

# Capítulo 7

## Tipos estructurados: clases y diccionarios

— No tendría un sabor muy bueno, me temo...  
— Solo no —le interrumpió con cierta impaciencia el Caballero— pero no puedes imaginarte qué diferencia si lo mezclas con otras cosas...

*Alicia en el país de las maravillas*, Lewis Carroll

El conjunto de tipos de datos Python que hemos estudiado se divide en tipos *escalares* (enteros y flotantes) y tipos *secuenciales* (cadenas y listas). En este capítulo aprenderemos a definir y utilizar tipos de datos definidos por nosotros mismos *componiendo* otros tipos de datos más sencillos. Los nuevos tipos de datos reciben el nombre de *clases*.

Por otra parte, los *diccionarios* permiten mantener correspondencias entre claves y valores, facilitando así la escritura de ciertos programas.

### 7.1. Tipos de datos «a medida»

#### 7.1.1. Lo que sabemos hacer

Supón que en un programa utilizamos el nombre, el DNI y la edad de dos personas. Necesitaremos tres variables para almacenar los datos de cada persona, dos variables con valores de tipo cadena (el nombre y el DNI) y otra con un valor de tipo entero (la edad):

```
1 nombre = 'Juan Pérez'  
2 dni = '12345678Z'  
3 edad = 19  
4  
5 otro_nombre = 'Pedro López'  
6 otro_dni = '23456789D'  
7 otra_edad = 18
```

Los datos almacenados en *nombre*, *dni* y *edad* corresponden a la primera persona y los datos guardados en *otro\_nombre*, *otro\_dni* y *otra\_edad* corresponden a la segunda persona, pero nada en el programa permite deducir eso con seguridad: cada dato está almacenado en una variable *diferente e independiente* de las demás. El programador debe saber en todo momento qué variables están relacionadas entre sí y en qué sentido para utilizarlas coherentemente.

Diseñemos un procedimiento que muestre por pantalla los datos de una persona y usémoslo:

```
1 def imprimir_persona_en_pantalla(nombre, dni, edad):  
2     print('Nombre:', nombre)  
3     print('DNI:', dni)  
4     print('Edad:', edad)  
5
```

```
6 imprimir_persona_en_pantalla(nombre, dni, edad)
7 imprimir_persona_en_pantalla(otro_nombre, otro_dni, otra_edad)
```

Al ejecutar ese fragmento de programa, por pantalla aparecerá:

```
Nombre: Juan Pérez
DNI: 12345678Z
Edad: 19
Nombre: Pedro López
DNI: 23456789D
Edad: 18
```

Funciona, pero resulta un tanto incómodo pasar tres parámetros cada vez que usamos el procedimiento. Si más adelante enriquecemos los datos de una persona añadiendo su domicilio, por ejemplo, tendremos que redefinir el procedimiento *imprimir\_persona\_en\_pantalla* y cambiar todas sus llamadas para incluir el nuevo dato. En un programa de tamaño moderadamente grande puede haber decenas o cientos de llamadas a la función.

Imaginemos ahora que nuestro programa gestiona la relación de personas que asiste a clase. No sabemos a priori de cuántas personas estamos hablando, así que hemos de gestionar una lista a la que iremos añadiendo cuantas personas sea necesario. Bueno, *una* lista no, sino *tres* listas «paralelas»: una para los nombres, una para los DNI y una para las edades. Entenderemos que los elementos de las tres listas que tienen el mismo índice contienen los tres datos que describen a una persona en nuestro programa. Este fragmento de código ilustra la idea:

```
1 nombre = ['Juan_Pérez', 'Pedro_López', 'Ana_García']
2 dni = ['12345678Z', '23456789D', '13577532B']
3 edad = [19, 18, 18]
4
5 for i in range(len(nombre)):
6     imprimir_persona_en_pantalla(nombre[i], dni[i], edad[i])
```

El bucle recorre con *i* los índices 0, 1 y 2 y, para cada uno, muestra los tres datos asociados a una de las personas. Por ejemplo, cuando *i* vale 1 se muestran los datos de Pedro López, que están almacenados en *nombre[1]*, *dni[1]* y *edad[1]*. Hemos ganado en comodidad (ya no hay que inventar un nombre de variable para cada dato de cada persona), pero hemos de estar atentos y mantener la coherencia entre las tres listas. Si, por ejemplo, queremos borrar los datos de Pedro López, tendremos que ejecutar tres operaciones de borrado (**del**):

```
1 del nombre[1]
2 del dni[1]
3 del edad[1]
```

Y si deseamos ordenar alfabéticamente la relación de personas por su nombre deberemos ser cuidadosos: cada intercambio de elementos de la lista *nombre*, supondrá el intercambio de los elementos correspondientes en las otras dos listas.

En resumen, es posible desarrollar programas que gestionan «personas» con esta metodología, pero resulta incómodo.

### 7.1.2. ... pero sabemos hacerlo mejor

Hay una alternativa a trabajar con grupos de tres variables independientes por persona: definir una «persona» como una lista con tres elementos. En cada elemento de la lista almacenaremos uno de sus valores, siempre en el mismo orden:

```
1 juan = ['Juan_Pérez', '12345678Z', 19]
2 pedro = ['Pedro_López', '23456789D', 18]
```

Trabajar así permite que los datos de cada persona estén agrupados, sí, pero también hace algo incómodo su uso. Deberemos recordar que el índice 0 accede al nombre, el índice 1 al DNI y el índice 2 a la edad. Por ejemplo, para acceder a la edad de Juan Pérez hemos de escribir

*juan[2]*. Es probable que cometamos algún error difícil de detectar si utilizamos los índices erróneamente. Podríamos facilitar el trabajo almacenando los índices en unas variables cuyos identificadores permitan recordar sus respectivos «significados»:

```
1 nombre = 0
2 dni = 1
3 edad = 2
4
5 juan = ['Juan_Pérez', '12345678Z', 19]
6 pedro = ['Pedro_López', '23456789D', 18]
```

Ahora, la edad de Juan Pérez se puede obtener escribiendo *juan[edad]*, que es bastante más elegante que *juan[2]*. La función que muestra por pantalla todos los datos de una persona tendría este aspecto:

```
1 def imprimir_persona_en_pantalla(persona):
2     print('Nombre:', persona[nombre])
3     print('DNI:', persona[dni])
4     print('Edad:', persona[edad])
```

Este procedimiento solo tiene un parámetro, así que, si añadimos nuevos datos a una persona, solo modificaremos el cuerpo del procedimiento, pero no todas y cada una de sus llamadas. Hemos mejorado, pues, con respecto a la solución desarrollada en el apartado anterior.

Siguiendo esta filosofía, también es posible tener listas de personas, que no serán más que listas de listas:

```
1 juan = ['Juan_Pérez', '12345678Z', 19]
2 pedro = ['Pedro_López', '23456789D', 18]
3
4 personas = [juan, pedro]
```

### Continuará

Seguro que a estas alturas ya te has encontrado con numerosas ocasiones en las que no te cabe una línea de programa Python en el ancho normal de la pantalla. No te preocupes: puedes partit una línea Python en varias para aumentar la legibilidad, aunque deberás indicarlo explícitamente. Una línea que finaliza con una barra invertida continúa en la siguiente:

```
1 a = 2 + \
2   2
```

¡Ojo! La línea debe *acabar* en barra invertida, es decir, el carácter que sigue a la barra invertida \ debe ser el salto de línea (que es invisible). Si a la barra invertida le sigue un espacio en blanco, Python señalará un error.

O, directamente:

```
1 personas = [ ['Juan_Pérez', '12345678Z', 19], \
2               ['Pedro_López', '23456789D', 18] ]
```

(Si te sorprende la barra invertida al final de la primera línea, lee el cuadro «Continuará»).

El nombre de Pedro López, por ejemplo, está accesible en *personas[1][nombre]*. Si deseamos mostrar el contenido completo de la lista podemos hacer:

```
1 for persona in personas:
2     imprimir_persona_en_pantalla(persona)
```

Solucionado. Bueno, no del todo. Si trabajamos con esta metodología nos aguardan nuevos problemas. Imagina que nuestro programa gestiona información sobre personas y coches y que de los coches necesitamos los siguientes datos: marca, modelo, matrícula y edad.

```
1 # Índices para personas
2 nombre = 0
3 dni = 1
4 edad = 2
5
6 juan = ['Juan Pérez', '12345678Z', 19]
7 pedro = ['Pedro López', '23456789D', 18]
8
9 # Índices para coches
10 modelo = 0
11 marca = 1
12 matrícula = 2
13 edad = 3      # Mal: edad ya estaba definida antes y valía 2
14
15 mi_coche = ['Biscúter', 'GTI', 'CSU0000UA', 42]
```

La variable *edad* ya estaba «ocupada» por «personas» y valía 2, así que asignarle ahora un valor distinto provocará errores cuando accedamos a la edad de una persona. ¿Cómo hacemos ahora para que la edad de un coche no se confunda con la edad de una persona? Tendremos que optar por:

- definir una nueva variable con otro nombre, por ejemplo, *edad\_coche*, en la que almacenamos el índice correspondiente (el valor 3),
- o almacenar la edad del coche en la posición de índice 2 de la lista de datos del coche (es decir, poner la matrícula en la última posición de la lista y la edad en la penúltima).

Cualquiera de las dos posibilidades es «antinatural», pues estamos siendo obligados a tomar decisiones acerca de cómo representar una información (coche) motivadas por la existencia de otro tipo de información (persona) que nada tiene que ver con la primera. No es una buena solución.

### 7.1.3. Lo que haremos: usar tipos de datos «a medida»

Lo ideal sería que Python proporcionara un tipo de datos básico «persona» del mismo modo que proporciona datos enteros, flotantes, listas, etc. Una variable cuyo contenido fuera de tipo «persona» albergaría en un solo paquete toda la información propia de una persona: su nombre, su dni y su edad. Pero, claro, pronto pediremos que Python disponga de un tipo de datos «coche» (con la marca, el modelo, la matrícula y la edad, por ejemplo), de un tipo de datos «empleado» (con el nombre, nómina, categoría profesional y dirección de una persona), de un tipo de datos «profesor» (con su nombre, DNI, asignaturas impartidas y horarios de tutorías), etc., es decir, pediremos que Python incorpore un tipo de datos para cada conjunto de datos hipotéticamente necesario en nuestros programas.

Python no puede anticiparse a cualquier necesidad de cualquier programador proporcionando infinitos tipos de datos, pero sí nos permite *definir nuevos tipos de datos* combinando tipos de datos existentes.

Podemos definir un tipo de datos nuevo, digamos *Persona*, que agrupe en un solo paquete los datos básicos que lo forman: el nombre (una cadena), el DNI (otra cadena) y la edad (un entero). Fíjate en que el nuevo tipo de datos es una composición de tipos de datos existentes. Los nuevos tipos de datos recibirán el nombre genérico de *clases*.

Definir nuevos tipos de datos nos obligará a aprender nuevas construcciones sintácticas del lenguaje Python, pero antes será mejor que veamos con algunos ejemplos cómo se usarán los nuevos tipos. Supongamos que hemos definido la clase *Persona*. «Crearemos» nuevas personas así:

```
1 juan = Persona('Juan_Pérez', '12345678Z', 19)
2 pedro = Persona('Pedro_López', '23456789D', 18)
```

Si necesitamos acceder al nombre, DNI o edad de Juan Pérez, podremos hacerlo así:

```
>>> print(juan.nombre)↵
Juan Pérez
>>> print(juan.dni)↵
12345678Z
>>> print(juan.edad)↵
19
```

Fíjate: la variable *juan* contiene un dato de tipo *Persona* que, en realidad, son tres datos (el nombre, el DNI y la edad). Cada uno de dichos datos recibe el nombre de *atributo* o *campo*. Puedes consultar el valor de cada campo por su nombre separándolo del identificador de la variable con un punto.

El nuevo tipo de datos «sabrá» mostrarse en pantalla con la función *print*:

```
>>> print(juan)↵
Nombre: Juan Pérez
DNI: 12345678Z
Edad: 19
```

¡Mucho mejor que definir una función *imprime\_persona\_en\_pantalla*!

Y aún más. Por ejemplo, imagina que necesitamos consultar con cierta frecuencia las iniciales de una persona. Podremos definir una función especial que efectúe el cálculo correspondiente a partir del nombre de la persona:

```
>>> print(juan.iniciales())↵
J. P.
```

Observa que *iniciales* se usa casi como si fuera el nombre de un campo de *juan*, pues se separa del identificador de la variable con un punto, pero se diferencia en el uso de paréntesis al final del identificador. Los paréntesis indican que *iniciales* es una función especial, un *método* que forma parte del tipo *Persona* y que lo estamos llamando (del mismo modo que se llama a una función). De todos modos, el uso de *iniciales* no debería resultarte demasiado extraño pues ya hemos usado métodos antes: al añadir un *dato* a una *lista* escribíamos *lista.append(dato)*, es decir, llamábamos sobre *lista* al método *append* con el dato que deseábamos añadir.

Para acabar esta exposición introductoria abordaremos el caso de la relación de estudiantes que asisten a clase y que antes resolvimos usando tres listas paralelas. Al disponer ahora de un tipo de datos *Persona* solo es necesario disponer de una lista:

```
1 alumnos = [Persona('Antonio_Pérez', '98761234Q', 20), \
2           Persona('Juan_Pérez', '12345678Z', 19), \
3           Persona('Pedro_López', '23456789D', 18)]
```

El siguiente fragmento de código mostrará los datos de todas las personas de la lista:

```
1 for i in range(len(alumnos)):
2     print(alumnos[i])
```

O, alternativamente:

```
1 for alumno in alumnos:
2     print(alumno)
```

Elegante, ¿no?

## 7.2. Definición de clases en Python

Vamos a definir un nuevo tipo de datos Python: la clase *Persona*. Presta atención a la sintaxis: es un poco enrevesada a primera vista.

```
persona.py
1 class Persona:
2     def __init__(self, nombre, dni, edad):
3         self.nombre = nombre
4         self.dni = dni
5         self.edad = edad
```

Analicémosla por partes. La primera línea empieza con la palabra reservada `class` y con ella indicamos que comienza la definición de un nuevo tipo de datos, de una nueva clase. A continuación aparece el identificador del nuevo tipo (*Persona*) y dos puntos. Las líneas 2 y siguientes empiezan más a la derecha y definen una función (empieza por `def`) llamada `__init__`. Las funciones que definimos dentro de una clase se denominan *métodos*. El método `__init__` tiene cuatro parámetros: el primero se llama `self`. *Todos los métodos tendrán como primer parámetro uno llamado self*. Le siguen tantos parámetros como campos tiene una variable del tipo *Persona*.<sup>1</sup> El cuerpo del método `__init__` (líneas 3 a 5) consiste en una serie de asignaciones de la forma:

```
self.atributo = parámetro
```

¿Qué es `self`? En inglés, «`self`» significa «uno mismo». Al asignar a `self.nombre` el valor del parámetro `nombre` estamos diciendo algo así como «el *nombre* de uno mismo es el valor del parámetro *nombre*». Es la forma de almacenar información en «uno mismo».

Ya puedes almacenar personas en variables. Cada vez que quieras crear una nueva persona, deberás hacerlo así:

```
1 toni = Persona('Antonio Pérez', '98761234Q', 20)
```

Cada vez que creas o «construyes» una nueva persona, Python llama automáticamente al método `__init__`. El método `__init__` es el denominado *constructor* de la clase *Persona*. Python interpreta esa sentencia como:

```
1 toni.nombre = 'Antonio Pérez'
2 toni.dni = '98761234Q'
3 toni.edad = 20
```

pues `self` equivale a `toni` en el ejemplo.

Cada persona creada es una *instancia* u *objeto* de la clase *Persona*. Puedes utilizar objetos de la clase *Persona* del mismo modo que utilizabas valores de otros tipos. Por ejemplo, puedes crear «personas» y almacenarlas en variables y/o en listas, a voluntad:

```
1 toni = Persona('Antonio Pérez', '98761234Q', 20)
2 juan = Persona('Juan Pérez', '12345678Z', 19)
3 pedro = Persona('Pedro López', '23456789D', 18)
4 alumnos = [toni, juan, pedro]
```

Si deseas acceder a la edad de `toni`, podrás utilizar la notación introducida en el apartado anterior:

```
>>> print(toni.edad)
20
>>> print(alumnos[0].edad)
20
```

Puedes acceder a los elementos de la lista `alumnos` como siempre. Este fragmento de programa, por ejemplo, muestra el dni de los integrantes de la lista:

<sup>1</sup>Más adelante veremos que no es necesario que haya tantos parámetros como atributos.

## POO

Este apartado constituye una introducción a uno de los aspectos básicos de la Programación Orientada a Objetos (POO). La POO es un paradigma (una forma) de programar muy extendida en las últimas décadas. Los otros dos pilares de la POO son la *herencia* y el *polimorfismo*, aunque no los trataremos en este curso.

Python es un lenguaje orientado a objetos, es decir, da soporte a las características propias de la POO. Entre los lenguajes orientados a objetos de uso más extendido se cuentan C++ y Java.

```
1 for alumno in alumnos:  
2     print(alumno.dni)
```

Alternativamente puedes recorrer la lista así:

```
1 for i in range(len(alumnos)):  
2     print(alumnos[i].dni)
```

Observa que en este caso hemos aplicado primero el operador de indexación a la lista y luego hemos accedido al campo *dni*. El contenido de *alumnos[i]* es una *Persona*, así que podemos añadir *.dni* para acceder a su campo *dni*.

► 389 Diseña un programa que pida por teclado los datos de varias personas y los añada a una lista inicialmente vacía. Cada vez que se lean los datos de una persona el programa preguntará si se desea continuar introduciendo nuevas personas. Cuando el usuario responda que no, el programa se detendrá.

► 390 Modifica el programa del ejercicio anterior para que, a continuación, muestre el nombre de la persona de más edad. Si dos o más personas tienen la mayor edad, el programa mostrará el nombre de todas ellas.

Dotemos a los objetos de la clase *Persona* de cierta «inteligencia»: hagamos que sepan devolvernos las iniciales de su nombre. Definiremos un método *iniciales* que devuelva una cadena:

```
persona.py  
1 class Persona:  
2     def __init__(self, nombre, dni, edad):  
3         self.nombre = nombre  
4         self.dni = dni  
5         self.edad = edad  
6  
7     def iniciales(self):  
8         cadena = ''  
9         for carácter in self.nombre:  
10             if carácter >= 'A' and carácter <= 'Z':  
11                 cadena = cadena + carácter + '.'  
12  
13     return cadena
```

Ahora nos detendremos a explicar paso a paso cómo hemos definido el nuevo método, pero antes, veamos si funciona:

```
>>> print(juan.iniciales())  
J. P.
```

¡Perfecto! Ya te habíamos dicho que todos los métodos tienen un primer parámetro llamado *self*, e *iniciales* no es una excepción. Cuando efectuamos una llamada a un método siempre lo haremos «sobre» una variable del tipo *Persona* y, en ese caso, *self* se interpreta como dicha variable. Cuando hemos ejecutado *juan.iniciales()*, el parámetro *self* se ha interpretado como *juan*, así que el acceso a *self.nombre* es, en ese caso, un acceso a *juan.nombre*, es decir, un acceso al atributo *nombre* de «uno mismo».

► 391 Diseña un método que permita determinar si una persona es menor de edad devolviendo cierto, si la edad es menor que 18, o falso, en caso contrario.

► 392 Diseña un método *nombre\_de\_pila* que devuelva el nombre de pila de una *Persona*. Supondremos que el nombre de pila es la primera palabra del atributo *nombre* (es decir, que no hay nombres compuestos).

¿Qué pasa si mostramos con *print* una variable de tipo *Persona*?

```
>>> print(juan)
<__console__.Persona object at 0x3cc2a50>
```

Mal. No sale lo que esperamos. Definamos un nuevo método que permitirá imprimir objetos de la clase *Persona*:

```
persona.py
1 class Persona:
2     def __init__(self, nombre, dni, edad):
3         self.nombre = nombre
4         self.dni = dni
5         self.edad = edad
6
7     def iniciales(self):
8         cadena = ''
9         for carácter in self.nombre:
10             if carácter >= 'A' and carácter <= 'Z':
11                 cadena = cadena + carácter + '.'
12
13     def __str__(self):
14         cadena = 'Nombre:{}\n'.format(self.nombre)
15         cadena = cadena + 'DNI:{}\n'.format(self.dni)
16         cadena = cadena + 'Edad:{}\n'.format(self.edad)
17
18     return cadena
```

Mmmm. Un método llamado *\_\_str\_\_*. ¡Qué nombre tan extraño! Bueno, ¿funciona ahora?

```
>>> print(juan)
Nombre: Juan Pérez
DNI: 12345678Z
Edad: 19
```

¡Ahora sí! Ciertos métodos tienen nombres especiales, como *\_\_init\_\_* y *\_\_str\_\_*. Python espera que un método con un nombre especial haga algo concreto. Por ejemplo, un método llamado *\_\_init\_\_* debe construir un objeto de la clase *Persona* y un método llamado *\_\_str\_\_* debe devolver una *cadena* con lo que queremos que se muestre por pantalla (el *str* del nombre del método es una abreviatura de «string», es decir, «cadena» en inglés). En nuestro caso, la cadena que hemos formado contiene todos los datos de una *Persona*. Lo realmente curioso acerca de los métodos especiales es que no tienes por qué llamarlos directamente: Python los llama automáticamente en ciertos casos. Por ejemplo, cuando haces *print* de un objeto de la clase *Persona*, Python le «pregunta» al objeto si tiene definido el método *\_\_str\_\_* y, si es así, muestra el resultado de ejecutar dicho método sobre el objeto. Como el resultado de ejecutar *\_\_str\_\_* es una cadena, Python muestra por pantalla esa cadena.

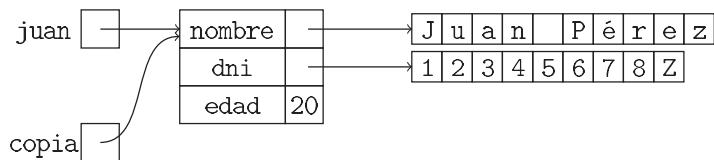
► 393 Modifica el programa del ejercicio anterior enriqueciendo el tipo de datos *Persona* con un nuevo campo: el sexo, que codificaremos con una letra ('M' para mujer y 'V' para varón). Añade a tu programa un método *\_\_str\_\_* que también imprima en pantalla cuál es el sexo de la persona.

### 7.2.1. Referencias y objetos

Hemos de tratar ahora un problema que ya nos es conocido. Fíjate bien:

```
>>> juan = Persona('Juan Pérez', '12345678Z', 19)
>>> copia = juan
>>> copia.edad = 20
>>> print(copia.edad)
20
>>> print(juan.edad)
20
```

¡Los cambios a *copia* afectan a *juan*! Estamos ante el mismo problema que apareció al trabajar con listas: la asignación de un objeto a otro *solo copia la referencia*, y no el contenido.



No solo la asignación se ve afectada por este hecho: también el paso de parámetros se efectúa transmitiendo a la función una referencia al objeto, así que *los cambios realizados a un objeto dentro de una función son «visibles» fuera, en el objeto pasado como argumento*.

Al trabajar con listas pudimos solucionar el problema obteniendo una copia de la lista a asignar con el operador de corte o la concatenación. Pero el operador de corte no tiene significado alguno para nuestros objetos. ¿Cómo solucionar el problema? Lo normal es que definamos un método capaz de generar una copia de nuestro objeto y lo usemos cuando sea menester:

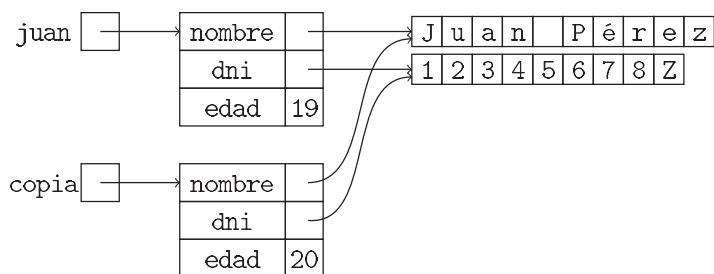
```
persona.py
1 class Persona:
2
3     ...
4
5     def copia(self):
6         nuevo = Persona(self.nombre, self.dni, self.edad)
7         return nuevo
```

Observa que hemos creado y devuelto una nueva *Persona* cuyos atributos tienen los mismos valores que tiene *self*, es decir, la *Persona* sobre la que invocamos el método.

Repitamos la prueba anterior:

```
>>> juan = Persona('Juan Pérez', '12345678Z', 19)
>>> copia = juan.copia()
>>> copia.edad = 20
>>> print(copia.edad)
20
>>> print(juan.edad)
19
```

¡Ahora sí! ¿Cómo ha quedado la memoria en este caso? Observa detenidamente la siguiente figura:

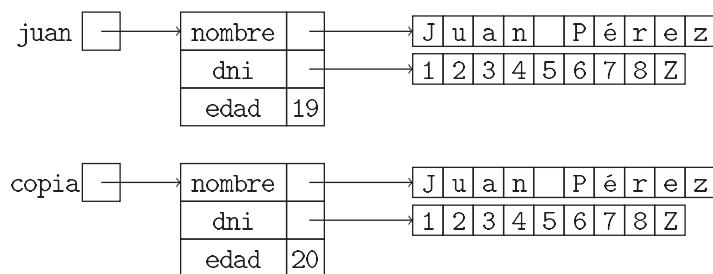


La verdad es que *juan* y *copia* ahora apuntan a objetos de la clase *Persona* diferentes, pero no completamente independientes: siguen compartiendo la memoria de las cadenas! ¿Por qué? Recuerda que la asignación de una cadena a otra se traduce en la copia de la referencia, no del contenido, y cuando se ejecuta *copia.nombre = juan.nombre*<sup>2</sup> el valor de *copia.nombre* es una referencia a la memoria apuntada por *juan.nombre*.

Como las cadenas son inmutables en Python, no hay problema alguno en que *juan* y *copia* comparten la memoria de sus campos *nombre* y *dni*. Aun así, si quisieramos que ambos tuvieran su propia zona de memoria para estos datos, deberíamos modificar el método de copia de la clase *Persona*:

```
persona.py
1 class Persona:
2
3     ...
4
5     def copia(self):
6         nuevo = Persona(self.nombre[:], self.dni[:], self.edad)
7         return nuevo
```

Tras ejecutar las sentencias del ejemplo con el nuevo método *copia* tenemos:



La gestión de la memoria es un asunto delicado y la mayor parte de los errores graves de programación están causados por un inapropiado manejo de la memoria. Python simplifica mucho dicha gestión, pero un programador competente debe saber qué ocurre exactamente en memoria cada vez que se maneja una cadena, lista u objeto.

► 394 ¿Qué mostrará por pantalla la ejecución del siguiente programa?

```
1 class Persona:
2
3     ...
4
5     def nada_útil(persona1, persona2):
6         persona1.edad += 1
7         persona3 = persona2
8         persona4 = persona2.copia()
9         persona3.edad -= 1
10        persona4.edad -= 2
11        return persona4
12
13 juan = Persona('Juan Pérez', '12345679Z', 19)
14 pedro = Persona('Pedro López', '23456789D', 18)
15 otro = nada_útil(juan, pedro)
16 print(juan)
17 print(pedro)
18 print(otro)
```

<sup>2</sup>En realidad, esta sentencia de asignación aparece en el programa expresada como *self.nombre = nombre* en el método *\_\_init\_\_* que se ejecuta al construir *copia*.

Haz un diagrama que muestre el estado de la memoria en los siguientes instantes:

- justo antes de ejecutar la línea 14,
  - justo antes de ejecutar la línea 10 en la invocación de *nada útil* desde la línea 14,
  - al finalizar la ejecución del programa.
- 

### 7.2.2. Un ejemplo: gestión de calificaciones de estudiantes

Desarrollemos un ejemplo completo. Vamos a diseñar un programa que gestiona la lista de estudiantes de una asignatura y sus calificaciones. De cada estudiante guardaremos su nombre, su grupo de teoría (que será la letra A, B o C), la nota obtenida en el examen y si ha entregado o no las prácticas. Tener aprobada la asignatura implica haber entregado las prácticas y haber obtenido en el examen una nota igual o superior a 5.

Deseamos hacer un programa que permita añadir estudiantes a la lista, mostrar la calificación de cada uno de ellos y efectuar algunas estadísticas sobre las notas, como obtener la nota media o el porcentaje de estudiantes que ha entregado las prácticas.

Definamos primero el tipo de datos *Estudiante*. Cada estudiante tiene cuatro campos (*nombre*, *grupo*, *nota* y *práctica*):

```
notas.py
1 class Estudiante:
2     def __init__(self, nombre, grupo, nota, práctica):
3         self.nombre = nombre
4         self.grupo = grupo
5         self.nota = nota
6         self.práctica = práctica
```

Los campos *nombre* y *grupo* serán cadenas, el campo *nota* será un flotante con el valor numérico de la evaluación del examen y el valor del campo *práctica* será *True* si la entregó y *False* en caso contrario.

Sería interesante definir una función que leyera por teclado los datos de un estudiante y nos devolviera un nuevo objeto *Estudiante* con sus campos cumplimentados.

```
notas.py
1 def lee_estudiante():
2     nombre = input('Nombre:')
3     grupo = input('Grupo (A,B,C):')
4     nota = float(input('Nota de examen:'))
5     entregada = input('Práctica entregada(s/n):')
6     práctica = entregada == 's'
7     return Estudiante(nombre, grupo, nota, práctica)
```

Ojo! *lee\_estudiante* no es un método, sino una función, y como tal se define fuera de la clase *Estudiante*. La función lee de teclado el valor de cada campo y construye un nuevo *Estudiante*, que es el valor que devuelve. Podemos pedir al usuario de nuestro programa que introduzca los datos de un estudiante así:

```
1 nuevo_estudiante = lee_estudiante()
```

El contenido de *nuevo\_estudiante* es un objeto de la clase *Estudiante*.

Diseñemos ahora una función que, dada una lista de estudiantes (posiblemente vacía), pida los datos de un estudiante y añada el nuevo estudiante a la lista:

```
notas.py
1 def lee_y_añade_estudiante(lista):
2     estudiante = lee_estudiante()
```

```
3     lista.append(estudiante)
```

Definamos ahora el método `__str__` para poder imprimir en pantalla un estudiante:

```
notas.py
1 class Estudiante:
2     ...
3
4     def __str__(self):
5         cadena = 'Nombre: {}{}'.format(self.nombre)
6         cadena = cadena + 'Grupo: {}'.format(self.grupo)
7         cadena = cadena + 'Nota examen: {:.3f}{}'.format(self.nota)
8         if self.practica:
9             cadena = cadena + 'Práctica entregada'
10            else:
11                cadena = cadena + 'Práctica no entregada'
12
13    return cadena
```

---

► 395 Diseña un procedimiento que, dada una lista de estudiantes, muestre por pantalla los datos de todos ellos.

► 396 Diseña un procedimiento que, dada una lista de estudiantes y un grupo (la letra A, B o C), muestre por pantalla un listado completo de dicho grupo.

---

Ahora nos gustaría conocer la calificación de un estudiante: Matrícula de Honor, Notable, Aprobado o Suspenso. No existe un campo *calificación* en los objetos de la clase *Estudiante*, así que deberemos implementar un método que efectúe los cálculos pertinentes a partir del valor de *práctica* y del valor de *nota*:

```
notas.py
1 class Estudiante:
2     ...
3
4     def calificación(self):
5         if not self.práctica:
6             return 'Suspenso'
7         else:
8             if self.nota < 5:
9                 return 'Suspenso'
10            elif self.nota < 7:
11                return 'Aprobado'
12            elif self.nota < 8.5:
13                return 'Notable'
14            elif self.nota < 10:
15                return 'Sobresaliente'
16            else:
17                return 'Matrícula de Honor'
```

Probemos si funciona:

```
>>> pepe = Estudiante('Pepe García', 'A', 7.7, True)
>>> print(pepe.calificación())
Notable
```

---

► 397 Define un método `está_aprobado` que devuelva `True` si el alumno ha aprobado la asignatura y `False` en caso contrario.

---

Podemos escribir ahora una función que muestre el nombre y la calificación de todos los estudiantes:

```
notas.py
1 def acta(lista):
2     for estudiante in lista:
3         print(estudiante.nombre, estudiante.calificación)
```

Si queremos obtener algunas estadísticas, como la nota media o el porcentaje de estudiantes que ha entregado las prácticas, definiremos y usaremos nuevas funciones:

```
notas.py
1 def nota_media(lista):
2     suma = 0
3     cantidad = 0
4     for estudiante in lista:
5         if estudiante.práctica:
6             suma += estudiante.nota
7             cantidad += 1
8     if cantidad != 0:
9         return suma / cantidad
10    else:
11        return 0.0
12
13 def porcentaje_de_prácticas_entregadas(lista):
14     if len(lista) != 0:
15         cantidad = 0
16         for estudiante in lista:
17             if estudiante.práctica:
18                 cantidad += 1
19         return cantidad / len(lista) * 100
20     else:
21         return 0.0
```

---

► 398 Diseña una función que devuelva el porcentaje de aprobados sobre el total de estudiantes (y no sobre el total de estudiantes que han entregado la práctica).

► 399 Diseña una función que reciba una lista de estudiantes y el código de un grupo (la letra A, B o C) y devuelva la nota media en dicho grupo.

---

Y esta otra función, por ejemplo, devuelve una lista con los estudiantes que obtuvieron la nota más alta:

```
notas.py
1 def mejores_estudiantes(lista):
2     nota_más_alta = 0
3     mejores = []
4     for estudiante in lista:
5         if estudiante.práctica:
6             if estudiante.nota > nota_más_alta:
7                 mejores = [estudiante]
8                 nota_más_alta = estudiante.nota
9             elif estudiante.nota == nota_más_alta:
10                 mejores.append(estudiante)
11
12 return mejores
```

Fíjate en que *mejores\_estudiantes* devuelve una lista cuyos componentes son objetos de la clase *Estudiante*. Si deseas listar por pantalla los nombres de los mejores estudiantes, puedes hacer lo siguiente:

```
1 los_mejores = mejores_estudiantes(lista)
2 for estudiante in los_mejores:
3     print(estudiante.nombre)
```

o, directamente:

```
1 for estudiante in mejores_estudiantes(lista):
2     print(estudiante.nombre)
```

---

► 400 Deseamos realizar un programa que nos ayude a gestionar nuestra colección de ficheros MP3. Cada fichero MP3 contiene una canción y deseamos almacenar en nuestra base de datos la siguiente información de cada canción:

- título,
- intérprete,
- duración en segundos,
- estilo musical.

Empieza definiendo la clase *MP3* y su método *\_\_init\_\_*. Cuando lo tengas, define dos nuevos métodos:

- *resumen*: devuelve una cadena con solo el título y el intérprete (en una sola línea).
- *\_\_str\_\_*: devuelve una cadena con todos los datos del fichero MP3 de modo que cada campo ocupe una línea.

A continuación, diseña cuantos métodos y funciones consideres pertinentes para implementar las siguientes acciones:

- a) añadir una nueva canción a la base de datos (que será una lista de objetos de la clase *MP3*),
- b) listar todas las canciones de un intérprete determinado (en formato resumido),
- c) listar todas las canciones de un estilo determinado (en formato resumido),
- d) listar todas las canciones de la base de datos (en formato completo),
- e) eliminar una canción de la base de datos dado el título y el intérprete.

(Nota: Si quieres que el programa sea realmente útil, sería interesante que pudieras salvar la lista de canciones a disco duro; de lo contrario perderás todos los datos cada vez que salgas del programa. En el próximo capítulo aprenderemos a guardar datos en disco y a recuperarlos, así que este programa solo te resultará realmente útil cuando hayas estudiado ese capítulo. De momento, si quieres usarlo ya, puedes utilizar el módulo *pickle* que describimos sucintamente en el capítulo anterior).

---

### 7.3. Algunas clases de uso común

Muchas aplicaciones utilizan ciertos tipos de datos estructurados. Un principio de diseño es la reutilización de código, es decir, no reescribir lo que ya hemos implementado cada vez que necesitemos usarlo. Nos vendrá bien disponer de módulos en los que hayamos implementado estas clases de datos. De ese modo, cada aplicación que necesite utilizar el tipo de datos en cuestión, solo tendrá que importar la clase correspondiente del módulo.

### 7.3.1. La clase fecha

Python no dispone de un tipo de datos fecha, y la verdad es que nos vendría bien en numerosas aplicaciones. Vamos a diseñar una clase *Fecha* y la implementaremos en un módulo *fecha* (es decir, en un fichero *fecha.py*).

Una fecha tiene tres valores: día, mes y año. Codificaremos cada uno de ellos con un número entero.

```
fecha.py
1 class Fecha:
2     def __init__(self, dia, mes, año):
3         self.dia = dia
4         self.mes = mes
5         self.año = año
```

Mmmm. Seguro que nos viene bien un método que muestre por pantalla una fecha.

```
fecha.py
1 class Fecha:
2     ...
3
4     def __str__(self):
5         return '{0}/{1}/{2}'.format(self.dia, self.mes, self.año)
```

---

► 401 Define un método llamado *formato\_largo* que devuelva la fecha en un formato más verboso. Por ejemplo, el 15/4/2002 aparecerá como 15 de abril de 2002.

Definamos ahora un método que indica si un año es bisiesto o no. Recuerda que un año es bisiesto si es divisible por 4, excepto si es divisible por 100 pero no por 400:

```
fecha.py
1 class Fecha:
2     ...
3
4     def en_año_bisiesto(self):
5         return self.año % 4 == 0 and (self.año % 100 != 0 or self.año % 400 == 0)
```

---

► 402 Diseña un método *válida* que devuelva cierto si la fecha es válida y falso en caso contrario. Para ello, debes comprobar que el mes esté comprendido entre 1 y 12 y que el día esté comprendido entre 1 y el número de días que corresponde al mes. Por ejemplo, la fecha 31/4/2000 no es válida, ya que abril tiene 30 días.

Ten especial cuidado con el mes de febrero: ¡tiene 28 o 29 días según el año! Usa, si te conviene, el método definido en el ejercicio anterior haciendo *self.en\_año\_bisiesto()*.

Diseñemos ahora una función (no un método) que lee una fecha por teclado y nos la devuelve:

```
fecha.py
1 class Fecha:
2     ...
3
4     def lee_fecha():
5         dia = 0
6         while dia < 1 or dia > 31:
7             dia = int(input('Día:'))
8         mes = 0
9         while mes < 1 or mes > 12:
10            mes = int(input('Mes:'))
11        año = int(input('Año:'))
12        return Fecha(dia, mes, año)
```

- 
- 403 Modifica la función *lee\_fecha* para que solo acepte fechas válidas, es decir, fechas cuyo día sea válido para el mes leído. Puedes utilizar el método *válida* desarrollado en el ejercicio anterior.

Nos gustaría comparar dos fechas para saber si una es menor que otra. Podemos diseñar una función o un método. Implementemos ambas y así remarcaremos las diferencias entre función y método. Empecemos por la función:

```
fecha.py
1 def fecha_es_menor(fecha1, fecha2):
2     if fecha1.año < fecha2.año:
3         return True
4     elif fecha1.año > fecha2.año:
5         return False
6     if fecha1.mes < fecha2.mes:
7         return True
8     elif fecha1.mes > fecha2.mes:
9         return False
10    return fecha1.día < fecha2.día
```

Si en un programa deseamos comparar dos fechas, *f1* y *f2*, lo haremos así:

```
1 ...
2 if fecha_es_menor(f1, f2):
3     ...
```

Vamos ahora a por el método:

```
fecha.py
1 class Fecha:
2     ...
3
4     def es_menor_que(self, la_otra_fecha):
5         if self.año < la_otra_fecha.año:
6             return True
7         elif self.año > la_otra_fecha.año:
8             return False
9         if self.mes < la_otra_fecha.mes:
10            return True
11        elif self.mes > la_otra_fecha.mes:
12            return False
13        return self.día < la_otra_fecha.día
```

Observa que también tiene dos parámetros, pero el primero es *self*, es decir, «uno mismo», y el otro es la segunda fecha, «la otra». ¿Cómo se usa el método? Así:

```
1 ...
2 if f1.es_menor_que(f2):
3     ...
```

A gusto del consumidor.

- 
- 404 Diseña un método que devuelva cierto si dos fechas son iguales y falso en caso contrario.

- 405 Diseña un método *añade\_un\_día* que añade un día a una fecha dada. La fecha 7/6/2001, por ejemplo, pasará a ser 8/6/2002 tras invocar al método *añade\_un\_día* sobre ella.

### ¿Cuántos días han pasado... dónde?

Trabajar con fechas tiene sus complicaciones. Una función que calcule el número de días transcurridos entre dos fechas cualesquiera no es trivial. Por ejemplo, la pregunta no se puede responder si no te dan otro dato: ¡el país! ¿Sorprendido? No te vendrá mal conocer algunos hechos sobre el calendario.

Para empezar, no existe el año cero, pues el cero se descubrió en Occidente bastante más tarde (en el siglo IX fue introducido por los árabes, que lo habían tomado previamente del sistema indio). El año anterior al 1 d. de C. (después de Cristo) es el 1 a. de C. (antes de Cristo). En consecuencia, el día siguiente al 31 de diciembre de 1 a. de C. es el 1 de enero de 1 d. de C. (Esa es la razón de que el siglo XXI empezara el 1 de enero de 2001, y no de 2000, como erróneamente creyó mucha gente).

Julio César, en el año 46 a. C. difundió el llamado calendario juliano. Hizo que los años empezaran en 1 de januarius (el actual enero) y que los años tuvieran 365 días, con un año bisiesto cada 4 años, pues se estimaba que el año tenía 365,25 días. El día adicional se introducía tras el 23 de febrero, que entonces era el sexto día de marzo, con lo que aparecía un día "bis-sextu" (o sea, un segundo día sexto) y de ahí viene el nombre bisiesto de nuestros años de 366 días. Como la reforma se produjo en un instante en el que ya se había acumulado un gran error, Julio César decidió suprimir 80 días de golpe.

Pero la aproximación que del número de días de un año hace el calendario juliano no es exacta (un año dura en realidad 365,242198 días, 11 minutos menos de lo estimado) y comete un error de 7,5 días cada 1000 años. En 1582 el papa Gregorio XIII promovió la denominada reforma gregoriana con objeto de corregir este cálculo inexacto. Este papa suprimió los bisiestos seculares (los que corresponden a años divisibles por 100), excepto los que caen en años múltiplos de 400, que siguieron siendo bisiestos. Para cancelar el error acumulado por el calendario juliano, Gregorio XIII suprimió 10 días de 1582: el día siguiente al 4 de octubre de 1582 fue el 15 de octubre de 1582. Como la reforma fue propuesta por un papa católico, tardó en imponerse en países protestantes u ortodoxos. Inglaterra, por ejemplo, tardó 170 años en adoptar el calendario gregoriano. En 1752 ya se había producido un nuevo día de desfase entre el cómputo juliano y el gregoriano, así que se suprimieron 11 días del calendario: al 2 de septiembre de 1752 siguió en Inglaterra el 14 de septiembre del mismo año. Por otra parte, Rusia no adoptó el nuevo calendario hasta ¡1918!, así que la revolución de su octubre de 1917 tuvo lugar en nuestro noviembre de 1917. Y no fue Rusia el último país occidental en adoptar el calendario gregoriano: Rumanía aún tardó un año más.

Por cierto, el calendario gregoriano no es perfecto: cada 3000 años (aproximadamente) se desfasa en 1 día. ¡Menos mal que no nos tocará vivir la próxima reforma!

Atención al último día de cada mes, pues su siguiente día es el primero del mes siguiente. Similar atención requiere el último día del año. Debes tener en cuenta que el día que sigue al 28 de febrero es el 29 del mismo mes o el 1 de marzo dependiendo de si el año es bisiesto o no.

► 406 Diseña un método que calcule el número de días transcurridos entre la fecha sobre la que se invoca el método y otra que se proporciona como parámetro. He aquí un ejemplo de uso:

```
>>> ayer = Fecha(1, 1, 2002)
>>> hoy = Fecha(2, 1, 2002)
>>> print(hoy.días_transcurridos(ayer))
1
```

(No tengas en cuenta el salto de fechas producido como consecuencia de la reforma gregoriana del calendario. Si no sabes de qué estamos hablando, consulta el cuadro «¿Cuántos días han pasado... dónde?»).

► 407 Modifica el método anterior para que sí tenga en cuenta los 10 días perdidos en la reforma gregoriana... en España.

► 408 Diseña un método que devuelva el día de la semana (la cadena 'lunes', o 'martes', etc.) en que cae una fecha cualquiera. (Si sabes en qué día cayó una fecha determinada, el número de días transcurridos entre esa y la nueva fecha módulo 7 te permite conocer el día de la semana).

### 7.3.2. Colas y pilas

En numerosas aplicaciones hemos de gestionar colas. Por ejemplo, en un programa de ayuda a la gestión de una consulta médica necesitamos manejar una cola de pacientes; o en la gestión de una lista de espera (una cola) de pasajeros en los vuelos con *overbooking*. Sería deseable disponer de un tipo de datos *Cola* que podamos utilizar en cualquier aplicación en la que haga falta. La cola podría comportarse como se muestra en este ejemplo (que, por simplificar, trabaja con números enteros):

```
>>> cola = Cola()↵
>>> cola.añade(5)↵
>>> cola.añade(8)↵
>>> cola.añade(4)↵
>>> print(cola)↵
5 8 4
>>> print(cola.primer())↵
5
>>> print(cola)↵
5 8 4
>>> cola.sacaPrimero()↵
>>> print(cola)↵
8 4
>>> cola.añade(9)↵
>>> print(cola)↵
8 4 9
>>> print(cola.tamaño())↵
3
>>> print(cola.esVacía())↵
False
>>> cola.sacaPrimero()↵
>>> cola.sacaPrimero()↵
>>> cola.sacaPrimero()↵
>>> print(cola)↵
>>> print(cola.esVacía())↵
True
>>> print(cola.primer())↵
None
```

Fíjate:

- La cola se construye sin ningún argumento (*cola = Cola()*): inicialmente la cola está vacía.
- El método *añade* permite añadir elementos al final de la cola.
- La función *print* muestra todos los elementos de la cola, empezando por el que entró en ella en primer lugar. Si la cola está vacía, no muestra nada.
- El método *primer* nos dice quién ocupa la primera posición de la cola. Si la cola está vacía, devuelve *None*.
- El método *sacaPrimero* elimina al primer elemento de la cola. Si la cola está vacía, no hace nada.
- El método *tamaño* nos dice cuántos elementos hay en la cola.
- El método *esVacía* nos dice si la cola está vacía o no, devolviendo cierto o falso.

Nuestro primer problema es decidir cómo guardamos la información propia de una cola y la respuesta es muy fácil: con una lista.

```
cola.py
1 class Cola:
2     def __init__(self):
3         self.cola = []
```

Hemos guardado la lista en el atributo `cola` y la hemos inicializado con la lista vacía. El constructor `__init__` no necesita parámetro alguno (salvo, `self`, naturalmente), pues una cola nueva siempre empieza estando vacía.

El método `añade` es sencillo de implementar: se limita a añadir a la lista un nuevo elemento por el final.

```
cola.py
1 class Cola:
2     ...
3
4     def añade(self, elemento):
5         self.cola.append(elemento)
```

El método `primero` nos dice quién ocupa la primera posición de la cola, es decir, quién ocupa `self.cola[0]`:

```
cola.py
1 class Cola:
2     ...
3
4     def primero(self):
5         if len(self.cola) == 0:
6             return None
7         else:
8             return self.cola[0]
```

Y el método `sacaPrimero` sencillamente borra el primer elemento de la lista:

```
cola.py
1 class Cola:
2     ...
3
4     def sacaPrimero(self):
5         if len(self.cola) > 0:
6             del self.cola[0]
```

El resto de métodos no plantea dificultad alguna, así que los mostramos todos sin más dilación:

```
cola.py
1 class Cola:
2     ...
3
4     def __str__(self):
5         cadena = ''
6         for elemento in self.cola:
7             cadena = cadena + str(elemento) + '\u2022'
8         return cadena
9
10    def tamaño(self):
11        return len(self.cola)
12
13    def esVacia(self):
14        return len(self.cola) == 0
```

---

► 409 Diseña un método `copia` que devuelva una nueva `Cola` cuyo contenido es el mismo de la `Cola` sobre la que se invoca el método. ¡Ojo con la memoria cuando saques una copia de `self.cola`!

► 410 Diseña un programa que gestiona una lista de espera de pacientes usando la clase `Cola` definida en este apartado. Cada paciente tiene un nombre y un número de la seguridad

social. El programa presentará un menú con las siguientes opciones: 1) añadir un paciente a la lista de espera; 2) atender al primer paciente de la lista; y 3) finalizar la ejecución del programa. Al seleccionar un paciente se solicitarán los datos del paciente y se añadirá este a la lista de espera (que es una cola). Al seleccionar la segunda opción, aparecerá en pantalla el nombre del paciente y se eliminará a este de la cola. La lista de espera respetará estrictamente el orden de llegada de los pacientes.

► 411 Modifica el programa anterior para que gestione dos colas: una para atención médica normal y otra para urgencias. Al añadir un paciente se preguntará si se trata de una urgencia o no. En el primer caso, se añadirá a una lista de espera de urgencias y en el segundo, a una lista de espera normal. Cada vez que se seleccione la segunda opción del menú aparecerá el nombre de un paciente en pantalla, pero tendrán preferencia aquellos que estén en la lista de espera de urgencias. Ningún paciente de la cola normal será atendido mientras haya uno solo en la cola de urgencias. Dentro de cada cola se respetará estrictamente el orden de llegada.

► 412 Implementa ahora el tipo *Pila*. Una pila es una lista de elementos en la que el primero que entra es el último en salir. Debes montar los siguientes métodos (además del constructor):

- *apila*: introduce un nuevo elemento en la pila,
  - *desapila*: extrae el último elemento introducido en la pila,
  - *cima*: devuelve el último elemento introducido en la pila (pero sin modificar la pila),
  - *tamaño*: dice cuántos elementos hay apilados,
  - *es\_vacía*: devuelve cierto si la pila está vacía y falso en caso contrario.
- 

### 7.3.3. Colas de prioridad

Las colas de prioridad son unas colas un tanto especiales: sus elementos se insertan en cualquier orden, pero el elemento que sale en primer lugar siempre es el que presenta mayor prioridad.

Tenemos varias alternativas para implementar una cola de prioridad<sup>3</sup>:

- a) representarla internamente mediante una lista que en todo momento está ordenada de mayor a menor prioridad,
- b) representarla internamente mediante una lista desordenada, buscando el elemento de mayor prioridad cada vez que se precise.

Desarrollaremos el segundo caso, pero te proponemos como ejercicio que desarrolles tú mismo el primero.

Las operaciones que podremos realizar con una cola de prioridad son:

- Construirla (*\_\_init\_\_*): crea una cola de prioridad vacía.
- Insertar un elemento (*inserta*): añade un elemento a la cola.
- Consultar el primer elemento (*primero*): devuelve el elemento de mayor prioridad, pero no lo elimina de la cola.
- Extraer (*extrae*): devuelve el elemento de mayor prioridad y lo elimina de la cola.
- Consultar su tamaño (*tamaño*): devuelve el número de elementos encolados.
- Consultar si está vacía (*es\_vacía*): devuelve cierto si la cola está vacía y falso en caso contrario.

<sup>3</sup>Con lo que sabemos hacer de momento, solo tenemos esas dos posibilidades. Ambas son muy ineficientes, pero aún es pronto para que sepas por qué. Las colas de prioridad pueden implementarse con *montículos* (*heaps*, en inglés) u otras estructuras de datos avanzadas.

Implementemos:

```
colaprioridad.py
1 class ColaPrioridad:
2     def __init__(self):
3         self.cola = []
4
5     def inserta(self, valor):
6         self.cola.append(valor)
7
8     def primero(self, valor):
9         if len(self.cola) == 0:
10             return None
11         maximo = self.cola[0]
12         for elemento in self.cola:
13             if elemento > maximo:
14                 maximo = elemento
15         return maximo
16
17     def extrae(self, valor):
18         if len(self.cola) == 0:
19             return None
20         indice = 0
21         for i in range(len(self.cola)):
22             if self.cola[i] > self.cola[indice]:
23                 indice = i
24         aux = self.cola[indice]
25         del self.cola[indice]
26         return aux
27
28     def tamaño(self):
29         return len(self.cola)
30
31     def es_vacia(self):
32         return len(self.cola) == 0
```

---

► 413 Implementa una cola de prioridad utilizando internamente una lista siempre ordenada.

► 414 Vamos a mejorar el programa de gestión de colas de pacientes desarrollado en el apartado anterior. Ahora vamos a clasificar los pacientes según la gravedad de su dolencia en 20 niveles de prioridad distintos. El nivel 20 es el más prioritario, el que requiere la más urgente atención, y el 1 es el menos prioritario.

Podríamos gestionar 20 colas distintas, una para cada prioridad, pero resulta más elegante gestionar una única cola de prioridad. Para ello, en lugar de encolar únicamente el nombre de un paciente, puedes encolar una lista con dos o más datos (como mínimo deberás almacenar la prioridad y el nombre del paciente).

---

### 7.3.4. Conjuntos

Es probable que en nuestras aplicaciones necesitemos utilizar conjuntos, es decir, colecciones de datos en las que cada elemento de un universo está o no está presente. Una lista no es un conjunto porque es posible que un elemento aparezca repetidas veces. Podemos simular el tipo de datos conjunto con una simple lista, pero teniendo siempre la precaución de no insertar un elemento si ya está presente. Para no complicar nuestros programas con constantes consultas a listas para determinar si los elementos a insertar están o no ya presentes, es mejor definir un nuevo tipo de datos que, internamente, realice las comprobaciones pertinentes: una clase *Conjunto*. Es más, podemos enriquecer el nuevo tipo de datos con operaciones de conjuntos útiles: la intersección, la unión, la diferencia, etc.

En primer lugar nos hemos de plantear qué atributos tendrá un *Conjunto*. Bastará con una lista que contenga los elementos presentes en el conjunto. Inicialmente, la lista estará vacía:

```
conjunto.py
1 class Conjunto:
2     def __init__(self):
3         self.elementos = []
```

Necesitamos definir ahora un método para la inserción de elementos. El método *inserta* recibirá dos argumentos: *self* (como siempre) y el elemento que deseamos insertar. Antes de añadir el elemento a la lista, nos preguntaremos si ya está presente, porque solo deberemos hacer algo en caso contrario:

```
conjunto.py
1 class Conjunto:
2     ...
3
4     def inserta(self, elemento):
5         if not (elemento in self.elementos):
6             self.elementos.append(elemento)
```

---

► 415 Enriquece la clase *Conjunto* con un método *elimina* que borre del conjunto un elemento dado. (Solo habrá que borrar el elemento de la lista si está presente, claro está).

---

Definamos ahora un método para imprimir un conjunto por pantalla. Estaría bien que los conjuntos aparecieran por pantalla con el aspecto «tradicional»: con sus elementos separados por comas y encerrados en un par de llaves.

```
conjunto.py
1 class Conjunto:
2     ...
3
4     def __str__(self):
5         cadena = '{'
6         if len(self.elementos) > 0:
7             for elemento in self.elementos[:-1]:
8                 cadena = cadena + str(elemento) + ','
9             cadena = cadena + str(self.elementos[-1])
10
11    return cadena + '}'
```

He aquí un ejemplo de uso de *Conjunto*:

```
>>> A = Conjunto()
>>> A.inserta(3)
>>> A.inserta(5)
>>> A.inserta(3)
>>> print(A)
{3, 5}
```

Otro método útil nos permite preguntar a un conjunto por su talla, es decir, el número de elementos que lo forman:

```
1 class Conjunto:
2     ...
3
4     def talla(self):
5         return len(self.elementos)
```

Nos vendrá bien disponer de un método que permita consultar si un elemento pertenece o no a un conjunto:

```
1 class Conjunto:
2     ...
3
4     def pertenece(self, elemento):
5         return elemento in self.elementos
```

► 416 Diseña un método *es\_vacio* que devuelva cierto si el conjunto está vacío y falso en caso contrario.

Vamos a por las operaciones entre conjuntos. Definamos la unión de conjuntos con un método que devuelva el conjunto resultante de unir al conjunto *self* otro conjunto:

```
1 class Conjunto:
2     ...
3
4     def unión(self, otro):
5         C = Conjunto()
6         C.elementos = self.elementos[:]
7         for elemento in otro.elementos:
8             C.inserta(elemento)
9         return C
```

Observa cómo se usa *unión*:

```
>>> A = Conjunto()
>>> A.inserta(3)
>>> A.inserta(5)
>>> B = Conjunto()
>>> B.inserta(10)
>>> C = A.unión(B)
>>> print(C)
{3, 5, 10}
```

► 417 Diseña un método *intersección* que devuelva un conjunto con la intersección de dos conjuntos (uno de ellos será aquel sobre el que se invoca el método y otro se pasará como parámetro).

► 418 Diseña un método *diferencia* que devuelva un conjunto con la diferencia entre dos conjuntos, es decir, con aquellos elementos que están en el primero, pero no en el segundo.

Finalmente, he aquí un método que consulta si otro conjunto dado está incluido en el conjunto (*self*):

```
1 class Conjunto:
2     ...
3
4     def incluye(self, otro):
5         for elemento in otro.elementos:
6             if not (elemento in self.elementos):
7                 return False
8         return True
```

## 7.4. Un ejemplo completo: gestión de un videoclub

En este apartado vamos a desarrollar un ejemplo completo y útil usando clases: un programa para gestionar un videoclub. Empezaremos creando la aplicación de gestión para un videoclub básico, muy simplificado, e iremos complicándola poco a poco.

### Más métodos especiales

Hemos visto que las clases admiten dos métodos especiales: `__init__` y `__str__`. No son los únicos métodos especiales. Podemos hacer que las clases se comporten de modo similar a los tipos de datos nativos de Python definiendo muchos otros métodos especiales. He aquí unos pocos:

- `__len__(self)`: Permite aplicar la función predefinida `len` sobre objetos de la clase. Debe devolver la «longitud» o «talla» del objeto. En el caso de colas y conjuntos, por ejemplo, correspondería al número de elementos. Si `A` es un *Conjunto*, podríamos usar `len(A)` si antes hubiésemos definido el método `__len__`.
- `__add__(self, otro)`: Permite aplicar el operador de suma (+) a objetos de la clase sobre la que se ha definido. Si, por ejemplo, `A` y `B` son conjuntos, la expresión `C = A + B` permite asignar al nuevo conjunto `C` la unión de ambos.
- `__mul__(self, otro)`: Permite aplicar el operador de multiplicación (\*) a objetos de la clase sobre la que se ha definido.
- `__cmp__(self, otro)`: Permite aplicar los operadores de comparación (<, >, <=, >=, ==, !=) a objetos de una clase. Debe devolver -1 si `self` es menor que `otro`, 0 si son iguales y 1 si `self` es mayor que `otro`.

Podemos, por ejemplo, definir `__cmp__` en *Persona* para que devuelva -1 cuando la edad `self.edad` es menor que `otro.edad`, 0 si son iguales y 1 si `self.edad` es mayor que `otro.edad`. Si `juan` y `pedro` son personas, podremos compararlas con expresiones como `juan < pedro` o `juan != pedro`.

Consulta la documentación de Python si quieres conocer todos los métodos especiales que puedes definir en tus clases. Tus programas pueden ganar mucho en elegancia si defines los métodos apropiados para cada clase.

#### 7.4.1. Videoclub básico

El videoclub tiene un listado de socios. Cada socio tiene una serie de datos:

- dni,
- nombre,
- teléfono,
- domicilio.

Por otra parte, disponemos de una serie de películas. De cada película nos interesa:

- título,
- género (acción, comedia, musical, etc.).

Supondremos que en nuestro videoclub básico solo hay un ejemplar de cada película.

Empocemos definiendo los tipos básicos con sus métodos especiales: el constructor `__init__` y el conversor a cadena `__str__`:

```
videoclub.py
1 class Socio:
2     def __init__(self, dni, nombre, teléfono, domicilio):
3         self.dni = dni
4         self.nombre = nombre
5         self.teléfono = teléfono
6         self.domicilio = domicilio
7
8     def __str__(self):
9         return 'DNI:{}\nNombre:{}\nTeléfono:{}\nDomicilio:{}\n' \
10            .format(self.dni, self.nombre, self.teléfono, self.domicilio)
11
```

```

12 class Película:
13     def __init__(self, título, género):
14         self.título = título
15         self.género = género
16
17     def __str__(self):
18         return 'Título:{}\nGénero:{}\n'.format(self.título, self.género)

```

Podemos definir también una clase *Videoclub* que mantenga y gestione las listas de socios y películas:

```

videoclub.py
1 class Videoclub:
2     def __init__(self):
3         self.socios = []
4         self.películas = []

```

Nuestra aplicación presentará un menú con diferentes opciones. Empecemos por implementar las más sencillas: dar de alta/baja a un socio, dar de alta/baja una película. La función *menú* mostrará el menú de operaciones y leerá la opción que seleccione el usuario de la aplicación. Nuestra primera versión será esta:

```

videoclub.py
1 def menú():
2     print('***VIDEOCLUB***')
3     print('1)Dar_de_alta_nuevo_socio')
4     print('2)Dar_de_baja_un_socio')
5     print('3)Dar_de_alta_nueva_pelicula')
6     print('4)Dar_de_baja_una_pelicula')
7     print('5)Salir')
8     opción = int(input('Escoge_opción:'))
9     while opción < 1 or opción > 5:
10         opción = int(input('Escoge_opción_(entre_1_y_5):'))
11     return opción

```

En una variable *videoclub* tendremos una instancia de la clase *Videoclub*, y es ahí donde almacenaremos la información del videoclub. Nuestra primera versión del programa presentará este aspecto:

```

videoclub.py
1 videoclub = Videoclub()
2
3 opción = menú()
4 while opción != 5:
5
6     if opción == 1:
7         print('Alta_de_socio')
8         socio = nuevo_socio()
9         if videoclub.contiene_socio(socio.dni):
10             print('Ya_existe_un_socio_con_DNI', dni)
11         else:
12             videoclub.alta_socio(socio)
13
14     elif opción == 2:
15         print('Baja_de_socio')
16         dni = input('DNI:')
17         if videoclub.contiene_socio(dni):
18             videoclub.baja_socio(dni)
19             print('Socio_con_DNI', dni, 'dado_de_baja')
20         else:

```

```

21         print('No existe ningún socio con DNI', dni)
22
23     elif opción == 3:
24         print('Alta de película')
25         película = nueva_película()
26         if videoclub.contiene_película(película, título):
27             print('Ya hay una película con título', película.título)
28         else:
29             videoclub.alta_pelicula(película)
30
31     elif opción == 4:
32         print('Baja de película')
33         título = input('Título: ')
34         if videoclub.contiene_película(título):
35             videoclub.baja_pelicula(título)
36         else:
37             print('No existe ninguna película llamada', título)
38
39
40 opción = menú()

```

Analicémoslo por partes. Empecemos por el fragmento de código que corresponde al alta de un socio. Lo primero que hacemos es pedir la creación de un nuevo socio mediante la función *nuevo\_socio*. Esta función leerá de teclado los datos. Definámolas:

```

videoclub.py
1 def nuevo_socio():
2     dni = input('DNI: ')
3     nombre = input('Nombre: ')
4     teléfono = input('Teléfono: ')
5     domicilio = input('Domicilio: ')
6     return Socio(dni, nombre, teléfono, domicilio)

```

El socio devuelto por *nuevo\_socio* puede haber sido dado de alta previamente en el videoclub, con lo que no sería procedente darlo de alta ahora. A continuación se «pregunta» al videoclub si ya tiene algún socio con el DNI del nuevo socio. Si es así, se muestra un aviso por pantalla, y si no, se da de alta al socio. Observa que se usan dos métodos del videoclub: *contiene\_socio*, que recibe un DNI y devuelve cierto o falso, y *alta\_socio*, que recibe un socio y lo añade a su lista de socios. Su definición sería:

```

videoclub.py
1 class Videoclub:
2     ...
3
4     def contiene_socio(self, dni):
5         for socio in self.socios:
6             if socio.dni == dni:
7                 return True
8         return False
9
10    def alta_socio(self, socio):
11        self.socios.append(socio)

```

Estudiemos ahora el fragmento de código para dar de baja a un socio. En primer lugar, se pide su DNI. Si el socio existe (lo que se averigua con el método *contiene\_socio*, definido justo antes), se le da de baja llamando al método *baja\_socio*, que recibe el DNI. Si ningún socio tiene el DNI suministrado, se advierte al usuario del programa con un aviso. Hemos de definir, pues, *baja\_socio*:

```

videoclub.py
1 class Videoclub:
2     ...
3
4     def baja_socio(self, dni):
5         for i in range(len(self.socios)):
6             if self.socios[i].dni == dni:
7                 del self.socios[i]
8                 break

```

---

► 419 Define tú mismo los procedimientos que dan de alta/baja una película.

---

De poca utilidad será el programa si no permite alquilar las películas. ¿Cómo haremos para representar que una película está alquilada a un socio? Tenemos (al menos) dos posibilidades:

- añadir un atributo a cada *Socio* indicando qué película tiene en alquiler (y si no tiene ninguna, su valor será **None**, por ejemplo),
- añadir un atributo a cada *Película* indicando a quién está alquilada (y si no está alquilada, su valor será **None**, por ejemplo).

Parece mejor la segunda opción: una operación que realizaremos con frecuencia es preguntar si una película está alquilada o no; por contra, preguntar si un socio tiene o no películas alquiladas parece una operación menos frecuente.

Así pues, tendremos que modificar la definición de la clase *Película*:

```

videoclub.py
1 class Película:
2     def __init__(self, título, género):
3         self.título = título
4         self.género = género
5         self.alquilada = None
6
7     def __str__(self):
8         cadena = 'Título:{0}\nGénero:{1}\n'.format(self.título, self.género)
9         if self.alquilada == None:
10             cadena = cadena + 'Disponible\n'
11         else:
12             cadena = cadena + 'Alquilada a:{0}\n'.format(self.alquilada)
13         return cadena

```

Observa que el atributo *alquilada* no se pasa como parámetro a *\_\_init\_\_*. La razón es muy simple: cuando «construimos» una nueva película no está alquilada a nadie, así que el atributo *alquilada* siempre empieza valiendo **None**. ¿Para qué pasar como argumento a *\_\_init\_\_* un valor que no aporta información alguna?

Añadamos ahora un método que permita alquilar una película (dado su título) a un socio (dado su DNI). La llamada a este método se asociará a la opción 5 del menú, y el final de ejecución de la aplicación se asociará ahora a la opción 6.

```

videoclub.py
1 ...
2
3 videoclub = Videoclub()
4
5 opción = menú()
6
7 while opción != 6:
8
9     if opción == 1:

```

```

10     ...
11
12 elif opción == 5:
13     print('Alquiler de película')
14     título = input('Título de la película:')
15     dni = input('DNI del socio:')
16     hay_pelicula = videoclub.contiene_pelicula(título)
17     hay_socio = videoclub.contiene_socio(dni)
18     if hay_pelicula and hay_socio:
19         videoclub.alquilar_pelicula(título, dni)
20     else:
21         if not hay_pelicula:
22             print('No hay película titulada', título)
23         if not hay_socio:
24             print('No hay socio con DNI', dni)
25
opción = menú()

```

Diseñemos el método *alquilar\_pelicula*. Supondremos que existe una película cuyo título corresponde al que nos indican y que existe un socio cuyo DNI es igual al que nos pasan como argumento, pues ambas comprobaciones se efectúan antes de llamar al método.

```

videoclub.py
1 class Videoclub:
2     ...
3
4     def alquilar_pelicula(self, título, dni):
5         for película in self.películas:
6             if película.título == título and película.alquilada == None:
7                 película.alquilada = dni
8                 break

```

En principio, ya está. El método *alquilar\_pelicula* recorre la lista de películas del videoclub y solo efectúa el alquiler cuando encuentra una con el título que nos dan y esta está disponible. Pero podemos mejorarlo: el método no nos informa de si finalmente alquiló o no la película en cuestión, lo que hace que no podamos informar al usuario de si la operación se realizó con éxito o no. Vamos a modificarlo para que devuelva cierto si alquiló efectivamente la película, y falso en caso contrario.

```

videoclub.py
1 class Videoclub:
2     ...
3
4     def alquilar_pelicula(self, título, dni):
5         for película in self.películas:
6             if película.título == título:
7                 if película.alquilada == None:
8                     película.alquilada = dni
9                     return True
10                else:
11                    return False

```

Ahora podemos modificar las acciones asociadas a la opción 5:

```

videoclub.py
1 videoclub = Videoclub()
2
3 opción = menú()
4 while opción != 6:
5
6     if opción == 1:

```

```

7     ...
8
9 elif opción == 5:
10    título = input('Título de la película:')
11    dni = input('DNI del socio:')
12    hay_pelicula = videoclub.contiene_pelicula(título)
13    hay_socio = videoclub.contiene_socio(dni)
14    if hay_pelicula and hay_socio:
15        if videoclub.alquilar_pelicula(título, dni):
16            print('Operación realizada')
17        else:
18            print('La película no está disponible')
19    else:
20        if not hay_pelicula:
21            print('No hay película titulada', título)
22        if not hay_socio:
23            print('No hay socio con DNI', dni)
24
opción = menú()

```

► 420 Añade una nueva funcionalidad al programa: la devolución de una película alquilada. Diseña para ello un método *devolver\_pelicula* que, dado el título de la película, devuelve **True** si estaba alquilada y **False** en caso contrario. Además, si estaba alquilada, el método la marcará como disponible (pondrá a **None** el valor del campo *alquilada*).

A continuación, añade una opción al menú para devolver una película. Las acciones asociadas son:

- pedir el nombre de la película;
- si no existe una película con ese título, dar el aviso pertinente y no hacer nada más;
- si existe la película pero no estaba alquilada, avisar al usuario y no hacer nada más;
- y si existe la película y estaba alquilada, marcarla como disponible.

► 421 Modifica los métodos que dan de baja a un socio o una película para que no se permita dar de baja una película alquilada ni a un socio que tiene alguna película en alquiler.

Si no fue posible dar de baja el socio o la película, el método correspondiente devolverá **False**. Si, por el contrario, se pudo dar de baja a uno u otro, devolverá **True**.

Modifica a continuación las acciones asociadas a las respectivas opciones del menú para que den los avisos pertinentes en caso de que no sea posible dar de baja a un socio o una película.

Finalmente, vamos a ofrecer la posibilidad de efectuar una consulta interesante a la colección de películas del videoclub. Es posible que un cliente nos pida que le recomendemos películas disponibles dado el género que a él le gusta. Un método de *Videoclub* permitirá obtener este tipo de listados.

```

videoclub.py
1 class Videoclub:
2     ...
3
4     def listado_por_género(self, género):
5         for película in self.películas:
6             if película.género == género and película.alquilada == None:
7                 print(título)

```

Solo resta añadir una opción de menú que pida el género para el que solicitamos el listado e invoque al método *listado\_por\_género*.

► 422 Modifica *listado\_por\_género* para que muestre todas las películas del videoclub, pero indicando al lado del título si está alquilada o disponible.

### 7.4.2. Un videoclub más realista

El programa que hemos hecho presenta ciertos inconvenientes por su simplicidad: por ejemplo, asume que solo existe un ejemplar de cada película y, al no llevar registro de las fechas de alquiler, permite que un socio alquile una película un número indeterminado de días. Mejoraremos el programa corrigiendo ambos defectos.

Tratemos en primer lugar la cuestión de la existencia de varios ejemplares por película. Está claro que la clase *Película* ha de sufrir algunos cambios. Tenemos (entre otras) dos posibilidades:

- a) hacer que cada objeto de la clase *Película* corresponda a un ejemplar, es decir, permitir que la lista *películas* contenga títulos repetidos (una vez por cada ejemplar).
- b) enriquecer cada película con un campo *ejemplares* que indique cuántos ejemplares tenemos.

Mmmm. La segunda posibilidad requiere un estudio más detallado. Con solo un contador de ejemplares no es suficiente. ¿Cómo representaremos el hecho de que, por ejemplo, de 5 ejemplares, 3 están alquilados, cada uno a un socio diferente? Será preciso enriquecer la información propia de una *Película* con una lista que contenga un elemento por cada ejemplar alquilado. Cada elemento de la lista deberá contener, como mínimo, algún dato que identifique al socio al que se alquiló la película.

Parece, pues, que la primera posibilidad es más sencilla de implementar. Desarrollaremos esa, pero te proponemos como ejercicio que desarrolles tú la segunda posibilidad.

En primer lugar, modificaremos el método que da de alta una película para que nos pida el número de ejemplares que añadimos al videoclub.

```
videoclub.py
1 class Videoclub:
2     ...
3
4     def alta_pelicula(self, película, ejemplares):
5         for i in range(ejemplares):
6             nuevo_ejemplar = Película(película.título, película.género)
7             self.películas.append(nuevo_ejemplar)
```

Al dar de alta ejemplares de una película ya no será necesario comprobar si existe ese título en nuestra colección de películas:

```
videoclub.py
1 ...
2
3     elif opción == 3:
4         print('Alta de película')
5         película = nueva_pelicula()
6         ejemplares = int(input('Ejemplares: '))
7         videoclub.alta_pelicula(película, ejemplares)
8     ...
```

Dar de baja un número de ejemplares de un título determinado no es muy difícil, aunque puede aparecer una pequeña complicación: que no podamos eliminar efectivamente el número de ejemplares solicitado, bien porque no hay tantos en el videoclub, bien porque alguno de ellos está alquilado. Haremos que el método que da de baja el número de ejemplares solicitado nos devuelva el número de ejemplares que realmente pudo dar de baja, de ese modo al menos «avisamos» a quien nos llama de lo que realmente hicimos.

```
videoclub.py
1 class Videoclub:
2     ...
3
4     def baja_pelicula(self, título, ejemplares):
5         bajas_efectivas = 0
```

```

6     i = 0
7     while i < len(self.películas):
8         if self.películas[i].título == título and self.películas[i].alquilada == None:
9             del self.películas[i]
10            bajas_efectivas += 1
11        else:
12            i += 1
13    return bajas_efectivas

```

Veamos cómo queda el fragmento de código asociado a la acción de menú que da de baja películas:

```

videoclub.py
1 ...
2
3     elif opción == 4:
4         print('Baja_de_pelicula')
5         título = input('Título:')
6         ejemplares = int(input('Ejemplares:'))
7         bajas = videoclub.baja_pelicula(título, ejemplares)
8         if bajas < ejemplares:
9             print('Solo_pude_dar_de_baja', bajas, 'ejemplares')
10        else:
11            print('Operación_realizada')
12
13 ...

```

El método de alquiler de una película a un socio necesita una pequeña modificación: puede que los primeros ejemplares encontrados de una película estén alquilados, pero no estamos seguros de si hay alguno libre hasta haber recorrido la colección entera de películas. El método puede quedar así:

```

videoclub.py
1 class Videoclub:
2     ...
3
4     def alquilar_pelicula(self, título, dni):
5         for película in self.películas:
6             if película.título == título:
7                 if película.alquilada == None:
8                     película.alquilada = dni
9                     return True
10    return False

```

Observa que solo devolvemos `False` cuando hemos recorrido la lista entera de películas sin haber podido encontrar una libre.

---

► 423 Implementa el nuevo método de devolución de películas. Ten en cuenta que necesitarás dos datos: el título de la película y el DNI del socio.

Ahora podemos modificar el programa para que permita controlar si un socio retiene la película más días de los permitidos y, si es así, que nos indique los días de retraso. Enriqueceremos la clase *Película* con nuevos atributos:

- *fecha\_alquiler*: contiene la fecha en que se realizó el alquiler.
- *días\_permitidos*: número de días de alquiler permitidos.

Parece que ahora hemos de disponer de cierto control sobre las fechas. Afortunadamente ya hemos construido una clase *Fecha* en este mismo capítulo. ¡Utilicémosla!

- 
- 424 Modifica el constructor de *Película* para añadir los nuevos atributos. Modifica a continuación *alta\_pelicula* para que pida también el valor de *días\_permitidos*.

Empezaremos por añadir una variable global, a la que llamaremos *hoy*, que contendrá la fecha actual<sup>4</sup>:

```
videoclub.py
1 from fecha import Fecha, lee_fecha
2 ...
3
4 # Programa principal
5 hoy = lee_fecha()
```

Cuando alquilemos una película no solo apuntaremos el socio al que la alquilamos, también recordaremos la fecha del alquiler:

```
videoclub.py
1 class Videoclub:
2 ...
3
4     def alquilar_pelicula(self, titulo, dni):
5         for pelicula in self.peliculas:
6             if pelicula.titulo == titulo:
7                 if pelicula.alquilada == None:
8                     pelicula.alquilada = dni
9                     pelicula.fecha_alquiler = hoy
10                    return True
11
12    return False
```

Otro procedimiento afectado por la introducción de fechas es el de devolución de películas. No nos podemos limitar a devolver la película: hemos de comprobar si se incurre en retraso y, por tanto, se debe pagar una multa.

- 
- 425 Modifica el método de devolución de películas para que tenga en cuenta la fecha de alquiler y la fecha de devolución. El método devolverá el número de días de retraso. (Supón que nuestra clase *Fecha* dispone de un método *días\_trascurridos* que devuelve el número de días transcurridos desde una fecha determinada).

Modifica las acciones asociadas a la opción de menú de devolución de películas para que tenga en cuenta el valor devuelto por el método *devolver\_pelicula* y muestre por pantalla el número de días de retraso (si es el caso).

- 
- 426 Modifica el método *listado\_por\_género* para que un mismo título de una película no aparezca más que una vez. Al lado del título aparecerá la cadena '**DISPONIBLE**' si hay al menos un ejemplar disponible y '**NO DISPONIBLE**' si todos los ejemplares están alquilados.

### 7.4.3. Listado completo

Nos ha salido un programa larguito. Vale la pena que mostremos un listado completo.

```
videoclub.py
1 =====
2 # videoclub
3 =====
4 # Programa para la gestión de videoclubs.
5 =====
6
7 from fecha import Fecha, lee_fecha
```

<sup>4</sup>Lo natural sería que la fecha actual se fijara automáticamente a partir del reloj del sistema. Puedes hacerlo usando el módulo *time*. Consulta el manual de la librería.

```

8
9 # _____
10 # Socio
11 #
12 # Clase para almacenar los datos relativos a un socio.
13 #
14 class Socio:
15     def __init__(self, dni, nombre, teléfono, domicilio):
16         self.dni = dni
17         self.nombre = nombre
18         self.teléfono = teléfono
19         self.domicilio = domicilio
20
21     def __str__(self):
22         return 'DNI: {} \nNombre: {} \nTeléfono: {} \nDomicilio: {} \n'.format(self.dni, self.nombre, self.teléfono, self.domicilio)
23
24 #
25 # Película
26 #
27 # Clase para almacenar los datos relativos a un ejemplar de una
28 # película.
29 #
30 #
31 class Película:
32     def __init__(self, título, género, días_permitidos):
33         self.título = título
34         self.género = género
35         self.alquilada = None
36         self.fecha_alquiler = None
37         self.días_permitidos = días_permitidos
38
39     def __str__(self):
40         cadena = 'Título: {} \nGénero: {} \n'.format(self.título, self.género)
41         if self.alquilada == None:
42             cadena = cadena + 'Disponible \n'
43         else:
44             cadena = cadena + 'Alquilada: {} \n'.format(self.alquilada)
45         return cadena
46
47 #
48 # Videoclub
49 #
50 # Almacena dos listas: una de socios y otra de películas. Los
51 # elementos de la primera lista son de la clase Socio, y los de la
52 # segunda, de la clase Película.
53 #
54 class Videoclub:
55     def __init__(self):
56         self.socios = []
57         self.películas = []
58
59     def contiene_socio(self, dni):
60         # Devuelve True si existe algún socio con DNI dni y False en caso
61         # contrario.
62         for socio in self.socios:
63             if socio.dni == dni:
64                 return True
65         return False
66
67     def contiene_pelicula(self, título):
68         # Devuelve True si existe alguna película del título que nos

```

```

69     # pasan y False en caso contrario.
70     for película in self.películas:
71         if película.título == título:
72             return True
73     return False
74
75 def alta_socio(self, socio):
76     # Añade un socio a la lista de socios.
77     # Requisito: no debe existir ningún socio con el mismo DNI.
78     self.socios.append(socio)
79
80 def baja_socio(self, dni):
81     # Elimina al socio cuyo DNI es igual a dni.
82     # Requisito: debe existir un socio con ese DNI.
83     for i in range(len(self.socios)):
84         if self.socios[i].dni == dni:
85             del self.socios[i]
86             break
87
88 def alta_pelicula(self, película, ejemplares):
89     # Da de alta un número dado de ejemplares de una película.
90     for i in range(ejemplares):
91         nuevo_ejemplar = Película(película.título, película.género, \
92                                   película.días_permitidos)
93         self.películas.append(nuevo_ejemplar)
94
95 def baja_pelicula(self, título, ejemplares):
96     # Da de baja un número de ejemplares de la película cuyo título nos
97     # suministran como argumento. Devuelve el número de ejemplares que
98     # se dio de baja efectivamente.
99     bajas_efectivas = 0
100    i = 0
101    while i < len(self.películas) and bajas_efectivas < ejemplares:
102        if self.películas[i].título == título and self.películas[i].alquilada == None:
103            del self.películas[i]
104            bajas_efectivas += 1
105        else:
106            i += 1
107    return bajas_efectivas
108
109 def alquilar_pelicula(self, título, dni):
110     # Alquila un ejemplar de la película cuyo título nos indican, al socio
111     # con DNI dni. Si no consigue efectuar el alquiler, devuelve False, y True
112     # si lo consigue. La fecha de alquiler se fija automáticamente al día
113     # actual.
114     # Requisito: debe existir un socio con el DNI suministrado.
115     for película in self.películas:
116         if película.título == título and película.alquilada == None:
117             película.alquilada = dni
118             película.fecha_alquiler = hoy
119             return True
120     return False
121
122 def devolver_pelicula(self, título, dni):
123     # Devuelve un ejemplar de la película cuyo título nos indican que
124     # estaba alquilada al socio con DNI dni. Devuelve el número de días
125     # de retraso, o -1 si ningún ejemplar de la película está alquilado
126     # al socio.
127     # Requisito: debe existir un socio con el DNI suministrado.
128     for película in self.películas:
129         if película.título == título and película.alquilada == dni:

```



```

191     género = input('Género:')
192     días_permitidos = input('Días_permitidos:')
193     return Película(título, género, días_permitidos)
194
195 #_____
196 # Programa principal
197 #
198
199 # Fijar fecha actual
200 hoy = lee_fecha()
201
202 videoclub = Videoclub()
203
204 opción = menú()
205 while opción != 8:
206
207     if opción == 0:
208         print('Cambiar_fecha_actual')
209         hoy = lee_fecha()
210
211     elif opción == 1:
212         print('Alta_de_socio')
213         socio = nuevo_socio()
214         if videoclub.contiene_socio(socio.dni):
215             print('Ya_existía_un_socio_con_DNI', dni)
216         else:
217             videoclub.alta_socio(socio)
218
219     elif opción == 2:
220         print('Baja_de_socio')
221         dni = input('DNI:')
222         if videoclub.contiene_socio(dni):
223             videoclub.baja_socio(dni)
224             print('Socio_con_DNI', dni, 'dado_de_baja')
225         else:
226             print('No_existe_ningún_socio_con_DNI', dni)
227
228     elif opción == 3:
229         print('Alta_de_película')
230         película = nueva_película()
231         ejemplares = int(input('Ejemplares:'))
232         videoclub.alta_pelicula(película, ejemplares)
233
234     elif opción == 4:
235         print('Baja_de_pelicula')
236         título = input('Título:')
237         ejemplares = int(input('Ejemplares:'))
238         bajas = videoclub.baja_pelicula(título, ejemplares)
239         if bajas < ejemplares:
240             print('Solo_pude_dar_de_baja', bajas, 'ejemplares')
241         else:
242             print('Operación_realizada')
243
244     elif opción == 5:
245         print('Alquilar_pelicula')
246         título= input('Título_de_la_pelicula:')
247         dni = input('DNI_del_socio:')
248         hay_pelicula = videoclub.contiene_pelicula(título)
249         hay_socio = videoclub.contiene_socio(dni)
250         if hay_pelicula and hay_socio:
251             if videoclub.alquilar_pelicula(título, dni):

```

```

252         print('Operación realizada')
253     else:
254         print('La película no está disponible')
255     else:
256         if not hay_pelicula:
257             print('No hay película titulada', título)
258         if not hay_socio:
259             print('No hay socio con DNI', dni)
260
261 elif opción == 6:
262     print('Devolver película')
263     título= input('Título de la película:')
264     dni = input('DNI del socio:')
265     hay_pelicula = videoclub.contiene_pelicula(título)
266     hay_socio = videoclub.contiene_socio(dni)
267     if hay_pelicula and hay_socio:
268         resultado = videoclub.devolver_pelicula(título, dni)
269         if resultado == 0:
270             print('Operación realizada')
271         elif resultado > 0:
272             print('Película entregada con un retraso de', resultado, 'días')
273         else:
274             print('La película', título, 'no está alquilada al socio', dni)
275     else:
276         if not hay_pelicula:
277             print('No hay película titulada', título)
278         if not hay_socio:
279             print('No hay socio con DNI', dni)
280
281 elif opción == 7:
282     print('Listado por género')
283     género = input('Género:')
284     videoclub.listado_por_género(género)
285
286 opción = menú()

```

El programa de gestión de un videoclub que hemos desarrollado dista de ser perfecto. Muchas de las operaciones que hemos implementado son ineficientes y, además, mantiene toda la información en memoria RAM, así que pierde toda la información al finalizar la ejecución. Tendremos que esperar al próximo capítulo para abordar el problema del almacenamiento de información de modo que «recuerde» su estado entre diferentes ejecuciones.

### Bases de datos

Muchos programas de gestión manejan grandes volúmenes de datos. Es posible diseñar programas como el del videoclub (con almacenamiento de datos en disco duro, eso sí) que gestionen adecuadamente la información, pero, en general, poco recomendable. Existen programas y lenguajes de programación orientados a la gestión de bases de datos. Estos sistemas se encargan de gestionar el almacenamiento de información en disco y ofrecen utilidades para acceder y modificar la información. Es posible expresar, por ejemplo, órdenes como «busca todas las películas cuyo género es *acción*» o «lista a todos los socios que llevan un retraso de 1 o más días».

El lenguaje de programación más extendido para consultas a bases de datos es SQL (Standard Query Language) y numerosos sistemas de bases de datos lo soportan. Existen, además, sistemas de bases de datos de distribución gratuita como MySQL o Postgres, suficientemente potentes para aplicaciones de pequeño y mediano tamaño.

En otras asignaturas de la titulación aprenderás a utilizar sistemas de bases de datos y a diseñar bases de datos.

#### 7.4.4. Extensiones propuestas

Te proponemos como ejercicios una serie de extensiones al programa:

► 427 Modifica el programa para permitir que una película sea clasificada en diferentes géneros. (El atributo *género* será una lista de cadenas, y no una cadena).

► 428 Modifica la aplicación para permitir reservar películas a socios. Cuando de una película no se disponga de ningún ejemplar libre, los socios podrán solicitar una reserva.

¡Ojo!, la reserva se hace sobre una película, no sobre un ejemplar, es decir, la lista de espera de *Matrix* permite a un socio alquilar el primer ejemplar de *Matrix* que quede disponible. Por otra parte, si hay, por ejemplo, dos socios con una misma película reservada, solo podrá alquilarse a otros socios cuando haya tres o más ejemplares libres.

► 429 Modifica el programa del ejercicio anterior para que las reservas caduquen automáticamente a los dos días. Es decir, si el socio no ha alquilado la película a los dos días, su reserva expira.

► 430 Modifica el programa para que registre el número de veces que se ha alquilado cada película. Mediante una nueva opción de menú, el programa mostrará la lista de las 10 películas más alquiladas hasta el momento.

► 431 Modifica el programa para que registre todas las películas que ha alquilado cada socio a lo largo de su vida. Al consultar los datos de un socio se mostrarán sus géneros favoritos.

► 432 Añade al programa una opción de menú para aconsejar al cliente. Basándose en su historial de alquileres, el programa determinará sus géneros favoritos y mostrará un listado con las películas de dichos géneros disponibles para alquiler.

#### 7.4.5. Algunas reflexiones

Hemos desarrollado un ejemplo bastante completo, pero lo hemos hecho poco a poco, incrementalmente. Hemos empezado por construir una aplicación para un videoclub básico y hemos ido añadiéndole funcionalidad paso a paso. Normalmente no se desarrollan programas de ese modo. Se parte de una *especificación* de la aplicación, es decir, se parte de una descripción de lo que debe hacer el programa. El programador efectúa un *análisis* de la aplicación a construir. Un buen punto de partida es determinar las *estructuras de datos* que utilizará. En nuestro caso, hemos definido dos clases, *Socio* y *Película*, y hemos decidido que mantendríamos una lista de socios y otra de películas como atributos de otra clase: *Videoclub*. Solo cuando se ha decidido qué estructuras de datos utilizar se está en condiciones de diseñar e implementar el programa.

Pero ahí no acaba el trabajo del programador. La aplicación debe ser *testeada* para, en la medida de lo posible, asegurarse de que no contiene errores. Solo cuando se está seguro de que no los tiene, la aplicación pasa a la fase de *explotación*. Y es probable (yo seguro!) que entonces descubramos nuevos errores de programación. Empieza entonces un *ciclo de detección y corrección de errores*.

Tras un período de explotación de la aplicación es frecuente que el usuario solicite la implementación de nuevas funcionalidades. Es preciso, entonces, proponer una nueva especificación (o ampliar la ya existente), efectuar su correspondiente análisis e implementar las nuevas características. De este modo llegamos a la producción de nuevas *versiones* del programa.

► 433 Nos gustaría retomar el programa de gestión de MP3 que desarrollamos en el ejercicio 400. Nos gustaría introducir el concepto de «álbum». Cada álbum tiene un título, unos intérpretes y una lista de canciones (ficheros MP3). Modifica el programa para que gestione álbumes. Deberás permitir que el usuario dé de alta y baje álbumes, así como que obtenga listados completos de los álbumes disponibles, listados ordenados por intérpretes, búsquedas de canciones en la base de datos, etc.