadas determinadas partes del proyecto de programación presentado, al tiempo que se podrá solicitar la modificación de alguna parte para que haga algo diferente.

El resultado de la entrevista puede cambiar por completo la calificación obtenida en el proyecto presentado.

Anexo I: Generación de diagrama de clases UML

Este anexo documenta las instrucciones para instalar el software necesario para generar el diagrama de clases de tu proyecto:

- **Instalar pylint**: Teniendo en Pycharm el proyecto abierto, ir a Settings > Project > Python Interpreter. Pulsamos el icono de **+**, buscamos en la lista **pylint** y, finalmente, pulsamos **Install Package**. Una vez instalado, damos a **Apply** y cerramos la configuración.
- **Instalar Graphviz**:
    i. Windows:
        1. Descargar Graphviz de la web oficial. El siguiente enlace lo descarga directamente: Enlace
        2. Tras instalarlo, abre el menú de inicio de Windows y escribe "Variables de entorno". Selecciona la opción "Editar las variables de entorno del sistema". En la ventana que aparece, selecciona "Variables de entorno". En la sección de "Variables del sistema", selecciona la variable **Path** y haz clic en **Editar**. Haz click en **Nuevo** y añade la ruta donde se instaló Graphviz: *C:\Program Files\Graphviz\bin*. Guarda los cambios.
        3. Comprueba que se ha instalado correctamente. Para ello, abre una terminal en Pycharm (icono abajo a la izquierda) y escribe el siguiente comando. Debería mostrar la versión de Graphviz, indicando que se ha instalado correctamente:

            dot -version

    ii. Mac:
        1. Instala Homebriew en caso de que no lo tengas. Comprobaremos si está instalado abriendo la aplicación **Terminal** y tecleando el siguiente comando. Si nos indica la versión, está instalado, y podemos pasar al paso 3. En caso contrario, sigue al paso 2.
        2. Las instrucciones para instalar Homebrew están en el siguiente enlace: Enlace. Copiaremos el comando que aparece justo debajo de "Instala Homebiew", pulsando al icono que aparece a la derecha del comando.
        3. Abriremos la aplicación **Terminal** en el Mac, pegaremos el comando que copiamos previamente en la web, y pulsaremos Intro. Esperaremos unos minutos hasta que se complete el proceso de instalación.
        4. Instalaremos Graphviz con el siguiente comando en la Terminal:

            brew install graphviz

        5. Comprueba que se ha instalado correctamente. Para ello, abre una terminal en Pycharm (icono abajo a la izquierda) y escribe el siguiente comando. Debería mostrar la versión de Graphviz, indicando que se ha instalado correctamente:

            dot -version

- Usar el comando pyreverse para generar el diagrama UML. Para ello, abriremos la terminal de Pycharm y pegaremos el siguiente comando:

    pyreverse -f ALL -o pdf -p Diagrama .

Este comando genera un diagrama Diagrama.pdf en el que se incluyen todos los miembros independientemente de su visibilidad (privados, protegidos y públicos). *Nota*: es importante incluir el punto al final del comando para que se genere correctamente el diagrama.

Proyecto 2

Tipos de datos abstractos

Convocatoria de enero de 2025

Curso 2024-25

Contenido

1. Introducción

En este documento se describe el **Proyecto 2** (Tipos de datos abstractos) de la asignatura **Tecnología de la Programación** de segundo curso del Grado en Matemáticas para la convocatoria de enero del curso 2024-2025.

Todo lo explicado en la parte de abstracción y TDAs de la asignatura es susceptible de ser usado para resolver este enunciado, donde el diseño seguirá las buenas prácticas de programación, y debe realizarse en parejas.

Este documento incluye también las normas de realización, entrega y defensa, así como los criterios generales de evaluación que se utilizarán para corregir el trabajo entregado.

2. Enunciado del proyecto

2.1. TDA Correo

Corresponde a la sesión de prácticas 10 en la que se pide la especificación, representación e implementación del tipo de datos abstracto Correo.

2.2. TDA Gestor de correo

Corresponde a la sesión de prácticas 11 en la que se pide la especificación, representación e implementación del tipo de datos abstracto Gestor de correo.

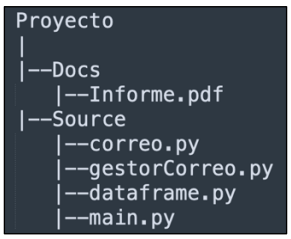
2.3. TDA Dataframe

Corresponde a la sesión de prácticas 12 en la que se pide la especificación, representación e implementación del tipo de datos abstracto Dataframe.

3. Normas de entrega del proyecto

La práctica deberá de presentarse por el Aula Virtual (AV), vía Tareas, en las fechas establecidas y que serán comunicadas por los profesores de la asignatura.

Todo deberá entregarse en **un único fichero comprimido en formato ZIP**. El fichero comprimido contendrá la siguiente estructura de directorios (no uses acentos o espacios en los nombres de los directorios/ficheros):



Las subcarpetas contendrán el siguiente contenido:

- La carpeta **Docs** contiene la documentación de la práctica, que incluye **un fichero en formato PDF** llamado *Informe.pdf* con la memoria descriptiva de la especificación, representación e implementación asociadas a cada TDA pedido.

- La carpeta **Source** contiene un archivo de código fuente por cada uno de los TDAs creados. Además, se debe adjuntar un archivo *main.py* que importe los tres TDAs anteriores y valide el correcto funcionamiento de cada uno de los TDAs.
  Nota: no hay que adjuntar un proyecto en Pycharm, únicamente los archivos fuente solicitados.

4. Documentación del proyecto

En la carpeta de la documentación (**Docs**), se entregará un documento en formato PDF llamado "**Informe.pdf**" que contendrá los nombres de los autores y los siguientes apartados. Este archivo constituye la documentación formal de:

- **Por cada TDA**:
  - **Especificación**. En relación a la especificación, documentaremos los tres apartados principales que la componen: 1) nombre y descripción, 2) dominio de los datos, y 3) especificación de las operaciones (descripción, precondiciones, información retornada y excepciones).
  - **Representación**. Describirá una estructura que indique los campos que representa el TDA. En el caso de ser necesaria una colección de datos, se debe justificar la estructura elegida. **Nota**: no se pide la función de abstracción y el invariante de la representación.
  - **Implementación**. Explicar cómo se ha programado la especificación y representación del TDA. Destacar y justificar las decisiones tomadas más importantes en relación al código. También se debe identificar claramente la parte privada y pública del TDA.
- **Programa principal.** Describir brevemente el fichero Python main.py que permite ejecutar el programa. Explica **con tus propias palabras** las ventajas de importar y usar los tres TDAs anteriormente mencionados.

5. Evaluación del proyecto

Este proyecto constituye el 50% de la nota de prácticas de la asignatura.

El **plagio (total o parcial) en cualquiera de las partes presentadas para su evaluación conllevará una valoración de 0 puntos en la nota final de la convocatoria correspondiente para todos los estudiantes implicados**. Los profesores usarán diversas herramientas de carácter informático para la detección de plagio sobre cualquier entrega realizada.

5.1. Requisitos mínimos

Para aprobar el proyecto, se deben cumplir los siguientes requisitos mínimos. De no cumplir alguno de ellos, **el proyecto se puede suspender automáticamente**:

- **Requisito 1**. Entregar en tiempo, a través de la tarea pertinente, un fichero .zip con todo el proyecto, incluyendo documentación y código, de acuerdo con las normas de entrega indicadas en el apartado 3.
- **Requisito 2**. La documentación debe cumplir las normas indicadas en el apartado 4.

- **Requisito 3**. El programa debe responder a las funcionalidades que se piden en el enunciado de este documento, apartado 2.
- **Requisito 4**. El programa **no puede hacer uso de tipos de datos propios de Python basados en colecciones**. Por ejemplo, está prohibido hacer uso de los tipos de datos tales como *dict*, *set*, *tuple*. El tipo *list* solo lo podremos usar de apoyo a una representación contigua, sin poder hacer uso de sus métodos y funciones asociadas, únicamente la representación a modo de array, como se ha visto en teoría.
- **Requisito 5**. El programa principal se ejecuta sin errores y lo programado funciona correctamente.

5.2. Criterios de evaluación y calificación

Los aspectos que se tendrán en cuenta durante la evaluación y calificación del trabajo presentado son los siguientes. Tanto el código como la documentación son de obligada entrega para poder evaluar la práctica.

- **Ponderación** de cada TDA sobre el total de la nota:
  - TDA Correo: **20%**
  - TDA Gestor Correo: **40%**
  - TDA Dataframe: **40%**
- **Código** (70% de la nota por cada TDA):
  - Funcionamiento correcto de los programas presentados (no pueden tener errores en tiempo de ejecución).
  - Originalidad de la solución al problema planteado.
  - Usar correctamente la metodología de la POO en todos los aspectos que han sido mostrados en las sesiones teórico-prácticas, destacando:
    - Empleo de la declaración de tipos explícita y uso correcto de los tipos de datos.
    - Eficiencia y correctitud de las funciones, métodos y algoritmos.
    - Definir correctamente las clases (métodos y atributos) y la visibilidad de cada miembro.
    - Uso correcto de sobrecarga de métodos.
    - Uso correcto de delegación de responsabilidades entre clases.
  - Uso adecuado de excepciones para gestionar errores en el código.
  - Se respeta la parte pública y privada de los TDAs cuando se invoca su funcionalidad desde el main.
  - Uso de docstring para documentar el código. **Nota**: no se pide generar documentación interactiva en formato HTML.
- **Documentación** (30% de la nota por cada TDA):
  - El informe debe contener todos los apartados mínimos solicitados.
  - Explicación, presentación y concreción del informe de prácticas.

5.3. Entrevista

Si los profesores lo consideran necesario, podríais ser citados para realizar una entrevista.

Durante la entrevista se preguntarán cómo están realizadas determinadas partes del proyecto de programación presentado, al tiempo que se podrá solicitar la modificación de alguna parte para que haga algo diferente.

El resultado de la entrevista puede cambiar por completo la calificación obtenida en el proyecto presentado.

Proyecto de prácticas

Tecnología de la Programación

Convocatoria de junio de 2025

Curso 2024-25

Contenido

1. PRIMERA PARTE: PROGRAMACIÓN ORIENTADA A OBJETOS

1.1. Introducción

Todo lo explicado en la parte de POO de la asignatura es susceptible de ser usado para resolver este enunciado, donde el diseño seguirá las buenas prácticas de programación, así como el criterio de la pareja.

1.2. Enunciado

1.2.1. UMUTickets

En este proyecto vamos a implementar en Python una versión sencilla de la aplicación **UMUTickets**, que permitirá que i) ciertas entidades organicen y oferten diferentes tipos de eventos, tales como deportivos o de ocio, y ii) los clientes podrán reservar y comprar entradas para los mismos.

Todos los elementos del sistema se modelarán con clases, que estarán relacionados entre sí mediante relaciones de uso, asociación, agregación, composición, herencia e interfaces.

En las siguientes subsecciones se comentan los detalles de las funcionalidades mínimas. Las clases pueden tener los atributos que se consideren necesarios para que la lógica del programa funcione correctamente.

1.2.2. Tipos de clientes y condición de socio

UMUTickets contará con un conjunto de **clientes**. Cada cliente tendrá un identificador de cliente que debería ser único.

Los clientes podrán ser **particulares** o **empresariales**.

- De los clientes **particulares** será necesario almacenar la **persona** asociada (nombre, apellidos, dirección postal y DNI). Adicionalmente, se deberá almacenar información sobre los datos de una tarjeta de crédito para poder realizar las reservas y las posteriores compras.
- De los clientes **empresariales**, es necesario guardar información sobre la **empresa** asociada (razón social, su CIF, su teléfono y su dirección postal) y la **persona** de contacto principal (nombre, apellidos, dirección postal y DNI), así como una tarjeta de crédito de manera análoga a los clientes particulares.

Un cliente puede ser socio de UMUTickets, lo que le ofrece ciertas ventajas:

- Dispondrán de un número de socio (que debería ser único y diferente del número de cliente).
- Dispondrán de una tarjeta de socio con un saldo económico asociado. El sistema deberá ofrecer la funcionalidad necesaria para realizar recargas de cantidades de 5, 10, 20 o 50 euros en el saldo de la tarjeta de socio, y que se harán desde la tarjeta de crédito asociada al cliente.

Los socios podrán renovar su condición de socio de manera mensual, trimestral o anual, y dependiendo de la periodicidad, tiene un coste diferente:

- Las renovaciones mensuales tienen un coste de 5 euros al mes.
- Las renovaciones trimestrales tienen un coste de 20 euros al trimestre.
- Las renovaciones anuales tienen un coste de 50 euros anuales.

En cualquier caso, las renovaciones se realizarán siempre con cargo a su tarjeta de socio, por lo que para realizarla debe comprobarse que existe suficiente saldo. Para cada renovación realizada por un socio se debería registrar la fecha de inicio de la renovación y la fecha de expiración prevista.

Además de poder hacerse socio, un cliente puede gestionar reservas y comprar entradas para los diferentes tipos de eventos ofertados.

1.2.3. Tipos de eventos y organización de eventos

UMUTickets gestionará **eventos.** Todos los tipos de eventos comparten un conjunto de características comunes:

- Un identificador único del evento.
- Un nombre.
- Una descripción del evento.
- Algunos eventos tendrán una fecha única del evento mientras que otros podrán tener un periodo de duración (una fecha de inicio y una fecha de fin).
- Dirección del evento.
- Una URL/link con la ubicación exacta del evento.
- Número de entradas disponibles.
- Precio original de la entrada al evento (sin incluir el I.V.A.)
- Organizador (que será una empresa organizadora o UMUTickets).

Sin embargo, UMUTickets no debe permitir crear eventos generales, pues sólo existen eventos específicos que calculan el precio final de sus reservas/entradas en función de criterios diferentes (explicados en el siguiente apartado 2.4). Inicialmente están previstos los siguientes tres **tipos de eventos** (aunque se añadirán más en el futuro), y que contendrán la siguiente información:

- **Deportivos**. Incluirán el deporte del que se trata, que en principio solo podrá ser de Fútbol, Tenis o Baloncesto.
- **Espectáculo audiovisual**. Incluirá el nombre del artista o grupo que realiza el espectáculo, así como una edad mínima recomendada para el mismo.
- **Feria empresarial**. Este evento solo estará dirigido a clientes que son empresas, por lo que deberá de tenerse en cuenta que solo este tipo de clientes podrán realizar reservas o comprar entradas. Además, en este tipo de evento el número máximo de entradas que el evento puede ofertar es de 100.

Los eventos podrán ser organizados por i) el elemento central **UMUTickets** o bien ii) por **empresas organizadoras**. Este tipo de empresas que son organizadoras de eventos, además de la razón social, CIF, teléfono y dirección postal, tendrán un listado de sus eventos organizados, así como un identificador de organizador que es único en el sistema.

En particular, la funcionalidad de gestionar eventos permite **crear eventos, modificarlos y anularlos**. UMUTicket tiene privilegios para gestionar cualquier evento existente en el sistema, mientras que una empresa organizadora sólo puede gestionar sus propios

eventos. En un futuro, se espera que más elementos puedan gestionar eventos a su manera.

1.2.4. Reservas y ventas de entradas

La gestión de las reservas y venta de entradas se realizará a través de UMTickets. El sistema ofrecerá todos los métodos necesarios para realizar la reserva y anulación, así como la venta de las entradas (con o sin reserva previa).

En particular, los clientes de UMUTickets pueden realizar **reservas** en los eventos ofrecidos, siempre y cuando sea anterior a 7 días previos a la fecha de inicio del evento y quede disponibilidad de entradas. La reserva tendrá una fecha y el precio de la entrada reservada, que dependerá de si el cliente es socio o no y del tipo de evento acorde a las siguientes condiciones:

- Todas las reservas, por el hecho de hacerse con una semana de antelación, tienen de base un 10% de descuento sobre el precio de la entrada al evento.
- En los **eventos que son ferias empresariales**, los clientes (empresariales) que son socios tendrán un descuento del 30% extra.
- Respecto a los **eventos deportivos y espectáculos audiovisuales**, los clientes que son socios tendrán un descuento extra del 20%. Sin embargo, el precio final de la reserva se incrementará si quedan pocas entradas:
  - **Eventos deportivos**: cuando quede un 25% del número máximo de entradas disponibles, el precio de la reserva resultante aumenta un 10% automáticamente.
  - **Espectáculos audiovisuales**: cuando quede un 10 % del número máximo de entradas disponibles, el precio de la reserva aumenta un 30% automáticamente.

Al realizar una reserva, se puede pagar directamente la entrada o bien más adelante. En cualquiera de los casos, deberá respetarse el precio de venta reflejado en la reserva. También existe la posibilidad de que un cliente **anule una reserva**, pero siempre y cuando no se haya realizado el pago en firme de la entrada. Además, como los dueños de UMUTickets no tienen claro si los eventos de feria empresarial son rentables para el año que viene, cada anulación de un evento de este tipo debería de anotarse, de cara a tener el **número total de anulaciones** de eventos de feria empresarial, y tomar decisiones para el año próximo.

Por otro lado, se puede realizar una compra directamente de una entrada sin reserva (si hay disponibilidad y se realiza antes del inicio del evento). El precio de venta no tendrá el descuento de la reserva, pero cumplirá las condiciones para socios y no socios, así como las reglas de incremento. Lógicamente, la venta de una entrada solo puede realizarse hasta el día de inicio del evento.

Cuando se vende una entrada (bien reservada o de manera directa), se registrará que se ha realizado la venta, la fecha de la venta, el número de entrada oficial, y el cliente al que se le ha realizado.

1.2.5. Gestión y estadísticas de UMUTickets

De cara al control interno, el sistema ofrecerá métodos que permitan:

1. Obtener el **beneficio** que ha obtenido UMUTickets de un evento particular, teniendo en cuenta que:
    - En los eventos deportivos, por cada entrada vendida a clientes no socios, obtiene un beneficio de un 10% del coste de dicha entrada; por cada entrada vendida a clientes socios, obtiene un beneficio de un 5% del coste de dicha entrada.
    - En los eventos de espectáculo audiovisual, por cada entrada vendida a clientes no socios, obtiene un beneficio de un 15 % del coste de la entrada; por cada entrada vendida a clientes socios, obtiene un beneficio de un 10 % del coste de la entrada.
    - En las ferias empresariales, por cada entrada vendida a clientes empresariales no socios, se obtiene un 10% del coste de esta, mientras que la vendida a un socio genera un 5% de beneficio con respecto a su coste.
2. Obtener el **número de socios actuales**, y los **ingresos totales** que está recibiendo UMUTickets con esa cantidad de socios.
3. Obtener el **porcentaje medio de anulaciones** sobre las reservas realizadas en las ferias empresariales.

1.2.6. Interfaz gráfica del sistema

En este proyecto, y como versión inicial, será necesario crear una interfaz gráfica para el sistema, que permita la siguiente funcionalidad:

- Ofrecer una ventana que permita dar de alta visualmente a un cliente de cualquiera de los dos tipos definidos por la aplicación. Los datos obtenidos se guardarán en un único fichero JSON, donde habrá posiblemente clientes precargados de ejecuciones anteriores, y aquellos nuevos deberán concatenarse a los ya existentes. Así, es imprescindible usar el fichero JSON en la resolución de la práctica. NO se pide que se realice el alta como socio; solo la información de cliente.
- Tenga un botón llamado “Cargar clientes” que lea el fichero anterior y cargue los clientes en el sistema. Además, deberá mostrarse el listado de clientes cargados en la GUI de forma que se visualice correctamente la información más relevante.

1.2.7. Validación de la aplicación

La validación de la aplicación se realizará mediante la implementación de un archivo *main* con una batería exhaustiva de pruebas que garanticen el correcto funcionamiento de la solución software implementada. El *main* deberá seguir los siguientes pasos:

- Lanzar la interfaz gráfica para que el usuario pueda dar de alta y cargar clientes, pudiendo visualizar desde la GUI el listado de clientes cargado.
- Cuando el usuario cierre la GUI, el *main* mostrará por terminal el listado de los clientes existentes en el fichero JSON.

- Implementación de las siguientes pruebas exhaustivas para verificar el correcto funcionamiento de la aplicación, haciendo uso tanto de clientes cargados del JSON como de nuevos que sean necesarios para la realización de las pruebas:
    - Gestión de clientes
        - Registrar un cliente particular y otro empresarial con datos correctos
        - Intentar registrar un cliente sin DNI válido
        - Intentar registrar un cliente sin datos de tarjeta de crédito
        - Modificar la dirección de un cliente particular y otro empresarial
        - Intentar modificar campos que tienen control de errores para verificar que no se pueden realizar
    - Gestión de socios
        - Convertir un cliente en socio y asignar un número de socio único
        - Recargar saldo de socio en una cantidad permitida
        - Intentar recargar una cantidad no permitida (por ejemplo, 7€)
        - Renovar la suscripción de socio con saldo suficiente
        - Intentar renovar la suscripción sin saldo suficiente
        - Consultar información de socios y su saldo
    - Gestión de eventos
        - Crear un evento deportivo con valores válidos
        - Crear un espectáculo audiovisual con nombre de artista y edad mínima
        - Crear una feria empresarial solo accesible a clientes empresariales
        - Intentar crear un evento sin nombre o sin fecha
        - Modificar la descripción de un evento existente
        - Intentar modificar el número de entradas cuando ya se han vendido reservas
        - Eliminar un evento sin reservas ni ventas
        - Intentar eliminar un evento con reservas activas
        - Intentar eliminar un evento con entradas ya vendidas.
    - Gestión de reservas
        - Un cliente particular reserva una entrada con más de 7 días de antelación
        - Un cliente socio particular reserva una entrada y recibe descuento
        - Un cliente socio empresarial reserva una entrada en una feria empresarial con descuento especial
        - Intentar reservar una entrada cuando quedan menos de 7 días para el evento
        - Intentar reservar un evento cuando no quedan entradas disponibles
        - Intentar reservar una feria empresarial con un cliente particular
        - Un cliente cancela una reserva sin haber pagado la entrada
        - Intentar cancelar una reserva ya pagada
    - Comprar entradas
        - Un cliente compra una entrada sin reserva previa
        - Un cliente compra una entrada con reserva previa y se mantiene el precio
        - Intentar comprar una entrada cuando el evento ya ha comenzado

        - Intentar comprar una entrada cuando el saldo de socio es insuficiente
    - Estadísticas y gestión de UMUTickets
        - Obtener el beneficio de UMUTickets para un evento deportivo
        - Obtener el beneficio de UMUTickets para un espectáculo audiovisual
        - Obtener el beneficio de UMUTickets para una feria empresarial
        - Consultar el número total de socios activos
        - Consultar los ingresos totales por cuotas de socios
        - Obtener el porcentaje de anulaciones en ferias empresariales

1.3. Documentación

En la carpeta de la documentación (**Docs**), se entregará:

1. Un documento en formato PDF llamado "**InformePOO.pdf**" que contendrá los nombres de los autores y los siguientes apartados. El objetivo de este documento es justificar las decisiones de diseño y/o implementaciones adoptadas.
    - **Estructura del proyecto**. Indicar brevemente cómo se ha estructurado el programa con paquetes justificando la agrupación de las clases relacionadas.
    - **Clases**. Listado de las clases indicando el propósito de cada una los tipos de relaciones que hay entre ellas. Por tanto, en herencia se debe indicar el nombre de las clases hijas más representativas. En el caso de que se trate de una clase abstracta, justifica por qué debe ser de este tipo. Indicar los aspectos de visibilidad y delegación que consideres más relevantes.
    - **Interfaces**. Listado de las interfaces indicando el propósito de cada una. A continuación, indica el nombre de las clases más representativas que la implementen.
    - **Gestión de entrada/salida.** Indicar cómo hemos resuelto el almacenamiento y recuperación de información en el proyecto. Mostrad un ejemplo del formato elegido para representar a varios clientes.
    - **Interfaz gráfica**. Explicar cómo se ha planteado las ventanas gráficas del proyecto, incluyendo capturas de pantalla de estas en las que se vean en funcionamiento.
    - **Programa principal.** Describir brevemente el fichero Python main.py que permiten ejecutar el programa, indicando los principales objetos creados y sus intercambios de mensajes.
2. Un diagrama de clases UML en formato PDF llamado **"Diagrama.pdf"** obtenido de forma automatizada a partir del código (ver Anexo I).
3. Subcarpeta (**Docs**/**API**) que estará formada por la documentación generada por la librería de Python Pydoctor (en formato .html), tal como está explicado en la Sesión 1 de prácticas. Con carácter general:
    - Cada módulo comenzará con un docstring explicando la funcionalidad y justificación de este.
    - Cada clase contendrá un docstring explicando el propósito de dicha clase.
    - El método de inicialización tendrá un docstring indicando los atributos propios y, en su caso, los atributos heredados más importantes. Cada

método (de instancia o de clase) explicará en su docstring qué es lo que hace y, si fuera necesario, las particularidades y ejemplos de uso de este.
- Los métodos Getter y propiedades no requieren documentación.
- Los métodos Setter documentarán las restricciones de uso que tengan.
- Los métodos de instancia y estáticos tendrán un docstring indicando una breve descripción, el significado de sus parámetros y una descripción de los datos devueltos.

2. SEGUNDA PARTE: TIPOS DE DATOS ABSTRACTOS

2.1. Introducción

Todo lo explicado en la parte de abstracción y TDAs de la asignatura es susceptible de ser usado para resolver este enunciado, donde el diseño seguirá las buenas prácticas de programación, y debe realizarse en parejas.

2.2. Enunciado del proyecto

2.2.1. TDA Correo (igual que en la convocatoria de febrero)

Un correo electrónico es una elemento digital que consta de varios elementos: un remitente, es decir, la dirección de correo electrónico de quien envía el mensaje; un destinatario, que es la dirección de correo a la que se envía el mensaje (en esta práctica solo consideraremos un único destinatario); un breve asunto, que resume el contenido del mensaje; el cuerpo del mensaje, que incluye el contenido principal que se desea comunicar; una fecha y hora de envío o recepción; un estado de lectura, que indica si el correo ha sido leído o no; y un estado de borrador, que indica si el correo aún no ha sido enviado. Además, un correo electrónico contendrá un identificador único que permitirá diferenciarlo de otros correos.

Además de estas características, el correo debe poder realizar una serie de operaciones para interactuar con sus datos de manera controlada. Entre estas operaciones se incluyen cambiar el estado de lectura para marcarlo como leído o no leído. Además, un correo debe permitir actualizar los datos del destinatario, remitente, asunto y cuerpo del correo, pero únicamente cuando el correo sea un borrador. Una vez enviado el correo ya no podrá editarse. También debe ser posible buscar una palabra específica dentro del correo, independientemente de si es en el asunto o en el cuerpo del mensaje, y determinar si dicha palabra se encuentra escrita en el correo.

A partir de esta descripción se debe diseñar el TDA Correo para que tenga las operaciones y la representación acorde a la descripción anterior.

2.2.2. TDA Gestor de correo (igual que en la convocatoria de febrero)

El objetivo en este apartado es elaborar un TDA que represente un gestor de correo básico. Trabajaréis en los aspectos de especificación, representación e implementación del TDA, para comprender cómo gestionar correos electrónicos de manera abstracta, sin preocuparnos por los detalles internos considerados para su implementación.

Un gestor de correo es una entidad digital que permite gestionar de forma eficiente los correos electrónicos de un usuario, brindando funcionalidades que van desde la redacción hasta la organización, envío y recepción de correos. Cada gestor de correo está asociado a un único usuario, cuya identidad se maneja mediante un nombre de usuario (dirección de correo electrónico) y una contraseña.

El gestor cuenta con varias bandejas para clasificar y organizar los correos según su estado. Estas bandejas son:

- **Bandeja de entrada**: almacena los correos recibidos por el usuario.
- **Bandeja de salida**: contiene los correos enviados.
- **Bandeja de borradores**: almacena los correos en estado de borrador, permitiendo que el usuario los edite hasta que decida enviarlos.
- **Bandeja de spam**: contiene los correos identificados como no deseados o sospechosos, en función de ciertos criterios de spam configurados.

Cada bandeja permite almacenar múltiples correos electrónicos, teniendo en cuenta que el orden de cómo se almacenan estos correos no es relevante en el contexto de este TDA. Así, el gestor de correo debe poder acceder a la colección de correos asociada a cada bandeja, así como el número de correos contenida en ella en un momento dado.

El gestor de correos contendrá, además, criterios de spam, que consisten en una colección de palabras clave que, al detectarse en el asunto o cuerpo de un correo entrante, provocan que este se añada automáticamente en la bandeja de spam, sin pasar por la bandeja de entrada. Esto permite filtrar de manera automática los correos no deseados en el momento de su recepción.

Las principales funcionalidades que ofrece un gestor de correo para interactuar con la información del usuario y los correos electrónicos son:

- **Cambiar contraseña**: el usuario podrá cambiar libremente la contraseña de su cuenta de correo.
- **Gestión de criterios de spam**: el usuario podrá añadir y eliminar criterios de spam, que servirán para clasificar de forma automática si un correo entrante debe ir a la bandeja de entrada o a la de spam. Esos criterios corresponden a un listado de palabras sin relación de orden.
- **Redactar un correo**: el gestor permite redactar correos, que se almacenan siempre inicialmente en la bandeja de borradores, pudiendo ser modificados libremente por el usuario. El gestor permitirá cambiar la dirección de correo electrónico del destinatario, el contenido del asunto, o el cuerpo del correo. Sin embargo, estas modificaciones solo podrán realizarse sobre aquellos correos en la bandeja de borradores.
- **Enviar y recibir un correo**: Una vez que el usuario del gestor de correo (lo denotaremos como *gestor emisor* por brevedad) decide que un correo de la

bandeja de borradores no necesita más modificaciones, podrá enviar dicho correo al gestor de correo de otro usuario (gestor receptor). El envío de un correo consta de los siguientes pasos:

  - **Acciones en el gestor emisor**: el correo seleccionado en el gestor de correo emisor, a partir de su identificador único de correo, pasará de la bandeja de borradores a la bandeja de enviados, cambiando el estado del correo, de borrador a enviado.
  - **Acciones en el gestor receptor**: el gestor emisor, al enviar el correo, deberá invocar a la funcionalidad de recepción de correos en el gestor receptor, quien añadirá el correo recibido al buzón de correo correspondiente. Así, si se detecta que el asunto o el mensaje del correo recibido contiene entre sus palabras al menos uno de los criterios de spam definidos en el gestor receptor, este correo electrónico se almacenará en la bandeja de spam. En caso contrario, se almacenará en la bandeja de entrada.

- **Cambiar el estado de lectura de un correo**: el usuario puede marcar un correo como leído o como no leído, pero únicamente en las bandejas de entrada y spam.
- **Eliminar un correo**: el gestor permite eliminar un correo electrónico de una bandeja de correo determinada, en base a su identificador único de correo. Se pueden eliminar correos de todas las bandejas.
- **Búsqueda de correos**: el gestor de correo permite obtener, en formato textual, la colección de correos que contienen una palabra o frase. Esta búsqueda se realizará en la totalidad del contenido del correo, esto es, tanto en su asunto como en el cuerpo del mensaje. Así, el gestor permitirá, de forma diferenciada, realizar la búsqueda por palabras clave tanto de forma individual por cada bandeja de correo, como sobre el total de las cuatro bandejas.

En base a todo lo anterior, provee la especificación, representación e implementación del TDA Gestor de Correo. De cara a la implementación, recuerda seguir las buenas prácticas de POO, en relación al uso de clases, delegación, etc. Además, **las colecciones que usemos en la implementación de este TDA deberán estar basadas en una estructura enlazada**.

2.2.3. TDA Dataframe *(CAMBIA respecto a enero)*

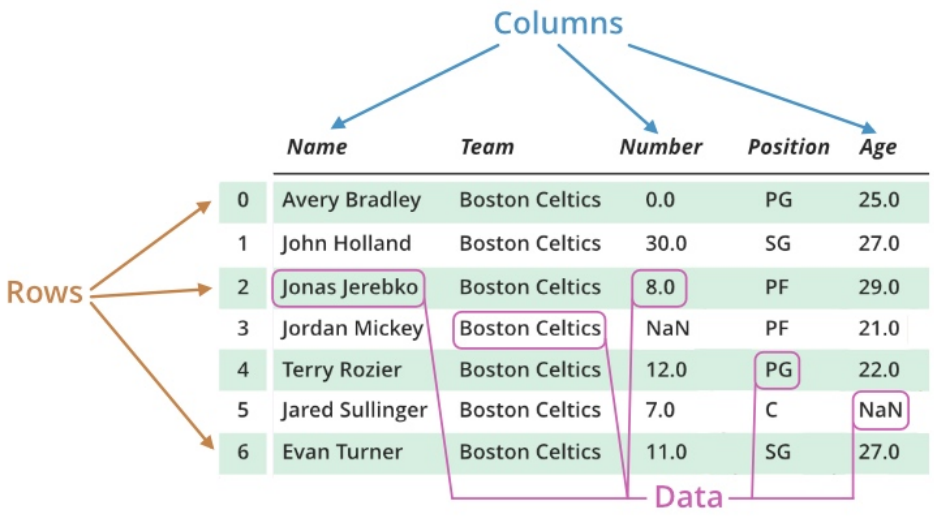
El objetivo en este apartado es elaborar un TDA que represente un DataFrame. Trabajaréis en los aspectos de especificación, representación e implementación del TDA, para comprender cómo gestionar información contenida en esta estructura de manera abstracta, sin preocuparnos por los detalles internos considerados para su implementación.

Un DataFrame es una estructura de datos tabular, ampliamente utilizada para gestionar y analizar datos en diferentes contextos, como bases de datos, procesamiento de información y análisis estadístico. Además, los DataFrames son ampliamente usados para gestionar la información usada por modelos de inteligencia artificial para aprender. Esta estructura se organiza en filas y columnas, donde cada columna tiene un nombre único, y cada fila puede contener datos de diferentes tipos.

Un DataFrame contiene varias características clave que lo hacen versátil para manipulación de datos:

- **Columnas**: Cada columna tiene un nombre único, que permite referenciarla directamente para consultarla o modificarla. Esto es, acceder a la columna a partir de su nombre. Todas las columnas deben tener **la misma longitud**, correspondiente al número de filas del DataFrame, así como el **mismo tipo de dato** (entero, real, cadena de texto, etc).
- **Filas**: Las filas contienen los datos del DataFrame y son organizadas por posición. Dicha posición corresponde a un índice secuencial que comienza siempre en cero. Cada fila puede ser accedida o manipulada mediante su índice correspondiente.
- **Valores nulos**: Las celdas del DataFrame pueden contener valores nulos (*None*), correspondiente a información faltante, algo muy común en entornos reales. Así, un DataFrame ofrece funcionalidades para trabajar eficientemente con estos valores.

La siguiente figura muestra una representación visual de un DataFrame. En este ejemplo, los valores nulos en las celdas se representan como *NaN* (Not a Number). Sin embargo, a nivel de implementación nosotros los consideraremos como *None*.



Las principales funcionalidades que ofrece el TDA DataFrame son las siguientes:

- **Inserción y eliminación de contenido:**
    - Insertar una nueva columna, pudiendo proporcionar una colección de valores específicos. En caso de no suministrarlos, se almacenarán valores nulos, correspondiendo a una ausencia de datos para esas celdas.
    - Insertar una fila, indicando la posición correspondiente sobre la que insertar la nueva fila.
    - Eliminar una columna a partir de un nombre de columna.
    - Eliminar una fila indicando la posición a eliminar.
- **Exploración y análisis:**
    - Verificar si una columna existe por su nombre.
    - Obtener las primeras (head) o las últimas (tail) n filas del DataFrame.

- Determinar la forma (shape) del DataFrame (número de filas y columnas).
- Calcular valores estadísticos básicos de únicamente todas las columnas que contengan valores numéricos (reales o enteros). En concreto, se mostrará, para cada columna: 1) el número de elementos que contiene, 2) el valor de la suma de los elementos, 3) la media, 4) la desviación típica, 5) el máximo, y 6) el mínimo. Un ejemplo de representación por consola, en el que se muestra únicamente una columna, podría ser el siguiente:

  Columna Edad:

  - Número de elementos: 10
  - Suma: 312
  - Media: 31.5
  - Desviación estándar: 5
  - Máximo: 40
  - Mínimo: 26

- **Procesamiento de datos:**
  - Reemplazar todos los valores nulos del DataFrame por un valor dado. Esto permite sustituir la ausencia de datos por un valor por defecto que sí puede ser interpretable para el cliente del TDA.
- **Visualización del DataFrame en formato textual:**
  - Visualizar un rango de filas, comprendidas entre dos índices.
  - Filtrar y mostrar aquellas filas que cumplan una determinada condición en base a los valores de una columna indicada. Por ejemplo, "mostrar todas las filas en las que el valor de la columna *edad* sea menor que 20". Así, un dataframe debe permitir realizar las siguientes comprobaciones en base los datos de una columna:
    - Mostrar las filas cuyo valor es **menor** que el valor suministrado
    - Mostrar las filas cuyo valor es **menor o igual** que el valor suministrado
    - Mostrar las filas cuyo valor es **igual** que el valor suministrado
    - Mostrar las filas cuyo valor es **distinto** al valor suministrado
    - Mostrar las filas cuyo valor es **mayor** que el valor suministrado
    - Mostrar las filas cuyo valor es **mayor o igual** que el valor suministrado

    Sin embargo, solo podrán realizarse estas comprobaciones si la columna no tiene valores nulos. En caso contrario, se daría una situación de error.
  - Mostrar los valores de una columna específica, referenciada por su nombre. Los elementos se mostrarán como un listado (todos en una única línea de la terminal).

- Visualizar el conjunto de filas que contienen, al menos, una celda con valor nulo. Deberá mostrarse también el índice de la fila para poder localizar fácilmente dónde hay una ausencia de datos.

Para la visualización de filas, sea una única o un conjunto dentro de un rango, mostraremos en primer lugar los nombres de las columnas en una primera línea. Tras ello, mostraremos todas las filas que sea necesario visualizar, una debajo de la anterior. Esto es, una visualización tabular como la ejemplificada en la figura correspondiente.

Notas importantes

Se recomienda el uso del entorno de depuración (*debugger*) de PyCharm para ir siguiendo paso a paso las actualizaciones que hace nuestro programa sobre el DataFrame y así poder identificar rápidamente los errores que se haya cometido en la implementación.

2.3. Documentación

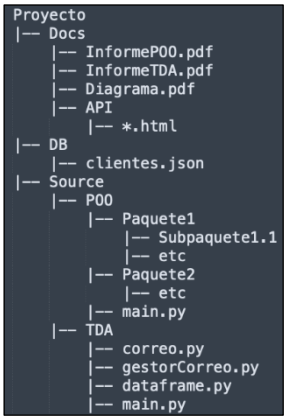
En la carpeta de la documentación (**Docs**), se entregará un documento en formato PDF llamado "**InformeTDA.pdf**" que contendrá los nombres de los autores y los siguientes apartados. Este archivo constituye la documentación formal de:

1. **Por cada TDA**:
   - **Especificación**. En relación a la especificación, documentaremos los tres apartados principales que la componen: 1) nombre y descripción, 2) dominio de los datos, y 3) especificación de las operaciones (descripción, precondiciones, información retornada y excepciones).
   - **Representación**. Describirá una estructura que indique los campos que representa el TDA. En el caso de ser necesaria una colección de datos, se debe justificar la estructura elegida. **Nota**: no se pide la función de abstracción y el invariante de la representación.
   - **Implementación**. Explicar cómo se ha programado la especificación y representación del TDA. Destacar y justificar las decisiones tomadas más importantes en relación al código. También se debe identificar claramente la parte privada y pública del TDA.
2. **Programa principal.** Describir brevemente el fichero Python main.py que permite ejecutar el programa. Explica **con tus propias palabras** las ventajas de importar y usar los tres TDAs anteriormente mencionados.

3. Normas de entrega del proyecto

La práctica deberá de presentarse por el Aula Virtual (AV), vía Tareas, en las fechas establecidas y que serán comunicadas por los profesores de la asignatura.

Todo deberá entregarse en **un único fichero comprimido en formato ZIP**. El fichero comprimido contendrá la siguiente estructura de directorios (no uses acentos o espacios en los nombres de los directorios/ficheros):



Las subcarpetas contendrán el siguiente contenido:

1. La carpeta **Docs** contiene la documentación de la práctica, que incluye:
   - **Un fichero en formato PDF** con el informe escrito por vosotros para cada una de las partes de la práctica: "InformePOO.pdf" e "InformeTDA.pdf".

     Nota importante: Los documentos deberán tener una presentación adecuada, incluyendo portada (autores, título, curso, asignatura, etc.), índice de contenidos, y cualquier otro aspecto que ayude a su buena presentación. Además, el texto deberá mostrarse con formato justificado y se debe cuidar la estética, así como hacer uso de un lenguaje correcto.
   - Una subcarpeta con la **documentación** generada automáticamente en formato .html por la herramienta **Pydoctor** (ver apartado 4) para la parte de POO. No generaremos documentación interactiva para la parte de TDAs.
   - Un **diagrama de clases UML** generado a partir del proyecto Python **en formato PDF** (ver apartado 3), también para la parte de POO únicamente.
2. La carpeta **Source** contiene el proyecto PyCharm completo, que contiene solo los archivos fuente (SIN incluir la carpeta **venv** del entorno virtual de Python). Así, habrá una carpeta para la parte de POO y otra para la de TDA, cada una con sus archivos y main.py correspondientes.
3. La carpeta **DB** contiene un fichero JSON con un listado de varios clientes dados de alta en el sistema para la primera parte de la práctica, sirviendo de base de datos para la ejecución.

4. Evaluación del proyecto

Cada parte de este proyecto constituye el 50% de la nota de prácticas de la asignatura, respectivamente.

El **plagio (total o parcial) en cualquiera de las partes presentadas para su evaluación conllevará una valoración de 0 puntos en la nota final de la convocatoria correspondiente para todos los estudiantes implicados**. Los profesores usarán diversas herramientas de carácter informático para la detección de plagio sobre cualquier entrega realizada.

4.1. Requisitos mínimos

Para aprobar el proyecto, se deben cumplir los siguientes requisitos mínimos. De no cumplir alguno de ellos, **el proyecto se puede suspender automáticamente**:

- **Requisito 1**. Entregar en tiempo, a través de la tarea pertinente, un fichero .zip con todo el proyecto, incluyendo documentación y código, de acuerdo con las normas de entrega indicadas en el apartado 3.
- **Requisito 2**. La documentación debe cumplir las normas indicadas en los apartados 1.3 y 2.3.
- **Requisito 3**. El programa debe responder a las funcionalidades que se piden en el enunciado de este documento, concretamente en los apartados 1.2 y 2.2.
- **Requisito 4 (solo aplicable a TDAs)**. El programa **no puede hacer uso de tipos de datos propios de Python basados en colecciones**. Por ejemplo, está prohibido hacer uso de los tipos de datos tales como *dict*, *set*, *tuple*. El tipo *list* solo lo podremos usar de apoyo a una representación contigua, sin poder hacer uso de sus métodos y funciones asociadas, únicamente la representación a modo de array, como se ha visto en teoría.
- **Requisito 5**. Ambos programas principales se ejecutan sin errores y lo programado funciona correctamente.

4.2. Criterios de evaluación y calificación

Los aspectos que se tendrán en cuenta durante la evaluación y calificación del trabajo presentado son los siguientes. Tanto el código como la documentación son de obligada entrega para poder evaluar la práctica.

4.2.1. Primera parte: POO (50% del total)

- **Código** (80% de la nota):
  - Funcionamiento correcto de los programas presentados (no pueden tener errores en tiempo de ejecución).
  - Originalidad de la solución al problema planteado.
  - Usar correctamente la metodología de la POO en todos los aspectos que han sido mostrados en las sesiones teórico-prácticas, destacando:
    - Empleo de la declaración de tipos explícita y uso correcto de los tipos de datos.
    - Eficiencia y correctitud de las funciones, métodos y algoritmos.
    - Definir correctamente las clases (métodos y atributos) y la visibilidad de cada miembro.
    - Uso correcto de sobrecarga de métodos.
    - Definir las relaciones entre las clases (uso, asociación, agregación, composición y herencia). Uso correcto de delegación de responsabilidades entre clases.
    - Empleo adecuado de clases abstractas e interfaces.
    - Crear una buena estructuración en paquetes y módulos de la solución construida.
  - Uso adecuado de excepciones para gestionar errores en el código.
  - Uso correcto de entrada/salida en base a los requisitos del problema.
  - Diseño y funcionamiento correcto de la interfaz gráfica.

- **Documentación** (20% de la nota):
    - El informe debe contener todos los apartados mínimos solicitados.
    - Explicación, presentación y concreción del informe de prácticas.
    - Completitud y calidad de la documentación HTML generada automáticamente (descripciones representativas, tipos de datos indicados, valores devueltos, etc).

4.2.2. Segunda parte: TDAs (50% del total)

- **Ponderación** de cada TDA sobre el total de la nota de la segunda parte:
    - TDA Correo: **20%**
    - TDA Gestor Correo: **40%**
    - TDA Dataframe: **40%**
- **Código** (70% de la nota por cada TDA):
    - Funcionamiento correcto de los programas presentados (no pueden tener errores en tiempo de ejecución).
    - Originalidad de la solución al problema planteado.
    - Usar correctamente la metodología de la POO en todos los aspectos que han sido mostrados en las sesiones teórico-prácticas, destacando:
        - Empleo de la declaración de tipos explícita y uso correcto de los tipos de datos.
        - Eficiencia y correctitud de las funciones, métodos y algoritmos.
        - Definir correctamente las clases (métodos y atributos) y la visibilidad de cada miembro.
        - Uso correcto de sobrecarga de métodos.
        - Uso correcto de delegación de responsabilidades entre clases.
    - Uso adecuado de excepciones para gestionar errores en el código.
    - Se respeta la parte pública y privada de los TDAs cuando se invoca su funcionalidad desde el main.
    - Uso de docstring para documentar el código. **Nota**: no se pide generar documentación interactiva en formato HTML.
- **Documentación** (30% de la nota por cada TDA):
    - El informe debe contener todos los apartados mínimos solicitados.
    - Explicación, presentación y concreción del informe de prácticas.

5. Entrevista

Si los profesores lo consideran necesario, podríais ser citados para realizar una entrevista.

Durante la entrevista se preguntarán cómo están realizadas determinadas partes del proyecto de programación presentado, al tiempo que se podrá solicitar la modificación de alguna parte para que haga algo diferente.

El resultado de la entrevista puede cambiar por completo la calificación obtenida en el proyecto presentado.

Anexo I: Generación de diagrama de clases UML

Este anexo documenta las instrucciones para instalar el software necesario para generar el diagrama de clases de tu proyecto:

- **Instalar pylint**: Teniendo en Pycharm el proyecto abierto, ir a Settings > Project > Python Interpreter. Pulsamos el icono de **+**, buscamos en la lista **pylint** y, finalmente, pulsamos **Install Package**. Una vez instalado, damos a **Apply** y cerramos la configuración.
- **Instalar Graphviz**:
    i. Windows:
        1. Descargar Graphviz de la web oficial. El siguiente enlace lo descarga directamente: Enlace
        2. Tras instalarlo, abre el menú de inicio de Windows y escribe "Variables de entorno". Selecciona la opción "Editar las variables de entorno del sistema". En la ventana que aparece, selecciona "Variables de entorno". En la sección de "Variables del sistema", selecciona la variable **Path** y haz clic en **Editar**. Haz click en **Nuevo** y añade la ruta donde se instaló Graphviz: *C:\Program Files\Graphviz\bin*. Guarda los cambios.
        3. Comprueba que se ha instalado correctamente. Para ello, abre una terminal en Pycharm (icono abajo a la izquierda) y escribe el siguiente comando. Debería mostrar la versión de Graphviz, indicando que se ha instalado correctamente:

            dot -version

    ii. Mac:
        1. Instala Homebriew en caso de que no lo tengas. Comprobaremos si está instalado abriendo la aplicación **Terminal** y tecleando el siguiente comando. Si nos indica la versión, está instalado, y podemos pasar al paso 3. En caso contrario, sigue al paso 2.
        2. Las instrucciones para instalar Homebrew están en el siguiente enlace: Enlace. Copiaremos el comando que aparece justo debajo de "Instala Homebiew", pulsando al icono que aparece a la derecha del comando.
        3. Abriremos la aplicación **Terminal** en el Mac, pegaremos el comando que copiamos previamente en la web, y pulsaremos Intro. Esperaremos unos minutos hasta que se complete el proceso de instalación.
        4. Instalaremos Graphviz con el siguiente comando en la Terminal:

            brew install graphviz

        5. Comprueba que se ha instalado correctamente. Para ello, abre una terminal en Pycharm (icono abajo a la izquierda) y escribe el siguiente comando. Debería mostrar la versión de Graphviz, indicando que se ha instalado correctamente:

            dot -version

- Usar el comando pyreverse para generar el diagrama UML. Para ello, abriremos la terminal de Pycharm y pegaremos el siguiente comando:

    pyreverse -f ALL -o pdf -p Diagrama .

Este comando genera un diagrama Diagrama.pdf en el que se incluyen todos los miembros independientemente de su visibilidad (privados, protegidos y públicos). *Nota*: es importante incluir el punto al final del comando para que se genere correctamente el diagrama.