

Web Partituras

Un ejemplo didáctico de compilación de páginas web con Python

Luis Sanjuán

febrero, 2025 <ver. 0.1>

Índice

Prólogo	2
Requisitos de software y entorno de trabajo	3
Organización del material	4
Fase 1. Modelo de registro de partitura	5
Formas de representación	6
Gramáticas y clases	8
Conversión entre representaciones	11
Código final	21
La primera <i>página</i> web	22
Fase 2. Presentación de un registro de partitura	22
El lenguaje de estilos CSS	22
Estilo de presentación de un registro de partitura	24
Fase 3. Modelo de un archivo de partituras	29
La clase <code>ScoreArchive</code>	29
Casos de prueba de los métodos de <code>ScoreArchive</code>	29
Implementación de los métodos principales de <code>ScoreArchive</code>	31
Ordenación de los registros del archivo: el método <code>sort</code>	32
Código final	34
Generación del fragmento web del fichero de partituras	34
Presentación de un archivo de partituras	35
Fase 4. Cubiertas de las partituras	35
Generación de cubiertas	35
Ampliación del estilo de presentación de un registro de partitura	37
Casos de prueba	39
Código final	40
Fase 5. Información extra sobre el compositor	41
Fase 6. Persistencia y bases de datos	41

Fase 7. La página web	41
Documentos HTML	41
Plantillas remozadas	43
Retoque del estilo	43
Validación de la página web	44
Fase 8. Script de ejecución	44

Índice de figuras

1. partituras - fase 1	23
2. biblio.css	26
3. score.css	28
4. partituras - fase 3	36
5. partituras - fase 4	40
6. partituras - fase 8	46

Prólogo ¹

El objetivo de este trabajo es explicar paso a paso cómo puede construirse una página web de un banco de partituras. En concreto, se creará un programa que produce automáticamente, a partir de un directorio de partituras en formato pdf, un documento HTML —página web— que muestra una colección de registros con información relevante acerca de cada partitura y un enlace para su descarga.

Se objetará que existen ya plataformas que permiten poner a disposición de usuarios autorizados el conjunto de aquellos documentos que hayamos subido al disco duro virtual del correspondiente proveedor de servicios en la Nube. Hay, no obstante, dos razones por las que un proyecto como el que ahora acometemos tiene pleno sentido.

En primer lugar, la creación de un programa especializado nos otorga un control absoluto sobre el contenido de lo que queremos presentar y la forma en la que, a nuestro juicio, ha de presentarse. Logramos, en definitiva, una flexibilidad y una extensibilidad que jamás podrá prometer un programa genérico, por lo demás sujeto a las opciones que sus programadores han prefijado de antemano y que, por añadidura, podrían cambiar en futuras versiones sin consideración alguna hacia nuestros intereses particulares.

La segunda razón es, si cabe, más importante, precisamente por ser la menos utilitaria: en el proceso de construcción de un programa propio accedemos al conocimiento y a la posibilidad de exploración personal del acervo de tecnologías, lenguajes y conceptos que están a la base de la Web moderna y, dicho sea de paso, constituyen asimismo los cimientos sobre los que se levantan las mentadas plataformas en la Nube.

¹Este documento se publica bajo una licencia CC BY-NC-SA 4.0 (Luis Sanjuán © 2025). En su primera versión, el texto es la formulación por escrito de la presentación en vivo que se realizó por primera vez con motivo de las *Jornadas Culturales 2025* en el Conservatorio Profesional de Música de Ávila (España).

Hechas estas observaciones iniciales, conviene aclarar cuál es el alcance de nuestra exposición, qué se espera de los asistentes o lectores y en qué forma se desarrollará la presentación.

Para empezar, el programa en su forma final no podrá ser, por inevitables limitaciones de tiempo, otra cosa que un prototipo. Es de esperar, no obstante, que en el proceso de su construcción se haga evidente que ese resultado provisional puede refinarse o ampliarse ulteriormente de acuerdo con los intereses específicos de los usuarios a los que va dirigido.

Por lo que respecta a la preparación previa que se presupone en el lector oyente, no puedo ocultar el hecho de que una comprensión plena de lo expuesto sólo podrá alcanzarla el programador competente. Conocimientos básicos de programación y de las tecnologías presentadas son suficientes para lograr una perspectiva iluminadora y motivadora en torno a los aspectos fundamentales desarrollados, aunque más de una zona del paisaje quede en penumbra. El resto de los lectores será capaz de ser testigo de la *magia* de la creación de programas, aunque no pueda seguir punto por punto los conceptos y técnicas involucrados. En todo caso, las ideas relevantes y cada tramo principal de la presentación se expondrán con la mayor claridad posible, esto es, sin dar por supuesto todo aquello que es habitual presuponer en este tipo de trabajos altamente técnicos.

Requisitos de software y entorno de trabajo

Para poder reproducir el contenido de lo que se va a presentar a continuación son necesarias las siguientes herramientas de software:

- Un editor de código
- Una distribución de Python

Aunque es posible, en principio, escribir código fuente en cualquier editor de texto plano, resulta mucho más productivo utilizar un editor especializado en programación. A lo largo de la presentación usaré fundamentalmente el editor Vim. Sin embargo, Vim no es recomendable para quien no lo conoce de primera mano. Una buena opción y muy extendida entre la comunidad de programadores es Visual Studio Code de Microsoft.

Puede ser, además, interesante añadir al editor una extensión o ayudante de IA para la edición de código, como Codeium o GitHub Copilot, ambos gratuitos en el momento en que escribo estas líneas.

Por su parte, la distribución de Python —el lenguaje de programación en que el código de nuestro programa se va a escribir—, viene preinstalada en los sistemas operativos Linux y MacOS. En Windows es seguramente necesario proceder a su instalación; la versión de la distribución de Python para el principiante o el usuario ocasional recomendada en el sitio oficial del lenguaje está disponible en la tienda de aplicaciones de Microsoft.

Una vez instaladas las herramientas antes citadas es necesario preparar nuestro entorno de trabajo. Las instrucciones que han de realizarse desde el terminal a tal efecto se exponen a continuación. [El primer paso se puede también llevar a cabo desde un navegador de ficheros.]

1. Creación de un directorio (carpeta) para el proyecto. Utilizaré en adelante el nombre **WebPartituras**.

```
mkdir WebPartituras
```

2. Creación de un entorno virtual de Python desde el directorio que acabamos de crear.

```
cd WebPartituras  
python -m venv .venv
```

3. Activación del entorno virtual de Python. Este paso es necesario cada vez que queremos trabajar en nuestro proyecto.

En Linux o MacOS:

```
source .venv/bin/activate
```

En Windows (cmd.exe):

```
venv\Scripts\activate.bat
```

En Windows (Powershell):

```
venv\Scripts\Activate.ps1
```

4. Desactivación del entorno virtual de Python. Este paso es conveniente cada vez que dejemos de trabajar en nuestro proyecto.

```
deactivate
```

5. Instalación de módulos externos. Puesto que nuestro programa hará uso de utilidades que no están incluidas en la distribución oficial de Python, es necesario disponer de ellas en el entorno virtual del proyecto. Su instalación —que requiere un entorno activado— se realiza mediante **pip**, un programa para gestionar paquetes de Python.

```
pip install -U pytest lark jinja2 pymupdf
```

6. Si la versión de Python es menor o igual a 3.10 —ejecute **python -v** para conocer su versión de Python— hay que incluir también este otro módulo:

```
pip install -U typing_extensions
```

7. Instalación de módulos externos para las fases 5 y 6 [omitidas en esta versión del documento]

```
pip install -U mwclient marshmallow
```

Organización del material

El material requerido y resultante de la exposición en cada una de sus fases se distribuye en sendos subdirectorios 01, 02 , etc.

El entorno virtual de Python se crea en el directorio raíz del proyecto para no tener que repetir su creación en cada paso. El directorio de partituras de ejemplo se ubica también en el directorio raíz bajo el nombre **scores**.

En los directorios correspondientes a cada fase se encuentran los materiales necesarios en cada iteración y el resultado de desarrollo en esa iteración. Así, excepto en las fases iniciales, cada uno de estos directorios tendrá esta estructura común:

```
dir
|- css
|   |-- score.css      # hoja de estilos
|-- img                 # imágenes que usa la página web
|-- partituras.html    # página (o fragmento) web
|-- partituras.py       # programa para generar la página web
|-- score.py            # código del modelo
|-- test_score.py       # tests del código del modelo
```

De los ficheros comentados, el código que iremos escribiendo a lo largo de la exposición, se encuentra en aquellos con extensión .py y .css. Los ficheros .html son producidos en la ejecución del programa, del mismo modo que lo son las imágenes de cubierta de las partituras en el directorio img.

Para ejecutar el código o los tests en cada iteración, tras activar el entorno virtual de Python, hay que moverse al directorio correspondiente a la iteración del momento. Por tanto, la ruta del directorio de partituras que se habrá que procesar estará en `../scores`.

Supongamos, por ejemplo, que estamos en la cuarta iteración del proceso de construcción. En ese caso, y suponiendo que hemos abierto el terminal en el directorio raíz del proyecto WebPartituras, los pasos que habrá que realizar son los siguientes:

```
source .venv/bin/activate # activa el entorno virtual de Python
cd 04                      # cambia al directorio de la iteración 4
partituras.py ../scores     # ejecuta del generador de la página
deactivate                  # desactiva el entorno virtual de Python
```

Por lo que respecta a este documento, se encuentra en el subdirectorio doc del directorio raíz. La fuente está escrita en Markdown y tiene extensión .md. Interesa saber la ubicación, porque los enlaces a los ficheros de código fuente que aparecen a lo largo de él, dan por supuesto que se abrió desde su ubicación original.

Fase 1. Modelo de registro de partitura

La ejecución de toda tarea compleja exige un proceder sistemático y mínimamente planificado. Una primera manifestación de este enfoque en la construcción de programas es lo que se conoce como *desarrollo iterativo*.

En esta exposición construiremos el programa en un serie de fases o iteraciones. En la primera fase se creará un boceto de página web que muestra un único registro de partitura. En la segunda fase se propondrán estilos de presentación de ese boceto de página. En una tercera fase el programa se extenderá a toda una colección de partituras. En fases posteriores se irán introduciendo una a una diversas características que enriquezcan el resultado de esta última fase.

Comencemos pues con la primera tarea, la producción de una página web —más exactamente, como se comprenderá más tarde, un boceto de ella— a partir del fichero pdf de una partitura.

Formas de representación

Tomemos como ejemplo la partitura en formato pdf del Preludio n.^o 1 del compositor Heitor Villa-Lobos en la edición de Max Eschig.

Esta descripción informal —en nuestra lengua— de la partitura se puede expresar formalmente a través de distintos lenguajes artificiales. Tales representaciones formales permiten el inmediato procesamiento de la información mediante procedimientos computacionales. Veamos, a continuación, las representaciones formales que utilizaremos a lo largo de esta tarea inicial.

Nombre de la partitura

La primera forma de representación puede proporcionarla el propio nombre del fichero de la partitura.

`Heitor_Villa+Lobos_Preludio-1_Max-Eschig.pdf`

Excluida la extensión `.pdf` del nombre, los campos de información —compositor, obra y editor, por este orden— aparecen delimitados por el signo ‘`_`’. El contenido de tales campos utiliza a su vez el signo ‘`-`’ para separar las distintas palabras de las que consta la información respectiva y el signo ‘`+`’ para separar las partes de un nombre compuesto, como lo es el apellido ‘Villa-Lobos’.

Diccionarios de Python

En el lenguaje Python podemos utilizar un *diccionario* (el tipo de dato `dict` de Python) para registrar esta misma información:

```
{  
    "composer": "Heitor Villa-Lobos",  
    "work": "Preludio 1",  
    "editor": "Max Eschig"  
}
```

Esta representación es, como se ve inmediatamente, más clara: cada *valor* de información está asociado a una *clave* —“`composer`”, “`work`”, “`editor`”— que define su sentido concreto, de modo que no dependemos del orden en que aparecen esos valores para entender a qué parte del registro de la partitura corresponden.

Por añadidura, un diccionario de Python puede extenderse fácilmente. Consideremos, en concreto, la siguiente extensión:

```
{  
    "composer": "Heitor Villa-Lobos",  
    "work": "Preludio 1",  
    "editor": "Max Eschig",  
    "score": Path("Heitor+Villalobos_Preludio-1_Max-Eschig.pdf")  
}
```

Aquí se ha añadido un nuevo elemento cuyo valor es la ruta en el sistema de fichero del pdf de la partitura, una información evidentemente crucial para el propósito que nos ocupa.

Este segundo diccionario incluye una novedad importante: la ruta del fichero de la partitura aparece enmarcada dentro de la expresión `Path(...)`, donde `Path` es precisamente un objeto para representar rutas en sistemas de ficheros. Se trata, ciertamente, de una indicación mucho más específica de este valor de información —la ubicación en el disco duro— que una mera cadena de caracteres.

Más cercana aún al contexto del problema sería esta otra representación, muy parecida a la anterior, pero con una sutil diferencia.

```
{  
    "composer": "Heitor Villa-Lobos",  
    "work": "Preludio 1",  
    "editor": "Max Eschig",  
    "score": Score(Path("Heitor+Villalobos_Preludio-1_Max-Eschig.pdf"))  
}
```

Ahora la ruta de la partitura aparece dentro de un objeto `Score` (partitura). La idea de esta representación es que la partitura misma contiene su ruta como una de sus propiedades, la propiedad inicial sin duda, pero se deja abierta la opción de que un objeto partitura pueda tener otras propiedades, como la del propio nombre de la partitura u otra característica suya que pueda ser interesante en el diseño del programa.

Objeto `ScoreRecord`

Aún más específica es la siguiente representación, donde la unidad de información tiene el nombre `ScoreRecord`, esto es, un registro de partitura y no un genérico diccionario de Python. La sintaxis es, por lo demás, ligeramente diferente.

```
ScoreRecord(  
    composer="Heitor Villa-Lobos",  
    work="Preludio 1",  
    editor="Eschig",  
    score=Score(Path("Heitor-Villa+Lobos_Preludio-1_Max-Eschig.pdf"))  
)
```

Elemento HTML

El registro de la partitura se puede, finalmente, representar mediante un elemento HTML. El lenguaje HTML es el medio de comunicación de la información en la Web. Esta variante ha de ser, por consiguiente, la representación final del programa, cuya meta no es otra que la de construir una página web.

```
<article class="score-record">  
  <div class="score-link">  
    <a download href="Heitor-Villa+Lobos_Preludio-1_Max-Eschig.pdf"></a>  
  </div>  
  <div class="score-info">  
    <p class="composer">Heitor Villa-Lobos</p>
```

```

<p class="work">Preludio 1</p>
<p class="editor">Max Eschig</p>
</div>
</article>

```

En HTML cada unidad de sentido es un *elemento* del lenguaje. Los elementos, predefinidos por el lenguaje, aparecen marcados mediante etiquetas (*tags*). La mayoría de los elementos requieren una etiqueta de inicio y otra de cierre, como todos los elementos que aparecen en este ejemplo. Los elementos pueden tener atributos. Algunos de estos atributos son comunes a todos los elementos, como el atributo `class`; otros son específicos de un elemento en particular, como los atributos `href` y `download` del elemento `a`.

En este caso hemos creado un elemento `article` de clase `score-record`, que albergará toda la información concerniente al registro de nuestra partitura. Este elemento contiene dos secciones (elementos `div`). La primera, de clase `score-link`, contiene un enlace (elemento `a`) que se refiere al fichero pdf descargable de la partitura. La segunda sección de clase `score-info` contiene los párrafos (elementos `p`) relativos al nombre del compositor, título de la obra y nombre del editor, respectivamente.

Nótese que el atributo `class` en los elementos anteriores nos ayuda a caracterizar mediante las categorías adecuadas los elementos HTML utilizados.

Gramáticas y clases

Desde el punto de vista del programador la representación mediante `dict` y la representación HTML están en lo fundamental a nuestra inmediata disposición: basta conocer las reglas de la sintaxis de Python relativas al tipo de datos `dict` para escribir un registro de partitura mediante un diccionario de Python. Por su parte, el conocimiento del lenguaje HTML nos capacita para crear una representación adecuada a nuestras intenciones de la información de una partitura.

Sin embargo, tanto la forma de representar la misma información en un nombre de fichero o mediante un objeto `ScoreRecord` debe ser descrita de antemano por el programador, ya que se trata de formas de representación ideadas por nosotros mismos y que no forman parte de ningún acervo lingüístico de uso común entre los programadores. En algún sentido ambas formas de representación utilizan un micro-lenguaje propio, que debemos especificar formalmente.

Algo semejante cabe decir de `Path` y `Score`. Mientras que Python suministra de fábrica un modelo para representar rutas en un sistema de ficheros (`Path`) y podemos hacer uso de él con tal de conocer su sintaxis, el modelo para la representación de una partitura (`Score`), en tanto elemento de nuestra propia cosecha, requiere de una definición expresa por nuestra parte.

Grámatica del nombre del fichero de una partitura

Mediante una gramática definimos las reglas de composición que producen un texto escrito construido a partir de ellas.

En nuestro caso tenemos que definir las reglas que permiten generar nombres de ficheros de partituras válidos de acuerdo con la explicación informal dada al principio de esta exposición.

Diremos que una partitura está compuesta de compositor, obra y, opcionalmente, editor —interesa, ciertamente, que el editor sea opcional, ya sea porque resulte irrelevante para la partitura del caso o porque sea desconocido—. Estos campos se separan uno de otro con el signo ‘_’. Expresado formalmente:

```
partitura: compositor "_" obra ("_" editor)?
```

donde el grupo en paréntesis terminado por ‘?’ indica un campo opcional, esto es, que puede estar presente o no.

compositor, obra y editor se componen de palabras:

```
compositor: palabras  
obra: palabras  
editor: palabras
```

Estas palabras están formadas por una o varias palabras separadas por ‘-’:

```
palabras: palabra ("-" palabra)*
```

donde el asterisco final indica que el grupo entre paréntesis puede aparecer 0 o más veces.

Por su parte, una palabra está formada por una o varias subpalabras separadas por ‘+’:

```
palabra: SUBPALABRA ("+" SUBPALABRA)*
```

Finalmente, las así llamadas SUBPALABRAs están formadas por uno o varios caracteres cualesquiera con tal de que sean distintos de los que hemos definido como delimitadores en las reglas anteriores, lo cual se especifica mediante una, así llamada, *expresión regular*:

```
SUBPALABRA: /[^-+_]+/
```

En inglés, y en un sólo bloque, la especificación de la gramática que hemos ideado para los nombres de ficheros de partituras quedaría como sigue [Las dos últimas líneas hacen posible que el analizador que utilizaremos para procesar los nombres de ficheros de acuerdo con estas reglas ignore los espacios en blanco que, por legibilidad, introducimos en esta especificación.]:

```
score: composer "_" work ("_" editor)?
```

```
composer: words  
work: words  
editor: words
```

```
words: word ("-" word)*  
word: SUBWORD ("+" SUBWORD)*
```

```
SUBWORD: /[^-+_]+/
```

```
%import common.WS
```

```
%ignore WS
```

Las clase ScoreRecord y Score

La definición de los objectos `ScoreRecord` para representar un registro de partitura la realizaremos en el lenguaje Python. En dicho lenguaje, como en muchos otros lenguajes de programación, se definen *clases* que establecen los ingredientes o miembros que todos los objetos de esa clase van a tener. Una definición inicial de la clase `ScoreRecord` en Python tendría el siguiente aspecto:

```
from dataclasses import dataclass

@dataclass
class ScoreRecord:
    composer: str
    work: str
    editor: str
    score: Score
```

La última parte de la definición es de suyo comprensible: un registro de partituras, esto es, un objeto `ScoreRecord`, es una *clase de datos* (*dataclass*) que consta de cuatro miembros: el nombre del compositor, el título de la obra, el nombre del editor y la partitura misma. Estos miembros son, por su parte, objetos de clase `str` (cadenas de texto) y de clase `Score` —aún por especificar—. La primera línea de la definición permite aplicar el constructo `dataclass`. En lenguaje Python se *importan* recursos, como aquí `dataclass`, de los *módulos* en los que aparecen definidos, el módulo `dataclasses` en este caso.

Por lo que respecta a la forma de representación de una partitura, utilizaremos la siguiente definición de la clase `Score`:

```
from dataclasses import dataclass, field
from pathlib import Path

@dataclass
class Score:
    path: Path
    name: str = field(init=False)
```

Según esta definición, todo objeto `Score` —toda partitura— tiene una ruta (`path`) de fichero y un nombre (`name`). La ruta del fichero es suficiente para crear un objeto partitura. El nombre de la partitura, sin embargo, aún siendo una propiedad que interesa adjuntar, no se utiliza en el proceso de *inicialización* (*initializing*). Justo eso es lo que significa la expresión `field(init=False)` en la definición.

Por ejemplo, el objeto

```
Score(Path("scores/Franciso-Tárrega_Adelita.pdf"))
```

representa la partitura en formato pdf de Adelita de Tárrega, que está dentro del directorio `scores` y cuyo nombre —como sabemos— es `Francisco-Tárrega_Adelita`.

Conversión entre representaciones

Diagrama de conversiones

En las anteriores secciones hemos introducido diversas formas de representación del registro de una partitura:

- nombre de la partitura
- `dict`
- `ScoreRecord`
- elemento HTML

En las representaciones `dict` y `HTML` nos limitamos a usar la reglas sintácticas y semánticas de los tipos de objetos `dict` de Python y del lenguaje HTML. En la representación mediante el nombre del fichero y en la representación como objeto `ScoreRecord` especificamos las propiedades de ambas mediante, respectivamente, la definición abstracta de una gramática y la definición de una clase de datos de Python.

Estas representaciones se usarán en distintas partes de nuestro programa. Cada una de ellas sirve a un propósito concreto.

Nombre de la partitura La representación inicial de acuerdo a la que nombramos nuestras partituras.

ScoreRecord La representación fundamental que maneja el programa.

dict Representaciones intermedias que usan partes del programa para realizar las conversiones fundamentales.

HTML La representación final sobre la que se construirá la página web.

El siguiente diagrama sintetiza las conversiones entre las distintas representaciones que habrá de realizar el programa en esta versión inicial.

`nombre_partitura -> (dict) -> ScoreRecord -> HTML`

Resulta evidente a partir del diagrama que el objeto central que el programa manipula es un `ScoreRecord`, el cual se crea a partir del nombre del fichero y desde el cual se produce el elemento HTML. Como representación intermedia entre el nombre del fichero y el `ScoreRecord` se recurre también a la representación `dict`.

Funciones y métodos

Antes de extraer las consecuencias que conlleva la reflexión anterior tenemos que introducir los conceptos de *función* y *método*.

Consideremos la siguiente operación matemática, que todos los niños saben realizar:

$$1 + 1 = 2$$

¿Tiene ella algo que ver con, por ejemplo, introducir un término de búsqueda en Google y obtener una lista de las páginas web que coinciden con ese término?

Aparentemente no pueden ser cosas más divergentes. Sin embargo, desde cierto punto de vista, son ejemplos del mismo concepto. Veámoslo.

Decimos que la suma consume dos números y produce como resultado otro número, la suma de aquellos. Por su parte, en la búsqueda dentro de Google se consume una palabra y se produce una lista de enlaces (URL) de las páginas web coincidentes.

Bosquejemos un diagrama a partir de esta descripción.

1. Suma de dos números:

```
número  |
         |-- suma --> número
número  |
```

2. Búsqueda en Internet:

```
palabra -- busca --> lista de URL
```

Las operaciones `suma` y `busca` son ambas *funciones*. La primera de ellas consume dos números —los *argumentos* de la función— y produce otro número; la segunda consume una palabra —el argumento de la función— y produce una lista de direcciones URL.

Una definición de estas funciones en Python tendría el siguiente aspecto:

1. Suma de dos números:

```
def suma(n: número, m: número) -> número:
    pass
```

2. Búsqueda en Google:

```
def busca(p: palabra) -> Lista[URL]:
    pass
```

Los argumentos de cada función tienen un nombre —`n`, `m`, `p`— elegido por nosotros. Tras su nombre viene especificado el tipo de dato del argumento respectivo. Después de la flecha que viene a continuación de los argumentos se indica el tipo de dato resultante. La palabra clave `pass` está aquí en lugar del código aún por escribir que haría que estas funciones produjeran el resultado deseado.

Consideremos ahora un par de ejemplos ligeramente distintos: pulsamos sobre un pájaro de un videojuego, que, al hacerlo, salta ; seleccionamos una palabra en el procesador de texto y pulsamos el botón de negrita, lo que provoca que esa palabra aparezca resaltada.

```
Pájaro.salta -> Pájaro
```

```
Palabra.negrita -> Palabra
```

En Python ambas operaciones se definirían como *métodos* de la clase de objetos `Pájaro` y `Palabra`, respectivamente, donde `Self` indica que el tipo de dato es de la clase en cuestión. Dicho de otra forma, el método `salta` consume y produce un `Pájaro` y el método `negrita` consume y produce una `Palabra`.

```
class Pájaro:
    def salta(self: Self) -> Self:
        pass
```

```
class Palabra:
    def negrita(self: Self) -> Self:
        pass
```

Más común es, sin embargo, suprimir la anotación del tipo `Self` para el primer argumento obligatorio de un método en Python, puesto que el propio nombre del argumento, `self`, apunta justamente a que se trata de un objeto de la clase en cuestión. Así pues, las siguientes serían formas de definición más habituales:

```
class Pájaro:
    def salta(self) -> Self:
        pass

class Palabra:
    def negrita(self) -> Self:
        pass
```

El concepto de método es, pues, semejante al de función. La diferencia esencial estriba en que la operación que efectúa el método se realiza sobre un objeto de la clase en la que tal método se define. Todos los objetos de esa clase realizan las operaciones así definidas: todos los pájaros del video-juego pueden saltar y todas las palabras en un procesador de texto pueden ponerse en negrita.

Resulta conveniente en ciertos contextos definir clases de objetos con sus propios métodos en lugar de simples funciones. Aunque el programa que vamos a desarrollar se puede escribir de un modo puramente funcional, me ha parecido más apropiado por razones que tienen que ver fundamentalmente con la organización de esta presentación utilizar principalmente clases y objetos.

Veamos, para terminar esta sección, la sintaxis de Python relativa al uso de las funciones y métodos previamente definidas. Nótese, en particular, la diferencia de sintaxis entre la invocación de un método y la invocación de una función.

```
suma(1, 1) # suma los números 1 y 1
busca("Ávila") # busca páginas web con el término Ávila
pepito = Pájaro("pepito") # crea un pájaro que llamamos pepito
pepito.salta # pepito salta
hola = Palabra("hola") # crea la palabra hola
hola.negrita # la palabra hola se pone en negrita
```

Diseño de los métodos de un registro de partitura.

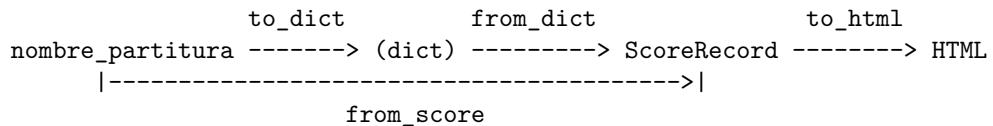
Tras esta digresión sobre las nociones de función y método, regresemos a nuestro asunto principal.

Recordemos el esquema de las conversiones entre las distintas formas de representar el registro de una partitura.

```
nombre_partitura -> (dict) -> ScoreRecord -> HTML
```

Cada una de estas transformaciones de una forma de representación en otra puede realizarla un método ².

²Hemos elegido un nombre apropiado, en inglés, para cada una de ellas. Utilizar el inglés evita que, en adelante, el código contenga una extraña mezcolanza de palabras castellanas e inglesas.



El propósito de cada una de estas operaciones es claro:

to_dict Convierte el nombre de la partitura en un diccionario
from_dict Crea un registro de partitura a partir de un diccionario
from_score Crea un registro de partitura a partir de ella
to_html Convierte un registro de partitura en HTML

Con excepción de **to_dict** todas las conversiones tienen como punto de origen o de llegada un **ScoreRecord**. En consecuencia, tales operaciones pueden y, en general, deben ser incorporadas como métodos en la definición de **ScoreRecord**.

Ahora bien, es necesario diferenciar los métodos que conciernen a un objeto **ScoreRecord** en concreto —aquí, **to_html** que transforma un determinado registro de partitura en su representación HTML—, de aquellos otros que sirven para crear objetos **ScoreRecord** —en el caso presente **from_dict** y **from_score**, los cuales construyen un **ScoreRecord** a partir de un diccionario, o bien a partir de la partitura misma—. Es evidente que estos últimos métodos no pueden ser aplicados a un determinado objeto de la clase **ScoreRecord**, ya que todavía no existe. Este tipo de métodos se llaman *constructores*, pues sirven para crear o construir objetos de la clase en cuestión. Python proporciona un constructor por defecto y que, sin necesidad de definición propia, puede invocarse en las clases de tipo **dataclass** como la nuestra. Tal es el caso del ejemplo inicialmente propuesto de [registro de partitura](#). Por su parte, constructores especializados como **from_dict** y **from_score** deben definirse en Python como métodos de clase (**classmethod**).

A la luz de todas estas observaciones nuestra clase **ScoreRecord** quedaría ampliada de la siguiente manera:

```

from dataclasses import dataclass

@dataclass
class ScoreRecord:
    composer: str
    work: str
    editor: str
    score: Score

    @classmethod
    def from_dict(cls, d: dict) -> Self:
        """Construct a ScoreRecord from the given dictionary."""
        pass

    @classmethod
    def from_score(cls, s: Score) -> Self:
        """Construct a ScoreRecord from the given score."""
        pass

```

```

def to_html(self) -> str:
    """Convert the ScoreRecord into an HTML element."""
    pass

```

Diseño de la clase Score

Antes de implementar los métodos de la clase `ScoreRecord` hemos de volver a la única de las operaciones de conversión que quedó sin explicar en las secciones precedentes, esto es, la operación `to_dict`, que convierte la partitura o, más exactamente, su nombre en un diccionario de Python.

Resulta evidente tras la discusión anterior que `to_dict` no puede definirse como un método de un registro de partitura. En concreto, un registro de partitura no es ni el origen ni el destino de la conversión a que esa operación se refiere. En efecto, su destino es, obviamente, un diccionario y su origen es, también claramente, la partitura misma. En otras palabras: `to_dict` consume una partitura y produce un diccionario. Pero —como ya sabemos— esto mismo se puede expresar como un método de la clase `Score`.

Una instancia de esa clase se construiría a partir de la ruta en el sistema de ficheros de la partitura:

```
Score(Path("Heitor-Villa+Lobos_Preludio-1_Max-Eschig.pdf"))
```

Tal clase tendría, por tanto, como campo fundamental —y el único necesario en la construcción de objetos de la clase— el campo `path` (ruta). Ahora bien, puesto que la conversión al diccionario se produce sobre el nombre de la partitura en lugar de sobre su ruta, tiene sentido que la clase `Score` tenga también un campo correspondiente, que denominamos `name`. Este campo puede computarse a partir del campo `path` muy fácilmente. Todo ello se puede expresar como sigue:

```

from dataclasses import dataclass, field
from pathlib import Path

@dataclass
class Score:
    path: Path
    name: str = field(init=False)

    def __post_init__(self):
        self.name = self.path.stem

```

El método especial `__post_init__` de una `dataclass` es precisamente el medio apropiado para computar los valores de aquellos campos de la clase —aquí `name`— que dependen de otros miembros primarios —aquí `path`—. El nombre de la partitura es simplemente lo que queda de suprimir la *extensión* y el *ancestro (parent)* de la ruta del fichero, esto es, lo que queda de extraer únicamente el *tallo (stem)* de la ruta. Así, por ejemplo, el nombre de la partitura `/home/luis/Partituras/Johann-Sebastian-Bach_Sarabande-995.pdf` sería `Johann-Sebastian-Bach_Sarabande-995`, donde la extensión `.pdf` y el ancestro `/home/luis/Partituras` han sido omitidos.

Si añadimos ahora el método `to_dict` a `Score` obtendríamos el siguiente diseño

inicial de la clase.

```
from dataclasses import dataclass, field
from pathlib import Path

@dataclass
class Score:
    path: Path
    name: str = field(init=False)

    def __post_init__(self):
        self.name = self.path.stem

    def to_dict(self) -> dict:
        """Convert the Score name into a dictionary."""
        pass
```

Ejemplos y verificaciones

Naturalmente, falta completar la definición de los métodos declarados en las clases `ScoreRecord` y `Score`, esto es, sustituir el enunciado `pass` en el *cuerpo* de tales métodos por el código real que realiza su propósito.

Pero antes de acometer esta última tarea es recomendable —y en buena medida necesario— crear ejemplos de uso de los métodos y de su resultado esperado, lo que técnicamente se denominan *test cases* (casos de prueba). A través de ellos nos aseguraremos de que nuestro código cumplirá su propósito correctamente, sea cual sea la implementación que barruntamos ahora o establezcamos en futuras versiones del programa.

En Python podemos construir estos tests mediante la utilidad `pytest`. Por ejemplo, si quisieramos verificar la función `suma`, anteriormente descrita, cuyo objetivo es sumar dos números, escribiríamos un caso de prueba de la siguiente forma:

```
def test_suma():
    assert suma(1, 1) == 2
```

Creemos, pues, unos pocos casos de prueba por cada uno de los métodos anteriormente declarados en `ScoreRecord` y `Score`. [Las funciones decoradas con `@pytest.fixture`, que producen los ejemplos que probar, son convenientes, aunque no necesarias, para escribir los tests subsiguientes de una forma más legible y sencilla.³]

```
import pytest
from pathlib import Path
```

```
# Examples -----
## Score
@pytest.fixture
def score1():
```

³En un producto acabado el número de ejemplos y tests debería ser considerablemente mayor.

```

    return Score(Path("Heitor-Villa+Lobos_Preludio-1_Max-Eschig.pdf"))

@pytest.fixture
def score2():
    return Score(Path("Heitor-Villa+Lobos_Preludio-1.pdf"))

## score dict
@pytest.fixture
def scoredict1(score1):
    return {
        "composer": "Heitor Villa-Lobos",
        "work": "Preludio 1",
        "editor": "Max Eschig",
        "score": score1
    }

@pytest.fixture
def scoredict2(score2):
    return {
        "composer": "Heitor Villa-Lobos",
        "work": "Preludio 1",
        "editor": "",
        "score": score2
    }

## ScoreRecord
@pytest.fixture
def scorerecord1(score1):
    return ScoreRecord(
        composer="Heitor Villa-Lobos",
        work="Preludio 1",
        editor="Max Eschig",
        score=score1
    )

## HTML
@pytest.fixture
def scorehtml1():
    return """
<article class="score-record">
    <div class="score-link">
        <a download href="Heitor-Villa+Lobos_Preludio-1_Max-Eschig.pdf">
        </a>
    </div>
    <div class="score-info">
        <p class="composer">Heitor Villa-Lobos</p>
        <p class="work">Preludio 1</p>
        <p class="editor">Max Eschig</p>
    </div>
</article>
"""

```

```

    """
# Utils -----
def normalize_html(element):
    """Format HTML element to a single line."""
    return ''.join([s.strip() for s in element.split("\n")])

# Tests -----
## Score methods
def test_score_to_dict(score1, scoredict1, score2, scoredict2):
    assert score1.to_dict() == scoredict1
    assert score2.to_dict() == scoredict2

## ScoreRecord methods
def test_scorerecord_from_dict(scoredict1, scorerecord1):
    assert ScoreRecord.from_dict(scoredict1) == scorerecord1

def test_scorerecord_from_score(score1, scorerecord1):
    assert ScoreRecord.from_score(score1) == scorerecord1

def test_scorerecord_to_html(scorerecord1, scorehtml1):
    assert (
        normalize_html(scorerecord1.to_html())
        == normalize_html(scorehtml1)
    )

```

Implementación de los métodos de Score y de ScoreRecord

Llegamos por fin a la fase de implementación: la creación del código que realiza el propósito de cada método de `Score` y `ScoreRecord`. No podemos, dado el carácter introductorio de esta exposición, explicar en detalle la implementación de los métodos de las clases `Score` y `ScoreRecord`. El lector que conozca bien Python encontrará la implementación que propongo fácil de seguir. Quien no, puede buscar la ayuda de un agente artificial. Hoy por hoy, la IA es notablemente certera a la hora de realizar esta clase de trabajos por nosotros. En todo caso, cada implementación se comentará hasta el punto en que, al menos, pueda ser entendida en términos generales.

Empecemos con los constructores de `ScoreRecord`. La implementación de `from_dict` es muy sencilla:

```

@classmethod
def from_dict(cls, d: dict) -> Self:
    """Construct a ScoreRecord from the given dictionary."""
    return cls(**d)

```

`from_dict` produce un objeto de la clase en que se define (`cls`), esto es, un objeto `ScoreRecord`, a partir de los campos del diccionario `d`, que se desempaquetan mediante el operador `**` para ser pasados como argumentos al constructor por defecto `ScoreRecord()`.

Por su parte, la implementación de `from_score` no es más que la composición

de `from_dict` y `to_dict`. En efecto, dada la partitura, `to_dict` la convierte en un diccionario a partir del que `from_dict` construye el `ScoreRecord`.

```
@classmethod
def from_score(cls, s: Score) -> ScoreRecord:
    """Construct a ScoreRecord from the given score."""
    return cls.from_dict(s.to_dict())
```

La implementación de `to_dict` exige más trabajo. Su propósito es, como se recordará, convertir el nombre de la partitura en un diccionario tomando como base la gramática presentada al principio:

```
GRAMMAR = """
score: composer "_" work ("_" editor)?
composer: words
work: words
editor: words

words: word ("-" word)*
word: SUBWORD ("+" SUBWORD)*

SUBWORD: /[^-+_-]+/

%import common.WS
%ignore WS
"""

"""

El módulo externo lark dispone de medios para escribir analizadores sintácticos basados en gramáticas de este tipo. En particular, el código siguiente:
```

```
from lark import Lark

parser = Lark(GRAMMAR, start="score")
parser.parse("Heitor-Villa+Lobos_Preludio-1_Max-Eschig")
```

produce un árbol sintáctico del nombre de la partitura de Villalobos:

```
score
  composer
    words
      word Heitor
      word
        Villa
        Lobos
  work
    words
      word Preludio
      word 1
  editor
    words
      word Max
      word Eschig
```

`lark` cuenta asimismo con herramientas para transformar árboles sintácticos como el precedente en cualquier objeto de nuestro interés. El código que sigue a continuación define un *transformador* de Lark, que denominamos `ScoreNameTransformer` donde La función `zip_longest` del módulo de Python `itertools` permite producir el diccionario deseado; en particular, hace posible que el campo del editor pueda ser una cadena vacía cuando el nombre de la partitura no incluye editor alguno.

```
from itertools import zip_longest

from lark import Token, Transformer

class ScoreNameTransformer(Transformer):
    FIELDS: ClassVar[list[str]] = ["composer", "work", "editor"]

    def score(self, items: list[str]) -> dict:
        return dict(zip_longest(self.FIELDS, items, fillvalue=""))

    def words(self, items: list[str]) -> str:
        return " ".join(items)

    def word(self, items: list[Token]) -> str:
        return "-".join(items)

    composer = words
    work = words
    editor = words
```

El método `to_dict` analiza el nombre de la partitura —basándose en la gramática establecida—, lo que produce un árbol sintáctico, el cual es luego transformado mediante el transformador `ScoreNameTransformer` en un diccionario. A este diccionario, que recoge únicamente la información acerca de la partitura, se le añade el campo `score`, cuyo valor es la ruta de la partitura. El diccionario así extendido es el resultado de `to_dict`

```
def to_dict(self) -> dict:
    """Convert the Score name into a dictionary."""
    parser = Lark(GRAMMAR, start="score")
    tree = parser.parse(self.name)
    scoreinfo = ScoreNameTransformer().transform(tree)
    return scoreinfo | {"score": self}
```

Queda, finalmente, por implementar el método que convierte nuestro registro de partitura en una representación HTML, objetivo último de la tarea.

La idea fundamental aquí es utilizar una plantilla HTML que funcione como modelo de cualquier registro de partitura. El módulo externo `Jinja2` nos prové de todo lo necesario para lograrlo.

En primer lugar, creamos una plantilla HTML mediante la sustitución de las partes de la representación HTML de un registro de partitura de ejemplo por las referencias a los campos de ese registro que convienen a cada elemento de tal

representación. Más sucintamente, dada la representación HTML del registro del Preludio 1 de Villalobos antes presentada:

```
<article class="score-record">
  <div class="score-link">
    <a download href="Heitor-Villa+Lobos_Preludio-1_Max-Eschig.pdf"></a>
  </div>
  <div class="score-info">
    <p class="composer">Heitor Villa-Lobos</p>
    <p class="work">Preludio 1</p>
    <p class="editor">Max Eschig</p>
  </div>
</article>
```

reemplazamos la información específica de esa partitura por referencias a los campos correspondientes de su representación como `ScoreRecord`:

```
SCORE_HTML_TEMPLATE = """
<article class="score-record">
  <div class="score-link">
    <a download href="{{scorerecord.score.path}}></a>
  </div>
  <div class="score-info">
    <p class="composer">{{scorerecord.composer}}</p>
    <p class="work">{{scorerecord.work}}</p>
    <p class="editor">{{scorerecord.editor}}</p>
  </div>
</article>
"""
```

Finalmente, creamos un objeto `Template` a partir de esa plantilla y ejecutamos sobre él el método `render`, que realiza la sustitución inversa, esto es, rellena las partes variables de la plantilla con los valores concretos del registro de partitura del caso. La implementación de `to_html` queda, pues, como sigue:

```
from jinja2 import Template

def to_html(self) -> str:
    """Convert the ScoreRecord into an HTML element."""
    return Template(SCORE_HTML_TEMPLATE).render(scorerecord=self)
```

Código final

Si unimos todas las piezas, obtenemos el código completo que resuelve nuestra tarea inicial en esta primera fase del desarrollo. Lo guardamos en el fichero `01/score.py`. Mientras que los casos de prueba se guardan en `01/test_score.py`, el cual debe logicamente importar las clases cuyos métodos han de verificarse:

```
from score import Score, ScoreRecord
```

Si ejecutamos la utilidad `pytest` dentro del directorio donde hemos guardado ambos archivos, el del código fuente y el de los tests, obtendremos el resultado

deseado: las cuatro unidades de test que hemos implementado superan la verificación, el programa es básicamente correcto, al menos hasta el punto en que hace lo que esperamos que haga en esos pocos casos de prueba.

La primera página web

Se objetará —con toda razón— que hasta ahora no nos hemos topado con nada parecido a una página web. ¡Ni siquiera hemos abierto el navegador! Lo único que hemos logrado —según parece— es construir un programa que permite obtener una representación HTML de un registro de partitura a partir de una partitura dada. ¡Y tampoco lo hemos probado con una partitura real!

Antes de pasar a la siguiente fase y por no dejar a nadie con la miel en los labios y un rictus de justificable disgusto, se adjunta una receta que abrirá desde el terminal el navegador con la dichosa página —más bien, un fragmento de ella si queremos ser totalmente sinceros, pero una que un navegador moderno es capaz de gestionar como página web—.

1. Edite un fichero de nombre `partituras.py` con el siguiente contenido:

```
import sys
from pathlib import Path

from score import Score, ScoreRecord

if __name__ == "__main__":
    score = Score(Path(sys.argv[1]))
    scorerecord = ScoreRecord.from_score(score)
    html_page = scorerecord.to_html()
    with open("partituras.html", "w") as f:
        f.write(html_page)
```

2. Ejecute el código. El segundo argumento es la partitura a partir de la que se construye el fragmento de página web. Puede elegir cualquier otra, siempre que se adecue a la gramática de nombres de partitura que definimos en las secciones precedentes.

```
python partituras.py Heitor-Villa+Lobos_Preludio-1_Max-Eschig.pdf
```

3. Abra la página `partituras.html` recién creada. Por ejemplo:

```
firefox partituras.html
```

Se mostrará una página web con [este aspecto](#):

Fase 2. Presentación de un registro de partitura

El lenguaje de estilos CSS

¿Todo el arduo trabajo y ese sofisticado aparato lógico para un resultado tan soso? —se preguntará el lector—. ¡Y encima, ni tan siquiera se ve el enlace para descargar la partitura! Las apariencias engañan, no obstante. En esta sección,

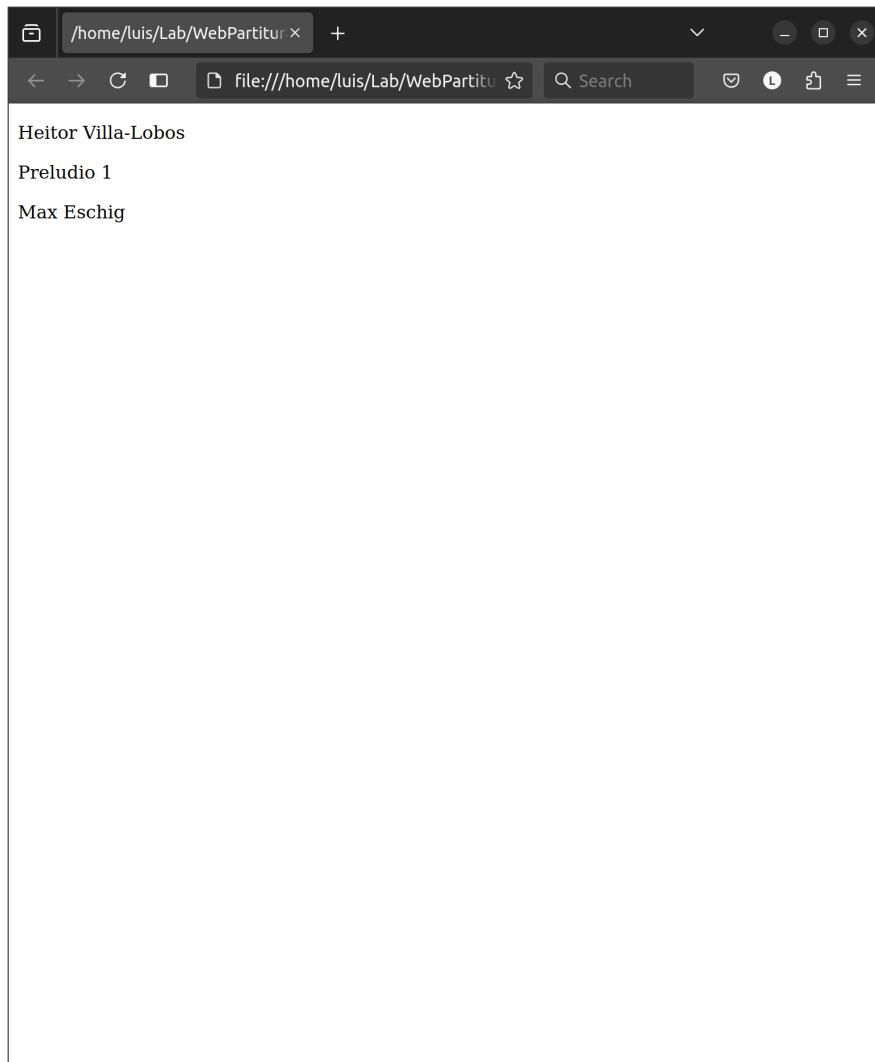


Figura 1: partituras - fase 1

más breve y directa que la anterior, se verá cómo hacer que la *apariencia* de nuestra página sea algo más atractiva.

Lo que el navegador nos mostró antes —Firefox, en este caso, pero el resultado es esencialmente el mismo en todos los navegadores convencionales— es una posible presentación del fragmento HTML producido por nuestro programa. Más exactamente, Firefox utiliza un *estilo* por defecto, sin duda, extremadamente austero, construido mediante el lenguaje de hojas de estilo CSS. Por medio de ese mismo lenguaje podemos dotar a nuestra página de un aspecto menos aburrido o, al menos, más acorde con nuestras pretensiones.

CSS es el acrónimo de *Cascading Style Sheets* —hojas de estilo en cascada—. Su idea fundamental es la siguiente: el estilo de presentación de una página web (documento HTML) se especifica a través de la definición de *reglas* para cada uno de los elementos HTML de la página cuyo estilo se desea personalizar. En estas reglas se establecen los valores de las propiedades de presentación de esos elementos como, por ejemplo, el tamaño de su fuente, la cantidad de márgen con respecto a otros elementos, la dimensión de la caja en que está inserto, etc.—.

Esta idea es ciertamente muy simple y la sintaxis con que se escriben las reglas de estilo es en lo fundamental muy fácil de entender y de aprender. Quizá la mayor dificultad inicial del lenguaje —que la hay— radica en la inmensidad de su vocabulario, es decir, en la gran variedad de propiedades que cada elemento puede tener y en la gran cantidad de valores posibles de esas propiedades.

Puesto que explorar CSS a fondo es una tarea larga que no puede acometerse en un trabajo de estas dimensiones, en esta sección nos limitaremos a utilizar CSS en la medida en que cubra nuestros humildes intereses. No se explicará la sintaxis, porque tras la lectura de dos o tres reglas resulta evidente de suyo. Tampoco se explica la pléthora de opciones y valores posibles, sino que se indica en cada caso la propiedad y el valor que conviene al diseño que deseamos obtener.

Estilo de presentación de un registro de partitura

Antes de empezar y por ser ésta una sección eminentemente práctica, se recomienda utilizar el navegador Firefox, que es, a día de la escritura de este documento, el único de entre los tres más usados —Chrome e Edge son los otros dos—, que lleva incorporado un editor de estilo que permite ir observando en vivo el resultado de aplicar las reglas CSS a medida que las vamos escribiendo. Para ello, abrimos con Firefox la página cuyo diseño se quiere elaborar, pulsamos **Ctrl+May+I** —lo que abre las herramientas de desarrollo del navegador—, seleccionamos la pestaña del editor de estilos y pulsamos el enlace para añadir una nueva hoja de estilo.

Estilo de cita bibliográfica

Como primer ejemplo —primer experimento— vamos a crear un estilo en que la información sobre la partitura se muestre como una cita bibliográfica.

En la primera regla establecemos el diseño del registro de partitura —el elemento de clase `score-record`— como una rejilla de dos columnas, con la primera columna con una anchura de cuatro emes —el tamaño de la letra *m* se puede utilizar en CSS como unidad de medida— y la segunda con la anchura necesaria, que dejamos que calcule automáticamente el navegador.

```
.score-record {
  display: grid;
  grid-template-columns: 4em auto;
}
```

En la segunda regla añadimos el texto ‘[PDF]’ como contenido del enlace de descarga de la partitura, de esta forma el enlace se hace visible y se puede pinchar en él.

```
.score-link a::after {
  content: "[PDF]";
}
```

En la tercera regla determinamos que los elementos de información —los párrafos relativos a compositor, obra y editor— se muestren en una única línea y no en líneas independientes.

```
.score-info p {
  display: inline;
}
```

En la cuarta regla añadimos un punto tras estos campos de información, como es convencional en los estilos de cita bibliográfica

```
.score-info p::after {
  content: ".";
}
```

Finalmente, utilizamos mayúsculas pequeñas (versalitas) como variante de la fuente del nombre del compositor.

```
.composer {
  font-variant: small-caps;
}
```

El resultado se muestra en la ventana izquierda de la [imagen](#) siguiente.

Estilo de carpeta desplegable

El siguiente experimento muestra la información como una carpeta desplegable de cierto carácter lúdico. Se trata de un estilo más complejo de diseñar. Se deja sin más comentario como ejemplo de que no siempre es necesario seguir los estilos de diseño tradicionales. Se recomienda al lector interesado consultar la documentación en [MDN Web Docs](#) si quiere desentrañar el significado de las reglas que el ejemplo utiliza.

```
.score-link {
  width: min-content;
}

.score-link a {
  display: inline-block;
  border: 1px solid black;
  border-top-right-radius: 10px;
  padding: 0.5rem;
```

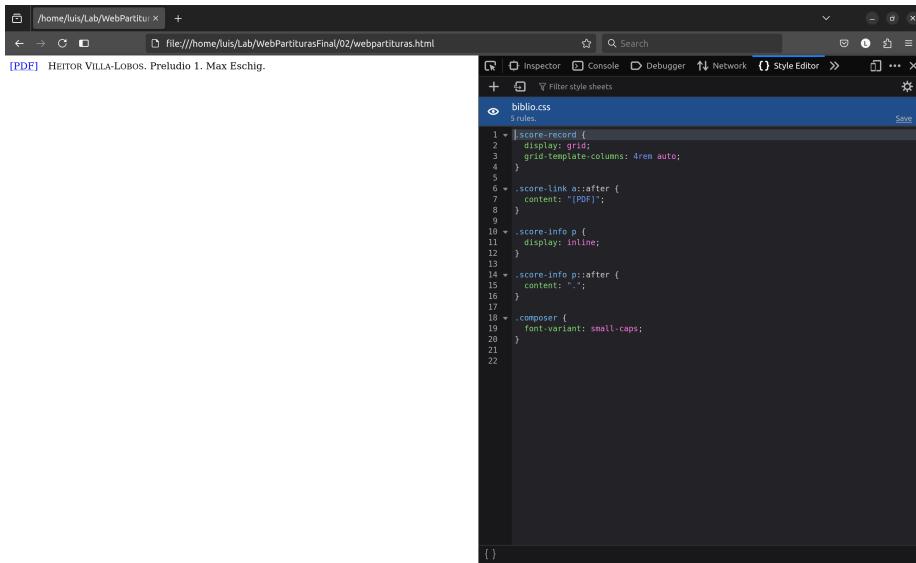


Figura 2: biblio.css

```

}

.score-link a::after {
  content: "?";
}

.score-link a:hover::after {
  content: "pdf";
}

.score-link+.score-info {
  display: none;
  border: 1px solid black;
  border-top-right-radius: 5px;
  padding: 0.5rem;
}

.score-link:hover+.score-info {
  display: inline-block;
}

```

Se invita al lector a que inspeccione el resultado, observe por sí mismo su componente interactivo y descubra qué reglas intervienen en la producción de ese efecto.

Estilo de miniatura de partitura

El último experimento es el que adoptaremos provisionalmente como estilo final. La información de la partitura se mostrará de un modo similar al que es habitual

en muchas páginas web que incluyen catálogos de partituras o de cualquier otro tipo de objeto que pueda descargarse y contenga información acerca de él: por medio de una imagen de la partitura —en esta ocasión, de una partitura en blanco— con información añadida debajo de ella.

Para empezar, creamos una regla que establezca que la caja en la que se ubica cada elemento presentado sea tal que incluya dentro de ella los bordes y rellenos. Esta es una regla frecuente entre los desarrolladores web que facilita la predictibilidad del diseño y la facilidad de calcular las dimensiones.

```
*, *::after {  
    box-sizing: border-box;  
}
```

El color de fondo de la página es rojo oscuro.

```
body {  
    background-color: darkred;  
}
```

El registro de partitura es una rejilla de dos filas, la primera de las cuales, que corresponde a la sección que contiene el enlace para la descarga de la partitura, tiene una altura de 420 píxeles. La anchura de la caja que contiene el registro de partitura es de 300 píxeles. El borde de dicha caja, de 3 píxeles de grosor, es de trazo continuo (`solid`) y de color amarillo. El color de fondo de la caja, blanco navajo. Además, hay un relleno entre los bordes superior e inferior y el contenido del registro de anchura equivalente a una eme.

```
.score-record {  
    display: grid;  
    grid-template-rows: 420px auto;  
    width: 300px;  
    border: 3px solid yellow;  
    background-color: navajowhite;  
    padding: 1rem 0;  
}
```

La fila superior, que incluye el enlace para la descarga de la partitura, es también una rejilla en la que todo su contenido aparece centrado. Posee una imagen de fondo de una partitura en blanco —guardada en el fichero `img/blank-score.jpg`—, con un tamaño de 280 por 420 píxeles, que no se repite y que está centrada.

```
.score-link {  
    display: grid;  
    place-content: center;  
    background-image: url(img/blank-score.jpg);  
    background-size: 280px 420px;  
    background-repeat: no-repeat;  
    background-position: center;  
}
```

El enlace mismo es un bloque cuyo contenido es otra imagen —un ícono que representa la acción de descargar— guardada en el fichero `img/download-icon.png`. El ícono está escalado al 80 % de su tamaño original.

```
.score-link a {
    display: inline-block;
    content: url(img/download-icon.png);
    transform: scale(80%);
}
```

La fila inferior, que contiene la información acerca de la partitura, es también una rejilla cuyo contenido aparece centrado y se separa de la sección precedente por un margen superior de una eme de altura.

```
.score-info {
    display: grid;
    place-content: center;
    margin-top: 1rem;
}
```

Finalmente, cada una de los elementos de información —los párrafos que indican el nombre del compositor, el título de la obra y el nombre del editor— carecen de márgenes, están centrados y tienen un relleno superior e inferior de 0.3 emes.

```
.score-info p {
    margin: 0;
    text-align: center;
    padding: 0.3rem 0;
}
```

He aquí el [resultado](#) que aparece en el navegador:

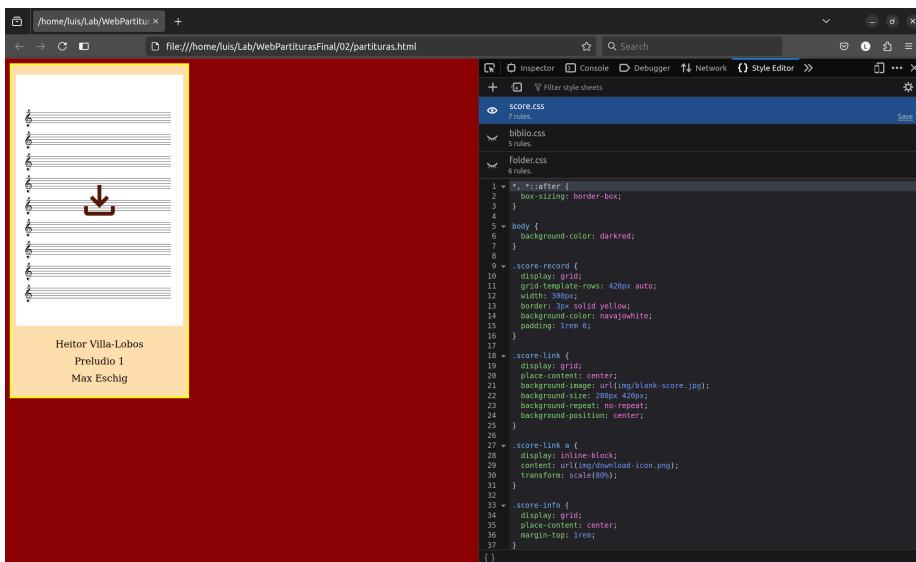


Figura 3: score.css

Fase 3. Modelo de un archivo de partituras

La clase ScoreArchive

Sobre la base de la infraestructura creada en la primera fase de nuestro trabajo, podemos acometer ahora la tarea que constituye el objetivo primordial que nos propusimos, la creación de una página web de partituras. El objeto mostrado en esa página es el conjunto de todos los registros de partitura producidos a partir de un directorio de ficheros pdf de ellas. A este conjunto de registros de partituras lo llamaremos —por brevedad— un archivo de partituras (*Score Archive*).

Si seguimos la misma estrategia aplicada para modelar un registro de partitura, deberíamos definir un tipo de objeto que represente un archivo de partituras. Los métodos principales de un archivo de partituras —como el lector atento habrá colegido ya por sí mismo— habrán de ser aquel que construye el archivo a partir de una lista de partituras y aquel otro que produce el documento HTML correspondiente a ese archivo. Expresado en lenguaje Python:

```
from collections import UserList

class ScoreArchive(UserList):
    @classmethod
    def from_scores(self, scores: list[Score]) -> Self:
        """Construct the ScoreArchive from the given scores."""
        pass

    def to_html(self) -> str:
        """Convert the ScoreArchive into an HTML element."""
        pass
```

Casos de prueba de los métodos de ScoreArchive

El preludio de la implementación de los métodos de `ScoreArchive` es —como ya sabemos— la creación de casos de prueba.

```
import pytest
from pathlib import Path
from score import Score, ScoreRecord, ScoreArchive

# Examples -----
## Score
@pytest.fixture
def score3():
    return Score(Path("Francisco-Tárrega_Adelita.pdf"))

@pytest.fixture
def score4():
    return Score(Path("Francesco-da-Milano_Fantasía_Ruggero-Chiesa.pdf"))

@pytest.fixture
def score5():
```

```

    return Score(Path("Anónimo_Greensleeves.pdf"))

## list of Score
@pytest.fixture
def scores1(score3, score4, score5):
    return [score3, score4, score5]

## ScoreRecord
@pytest.fixture
def scorerecord3(score3):
    return ScoreRecord.from_score(score3)

@pytest.fixture
def scorerecord4(score4):
    return ScoreRecord.from_score(score4)

@pytest.fixture
def scorerecord5(score5):
    return ScoreRecord.from_score(score5)

## ScoreArchive
@pytest.fixture
def scorearchive1(scorerecord3, scorerecord4, scorerecord5):
    return ScoreArchive([scorerecord3, scorerecord4, scorerecord5])

## HTML
@pytest.fixture
def scorearchivehtml1():
    return """
        <section class="score-archive">
            <article class="score-record">
                <div class="score-link">
                    <a download href="Francisco-Tárrega_Adelita.pdf">
                    </a>
                </div>
                <div class="score-info">
                    <p class="composer">Francisco Tárrega</p>
                    <p class="work">Adelita</p>
                    <p class="editor"></p>
                </div>
            </article>
            <article class="score-record">
                <div class="score-link">
                    <a download href="Francesco-da-Milano_Fantasía_Ruggero-Chiesa.pdf">
                    </a>
                </div>
                <div class="score-info">
                    <p class="composer">Francesco da Milano</p>
                    <p class="work">Fantasía</p>
                    <p class="editor">Ruggero Chiesa</p>
                </div>
            </article>
        </section>
    """

```

```

        </div>
    </article>
<article class="score-record">
    <div class="score-link">
        <a download href="Anónimo_Greensleeves.pdf">
            </a>
    </div>
    <div class="score-info">
        <p class="composer">Anónimo</p>
        <p class="work">Greensleeves</p>
        <p class="editor"></p>
    </div>
</article>
</section>
"""

# Tests -----
## ScoreArchive methods
def test_scorearchive_from_scores(scores1, scorearchive1):
    assert ScoreArchive.from_scores(scores1) == scorearchive1

def test_scorearchive_to_html(scorearchive1, scorearchivehtml1):
    assert (
        normalize_html(scorearchive1.to_html())
        == normalize_html(scorearchivehtml1)
    )

```

Implementación de los métodos principales de ScoreArchive

El método `from_scores` construye un `ScoreArchive` a partir de una lista de partituras. Para ello convierte cada partitura de esa lista en un registro de partitura (`ScoreRecord`) —por medio del constructor `from_score` de `ScoreRecord`—. El resultado es una lista de registros de partitura que se pasa al constructor por defecto de `ScoreArchive`. Naturalmente este método es, como lo era el método `from_score` de `ScoreRecord` y por las mismas razones allí expuestas, un método de clase:

```

@classmethod
def from_scores(cls, scores: list[Score]) -> Self:
    """Construct the ScoreArchive from the given scores."""
    return cls([ScoreRecord.from_score(s) for s in scores])

```

La implementación del método `to_html` de `ScoreArchive` es prácticamente la misma que la del método del mismo nombre en la clase `ScoreRecord`:

```

def to_html(self) -> str:
    """Convert the ScoreArchive into an HTML element."""
    return Template(ARCHIVE_HTML_TEMPLATE).render(scorearchive=self)

```

Lo único que cambia es la plantilla HTML que ha de procesarse, la cual naturalmente tiene que contar ahora con un nuevo elemento —aquí usaremos el elemento `section` de HTML de clase `score-archive`— que agrupe un número

arbitrario de registros de partituras. Con ese fin se utiliza una expresión `for` —un bucle— que procesa cada uno de los ítems que forman parte de los datos del archivo:

```
ARCHIVE_HTML_TEMPLATE = """
<section class="score-archive">
{%
  for scorerecord in scorearchive%}
    <article class="score-record">
      <div class="score-link">
        <a download href="{{scorerecord.score.path}}"></a>
      </div>
      <div class="score-info">
        <p class="composer">{{scorerecord.composer}}</p>
        <p class="work">{{scorerecord.work}}</p>
        <p class="editor">{{scorerecord.editor}}</p>
      </div>
    </article>
{%
  endfor%}
</section>
"""
```

Ordenación de los registros del archivo: el método `sort`.

Antes de completar la tarea y con la mirada puesta en lo que el usuario acabará viendo en la página web resultante, interesa implementar un método más en la clase `ScoreArchive`, uno que permita ordenar los registros de partituras. Es evidente que una presentación sin orden ni concierto será por fuerza mucho menos navegable que una adecuadamente organizada.

La ordenación que el usuario probablemente espera encontrar o, cuando menos, una que agilizará la navegación es la que muestra los registros ordenados por nombre de compositor y título de obra. Así, por ejemplo, en lugar de la serie

- Johann Sebastian Bach, Sarabande BWV 995
- Anónimo, Greensleeves
- Francisco Tárrega, Adelita
- Johann Sebastian Bach, Preludio BWV 999
- Carl Philipp Emanuel Bach, Sonata

deberíamos preferir esta otra:

- Anónimo, Greensleeves
- Carl Philipp Emanuel Bach, Sonata
- Johann Sebastian Bach, Preludio BWV 999
- Johann Sebastian Bach, Sarabande BWV 995
- Francisco Tárrega, Adelita

Obsérvese, en particular, que se ordena alfabéticamente por el apellido del compositor primero, por su nombre de pila y finalmente por el título de la obra. Técnicamente se llaman *claves* a las propiedades que se utilizan para ordenar. Aquí, las claves son el apellido, el nombre y la obra, por ese orden.

Ahora bien, en los datos de nuestros registros el nombre del compositor es

el nombre completo. Debemos, por tanto, extraer las partes del nombre que usaremos como claves iniciales.

Las siguientes expresiones de Python, donde `full_name` es el nombre del compositor tal como consta en nuestros registros, sirven a ese propósito:

```
composer_last_name = full_name.split(" ")[-1]
composer_first_name = full_name.split(" ")[:-1]
```

La función de Python `sorted` ordena cualquier secuencia de acuerdo con una función que produce la clave del ordenamiento —o grupo de claves, cuando la ordenación es múltiple, como en nuestro problema—. Una función de este tipo sería la siguiente:

```
def composer_and_work(sr: ScoreRecord) -> tuple:
    composer_names = sr.composer.split(" ")
    composer_last_name = composer_names[-1]
    composer_first_name = composer_names[:-1]
    return (composer_last_name, composer_first_name, sr.work)
```

Esta función se pasa como clave de `sorted`, que devuelve a una lista ordenada de los registros con la que se reconstruye el `ScoreArchive`.

El método `sort` de `ScoreArchive` quedaría implementado, pues, como sigue:

```
def sort(self) -> Self:
    """Sort the ScoreRecord by composer and work."""
    def composer_and_work(sr: ScoreRecord) -> tuple:
        composer_names = sr.composer.split(" ")
        composer_last_name = composer_names[-1]
        composer_first_name = composer_names[:-1]
        return (composer_last_name, composer_first_name, sr.work)
    return ScoreArchive(sorted(self, key=composer_and_work))
```

Un caso de prueba, formulado sobre los ejemplos anteriores, es el siguiente:

```
@pytest.fixture
def score6():
    return Score(Path("Johann-Sebastian-Bach_Sarabande-BWV-995.pdf"))

@pytest.fixture
def score7():
    return Score(Path("Johann-Sebastian-Bach_Preludio-BWV-999.pdf"))

@pytest.fixture
def score8():
    return Score(Path("Carl-Philipp-Emanuel-Bach_Sonata.pdf"))

@pytest.fixture
def scorerecord6(score6):
    return ScoreRecord.from_score(score6)

@pytest.fixture
def scorerecord7(score7):
```

```

    return ScoreRecord.from_score(score7)

@pytest.fixture
def scorerecord8(score8):
    return ScoreRecord.from_score(score8)

@pytest.fixture
def scorearchive2(scorerecord3, scorerecord5, scorerecord6,
                  scorerecord7, scorerecord8):
    return ScoreArchive([scorerecord6, scorerecord5, scorerecord3,
                        scorerecord7, scorerecord8])

@pytest.fixture
def scorearchive3(scorerecord3, scorerecord5, scorerecord6,
                  scorerecord7, scorerecord8):
    return ScoreArchive([scorerecord5, scorerecord8, scorerecord7,
                        scorerecord6, scorerecord3])

def test_scorearchive_sort(scorearchive2, scorearchive3):
    assert scorearchive2.sort() == scorearchive3

```

Código final

Si recopilamos todas las piezas y las incluimos en nuestro fichero fuente `score.py` tendremos el código que resulta de esta tercera fase en el desarrollo del programa, y que puede verse en [03\(score.py\)](#):

Por su parte, el código ampliado de los tests se puede consultar en el fichero [03/test_score.py](#) dentro del directorio del proyecto.

Generación del fragmento web del fichero de partituras

Al final de la primera sección tuvimos la oportunidad de ver por vez primera el resultado del programa en el navegador. Es momento ahora de modificar el código creado entonces para que tenga en cuenta los resultados obtenidos en esta fase del desarrollo

El fichero de código fuente `partituras.py` generaba el fragmento HTML y lo guardaba en el fichero `partituras.html`, que contenía un boceto de la página web, el cual abríamos luego en el navegador. Entonces se trataba de producir el HTML a partir del fichero de una única partitura. En esta ocasión se trata de generarla a partir de un directorio de partituras. En concreto, el método `iterdir()` aplicado sobre un directorio sirve para procesar en bucle los ficheros contenidos en él. Por medio de ese método creamos una lista de objetos `Score`, a partir de la cual se construye el archivo de partituras, que se ordena y cuyo método `to_html` genera el nuevo fragmento HTML, que guardamos igualmente en el fichero `partituras.html`

```

import sys
from pathlib import Path

```

```

from score import Score, ScoreArchive

if __name__ == "__main__":
    scores = [Score(s) for s in Path(sys.argv[1]).iterdir()]
    scorearchive = ScoreArchive.from_scores(scores).sort()
    html_page = scorearchive.to_html()
    with open("partituras.html", "w") as f:
        f.write(html_page)

```

Presentación de un archivo de partituras

En la segunda fase del desarrollo del programa utilizamos el lenguaje CSS para especificar la apariencia de un registro de partitura (elemento de clase `score-record`). Es el turno ahora de hacer lo propio con el archivo de partituras (elemento de clase `score-archive`).

Este elemento será también una rejilla con un número de celdas que rellene el ancho de la pantalla sin desboardarlo y cuyo tamaño sea de 300px. Las celdas aparecen centradas en su eje horizontal y separadas por un canal (`gap`) de 1.5 emes de anchura:

```

.score-archive {
    display: grid;
    grid-template-columns: repeat(auto-fill, 300px);
    justify-content: center;
    grid-gap: 1.5rem;
}

```

Añadimos esta regla a las anteriores y las guardamos en el fichero `css/score.css`.

Tras ejecutar

```
python partituras.py ../scores
```

se genera el fichero `partituras.html`, al que aplicamos la hoja de estilo recién creada desde la pestaña `Style` de las herramientas para desarrolladores de Firefox. El `resultado` se muestra a continuación.

Fase 4. Cubiertas de las partituras

Generación de cubiertas

La forma actual de la página que se muestra en el final de la sección anterior cumple ciertamente todos nuestros objetivos, pero adolece de un problema de usabilidad —por utilizar el un tanto malsonante neologismo que suelen emplear los desarrolladores web y de aplicaciones—. En concreto, no es fácil para el usuario orientarse entre los distintos registros cuando todos comparten el mismo aspecto de una partitura en blanco. El ojo se marea ante lo casi idéntico. La solución habitual en estos casos es utilizar una imagen en miniatura de la cubierta o primera página del objeto que se registra, en nuestro caso una partitura.

Para dotar a nuestro programa de esta capacidad haremos uso de un módulo que es capaz de extraer como imágenes las páginas de un documento pdf, aquí el

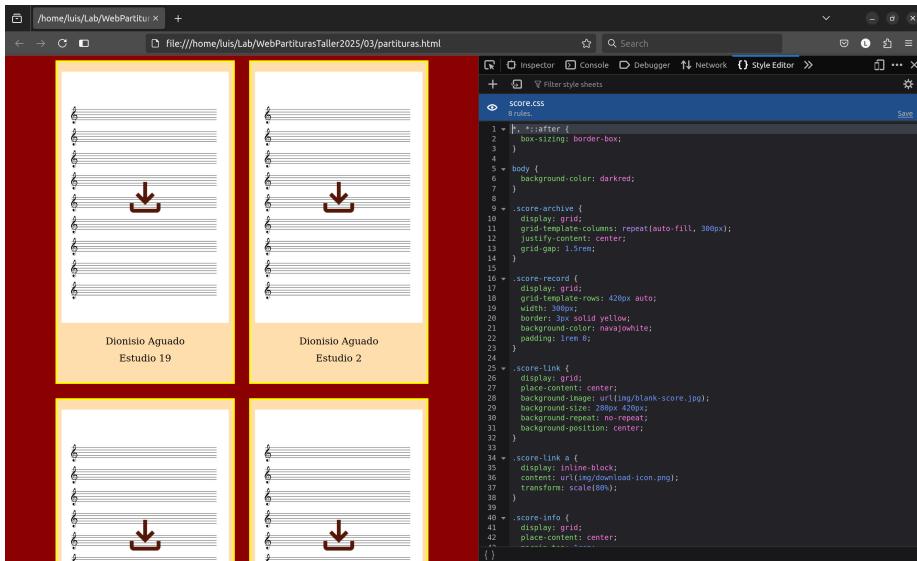


Figura 4: partituras - fase 3

módulo `pymupdf`. La cubierta (`cover`) de ese documento deberá aparecer como imagen en sustitución de la partitura en blanco en el diseño precedente.

La primera extensión necesaria de nuestro código consiste en diseñar una clase, que llamaremos `CoverGenerator` (generador de cubiertas), cuyos campos serán la ruta del fichero pdf y la ruta del fichero de imagen de la cubierta extraída —aquí, la imagen de la primera página del pdf—. Las instancias de esa clase, esto es, los objetos generadores de cubiertas poseen un método, que llamaremos `make_cover` encargado de producir y guardar la imagen de la cubierta en caso de que aún no haya sido generada —no interesa que cada vez que ejecutemos el programa la imagen tenga que regenerarse—.

El código de esta clase puede implementarse en pocas líneas:

```

from pymupdf import Document as PdfDocument

@dataclass
class CoverGenerator:
    pdf: Path
    cover: Path

    def make_cover(self) -> None:
        """Creates a cover from pdf if it does not exist and
        save it to cover."""
        if not self.cover.exists():
            print(f"Creating cover for {self.pdf} ...")
            cover_image = PdfDocument(self.pdf).get_page_pixmap(0)
            cover_image.save(self.cover)

```

Lo único reseñable de este código es que utiliza las herramientas suministradas

por los objetos `Document` de `pymupdf` —que aquí, por claridad, invocamos bajo el nombre `PdfDocument`—. En particular, el método `get_page_pixmap` produce una imagen de la página dada como argumento, aquí 0, la primera página del pdf —en programación el 0 suele ser el índice del primer elemento de una serie de objetos—. Esta imagen se guarda luego en el fichero pasado como primer argumento a `save`. El formato de la imagen se deduce de la extensión de ese fichero.

También es digno de mención el hecho de que el tipo de dato devuelto por el método `make_cover` sea `None`. Lo que este término indica es que tal método en realidad no produce ningún resultado en particular, aunque su ejecución implique que se cree una imagen y se guarde en el sistema de ficheros. Métodos con esa firma en su tipo de datos de salida, que se desmarcan y diferencian de las funciones puras, aparecen justo allí donde lo que importa es el efecto colateral (*side effect*) que desencadenan y no su posible resultado, típicamente en operaciones de entrada y salida (*IO operations*).

Acabamos de diseñar una herramienta para la generación de cubiertas. ¿Cómo y dónde usarla? Puesto que la imagen lo es de la partitura, tiene sentido que sean los propios objetos `Score` —que, como se recordará, son los modelos de las partituras— los que deleguen en el generador de cubiertas el proceso de creación de dicha imagen. Un modo de implementar esta idea es hacer que las instancias de la clase `Score` sean las que entre sus campos incluyan la ruta del fichero de la imagen de cubierta y que en el proceso de construcción de un objeto `Score` se genere y guarde dicha imagen. Podemos extender el método `__post_init__` de `Score` para lograr ambas cosas.

```
def __post_init__(self):
    self.name = self.path.stem
    self.cover = Path(COVERS_DIR, self.name).with_suffix(COVER_FORMAT)
    CoverGenerator(self.path, self.cover).make_cover()
```

La ruta del fichero de la cubierta se determina a partir del propio nombre de la partitura al que se prefija la ruta del directorio de todas las cubiertas y al que se añade la extensión del formato de la imagen, aquí `.png`. Esa ruta es la que se pasa al constructor del generador de partituras como segundo argumento; mientras que su primer argumento es la ruta de la propia partitura.

Ampliación del estilo de presentación de un registro de partitura

Hasta aquí hemos conseguido que, cuando se cree un objeto `Score`, se genere, si aún no existe, una imagen de su cubierta en el directorio de las cubiertas, el cual no es otro que el de las imágenes que usa nuestra hoja de estilo. El siguiente paso lógico sería modificar el estilo para que, en lugar de incluir una imagen de una partitura en blanco, contenga, para cada una de los registros de partituras, una imagen específica de su cubierta. Ahora bien, a poco que meditemos sobre el asunto, nos percataremos de que la deseada modificación no parece sin más posible. En efecto, las reglas definidas en `score.css` se aplican a *todos* los elementos del documento HTML y lo que ahora estamos intentando es que *cada uno* de aquellos que representan los registros de partituras tenga una imagen de fondo diferente.

Una solución posible a este problema aparentemente insoluble sería cambiar el diseño del HTML; en concreto, hacer que el contenido del enlace del elemento de clase `score-link` fuese una imagen. Por ejemplo, para la partitura de *Greensleeves*:

```
<article class="score-record">
  <div class="score-link">
    <a download href="scores/Anónimo_Greensleeves.pdf">
      
    </a>
  </div>
</article>
```

Una solución alternativa, una en la que mantenemos intacta la representación HTML del registro de la partitura, consiste en establecer la imagen de fondo del elemento `score-link` dentro de él mismo a través del atributo `style`. Mediante esa estrategia el ejemplo tendría el siguiente aspecto:

```
<article class="score-record">
  <div class="score-link"
    style="background-image: url(img/Anónimo_Greensleeves.png)">
    <a download href="scores/Anónimo_Greensleeves.pdf"></a>
  </div>
</article>
```

En efecto, el atributo `style` permite definir en el mismo lenguaje CSS las propiedades estilísticas del elemento al que se añade.

Es más, en caso de que se produzca un conflicto entre diferentes definiciones de la misma propiedad, como sucede si mantenemos la definición de la propiedad `background-image` en la regla correspondiente a los elementos de clase `score-link` dentro de la hoja de estilo, la definición interna a cada elemento `score-link` de esa misma propiedad tendría una *precedencia* mayor, esto es, el conflicto se resolvería a favor de ella. Así pues, podemos sin miedo dejar intacta la hoja de estilo y modificar únicamente la plantilla HTML de la siguiente manera:

```
ARCHIVE_HTML_TEMPLATE = """
  <section class="score-archive">
    {%
      for scorerecord in scorearchive%
    %}
      <article class="score-record">
        <div class="score-link"
          style="background-image: url('{{scorerecord.score.cover}})">
          <a download href="{{scorerecord.score.path}}"></a>
        </div>
        <div class="score-info">
          <p class="composer">{{scorerecord.composer}}</p>
          <p class="work">{{scorerecord.work}}</p>
          <p class="editor">{{scorerecord.editor}}</p>
        </div>
      </article>
    {%
      endfor%
    %}
  </section>
"""
```

Casos de prueba

¿Qué hay de los casos de prueba? ¿Rebajamos ahora su importancia? En absoluto. El método renovado `make_cover` de `CoverGenerator` debería ser también verificado. Desgraciadamente la creación de casos de prueba para procedimientos que implican operaciones de *input-output* como la lectura o escritura de ficheros son más difíciles de construir y no hay tiempo, dada la dimensión de este trabajo, de exponerlos aquí.

Podemos, sin embargo, aprovechar la ocasión para verificar parte del trabajo del método `__post_init__` de la clase `Score`, a saber, el que la ruta de la imagen de la cubierta computada por el método sea la que esperamos que sea o que el nombre de la partitura sea el que debe ser con independencia del directorio en que se encuentre, algo, por cierto, esto último, que no deberíamos haber dado por sentado en las secciones anteriores.

Pero antes de codificar estos tests debemos modificar ligeramente la definición de la clase `Score` y de su método `__post_init__`. para que, bajo demanda, pueda desactivarse la generación de las imágenes de las cubiertas. De hecho, si ejecutamos ahora `pytest` con los ejemplos de los anteriores casos de prueba, se producirán errores debidos a que esos ejemplos no lo son de partituras reales y `make_cover` no encontrará los ficheros a los que fingidamente se refieren. Este paso es, pues, necesario, si queremos crear tests que no impliquen la creación o existencia de ficheros concretos.

La modificación consiste en crear un campo con clave en la clase `Score`, que llamaremos `with_cover`. Cuando el valor de `with_cover` es `True` —el valor por defecto—, la imagen de la cubierta se generará en el momento de la creación del objeto partitura; en caso contrario, ese paso no se realiza. El método `__post_init__` se actualiza para que incluya esa condición, como se muestra en el siguiente fragmento de código

```
from dataclasses import KW_ONLY

@dataclass
class Score:
    path: Path
    name: str = field(init=False)
    cover: Path = field(init=False)
    _: KW_ONLY
    with_cover: bool = True

    def __post_init__(self, gen_cover=True):
        self.name = self.path.stem
        self.cover = Path(COVERS_DIR, self.name).with_suffix(COVER_FORMAT)
        if with_cover:
            CoverGenerator(self.path, self.cover).make_cover()
```

Tras el cambio, hay que editar todos los ejemplos existentes de partituras en `test_score.py` para que tengan en cuenta esta funcionalidad. El ejemplo siguiente efectúa un cambio de esa índole para crear un caso de prueba de el método `__post_init__`:

```

@pytest.fixture
def score9():
    return Score(Path("scores/Tárrega_Gran-Vals.pdf"),
                 with_cover=False)

def test_score_postinit(score9):
    # Assume: covers directory is 'img' and cover extension is '.png'
    assert score9.name == "Tárrega_Gran-Vals"
    assert score9.cover == Path("img/Tárrega_Gran-Vals.png")

```

También hay que cambiar el ejemplo `scorearchivehtml1` en `test_score.py` para que tenga en cuenta la modificación de la plantilla HTML comentada en el apartado inmediatamente anterior.

Código final

Reuniendo de nuevo todas las piezas obtenemos la siguiente versión ampliada de `score.py` se puede consultar en [04/score.py](#)

Tras ejecutar `partituras.py` como en anteriores ocasiones, la página `partituras.html` queda como muestra la [imagen](#) siguiente:

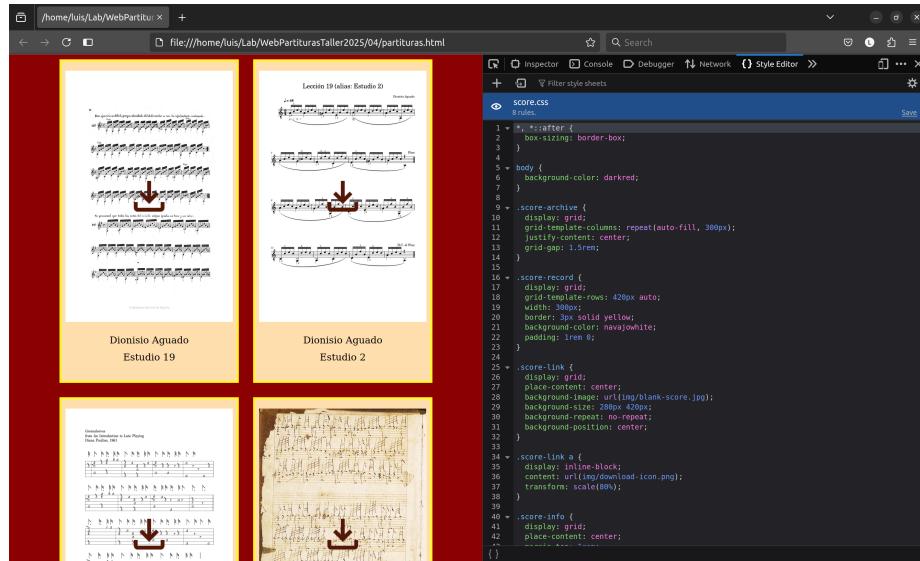


Figura 5: partituras - fase 4

Fase 5. Información extra sobre el compositor ⁴

Fase 6. Persistencia y bases de datos ⁵

Fase 7. La página web ⁶

Documentos HTML

Hasta aquí hemos estado utilizando los términos página web, texto HTML y fragmento HTML sin mayor precisión ni explicación.

Digámoslo clara y abiertamente de una vez. Hasta ahora, pese a las apariencias, no hemos producido ninguna página web, término coloquial para referirse a lo que técnicamente se denomina *documento HTML*. Y sin embargo —se dirá—, hemos abierto una página web en el navegador. ¿No es eso cierto? Ciento es, y la explicación de esta contradicción entre la premisa mayor y la menor es que lo que, en realidad, sucedía —como hemos sugerido de paso en otro momento— es el que navegador estaba completando la página web con los fragmentos HTML con que lo alimentábamos.

Estructura del documento

Por tanto, el mejor modo de empezar a *sentir* lo que es una página web de verdad es pedirle a ese solícito navegador que nos muestre el trabajo que está haciendo en la trastienda mientras nos dibuja la página a partir de nuestros inopinados fragmentos. Para ello, abrimos las herramientas de desarrollo —**Ctrl+May+I**— y seleccionamos la pestaña *Inspector*. Veremos efectivamente un código HTML, pero con nuevos elementos.

```
<html>
  <head></head>
  <body>
    <section class="score-archive">...</section>
  </body>
</html>
```

Observamos que nuestro fragmento HTML del archivo de partituras —el elemento `section` de clase `score-archive`— aparece dentro del elemento `body`, el cual, junto con el elemento precedente `head`, está envuelto en el elemento raíz —del que todos los demás son hijos—: el elemento `html`.

Esta es la estructura básica de todo documento HTML. Ahora bien, si abrimos la pestaña *Console* nos mostrará la advertencia de que la página está en *quirks mode* —lea el lector qué significa esto siguiendo el enlace de información extra asociado a la advertencia—.

⁴Este capítulo se omite por falta de tiempo para la exposición.

⁵Este capítulo se omite por falta de tiempo para la exposición.

⁶Este capítulo se adapta dando por supuesto que la Fase 5 y la Fase 6 no se han realizado.

El tipo de documento

Para que la página sea acorde con la especificación moderna de HTML hemos de incluir al principio de ella la declaración del tipo de documento:

```
<!DOCTYPE html>
```

Es evidente que todos estos añadidos hay que dejarlos reflejados en nuestras plantilla HTML. Pero antes de hacerlo, incluiremos alguna línea más, que comentamos paso a paso.

Información acerca del propio documento e idioma.

Dentro del elemento `head` del documento HTML se suele introducir información acerca del propio documento: meta-information.

Los elementos mínimos de esa información son el título del documento, la codificación de caracteres en que el documento está escrito —hoy en día `utf-8`— y su relación con otros recursos externos—, como la hoja de estilo que ha de aplicarse, cuya ubicación se establece aquí.

```
<head>
  <title>WebPartituras</title>
  <meta charset="utf-8">
  <link href="css/score.css" rel="stylesheet">
</head>
```

Por su parte el idioma del documento se declara como atributo del elemento `html`:

```
<html lang="es">
```

Elemento `h1`

Dado que el elemento `title` de `head` no se muestra dentro de la página —lo consumen sólo los robots, si bien un rastro de él aparezca normalmente como título de la pestaña del navegador—, aprovechamos la ocasión para introducir un nuevo elemento, `h1`, que se utiliza para hacer visible en el interior de la página su propio título.

```
<body>
  <h1>Web Partituras</h1>
  <section class="score-archive">...</section>
</body>
```

Como cualquier elemento dentro de `body` podemos especificar para `h1` una regla de estilo. Por ejemplo:

```
h1 {
  font-size: 300%;
  color: navajowhite;
  text-align: center;
  padding: 1.4rem 0;
  margin: 3rem;
```

```
    border: 4px solid navajowhite;
}
```

Plantillas remozadas

Lo que resta es añadir los cambios a nuestras plantillas HTML, por ejemplo:

```
ARCHIVE_HTML_TEMPLATE = """
<!DOCTYPE html>
<html lang="es">
<head>
    <title>Web Partituras</title>
    <meta charset="utf-8">
    <link href="css/score.css" rel="stylesheet">
</head>
<body>
    <h1>Web Partituras</h1>
    <section class="score-archive">
        {%
            for scorerecord in scorearchive %}
            <article class="score-record">
                <div class="score-link"
                    style="background-image: url('{{scorerecord.score.cover}})">
                    <a download href="{{scorerecord.score.path}}></a>
                </div>
                <div class="score-info">
                    <p class="composer">{{scorerecord.composer}}</p>
                    <p class="work">{{scorerecord.work}}</p>
                    <p class="editor">{{scorerecord.editor}}</p>
                </div>
            </article>
        {%
            endfor %}
    </section>
</body>
</html>
"""
```

Retoque del estilo

Si reconstruimos de nuevo las páginas web con las novedades comentadas, notaremos algo perturbador: los iconos de descarga de la partitura no aparecen y la partitura no puede descargarse.

La razón es la siguiente: el navegador busca las imágenes tomando como raíz el directorio en el que reside el fichero que apunta a ellas. Así, por ejemplo, si nuestro fichero `css/score.css` apunta a la imagen en `img/download-icon.png` el navegador buscará la imagen en `css/img/downlaod-icon.png`, un ruta que no existe. Por el contrario, si es desde el propio fichero `partituras.html` desde donde apuntamos hacia la imagen `img/Anonimo_Greensleeves.png`, la imagen se buscará en el lugar correcto, uno que sí existe —y es por ello por lo que las imágenes de las partituras no han desaparecido—.

Para resolver el problema, se añade a la ruta de la imagen un símbolo `../`, que indica que hay que mirar en el directorio padre del directorio donde está la hoja de estilo:

Así, en lugar de

```
background-image: url(img/download-icon.png)
```

debe aparecer

```
background-image: url(..img/download-icon.png)
```

Un efecto positivo de ver cómo, por arte de magia negra, el ícono de descarga desaparecía de la página, pudo ser cierto alivio visual. El ícono tenía algún sentido cuando el número de registros era insignificante, con más registros puede resultar cargante. Eliminémoslo, pues, y modifiquemos el estilo para que, aun sin él, pueda seguir accediéndose a la descarga de la partitura:

```
.score-link a {  
    width: 280px;  
    height: 420px;  
}
```

El conjunto de estos cambios en el estilo se encuentra en [07/css/score.css](#).

Validación de la página web

Es aconsejable y signo de buena práctica entre los desarrolladores web someter nuestra página a la aprobación del W3C, que es la organización encargada del desarrollo de muchos estándares de la *World Wide Web*, entre ellos el del lenguaje HTML.

W3C proporciona un servicio de validación del HTML de nuestras páginas en <https://validator.w3.org/>. Si cargamos nuestra página en el validador sabremos si contiene errores y si es conforme con el estándar actual de HTML.

Cargada nuestra página, el validador nos indica que no contiene errores, aunque sugiere la posibilidad de incluir títulos en cada registro de partitura y uno para el archivo en general. Esta opción puede ser interesante si pensamos en navegadores no visuales. Dejamos al lector que investigue cómo modificar la plantilla para que introduzca un título del archivo y sendos títulos por registro, pero de tal manera que estos títulos no se muestren en la página dibujada en nuestros navegadores visuales.

Fase 8. Script de ejecución

Construido el generador de la página web y la propia página web —así como la base de datos— resulta inconveniente tener que activar el entorno virtual de desarrollo de Python para ejecutar `partituras.py` cada que vez que tenemos que actualizar el archivo de partituras o cuando queremos crear uno nuevo. Incluso puede resultar inconveniente tener que abrir el navegador con la página web creada.

Un guión del shell es un mecanismo sencillo para automatizar todos estos pasos. Seguiremos todavía en el terminal, pero desde allí bastará ejecutar `./crear_webpartituras <directorio-de-partituras>` para producir la página a partir del directorio de partituras `<directorio-de-partituras>` y que ésta se muestre en el navegador.

Los guiones del shell pueden requerir personalización dependiendo del sistema operativo desde el que se lanzan. El script siguiente está escrito para ser ejecutado desde un ordenador Linux y probablemente pueda ejecutarse tal cual o con mínimos cambios desde MacOSX.

```
!/usr/bin/env bash

SCORE_DIR="$1"

# activate Python virtual environment in parent directory
source ../.venv/bin/activate

# execute partituras.py
python partituras.py "$SCORE_DIR"

# deactivate environment
deactivate

# open webpage in browser
open partituras.html
```

Una vez guardado el script bajo el nombre `crear_partituras`, hay que asignar, además, permiso de ejecución a nivel de usuario al fichero que lo contiene.

```
chmod u+x crear_partituras

Ejecutemos el nuevo script y disfrutemos del resultado.

./crear_partituras ../scores
```

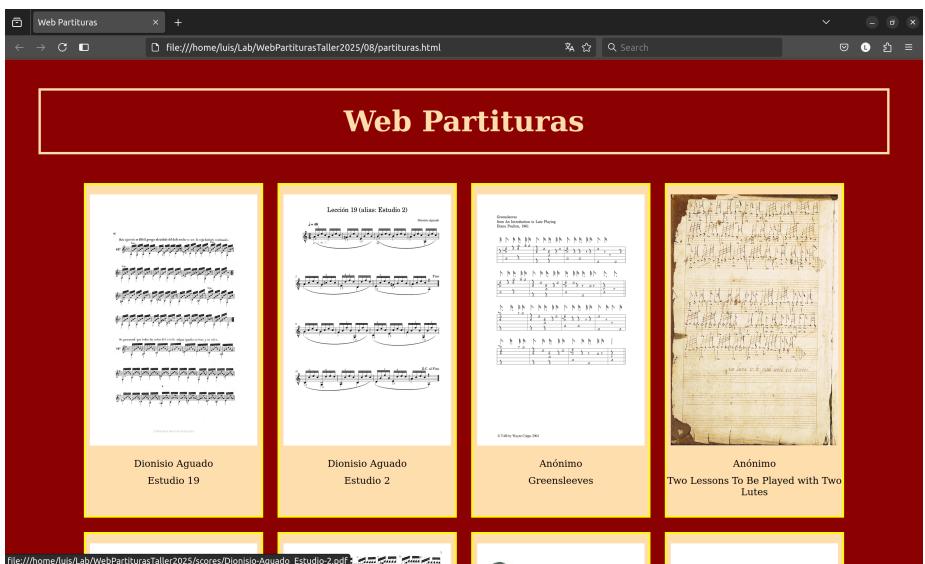


Figura 6: partituras - fase 8