Clases y objetos

March 17, 2023

Authors: Simon Pickin, Rubén Rafael Rubio Cuéllar, Jorge Carmona Ruber

Imágenes y paletas con clases

Introducción

Se dice que un lenguaje de programación que ofrece la posibilidad de crear clases y objetos implementa el *paradigma orientado a objetos*. Se suele reservar el término *lenguaje orientado a objetos* para un lenguaje como Java, en el que cualquier instrucción tiene obligatoriamente que estar dentro de una clase. Con esta terminología, Python implementa el paradigma orientado a objetos pero no es un lenguaje orientado a objetos, ya que permite definir funciones fuera de cualquier clase y deja al programador decidir la extensión del uso que quiere hacer de las clases y objetos. El objetivo general de esta práctica es enseñaros a emplear bien las clases y objetos en Python, lo que implica entender los conceptos básicos de la programación orientada a objetos (POO). Con este fin, en el programa que vamos a diseñar e implementar, vamos a hacer un uso bastante amplio de clases y objetos. El objetivo específico de está práctica es *refactorizar*[1] el programa de la práctica 3 para convertirlo en un programa que utiliza clases y objetos.

La primera tarea es pensar en las clases que vamos a necesitar. Las más evidentes son una clase `Colour` que representa colores RGB, una clase `Palette` que representa paletas, es decir, conjuntos de colores RGB, y una clase `Image2D` que representa una imagen digital cuyos píxels contienen colores RGB (podría también representar de manera ineficiente una imagen en escala de grises ya que si los tres componentes de un color tienen el mismo valor, se trata de un tono de gris). Sin embargo, veremos que será útil también definir una clase para imágenes unidimensionales de tal manera que una paleta sea un tipo específico de imagen unidimensional. La segunda tarea es decidir dónde colocar el código proporcionado en el enunciado de la práctica 3. Puesto que vamos a hacer un uso amplio de las clases, vamos a colocar cada función de este código en una de las clases mencionadas.

La clase `Colour`

En cuanto a las tres funciones de distancia, puesto que tratan relaciones entre colores, su lugar lógico sería como métodos de la clase `Colour`. Pero ¿como *métodos de instancia* o *métodos estáticos*? A primera vista parece razonable invocarlos siempre sobre un objeto concreto: se entendería la invocación de un tal método como el envío al objeto de la instrucción "dáme la distancia entre tú y este otro color". Sin embargo, definirlos como métodos de instancia complicaría pasarlos como parámetros a los métodos que derivaremos de las funciones `img_subst`, `new_list` y `colores_mas_representativos` tal como hicimos en la práctica anterior. Ahora bien, antes de convertir estas tres funciones en métodos estáticos de la clase `Colour`, tenemos que decidir cúal va a ser la representación interna de esta clase, es decir, ¿qué tipo(s) de atributo(s) usaremos para almacenar los datos del color? Decidimos almacenar los colores internamente en un solo atributo, con nombre `col_data`, como una tupla de tres enteros (con valores en el intervalo `[0, 255]`).

Para completar la clase `Colour`, vamos a añadir los métodos `__eq__`, `__str__`, `__hash__`, `externalize`, `show` y `getCmpnt`. Por tanto, con anotaciones de tipo, tenemos:

[1] Refactorizar un código significa modificarlo sin cambiar su funcionalidad, p.ej. para mejorar su estructura y/o su eficiencia.

[1]:
```python
from typing import Any

class Colour:
    """
    Objects of this class represent colours. The internal representation
    (i.e. type of the attribute col_data) is an RGB 3-tuple, i.e. a 3-tuple
    of integers in the interval [0, 255].
    """

    # Falta el código para validar los datos de entrada
    def __init__(self, R:int, G:int, B:int):
        self.col_data = (R, G, B)

    def __eq__(self, other:Any) -> bool:
        pass # Tu código va aquí

    def __hash__(self) -> int:
        pass # Tu código va aquí

    def __str__(self) -> str:
        pass # tu código va aquí

    def externalize(self) -> IntColorData:
        return self.col_data

    def show(self):
        pass # Tu código va aquí (llama a 'show' de 'Image1D')

    def getCmpnt(self, index:int) -> int:
        pass # Tu código va aquí

    @staticmethod
    def dist1(col1:'Colour', col2:'Colour') -> int:
        result = 0
        for i in range(3):
            result += (col1.col_data[i]-col2.col_data[i])**2
        return result

    @staticmethod
    def dist2(col1:'Colour', col2:'Colour') -> int:
        result = 0
        for i in range(3):
            result = max(result, abs(col1.col_data[i]-col2.col_data[i]))
        return result

    @staticmethod
    def dist3(col1:'Colour', col2:'Colour') -> int:
        result = 0
        coeffs = [2, 4, 3]
        for i in range(3):
            result += coeffs[i]*(col1.col_data[i]-col2.col_data[i])**2
        return result
```

La clase `Image2D`

En cuanto a las funciones del fichero `imagen.py`, puestos a elegir entre las clases presentadas anteriormente, claramente deberían colocarse como métodos de la clase `Image2D`. Pero ¿como *métodos de instancia* o *métodos estáticos*? Observa que tendría sentido invocar el método `show`, resp. el método `save`, de un objeto concreto: se entendería la invocación de un tal método como el envío al objeto de la instrucción "muéstrate en pantalla", resp. "guárdate en un fichero con este nombre". Sin embargo, no tendría mucho sentido invocar el método `read` (para leer una imagen de fichero) de un objeto concreto. Para seguir tratando las implementaciones como una caja negra, los tres métodos de la clase `Image2D` simplemente llamarán a las funciones correspondientes del módulo `imagen` (el fichero `imagen.py` ha sido ligeramente modificado respecto a la práctica 3).

Pero antes de convertir estas tres funciones en métodos de la clase `Image2D`, tenemos que decidir cuál va a ser la representación interna de esta clase, es decir, ¿qué tipo(s) de atributo(s) usaremos para almacenar los datos de la imagen? Puesto que queremos practicar al máximo el uso de clases y objetos, vamos a almacenar las imágenes internamente en un solo atributo con nombre `img2D_data` como una lista de objetos de la clase `Image1D`. Esta última clase, que definiremos a continuación, representa una imagen unidimensional. Esta decisión nos obliga escribir un método de conversión desde el formato interno hasta el formato externo para los métodos `show` y `save`, denotémoslo `externalize`, y un método de conversión en el sentido contrario para el método `read`, denotémoslo `internalize`. El último método se llamará desde el método `__init__` que se llamará, a su vez, desde el método `read` al crear un nuevo objeto de la clase `Image2D` a partir de los datos leídos del fichero. Lo lógico sería que el método `externalize` de la clase `Image2D` devuelva una lista del tipo que devuelve el método `externalize` de la clase `Image1D`. En cuanto al método `internalize`, para dar más flexibilidad al método `__init__`, decidimos que puede aceptar cualquiera de los siguientes tipos de datos:

- listas de objetos de la clase `Image1D` (anotación de tipo: `list[Image1D]`),
- listas de listas de objetos de la clase `Colour` (anotación de tipo: `list[list[Colour]]`),
- listas de listas de listas de 3 enteros (anotación de tipo: `list[list[list[int]]]`),
- listas de listas de tuplas de tres enteros (anotación de tipo: `list[list[tuple[int,int,int]]]`.

Para realizar su función, ambos métodos deberían usar los métodos del mismo nombre de la clase `Image1D`.

Respecto a la función `muestra`, puesto que sus datos de entrada constan de una imagen y una función de distancia, sería lógico colocarla también como método de la clase `Image2D` (aquí cambiando su nombre a `sample_palette`). Observa que el método derivado de esta función supone la existencia de un método estático de la clase `Palette`, que llamaremos `empty_palette`, que devuelve una nueva paleta vacía; y un método de instancia de la clase `Palette`, que llamaremos `add_colour`, que añade a la paleta el objeto de la clase `Colour` proporcionado como argumento, si no está presente ya. Observa también que el método devuelve un objeto de la clase `Palette`.

Finalmente, para terminar la clase `Image2D` añadimos un método derivado de la función `img_subst` y el método `__getitem__` llamado implícitamente en la penultima línea del método `sample_palette`. Por tanto, con anotaciones de tipo y simplificando las tres funciones ya que solo vamos a tratar imágenes en color, tenemos:

[4]:
```
from imagen import read, show, save
from typing import Callable, Any, Union

IntColourData = tuple[int, int, int]
ExtColourData = Union[IntColourData, list[int], 'Colour']
ExtDataImage1D = list[ExtColourData]
ExtDataImage2D = Union[list[ExtDataImage1D], list['Image1D']]
DistFunction = Callable[['Colour','Colour'],int]


class Image2D:
    """
    Objects of this class represent colour images. The internal representation
```

```
    (i.e. type of the attribute img2D_data) is a list of 1-D images.
    """

    # Tomaremos las anotaciones de tipo como precondición; no comprobaremos tipos
    def __init__(self, colours:ExtDataImage2D):
        # https://docs.python.org/3/library/stdtypes.html#truth-value-testing
        if colours:
            # excepción si alguno de los elementos tiene longitud distinta
            pass # Tu código va aquí
            self.img2D_data = self.internalize(colours)
        else:
            self.img2D_data = []

    @staticmethod
    def internalize(colours:ExtDataImage2D) -> list['Image1D']:
        """
        Converts a matrix of RGB-data or Colour objects or a list of 'Image1D'
        objects to a list of 'Image1D' objects (RGB-data: a 3-tuple of ints or
        a list of 3 ints).
        """
        pass # Tu código va aquí

    def externalize(self) -> list[list[IntColourData]]:
        """
        Externalizes the internal data as a matrix of RGB 3-tuples.
        """
        pass # Tu código va aquí

    # nuevo algoritmo
    def sample_palette(self, k:int, l:int) -> 'Palette':
        height, width = len(self.img2D_data), len(self.img2D_data[0])
        xstep, ystep = width // (l + 1), height // (k + 1)
        palette = Palette.empty_palette()
        for y in range(ystep, height - ystep + 1, ystep):
            for x in range(xstep, width - xstep + 1, xstep):
                palette.add_colour(self[y][x])
        return palette

    def __getitem__(self, i):
        pass # tu código va aquí

    def img_subst(self, palette:'Palette', dist_fun:DistFunction) -> 'Image2D':
        pass # tu código va aquí

    @staticmethod
    def read(path:str) -> 'Image2D':
        return Image2D(read(path))

    def show(self):
        show(self.externalize())

    def save(self, path:str):
        save(path, self.externalize())
```

La clase `Image1D`

La clase `Image1D` representa una imagen con una sola fila para la cual elegimos como representación interna una lista de objetos de la clase `Colour`. Esta clase implementará el método estático `internalize` y el método de instancia `externalize` para convertir entre las representaciones internas y externas. El método `externalize` devolverá una lista del tipo que devuelve el método externalize de la clase `Colour`. El método `internalize` debería aceptar cualquiera de los siguientes tipos:

- listas de objetos de la clase `Colour`(anotación de tipo: `list[Colour]`),
- listas de listas de 3 enteros (anotación de tipo: `list[list[int]]`),
- listas de tuplas de tres enteros (anotación de tipo: `list[tuple[int,int,int]]`).

Para completar esta clase, añadimos los métodos `show`, `__str__`, `__getitem__` y `__len__`. Observa que el método `__str__` de `Image1D` llamará al método `__str__` de `Colour` (un fenómeno típico). Por tanto, con anotaciones de tipo, tenemos:

[ ]:
```python
class Image1D:
    """
    Objects of this class represent 1D-images. The internal representation
    (i.e. type of the attribute img1D_data) is a list of objects of class Colour
    """

    # Tomaremos las anotaciones de tipo como precondición; no comprobaremos tipos
    def __init__(self, colours:ExtDataImage1D):
        # https://docs.python.org/3/library/stdtypes.html#truth-value-testing
        if colours:
            self.img1D_data = Image1D.internalize(colours)
        else:
            self.img1D_data = []

    @staticmethod
    def internalize(colours:ExtDataImage1D) -> list[Colour]:
        """
        Converts a list of RGB-data or 'Colour' objects into a list of 'Colour'
        objects (RGB-data: a 3-tuple of ints or a list of 3 ints).
        """
        pass # tu código va aquí

    def externalize(self) -> list[IntColourData]:
        """
        Externalises the internal data as a list of RGB 3-tuples
        """
        pass # tu código va aquí

    def show(self):
        pass # tu código va aquí (llama a 'show' de 'Image2D')

    def __str__(self) -> str:
        pass # tu código va aquí

    def __getitem__(self, i):
        pass # tu código va aquí

    def __len__(self):
        pass # tu código va aquí
```

La clase `Palette`

Finalmente, toca la clase `Palette`. Tal como hemos explicado anteriormente, queremos que una paleta sea un tipo específico de imagen unidimensional. A nivel de código, esta idea se traduce al requisito: la clase `Palette` es una subclase de la clase `Image1D` (del mismo modo que `ZeroDivisionError` es una subclase de `ArithmeticError` que es una subclase de `Exception`, ver la jerarquía de clases de excepciones de Python), lo que significa que la clase `Palette` tiene todos los atributos y métodos de la clase `Image1D` sin tener que definirlos. Se dice que la clase `Palette` *hereda* los atributos y métodos de la clase `Image1D`. La sintaxis Python para especificar esta relación de subclase es `class Palette(Image1D)`. ¿Cuáles son las propiedades esenciales que determinan cuándo una imagen unidimensional es una paleta? Pues en una paleta no hay repetición de colores y el orden de colores no importa (no corresponden a píxeles).

Ya hemos colocado como métodos todas las funciones que venían en el enunciado de la práctica anterior y también hemos colocado como método la función desarrollada en el ejercicio 1 de esta práctica. Las funciones de los dos últimos ejercicios tienen más que ver con paletas que con imágenes (aunque se podría discutir este punto) por lo que decidimos colocarlas como métodos de la clase `Palette`, cambiando los nombres a `new_palette` y `representative_palette` respectivamente.

En la última práctica, para no repetir código entre los ejercicios 1 y 2, se aconsejó definir una función que, dado un color, una paleta y una función de distancia, calcula cuál es el elemento de la paleta que está más cerca del color. Por tanto, también vamos a añadir a la clase `Palette` un método que corresponde a esta función, que llamaremos `nearest` y, asimismo, vamos a solucionar de una manera coherente con la POO el problema de que la función del ej. 1 necesita que la función `nearest` devuelva el elemento de la paleta, mientras que la función del ej. 2 necesita que la función `nearest` devuelva el índice de este elemento de la paleta.

Recuerda que al colocar el método `sample_palette` en la clase `Image2D`, hemos supuesto que en la clase `Palette` existen métodos `empty_palette` y `add_colour`. Para completar la clase `Palette` vamos a añadir los métodos `__contains__` y `__eq__`. Por tanto, con anotaciones de tipo, tenemos:

[ ]:
```
class Palette(Image1D):
    """
    Objects of this class represent palettes. A palette is a 1D-image in which
    there are no duplicate colours and in which the order is arbitrary.
    """

    # Falta el código para quitar duplicados
    def __init__(self, colours:ExtDataImage1D):
        super().__init__(colours)

    def __contains__(self, elem:Any) -> bool:
        pass # tu código va aquí

    # usado por el método 'representative_palette' de esta clase
    def __eq__(self, other:Any) -> bool:
        pass # tu código va aquí

    # añadir un color a la paleta solo si no está presente ya
    def add_colour(self, col:Colour):
        pass # tu código va aquí

    @staticmethod
    def empty_palette() -> 'Palette':
        pass # tu código va aquí

    def nearest(self, colour:Colour, dist_fun:DistFunction) -> tuple[int, Colour]:
```

```
        pass # tu código va aquí

    # versión de 'nearest' para llamarse desde un objeto de la misma clase
    def nearest_index(self, colour:Colour, dist_fun:DistFunction) -> int:
        return self.nearest(colour, dist_fun)[0]

    # versión de 'nearest' para llamarse desde un objeto de otra clase
    def nearest_colour(self, colour:Colour, dist_fun:DistFunction) -> Colour:
        return self.nearest(colour, dist_fun)[1]

    def new_palette(self, img2D:'Image2D', dist_fun:DistFunction) -> 'Palette':
        pass # tu código va aquí

    def representative_palette(self, img2D:'Image2D', dist_fun:DistFunction)\
                        -> 'Palette':
        pass # tu código va aquí
```

Aumentar la eficiencia

Ahora vamos a refactorizar el código de nuestro programa con el fin de aumentar su eficiencia. Observa que, dado un color que aparece en una imagen pasada como argumento al método `new_palette`, cada vez que el algoritmo naif encuentra un píxel de este color, repite el cálculo de la distancia entre el color y cada color de la paleta. Si la imagen contiene colores que se repiten en muchos píxeles, esta repetición de cálculo tendrá un efecto significativo sobre el tiempo de ejecución del algoritmo. Para evitarlo, bastaría con empezar por calcular, para cada color de la imagen, el número de píxeles en que aparece. Podremos almacenar esta información en un diccionario cuyas claves son colores (objetos de la clase `Colour`) y cuyos valores son frecuencias (enteros). Además, calcular la nueva paleta a partir de este diccionario en vez de a partir de la imagen simplificará el código del método `new_palette`.

Por tanto, primero definimos un método llamado `calculate_histogram` que, a partir de la imagen, genera un tal diccionario de frecuencia de colores. Para nuestra conveniencia, también definimos un nuevo alias de tipos:

```
Histogram = dict[Colour, int]
...
class Image2D
    ...
    def calculate_histogram(self) -> Histogram:
        # tu código va aquí
```

Luego modificamos el método `new_palette` para que acepte un histograma en vez de una imagen:

```
class Palette
    ...
    def new_palette(self, hist:Histogram, dist_fun:DistFunction) -> 'Palette':
        # tu código va aquí
```

Finalmente, tenemos que modificar al método `representative_palette` para que, antes de empezar a invocar a la nueva versión de `new_palette`, invoque a `calculate_histogram`.

Ejecuta la nueva versión de `representative_palette` y observa la ganancia en tiempo de ejecución.

Ayudas / avisos

Pruebas

Recuerda que es esencial escribir también código para probar el código que constituye tu solución a esta práctica, aunque no debes incluirlo en la entrega.

Modificaciones que parecen no tener ningún efecto

Supongamos que ejecutamos un código (con la flecha verde de Spyder) que contiene la siguiente definición de clase:

```
class Point
    ...
    def distance(self, other):
        ...
```

Una vez ejecutado el código, en la consola de Spyder, instanciamos dos objetos y calculamos la distancia entre ellos con el método `distance`:

```
In [..]: p = Point(3,4)
In [..]: q = Point(2,3)
In [..]: una_distancia = p.distance(q)
```

Luego, decidimos introducir modificaciones en el código del método `distance`, que hacen que devuelva un valor distinto. A continuación, volvemos a ejecutar el código que contiene la definición del método `distance` (con la flecha verde de Spyder) y volvemos a calcular la distancia entre `p` y `q` en la consola:

```
In [..]: otra_distancia = p.distance(q)
```

Para nuestro desconcierto, observamos que los valores de `una_distancia` y `otra_distancia` son iguales, es decir, las modificaciones que hemos introducido en el código del método `distance` no han tenido ningún efecto. ¿Cómo puede ser?

El problema es que un objeto está siempre ligado a la última versión de la clase ejecutada antes de (la ejecución de) su creación[2]. Si no volvemos a ejecutar la creación de los objetos `p` y `q`, no serán conscientes de las modificaciones que se han introducido en el método `distance` después de su creación. Si introducimos modificaciones en el código de unas clases después de haber creado muchas instancias, es muy fácil olvidar crear de nuevo alguna de estas instancias y, en consecuencia, ejecutar una mezcolanza de distintas versiones de las clases del programa con resultados impredicibles y mensajes de error perturbadores. Para evitar este problema, *IPython* (el programa que se ejecuta en la consola de *Spyder*) ofrece la posibilidad de borrar el contenido de todas las variables mediante la instrucción `reset`. *Spyder* ofrece la misma posibilidad mediante el icono del cubo de basura que se encuentra en la pestaña de la consola, arriba a la derecha. Finalmente, *Spyder* también ofrece la posibilidad de configurar, por fichero, el borrado automático de todas las variables antes de cada ejecución en *Run - Configuration per file - Run file with custom configuration+Remove all variables before execution.*

Invocar a los métodos de instancia

En el código siguiente creamos un objeto de la clase `Colour` que almacenamos en la variable `colour`. Luego invocamos al método de instancia `getCmpnt` de este objeto para obtener el valor del primer componente

[2]Para los que vuelven a preguntar ¿por qué?, como quizás sabéis, un IDE como *Spyder* no interpreta el código fuente cada vez que se ejecuta un código. La primera vez que se ejecuta el código fuente la interpretación se hace en dos fases: traducción de los elementos del código fuente a código de máquina abstracta, o bytecode, que se guarda, e interpretación de este bytecode (es decir, traducción del bytecode a código máquina y ejecución del resultado) por una *máquina virtual*. Si se vuelve a ejecutar un elemento traducido previamente a bytecode, esta vez el IDE interpreta directamente este bytecode, ahorrando el tiempo de traducción de código fuente a bytecode. La representación de un objeto, generado al ejecutar la instrucción en la que se crea, incluye una referencia al bytecode de la clase (para poder acceder al bytecode de los métodos de la clase). Aunque se modifique la clase y se vuelva a generar el bytecode correspondiente, la representación del objeto seguirá referenciando al bytecode generado a partir de la versión anterior de la clase.

(rojo) y lo almacenamos en la variable `component`:

```
colour = Colour(200, 200, 200)
component = colour.getCmpnt()
```

En Python, a pesar de que `getCmpnt` es un método de instancia, si usamos la sintaxis estándar para la invocación de un método estático, es decir:

```
colour = Colour(200, 200, 200)
component = Colour.getCmpnt(colour)
```

también funciona y da el mismo resultado.

Sin embargo, en la solución que entregas de esta práctica, cualquier invocación de un método de instancia mediante la sintáxis estándar de invocación de métodos estáticos **está prohibida** y puede resultar en una bajada de nota. ¿Por qué? Porque invocar a un método de instancia con la sintaxis correspondiente a un método estático es una posibilidad particular de Python que no es trasladable a otros lenguajes, y su uso dificulta entender bien los conceptos.

Uso de `NumPy`

En esta práctica, el uso de la biblioteca `NumPy` **está prohibido** y puede resultar en una bajada de nota.