Capítulo 3-3
Modelo de objetos

1. Todo es un objeto

1.1 Principios

1.1.1 Qué sentido dar a «objeto»

Python es un lenguaje que utiliza varios paradigmas y, entre ellos, el paradigma orientado a objetos. Este se elaboró durante los años 1970 y es, ante todo, un concepto. Un objeto representa:

– un objeto físico:
  – parcela de terreno, bien inmueble, apartamento, propietario, inquilino...;
  – coche, piezas de un coche, conductor, pasajero...;
  – biblioteca, libro, página de un libro...;
  – dispositivo de hardware, robot...;
– un objeto informático:
  – archivo (imagen, documento de texto, sonido, vídeo...);
  – servicio (servidor, cliente, sitio de Internet, servicio web...);
  – un flujo de datos, pool de conexiones...;
– un concepto:
  – portador de alguna noción que pueda compartir;
  – secuenciador, ordenador, analizador de datos...

Observación

Aprovechamos para llamar la atención del lector sobre ciertos aspectos cognitivos: a partir del momento en que se modelizan personas como objetos, es fácil olvidar que detrás del código hay seres vivos que piensan, y cuya libertad puede ser limitada por las elecciones que se hacen sobre la manera de modelizar, la cantidad o la calidad de las informaciones que se decide tratar o conservar.

Aquí vemos un ejemplo muy simple y concreto: un gestor de horarios que solo permite crear franjas cada media hora puede tener una repercusión real sobre el funcionamiento de un servicio que necesita una granularidad más fina. Por lo tanto, influye de manera muy negativa en los usuarios que, antes de la llegada del software, tenían horarios que permitían más libertad.

Además, a partir del momento en que almacena datos de sus usuarios, es responsable de su conservación y seguridad; por eso tiene que asegurarse de que no se puedan filtrar, robar o incluso corromper.

Hay organismos que permiten dar una visión correcta de las reglas básicas que se deben seguir para tratar datos de este tipo, y también legislaciones, como el RGPD, que a su vez determinan algunos principios útiles.

Para modelizar como objeto, es necesario seguir algunos principios.
Uno de los principios es la encapsulación de datos. Esto significa que cada objeto posee en su seno no solo los datos que lo describen y que contiene (bajo la forma de atributos), sino también el conjunto de métodos necesarios para gestionar sus propios datos (modificación, actualización, compartición...).
El desarrollo orientado a objetos consiste, simplemente, en crear un conjunto de objetos que representa de la mejor forma posible aquello que modelan y en gestionar sus interacciones.
Cada funcionalidad se modela, de este modo, bajo la forma de interacciones entre objetos. De su correcto modelado y de la naturaleza de sus interacciones dependen la calidad del programa y también su estabilidad y mantenibilidad.
El paradigma orientado a objetos define, entonces, otros mecanismos para dar respuesta a las distintas problemáticas que se le plantean al desarrollador: polimorfismo, interfaces, herencia, sobrecarga de métodos, sobrecarga de operadores…
Es aquí donde se diferencian los lenguajes entre sí, pues cada uno propone soluciones que le son propias utilizando o no ciertos mecanismos del lenguaje orientado a objetos y de forma más o menos fiel a su espíritu.

1.1.2 Adaptación de la teoría de objetos en Python

En lenguajes como PHP, por ejemplo, se agrega una **semántica de objetos** que permite a los desarrolladores escribir de forma similar a un lenguaje orientado a objetos. Esto se realiza en dos etapas: la posibilidad de declarar clases (con interfaces y herencia simple) y la posibilidad de crear instancias de estas clases y acceder a los atributos de los métodos. Pero no es más que una semántica de objetos, puesto que detrás se trata en realidad de tablas (que contienen los atributos) que se asocian a una lista de métodos que pueden aplicarse al objeto.

La implementación está, por tanto, muy lejos de un paradigma orientado a objetos, aunque la semántica esté presente y sea suficiente para este lenguaje.

En los lenguajes orientados a objetos, como Java, el paradigma orientado a objetos está en el núcleo del lenguaje y, por tanto, de la gramática. Se han realizado adaptaciones del concepto para amoldarse a distintos escenarios técnicos o a una filosofía propia del lenguaje. No se dispone de herencia múltiple, y el concepto de interfaz se ha transformado en su totalidad para ofrecer una alternativa. Como no existen más que objetos, es necesario pasar por el proceso de bootstrap y las arquitecturas se han vuelto difíciles o restrictivas debido a limitaciones técnicas que debían respetarse.

C++ también propone sus propias adaptaciones e innovaciones. El modelo orientado a objetos que ofrece es la referencia absoluta de un lenguaje de bajo nivel estáticamente tipado.

Estas son las características esenciales que diferencian estos lenguajes del lenguaje de programación Python y que hacen que el modelo de objetos de Python sea, necesariamente, muy diferente.

Pero, además de ser diferente, el lenguaje ha tratado de aprovechar sus cualidades básicas que lo diferencian de otros lenguajes para adaptar completamente la teoría de objetos a su filosofía y encontrar aplicaciones particularmente novedosas que permitan proponer un conjunto a la vez completo, preciso y con buen rendimiento.

Por este motivo se encuentran tantas diferencias. Por tanto, la forma de trabajar de Python está completamente adaptada al lenguaje, aunque no puede decirse que el modelo de objetos de Python sea mejor que el de C++, por ejemplo. El modelo de C++ está adaptado a C++ y el de Python lo está a Python. Si se hubieran retomado los conceptos de C++ en Python, estos no habrían encontrado lugar, y viceversa.

Al final, cuando se viene de trabajar en otro lenguaje, adquirir práctica puede resultar más o menos fácil en un primer lugar, aunque para comprender realmente las diferencias y sutilidades es necesario ir más allá en el modelo de objetos, en la teoría, y comprender las elecciones realizadas y su adaptación a las características del lenguaje.

Por ello, no se deje sorprende por el hecho de que no existan las palabras clave **`new`** o **`this`**, que la firma de los métodos sea diferente, sino comprenda la filosofía general y saque provecho de las posibilidades que se ofrecen.

Python se ha creado en un momento en el que los lenguajes de referencia ya existían y habían marcado su tiempo. Ha aprovechado su experiencia y sacado el mejor provecho. A día de hoy, el propio lenguaje Python es una fuente de inspiración.

1.1.3 Generalidades

El objeto es uno de los pilares esenciales de Python, que decide proporcionar un lenguaje donde todo es un objeto, con el objetivo de responder de manera sencilla y eficaz a problemáticas complejas, permitir una gran flexibilidad y ofrecer una gran libertad de acción a los desarrolladores, como veremos en este capítulo.

Python tiene un único principio, que es «todo es un objeto», lo cual no es simplemente un concepto genérico. En efecto, si es evidente que una instancia es un objeto, el hecho de que todo sea un objeto quiere decir que la propia clase es un objeto, que un método es un objeto y que una función es un objeto.

Esto significa que todas las clases, funciones y métodos disponen de atributos y de métodos particulares, y que pueden modificarse tras su creación.
De este modo, es posible declarar clases, métodos y funciones de manera imperativa, mediante el uso de las palabras clave **`class`** o **`def`**, aunque también pueden declararse por asignación, abriendo así posibilidades muy interesantes.
Pero Python no es un lenguaje doctrinal con una única visión y buscando imponerla. Si bien el objeto está en el núcleo de sus funcionalidades, los demás paradigmas no se han rechazado o dejado de lado. Son tan importantes los unos como los otros.
En efecto, en función de la tarea que quiera cumplir, habrá una única manera evidente de proceder y aun así se podrá recurrir a uno de los tres paradigmas: imperativo, orientado a objetos o funcional.
Python no preconiza la superioridad del objeto, ni busca impedir la programación imperativa para obligar a que se utilicen objetos simplemente porque el objeto sea un enfoque más moderno o más de moda.
Es, por otro lado, interesante, cuando se conocen varios lenguajes, ver cómo Python es capaz de vincular la experiencia imperativa con la orientación a objetos y hacer emerger lo mejor de cada una.
Los debutantes que ya conozcan alguno de los paradigmas podrán desarrollar utilizando preferentemente el paradigma que conozcan y, a continuación, descubrir los demás poco a poco, en función de su experiencia.

1.2 Clases

1.2.1 Introducción

Una clase se define, simplemente, así:

```
>>> class A:
...     pass
```

Esta definición es de naturaleza imperativa, en el sentido de que una clase es un bloque que contiene un conjunto de instrucciones imperativas que se recorren y ejecutan unas detrás de otras.
Estas instrucciones pueden ser un docstring, por ejemplo:

```
>>> class A:
...     """Descripción de mi clase"""
```

Existen, en realidad, dos formas de describir una clase: bien utilizando este modo imperativo, descriptivo, que pone de relieve la encapsulación (utilizada a menudo), o bien mediante un prototipo, que también permite Python, de manera similar a JavaScript, como veremos más adelante.

1.2.2 Declaración imperativa de una clase

Una clase puede contener instrucciones declarando una variable, que se convierte en un atributo de clase, o una función, que se convierte en un método.

```
>>> class A:
...     """Descripción de mi clase"""
...     atributo = "Esto es un atributo"
...     def método(self, *args, **kwargs):
...         return "Esto es un método"
```

Una clase encapsula, así, con claridad todos sus datos, que son accesibles:

```
>>> A.__doc__
'Descripción de mi clase'
>>> A.atributo
'Esto es un atributo'
>>> A.método
<function método at 0x257ea68>
```

De este modo, el método es un atributo como los demás, pues cuando se accede sin invocarlo se devuelve la instancia correspondiente al método.
La única complejidad en la creación de clases se desprende de la complejidad funcional, del correcto modelado de objetos y de sus relaciones. Se recomienda trabajar de modo que los datos de una clase o de una instancia le pertenezcan y estén gestionados por la propia instancia, y no desde el exterior.

1.2.3 Instancia

Creemos una instancia:

```
>>> a = A()
```

Y accedamos al contenido de la instancia, definido en la clase:

```
>>> a.__doc__
'Descripción de mi clase'
>>> a.atributo
'Esto es un atributo'
>>> a.método
<bound method A.método of <__main__.A object at 0x25dfa90>>
```

Los atributos y métodos de la clase están, ahora, disponibles para la instancia, puesto que incluye un vínculo hacia los elementos de la clase, tal y como sugiere el término «bound».
Como puede verse, el hecho de acceder al método devuelve, simplemente, un objeto, aunque no invoca al método. Para realizar dicha llamada es necesario agregar los paréntesis y pasar los eventuales argumentos.
Recordemos que la firma del método espera un parámetro. Este parámetro representa, en realidad, la instancia. El vínculo entre el primer argumento del método definido en la clase y la instancia se realiza de manera natural:

```
>>> a.método()
'Esto es un método'
```

Es la gramática del lenguaje la encargada de informar el primer argumento situando la instancia. En Python no se hace magia, no existe ninguna variable mágica que represente automáticamente la instancia en curso; esta última está realmente visible y presente en la firma. Por convención, se denomina **`self`**.

La noción de interconexión entre una instancia y su clase es un elemento importante que debe dominarse. En efecto, si de una u otra manera se modifican los elementos de la clase, entonces se modifican también los elementos de la instancia:

```
>>> class B:
...     a = 'Otro atributo'
...     def m(self, *args, **kwargs):
...         return 'Otro método'
...
>>> A.atributo = B.a
>>> A.método = B.m
>>> a.atributo
'Otro atributo'
>>> a.método()
'Otro método'
```

Igual que la instancia no tiene su propio atributo, su valor es el de la clase y, como el atributo de la clase es dinámico, cualquier cambio realizado sobre este afectará a la instancia. No tener claro este aspecto puede llevarnos a generar errores inesperados. No obstante, preste atención: no es así como declaramos los atributos únicos de cada instancia, sino que se pasan al constructor, como veremos más adelante.

Informar los atributos directamente a nivel de la clase sirve, también, para que se compartan entre todas las instancias. Son, de algún modo, atributos de clase, en la semántica de Python, lo que equivale a atributos estáticos en la mayoría de los lenguajes.

Si bien estos atributos son de la clase, nada impide que un atributo con el mismo nombre aparezca en una instancia.

En este momento, podemos considerar que el atributo de la clase contiene el valor por defecto y el atributo de la instancia contiene el valor asociado de manera durable a la instancia.

Cuando el atributo de una instancia se modifica y recibe otro valor diferente al de la clase, se encuentra desconectado del atributo de la clase.

```
>>> a.atributo = 'Atributo de instancia'
>>> A.atributo
'Otro atributo'
>>> a.atributo
'Atributo de instancia'
```

El atributo de la instancia está conectado al de la clase. Ahora:

```
>>> a.atributo = A.atributo
```

Nos contentamos con realizar una asignación, sin cambiar de valor:

```
>>> A.atributo = 'B'
>>> a.atributo
'A'
```

1.2.4 Objeto en curso

Se denomina objeto «en curso» a la instancia en curso de la clase.
En Python, dicha instancia se denomina **`self`**, aunque no es más que una convención. Lo que importa es que el objeto en curso es, sistemáticamente, el primer objeto que recibe como parámetro un método, y dicho vínculo se establece de forma automática.
En la mayoría de los lenguajes existe una palabra clave **`this`** que permite ejecutar un método como una función, un poco de forma mágica, pues **`this`** representa a la instancia en curso.
Como a Python no le gusta la magia y quiere preservar la legibilidad, se contenta con exigir un primer argumento que representa a la instancia y el vínculo se establece a bajo nivel, pero no hay ningún elemento mágico de por medio. Lo que se utiliza en la función es, simplemente, variables que se presentan en la firma del método.
Cabe destacar que no se utiliza la palabra clave **`this`** ni la palabra clave **`new`** para crear la instancia.

1.2.5 Declaración por prototipo de una clase

La programación orientada a objetos por prototipo consiste en crear una clase y, a continuación, asignarle atributos y métodos como se hace, por ejemplo, en JavaScript.
Esto es muy diferente a la programación orientada a objetos clásica, puesto que nos contentamos con declarar una clase que es un recipiente vacío con un nombre y, a continuación, se le agregan atributos y métodos.
Estos métodos pueden ser, para ciertos lenguajes, simples funciones que transforman un objeto que se pasa como parámetro o que reciben un objeto como parámetro para devolver otro objeto sin que exista ningún vínculo entre ellos, salvo el hecho de agregarse en la misma clase.
El recurso de una palabra clave permite, por tanto, crear un vínculo artificial pero suficiente entre los métodos de una misma clase y sus propiedades. Esto puede parecer una agregación de propiedades y de funciones similares a lo que serían atributos y métodos.
Semánticamente, el uso de una clase así es idéntico al de una clase declarada de manera clásica, aunque los mecanismos internos sean totalmente distintos.
Esto no entra, en absoluto, en el espíritu de la programación orientada a objetos, pues si bien la encapsulación se resuelve de una manera diferente, aunque comprensible, los demás mecanismos tales como la instanciación, la diferenciación de instancias o el polimorfismo, por ejemplo, no pueden resolverse, o bien se resuelven de manera poco satisfactoria. Además, ciertos lenguajes hacen todas las clases puramente estáticas.
Estos lenguajes son, entonces, una interpretación del paradigma orientado a objetos bastante reducida, aunque por el contrario representan una ventaja indiscutible, que es la capacidad evolutiva, dado que, en cualquier momento, es posible agregar o modificar funciones.
En efecto, en la mayoría de los lenguajes, una vez declarada la clase, es imposible agregar nuevos métodos o atributos. En ocasiones, una permisividad natural permite agregar atributos de manera lateral. No obstante, esto es una limitación importante que hace que la programación orientada a objetos por prototipo encuentre su verdadero lugar.

En lo relativo a Python, esto es muy distinto. Por un lado, su lectura extrema del paradigma orientado a objetos hace que las propias clases, funciones y métodos sean objetos sobre los que es posible actuar como con cualquier otro objeto. Por otro lado, el hecho de que sea dinámico implica que, en todo momento, sea posible realizar una asignación o una modificación.
De este modo, es posible declarar una clase y, a continuación, añadir más tarde un atributo, por agregación. Para comenzar, creemos una clase de manera declarativa, como hemos hecho hasta ahora:

```
>>> class Declarativa(object):
...     """Clase escrita de manera declarativa"""
...
...     atributo_de_clase = 42
...
...     def __init__(self, name):
...         self.name = name
...         self.subs = []
...
...     def __str__(self):
...         return "{} ({})".format(self.name, ", ".join(self.subs))
...
 ...    def mostrar(self):
...         print(self)
```

Ahora podemos utilizar este objeto:

```
>>> a = Declarativa("test")
>>> a.subs.append("cosa", "chisme")
>>> print(a)
test (cosa, chisme)
>> dir(a)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'atributo_de_clase']
>>> Declarativa.mostrar
<function Declarativa.mostrar__ at 0x7fb456895bf8>
```

Presentaremos, ahora, el código equivalente al anterior, escrito mediante prototipo. Veremos que primero se escriben los métodos:

```
>>> def proto__init__(self, name):
...     self.name = name
...     self.subs = []
...
>>> def proto__str__(self):
...     return "{} ({})".format(self.name, ", ".join(self.subs))
...
>>> Prototipo = type("Prototipo", (object,), {
```

```
    "__init__": proto__init__,
    "__str__": proto__str__,
    "atributo_de_clase": 42})
```

También es posible agregar funciones más tarde:

```
>>> def mostrar(self):
...     print(self)
...
>>> Prototipo.mostrar = mostrar
```

El resultado es completamente idéntico a nuestra clase declarada de manera clásica:

```
>>> dir(Prototipo)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'atributo_de_clase']
```

Esta forma de operar no es un error de programación o de diseño, es algo natural y que está previsto en Python.

Como un método no es más que una función encapsulada en una clase (si bien sigue algunas reglas particulares suplementarias que se presentan en la sección Métodos), Python no tiene ningún problema con esta forma de trabajar.

```
>>> def m(self):
...     return "Definido por prototipo"
...
>>> A.método = m
```

Esta forma de trabajar es bastante diferente a la de los lenguajes específicamente cualificados como «programación orientada a objetos por prototipo», como por ejemplo JavaScript, uno de los más conocidos y utilizados en este dominio.

Sin embargo, es bastante limitada en Python, a parte de las librerías que utilizan masivamente nociones complejas, tales como metaclases; por ejemplo, para resolver requisitos específicos de diseño.

La gran ventaja de esta técnica es que permite modificar las clases en cualquier momento, o extenderlas tanto como se quiera. Podemos perfectamente declarar una clase de la manera habitual, y después, en otro módulo importarla y agregarle métodos o atributos.

1.2.6 Tuplas con nombre

Existen muchos casos de uso en los que se necesita la flexibilidad de un objeto pero no se desea pasar demasiado tiempo escribiendo una clase. Para ello, existen las tuplas con nombre:

```
>>> from collections import namedtuple
>>> Punto = namedtuple('Punto', ['x', 'y'])
```

`Punto` es una clase particular que dispone de dos atributos **`x`** e **`y`**. Puede instanciarse:

```
>> p = Punto(4, 2)
```

Capítulo 9

Trabajar en 3D con Pygame

1. Introducción

Los juegos desarrollados durante este libro son exclusivamente en dos dimensiones (2D). En la práctica, Pygame está más asociado con el desarrollo de juegos 2D, aunque permite la visualización tridimensional.

Hay varios videojuegos que son tridimensionales o utilizan gráficos que hacen que se acerquen a las tres dimensiones. En particular, podemos mencionar los llamados juegos FPS (*First Person Shooter*, videojuego de disparos con vista en primera persona, en español). Este tipo de juego suele consistir en la encarnación de un personaje con un arma de fuego dentro de un 3D inmersivo. El objetivo es derribar a los enemigos que se va encontrando. En general, el personaje principal se mueve a través de una red de pasillos dibujados en tres dimensiones.

Debido a su base SDL, Pygame es una herramienta muy orientada a las 2D. Todos los juegos que se han puesto como ejemplos en este libro, son juegos bidimensionales. Entonces, ¿qué necesitarían para ser juegos en 3D? La respuesta puede parecer obvia: una decoración en 3D y una visualización en 3D. En otras palabras, su algoritmo sigue siendo el mismo: la lógica y el bucle del juego en sí, se supone que no necesitan modificaciones excesivas. Pero queda por ver cómo hacer 3D con Pygame. La respuesta se puede resumir en una palabra: OpenGL.

2. La librería 3D OpenGL

OpenGL (*Open Graphics Library*) es una librería que permite hacer 3D. Se utiliza en muchos dominios más allá de los videojuegos: simulación científica, CAO, realidad aumentada, etc. Su objetivo es permitir crear objetos 3D a los que se puedan asociar texturas. OpenGL se encarga de la visualización de estos objetos 3D teniendo en cuenta la orientación, la distancia y los aspectos relativos a la luz, como la sombra proyectada, la transparencia, etc.

Desde luego, el objetivo aquí no es enseñarle OpenGL. De hecho, OpenGL es un mundo amplio y complejo y se necesitaría mucho más que estas pocas páginas para aprender a programar de forma autónoma con esta herramienta. El objetivo es acompañarle en sus primeros pasos en 3D en un entorno Pygame.

3. OpenGL en Python/Pygame

3.1 PyOpenGL

Hay un módulo de Python que permite trabajar con OpenGL: `PyOpenGL`. Si no está instalado en su máquina, lanzar el comando `pip` en el terminal le permite hacerlo.

```
pip install PyOpenGL
```

Para usarlo en código Python, simplemente importe el módulo como se hace normalmente (para Pygame, por ejemplo). En general, procedemos a importar escribiendo estas dos líneas:

```
from OpenGL.GL import *
from OpenGL.GLU import *
```

3.2 Las nociones fundamentales: vértice y arista

Para definir objetos 3D en OpenGL, utilizaremos dos conceptos fundamentales: vértice (*vertex* en inglés) y arista (*edge* en inglés).

Esquemáticamente, podemos considerar que los vértices son puntos en el espacio y, por lo tanto, tienen sus coordenadas en tres dimensiones. Por ejemplo: (1, 2, -4). Las aristas son los segmentos que conectan los vértices.

De esta manera, si definimos dos vértices:

- vértice 1 situado en las coordenadas (1, 0, 0),
- vértice 2 situado en las coordenadas (0, -1, 0),
- entonces podemos definir la arista 1 que conecta el vértice 1 con el vértice 2.

Esto es equivalente a pensar que definir los objetos 3D en OpenGL es, ante todo, definir una colección de puntos tridimensionales (vértices) y luego definir las aristas que conectan estos vértices entre sí.

3.3 PyOpenGL y Pygame

Cuando creamos la ventana de juego Pygame, especificamos un modo de funcionamiento relativo a OpenGL. El valor utilizado es: `pygame.DOUBLEBUF | pygame.OPENGL`.

Por ejemplo, tenemos este tipo de código:

```
import pygame
pygame.init()
ZONA = (800,800)
pygame.display.set_mode(ZONA, pygame.DOUBLEBUF|pygame.OPENGL)
```

4. PyOpenGL/Pygame: el ejemplo del cubo

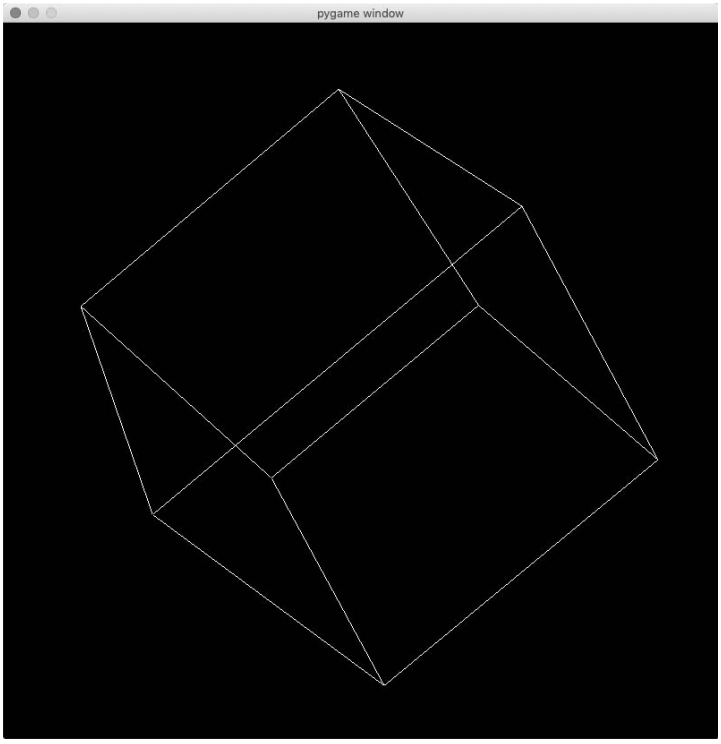
Tomemos el ejemplo clásico de un cubo representado en tres dimensiones, que sometemos a una rotación continua. La idea es mostrar solo las aristas del cubo; no definimos ninguna textura o coloración de las caras.

4.1 El código global

Comencemos presentando el código global del ejemplo, antes de proceder a la explicación detallada del programa.

```
import pygame
from OpenGL.GL import *
from OpenGL.GLU import *

VERTICES = (
    (1, -1, -1),
    (1, 1, -1),
    (-1, 1, -1),
    (-1, -1, -1),
    (1, -1, 1),
    (1, 1, 1),
    (-1, -1, 1),
    (-1, 1, 1)
```

```
    )

ARISTAS = (
    (0,1),
    (0,3),
    (0,4),
    (2,1),
    (2,3),
    (2,7),
    (6,3),
    (6,4),
    (6,7),
    (5,1),
    (5,4),
    (5,7)
    )


def Cubo():
    glBegin(GL_LINES)
    for arista in ARISTAS:
        for vertice in arista:
            glVertex3fv(VERTICES[vertice])
    glEnd()

pygame.init()
ZONA = (800,800)
pygame.display.set_mode(ZONA, pygame.DOUBLEBUF|pygame.OPENGL)

gluPerspective(45, (ZONA[0]/ZONA[1]), 0.1, 50.0)
glTranslatef(0.0,0.0, -5)

while True:
 for event in pygame.event.get():
   if event.type == pygame.QUIT:
     pygame.quit()
     quit()

 glRotatef(1, 3, 1, 3)
 glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
 Cubo()
 pygame.display.flip()
 pygame.time.wait(10)
```



Captura de pantalla de la ventana del cubo 3D en rotación

4.2 Explicación detallada del código

Empezamos importando Pygame y lo que se necesita para usar OpenGL.

```
import pygame
from OpenGL.GL import *
from OpenGL.GLU import *
```

Defina una lista de vértices.

```
VERTICES = (
 (1, -1, -1),
 (1, 1, -1),
 (-1, 1, -1),
 (-1, -1, -1),
 (1, -1, 1),
 (1, 1, 1),
 (-1, -1, 1),
 (-1, 1, 1)
)
```

Por lo tanto, el cubo tiene dos unidades en cada lado y se define entre -1 y +1 en cada uno de los tres ejes.

Los ocho vértices participan en la definición del cubo:

- Vértice 0: (1, -1, -1)
- Vértice 1: (1, 1, -1)
- Vértice 2: (-1, 1, -1)
- Vértice 3: (-1, -1, -1)
- Vértice 4: (1, -1, 1)
- Vértice 5: (1, 1, 1)
- Vértice 6: (-1, -1, 1)
- Vértice 7: (-1, 1, 1)

A continuación, defina los bordes del cubo.

```
ARISTAS = (
 (0,1),
 (0,3),
 (0,4),
 (2,1),
 (2,3),
 (2,7),
 (6,3),
 (6,4),
 (6,7),
 (5,1),
 (5,4),
 (5,7)
)
```

por ejemplo, `(2,7)` significa "la arista entre los vértices 2 y 7".

A continuación, cree la función que genera el cubo. Para ello, utilice las siguientes funciones OpenGL:

- `glBegin` se utiliza para declarar el inicio del procesamiento. La función recibe como argumento el tipo de elemento geométrico dibujado, aquí `GL_LINES` lo que significa que dibujamos líneas.
- `glEnd` se utiliza para indicar el final de este procesamiento específico.
- `glVertex3fv` permite dibujar aristas.

```
def Cubo():
 glBegin(GL_LINES)
 for arista in ARISTAS:
   for vertice in arista:
     glVertex3fv(VERTICES[vertice])
 glEnd()
```

Además de las líneas (*GL_LINES*), el resto de elementos geométricos que puede dibujar con OpenGL son:

`GL_POINTS`: para dibujar puntos (vértices no conectados).

`GL_LINES`: para dibujar líneas (que es lo que usamos aquí).

`GL_LINE_STRIP`: para dibujar líneas conectadas (de un paso al siguiente).

`GL_LINE_LOOP`: para dibujar líneas conectadas, pero que finalmente forman un bucle.

`GL_TRIANGLES`: para dibujar triángulos.

`GL_QUADS`: para dibujar cuadriláteros.

`GL_POLYGON`: para dibujar polígonos convexos.

Ahora podemos inicializar Pygame y definir una ventana de juego dedicada a OpenGL.

```
pygame.init()
ZONA = (800,800)
pygame.display.set_mode(ZONA, pygame.DOUBLEBUF|pygame.OPENGL)
```

Después, manejamos la perspectiva, es decir, la característica que define cómo el observador verá el sujeto. Usamos la función OpenGL `gluPerspective` que recibe cuatro argumentos:

- El ángulo del campo de visión (en grados).
- El marco de visualización.
- La distancia desde el plano del sujeto más cercano.
- La distancia desde el plano del sujeto más lejano.

```
(gluPerspective(45, (ZONA[0]/ZONA[1]), 0.1, 50.0)
```

Seguidamente hacemos una ligera traslación a lo largo del eje z para obtener una mayor comodidad. De hecho, permite alejar un poco el cubo. De lo contrario estaríamos demasiado cerca de él.

```
GlTranslatef(0.0,0.0, -5)
```

Luego inicie el bucle del juego.

```
while True:
```

Como de costumbre, ocúpese de la acción de abandonar el programa.

```
for event in pygame.event.get():
 if event.type == pygame.QUIT:
   pygame.quit()
   quit()
```