Pilas: Asteroides

asteroides.py

Vamos a construir, paso a paso, otro de los ejemplos que incluye como minijuego Pilas. Se trata de **asteroides.py**

El juego está inspirado en el clásico asteroides de las máquinas recreativas. Se trata de una nave espacial, dirigida con las flechas del cursor, que se enfrenta a una lluvia de asteroides que hay que destuir. El detalle importante es que al disparar a un asteroide, éste se destruye en trozos más pequeños que son más difíciles de destruir. Por otra parte, se trata de un mundo esférico, y si nosotros o las rocas nos salimos por un extremo de la ventana, aparecemos por el extremo contrario.

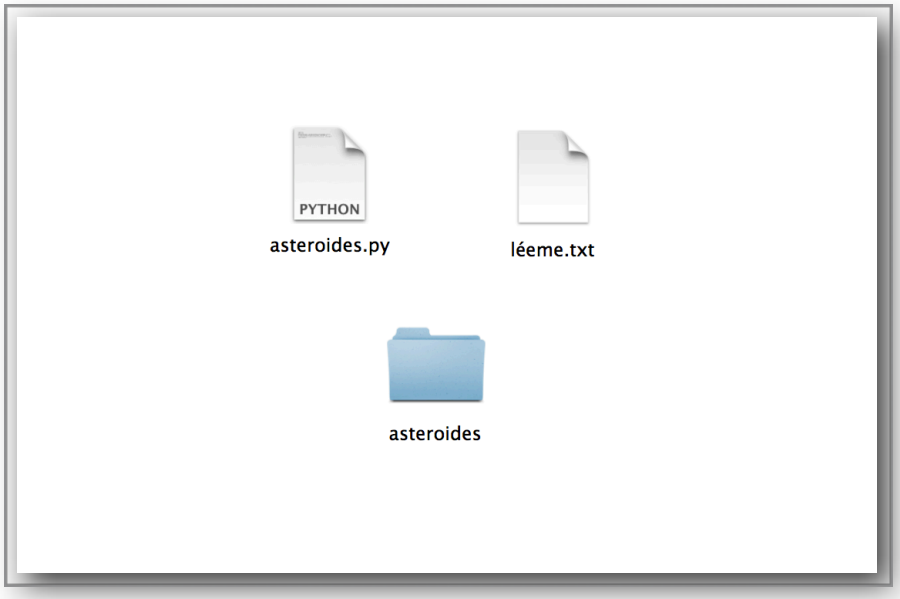
Hay varios detalles más; un marcador, sonido, un contador de vidas y una pantalla de presentación de juego y otra de ayuda. Además, contaremos con varios niveles de dificultad, a medida que destruimos todos los asteroides. Todo ello nos permitirá aprender nuevas técnicas, empezando por la **modularización**. y la organización en torno a **escenas**.

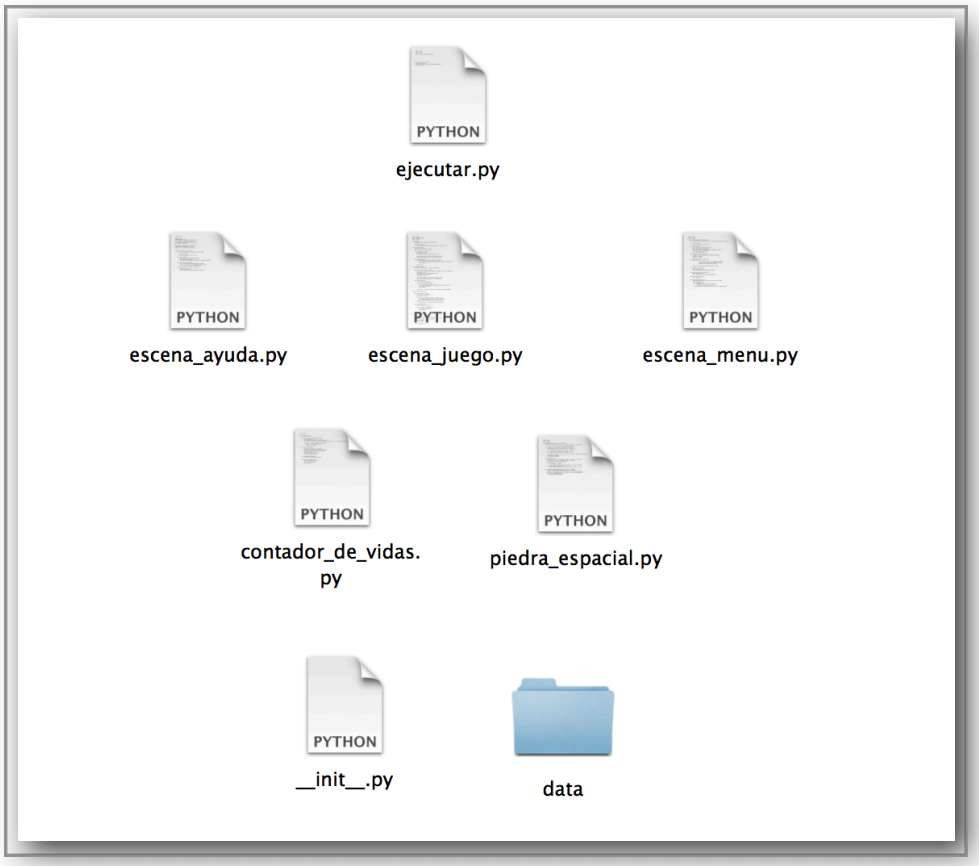
La **modularización** intenta identificar partes reutilizables de un programa y desacoplarlas en archivos diferentes. Por ejemplo, si el comportamiento de los asteroides lo implementamos por separado, más adelante podemos cambiar la forma en la que se crean, se desplazan o se destruyen sin interferir en nada más del videojuego. Además,

permite que el código sea manejable, al estar divido en piezas más pequeñas y no un solo archivo de una longitud inabarcable.

Por otra parte, el concepto de **escena**, al que aludimos en el tutorial anterior, es una manera más apropiada de dividir un videojuego en secciones diferentes sin que interfieran unas con otras y guardando el estado en el que se encuentran. En el caso que nos ocupa, tenemos tres escenas naturales; la de la pantalla de inicio, la de las instrucciones y la del propio juego.

Éste es el esquema de archivos que vamos a construir y que vamos a ver paso a paso:





¡Organizarse el algo muy importante! Cuanto más claro dejemos el acceso a nuestros jugadores, más fácil será para ellos identificar qué deben hacer para jugar con nuestro videojuego. He visto demasiadas veces cómo, al abrir la carpeta de un juego, aparecen montonadas de archivos sin que quede claro cuál es el ejecutable o qué hay que hacer. Así que nosotros vamos a crear una carpeta en la que aparecerán tres cosas:

- **Un archivo .py** que será sobre el que hay que hacer doble click (o arrastrar sobre la ventana de Pilas) para ejecutar nuestro juego.
- **Un archivo .txt** (podría ser un pdf o similar) donde incluiremos instrucciones, observaciones o licencias.
- **Una carpeta** en donde irá ubicado todo el resto del material que necesita el juego.

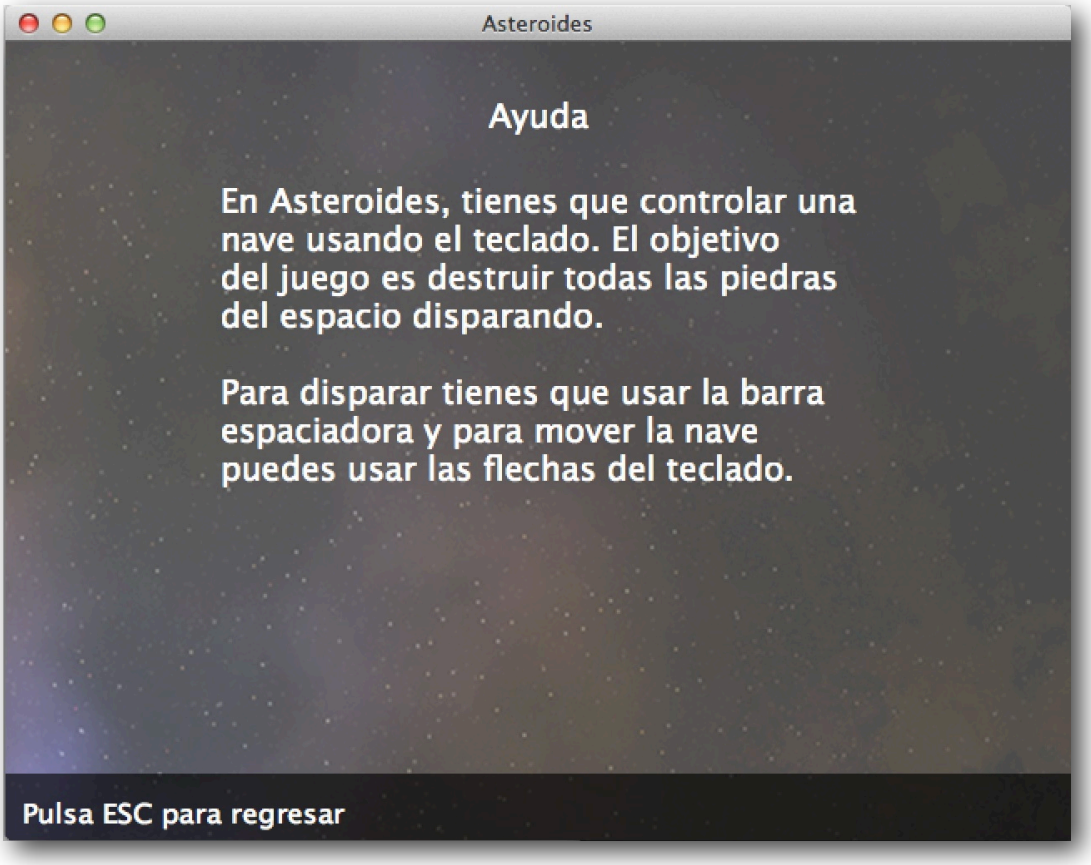
En esta última carpeta es donde tendremos implementada nuestra modularidad; el código separado en partes reusables, escenas y material multimedia. Como puedes ver, éstos son los archivos que vamos a poner:

- **ejecutar.py** . Se trata del primer archivo que se ejecuta y se encarga de cargar la primera escena.
- Las tres escenas de las que constará el juego:
  - **escena_menu.py** es la escena de inicio, con el menú de selección del juego que da acceso a las demás partes.
  - **escena_ayuda.py** contiene la pantalla con las breves instrucciones.
  - **escena_juego.py** es la implementación del juego propiamente dicho.
- **contador_de_vidas.py** se encarga, como es obvio, del marcador que indica cuántas vidas le quedan al jugador. Lo hemos separado, como hemos indicado, atendiendo a la modularidad (del mismo modo que, si hubiéramos optado por un marcador de puntos más elaborado, podríamos haberlo separado también).
- **piedra_espacial.py** define los asteroides y su comportamiento. Es muy útil tener en módulos separados los diferentes personajes más elaborado para, de esta manera, poderlos modificar con facilidad e incluso usarlos en otros proyectos.
- **__init__.py** es un archivo (vacío en este caso) que debe estar necesariamente en cualquier módulo importable de python, igual que en las definiciones de las clases.
- La **carpeta data**. Allí incluiremos el resto del material que usamos que no sean archivos de python, generalmente imágenes y sonidos propios que no incorpora Pilas. En este proyecto, se trata de las imágenes de fondo de la ayuda y de la pantalla de menú, el título del juego y el icono que marca el número de vidas que le quedan al jugador.

¿Tienes ya creada la estructura de carpetas? ¿Estás preparado?... ¡Escribamos código!

escena_ayuda.py

Vamos a empezar por la escena más sencilla, la que muestra la pantalla de ayuda. Lo que queremos mostrar tiene el siguiente aspecto:



¿Qué elementos tenemos aquí?

- Una imagen de fondo de un campo de estrellas

  Esta imagen o la que queramos usar, la tendremos que poner en la carpeta **data**. La nuestra, tiene el nombre **ayuda.png**

- Un texto que muestra la ayuda

  En efecto, acertaste; optaremos por usar un **actor** de Pilas de tipo **Texto**.

- Un mensaje de aviso

  Indicará que para volver a la pantalla principal hay que pulsar la tecla 'ESC'. Ya nos resulta familiar, ¿verdad? Se trata de una función **pilas.avisar()**

Una vez visto en detalle, vamos al código que lo implementa:

```
# -*- coding: utf-8 -*-

import pilas

MENSAJE_AYUDA = """
En Asteroides, tienes que controlar una
nave usando el teclado. El objetivo
del juego es destruir todas las piedras
del espacio disparando.

Para disparar tienes que usar la barra
espaciadora y para mover la nave
puedes usar las flechas del teclado.
"""


class Ayuda(pilas.escena.Base):
    "Es la escena que da instrucciones de cómo jugar."

    def __init__(self):
        pilas.escena.Base.__init__(self)

    def iniciar(self):
        pilas.fondos.Fondo("data/ayuda.png")
        self.crear_texto_ayuda()
        self.pulsa_tecla_escape.conectar(self.cuando_pulsa_tecla)

    def crear_texto_ayuda(self):
        pilas.actores.Texto("Ayuda", y=200)
        pilas.actores.Texto(MENSAJE_AYUDA, y=150)
        pilas.avisar("Pulsa ESC para regresar")

    def cuando_pulsa_tecla(self, *k, **kw):
        import escena_menu
        pilas.cambiar_escena(escena_menu.EscenaMenu())
```

Lo primero (a parte de la importación obligatoria de Pilas) es la definición de la constante **MENSAJE_AYUDA**. Observa que su nombre está en mayúsculas; es un convenio habitual, de manera que si encontramos en el código una variable cuyo nombre está en mayúsculas, sabremos que se refiere a un valor que no cambia (es decir, una constante). En nuestro caso, contiene el texto de ayuda que se muestra en la pantalla.

A continuación, **creamos la definición de la clase que va a representar nuestra escena**. Fíjate bien en el proceso, pues el mecanismo será siempre similar para cualquier escena que queramos crear en nuestros juegos. ¿Has observado cómo la definimos?

```
class Ayuda(pilas.escena.Base):
    "Es la escena que da instrucciones de como jugar."
```

Lo primero que notarás es que todas las escenas derivan de una misma clase, la clase padre de todas las escenas, **pilas.escena.Base**. Y lo segundo es el texto (suelto) que aparece justo debajo de la declaración de la clase. ¿Recuerdas las formas de documentarte a través del código? Se apoyan, precisamente, en estas **cadenas de texto de documentación** y si pides ayuda sobre esta clase con la función **help()** será este texto el que te aparezca.

Bien. Siguiendo adelante, verás, como ya viene siendo habitual, la inicialización de la clase como una clase de tipo escena, llamando al método **__init__()** de la clase padre desde el método **__init__()** de nuestra propia clase. Ja, ja, ¿no es un trabalenguas? En cualquier caso, a estas alturas, esto debería estar claro para ti.

Es en el método **iniciar()** donde debemos colocar todo aquello que queremos que aparezca al comenzar la escena. En primer lugar, hemos incluido el fondo para la pantalla de ayuda, **ayuda.png**, imagen que está dentro de la carpeta **data**.

Y, a continuación y como es lógico, hay que añadir el texto de la ayuda y habilitar el abandono de la pantalla de algún modo. Para ambas tareas usamos métodos que definimos para la instancia de la clase. En el primer caso, **crear_texto_ayuda()** y en el segundo, **cuando_pulsa_tecla()**, conectándo a la tecla **escape**:

```
        self.crear_texto_ayuda()
        self.pulsa_tecla_escape.conectar(self.cuando_pulsa_tecla)
```

Si observas la definición de la primera de las funciones, todo debería resultarte comprensible: se escribe el texto **"Ayuda"** en el centro a la altura **y=200**, se escribe a continuación el texto alnacenado en la constante **MENSAJE_AYUDA** en otra posición y se muestra la advertencia de que se pulse la tecla **escape** para salir.

La segunda función es más interesante, empezando por su declaración:

```
def cuando_pulsa_tecla(self, *k, **kw):
```

Esta, a primera vista, extraña forma de declarar los argumentos de una función es un estándar en Python. De forma rápida, recuerda que además de los argumentos obligatorios, Python puede aceptar un número variable de argumentos que es recogido como una lista y también argumentos con valores por defecto y etiquetados, que son recogidos como un diccionario. Desde ese punto de vista, ***k** representa a todos los posibles argumentos de tipo lista y ****kw** representa a todos los posibles argumentos de tipo diccionario. Más información en la wiki o aquí.

¿Qué es lo que se hace en el método **cuando_pulsa_tecla()**? Lo que se pretende es pasar a la escena principal, abandonando la ayuda. ¿Dónde está la escena definida la escena principal? En el archivo/módulo **escena_menu.py**. Importémoslo, entonces:

```
import escena_menu
```

Allí, como veremos enseguida, está definida la clase de escena **EscenaMenu**. Así que debemos instanciar (crear) un objeto de este tipo y decirle a Pilas que cambie a esta escena. Dicho y hecho:

```
pilas.cambiar_escena(escena_menu.EscenaMenu())
```

¿Comprendes cómo funciona? Para cambiar de escena usamos el método de Pilas **pilas.cambiar_escena()** y le pasamos la escena en cuestión, en nuestro caso creándolo al mismo tiempo, un objeto del tipo **EscenaMenu** definido en el módulo **escena_menu**: **escena_menu.EscenaMenu()**. Pilas se encarga del resto.

Para. Respira. Comprende. ¡El mecanismo de escenas ya es tuyo! Ya puestos, podemos pasar a la siguiente escena. ¿Adivinas? Sí, **EscenaMenu**.

escena_menu.py

Ésta es la pantalla principal del juego, el menú inicial:

¿En qué consiste esta pantalla?

- Se muestran las opciones, que se iluminan al paso del ratón y dan lugar al resto de las escenas.
- El título del juego aparece por la parte superior hasta que alcanza su posición final.
- Se muestra un fondo estrellado, difuminado en su parte inferior para realzar el texto de las opciones del menú.
- Aparecen unos asteroides desplazándose por la ventana, lo que ambienta el juego de forma adecuada y queda visualmente atractivo.

La imagen de fondo es la del apartado anterior, ligeramente modificada. La hemos denominado **menu.png**. Del mismo modo, el título es una imagen, no un texto, de forma que quede más aparente. Su imagen correspondiente es **titulo.png**.

¡Veamos el código!

```
# -*- coding: utf-8 -*-

import pilas
import random


class EscenaMenu(pilas.escena.Base):
    "Es la escena de presentación donde se elijen las opciones del juego."

    def __init__(self):
        pilas.escena.Base.__init__(self)

    def iniciar(self):
        pilas.fondos.Fondo("data/menu.png")
        self.crear_titulo_del_juego()
        pilas.avisar(u"Use el teclado para controlar el menú.")
        self.crear_el_menu_principal()
        self.crear_asteroides()

    def crear_titulo_del_juego(self):
        logotipo = pilas.actores.Actor("data/titulo.png")
        logotipo.y = 300
        logotipo.y = [200]

    def crear_el_menu_principal(self):
        opciones = [
                ("Comenzar a jugar", self.comenzar_a_jugar),
                ("Ver ayuda", self.mostrar_ayuda_del_juego),
                ("Salir", self.salir_del_juego)
               ]
        self.menu = pilas.actores.Menu(opciones, y=-50)

    def comenzar_a_jugar(self):
        import escena_juego
        pilas.cambiar_escena(escena_juego.Juego())

    def mostrar_ayuda_del_juego(self):
        import escena_ayuda
        pilas.cambiar_escena(escena_ayuda.Ayuda())

    def salir_del_juego(self):
        pilas.terminar()

    def crear_asteroides(self):
        fuera_de_la_pantalla = [-600, -650, -700, -750, -800]
        import piedra_espacial
        for n in range(5):
```

```
x = random.choice(fuera_de_la_pantalla)
y = random.choice(fuera_de_la_pantalla)
piedra_espacial.PiedraEspacial([], x=x, y=y, tamano="chica")
```

Ahora que hemos entendido el mecanismo de las escenas, vemos que se define el tipo de escena **EscenaMenu** y que se en su método **iniciar()**, además de poner el fondo y de lanzar el aviso de que el menú también se puede controlar con el menú, se llama a las funciones **crear_titulo_del_juego()**, **crear_el_menu_principal()** y **crear_asteorides()**. Empecemos por el primero, que se encarga de mostrar el título del juego animado. ¿Cómo lo hace? Sencillo; se crea el actor con la imagen del título, que se almacena en la variable logotipo, y tras situarlo en la coordenada **y=300** (fuera de la pantalla, recuerda que la ventana de Pilas, por defecto, tiene **480** píxeles de alto y la coordenada **y=240** marca el borde superior) se genera una animación de movimiento usando la notación de listas:

```
logotipo.y = [200]
```

Lo anterior hace que **logotipo** se mueva desde su posición actual a la posición **y=200**, dentro de la pantalla y en la parte superior. ¡Conseguido!

La siguiente es la función **crear_el_menu_principal()**, que hace lo que indica su nombre de una manera sencilla, gracias al tipo de actor predefinido de Pilas **pilas.actores.Menu**:

```
def crear_el_menu_principal(self):
    opciones = [
            ("Comenzar a jugar", self.comenzar_a_jugar),
            ("Ver ayuda", self.mostrar_ayuda_del_juego),
            ("Salir", self.salir_del_juego)
           ]
    self.menu = pilas.actores.Menu(opciones, y=-50)
```

Como verás, además de indicarle la coordenada **y** donde queremos que aparezca, al crear el actor de tipo **pilas.actores.Menu** le pasamos como argumento la lista **opciones** que hemos definido justo antes. Esa lista de opciones consiste en varias tuplas en las que el primer elemento es el texto que se quiere mostrar y el segundo es la función que se ejecutará al seleccionarlo. Y cada una de esas funciones, simplemente importa la escena deseada y la invoca. ¡Habíamos dicho que era sencillo! Por ejemplo, la opción **"Ver ayuda"** hace que se ejecute el método **mostrar_ayuda_del_juego()**:

```
def mostrar_ayuda_del_juego(self):
    import escena_ayuda
    pilas.cambiar_escena(escena_ayuda.Ayuda())
```

¿Recuerdas **escena_ayuda.py** y la escena **Ayuda** del apartado anterior? Seguro que sí.

Pasemos ahora a la función **crear_asteroides()**, la última que necesitamos para darle vistosidad al menú del juego. Observa su código:

```
def crear_asteroides(self):
    fuera_de_la_pantalla = [-600, -650, -700, -750, -800]
    import piedra_espacial
    for n in range(5):
        x = random.choice(fuera_de_la_pantalla)
        y = random.choice(fuera_de_la_pantalla)
        piedra_espacial.PiedraEspacial([], x=x, y=y, tamano="chica")
```

De las líneas anteriores, deberías comprender que se generan **5** asteorides, llamando a su constructor **PiedraEspacial()** del módulo, importado previamente, **piedra_espacial**. Fíjate que su posición se elige al azar de entre la lista **fuera_de_la_pantalla** (que, en efecto, contiene coordenadas fuera de la ventana del juego) a través de la función **random.choice()**. Al constructor del asteroide se le pasa, también, el valor **"chica"** para el argumento **tamano**.

¿Cómo está implementada la clase **PiedraEspacial**? Me temo que tendrás que esperar un poco, pues ahora le toca al turno a la escena de juego. Pero ¡no desesperes! Vendrá enseguida... La escena **escena_juego** se apoya en unas cuantas clases y entre ellas estará nuestro asteroide...

escena_juego.py

Como puedes imaginarte, el meollo del juego reside aquí y, por ello, la complicación es un poco mayor. ¡Vamos allá, sin más preámbulos! ¡Abróchate el cinturón!

```
# -*- coding: utf-8 -*-

import pilas
import piedra_espacial
import random


class Estado:
    "Representa un estado dentro del juego."

    def actualizar(self):
        pass  # Tienes que sobrescribir este metodo...


class Jugando(Estado):
    "Representa el estado de juego."

    def __init__(self, juego, nivel):
        self.nivel = nivel
        self.juego = juego
        self.juego.crear_piedras(cantidad=nivel * 3)

        # Cada segundo le avisa al estado que cuente.
        pilas.mundo.agregar_tarea(1, self.actualizar)

    def actualizar(self):
        if self.juego.ha_eliminado_todas_las_piedras():
            self.juego.cambiar_estado(Iniciando(self.juego, self.nivel + 1))
            return False

        return True


class Iniciando(Estado):
    "Estado que indica que el juego ha comenzado."

    def __init__(self, juego, nivel):
        self.texto = pilas.actores.Texto("Iniciando el nivel %d" % (nivel))
        self.nivel = nivel
        self.texto.color = pilas.colores.blanco
        self.contador_de_segundos = 0
        self.juego = juego
```

```
        # Cada un segundo le avisa al estado que cuente.
        pilas.mundo.agregar_tarea(1, self.actualizar)

    def actualizar(self):
        self.contador_de_segundos += 1

        if self.contador_de_segundos > 2:
            self.juego.cambiar_estado(Jugando(self.juego, self.nivel))
            self.texto.eliminar()
            return False

        return True    # para indicarle al contador que siga trabajado.


class PierdeVida(Estado):

    def __init__(self, juego):
        self.contador_de_segundos = 0
        self.juego = juego

        if self.juego.contador_de_vidas.le_quedan_vidas():
            self.juego.contador_de_vidas.quitar_una_vida()
            pilas.mundo.agregar_tarea(1, self.actualizar)
        else:
            juego.cambiar_estado(PierdeTodoElJuego(juego))

    def actualizar(self):
        self.contador_de_segundos += 1

        if self.contador_de_segundos > 2:
            self.juego.crear_nave()
            return False

        return True


class PierdeTodoElJuego(Estado):

    def __init__(self, juego):
        # Muestra el mensaje "has perdido"
        mensaje = pilas.actores.Texto("Lo siento, has perdido!")
        mensaje.color = pilas.colores.blanco
        mensaje.abajo = 240
        mensaje.abajo = [-20]

    def actualizar(self):
        pass
```

```
class Juego(pilas.escena.Base):
    "Es la escena que permite controlar la nave y jugar"

    def __init__(self):
        pilas.escena.Base.__init__(self)

    def iniciar(self):
        pilas.fondos.Espacio()
        self.pulsa_tecla_escape.conectar(self.cuando_pulsa_tecla_escape)
        self.piedras = []
        self.crear_nave()
        self.crear_contador_de_vidas()
        self.cambiar_estado(Iniciando(self, 1))
        self.puntaje = pilas.actores.Puntaje(x=280, y=220,
                                             color=pilas.colores.blanco)

    def cambiar_estado(self, estado):
        self.estado = estado

    def crear_nave(self):
        nave = pilas.actores.Nave()
        nave.aprender(pilas.habilidades.SeMantieneEnPantalla)
        nave.definir_enemigos(self.piedras, self.cuando_explota_asteroide)
        self.colisiones.agregar(nave, self.piedras, self.explotar_y_terminar)

    def cuando_explota_asteroide(self):
        self.puntaje.aumentar(1)

    def crear_contador_de_vidas(self):
        import contador_de_vidas
        self.contador_de_vidas = contador_de_vidas.ContadorDeVidas(3)

    def cuando_pulsa_tecla_escape(self, *k, **kw):
        "Regresa al menu principal."
        import escena_menu
        pilas.cambiar_escena(escena_menu.EscenaMenu())

    def explotar_y_terminar(self, nave, piedra):
        "Responde a la colision entre la nave y una piedra."
        nave.eliminar()

        self.cambiar_estado(PierdeVida(self))

    def crear_piedras(self, cantidad):
        "Genera una cantidad especifica de naves en el escenario."
        fuera_de_la_pantalla = [-600, -650, -700, -750, -800]
        tamanos = ['grande', 'media', 'chica']
```

```
    for x in range(cantidad):
        x = random.choice(fuera_de_la_pantalla)
        y = random.choice(fuera_de_la_pantalla)
        t = random.choice(tamanos)

        piedra_nueva = piedra_espacial.PiedraEspacial(self.piedras, x=x,
                                                      y=y, tamano=t)
        self.piedras.append(piedra_nueva)

def ha_eliminado_todas_las_piedras(self):
    return len(self.piedras) == 0
```

¡No te asustes! Hay mucho que hacer y mucho que aprender. Pero, paso a paso, todo es llevadero y puede comprenderse completamente...

Un primer punto importante es apreciar que, dentro del juego, el jugador puede estar en diferentes **estados**; al comienzo, cuando aparece su nave y todavía no se ha empezado a jugar, cuando se está jugando, cuando te destruyen... Piensa en esos estados como si fueran mini-escenas dentro de la escena de juego, en las que se tienen que ajustar cosas cuando ocurran determinadas circustancias. ¿Cómo controlar esto? Una tentativa interesante para hacerlo, inicialmente de manera abstracta, no concretada, es definir una clase de objetos que contengan esa información. Vamos a llamar a esa clase, en un alarde de originalidad, **Estado**. Recuerda que Pilas llama al método **actualizar()** de cada objeto en cada fotograma, así que vamos a implementarlo también:

```
class Estado:
    "Representa un estado dentro del juego."

    def actualizar(self):
        pass  # Tienes que sobrescribir este metodo...
```

Más adelante, para concretar dichos estados, solo tendremos que especificar la implememtación del método **actualizar()** y, si lo deseamos, usar el método **__init__()** para añadir alguna característica más en la que apoyarse.

¿Cómo cambiar el estado del jugador? Definamos una función para eso. Si te fijas, en el código de la escena, la función es **cambiar_estado()**

```
    def cambiar_estado(self, estado):
        self.estado = estado
```

¿Lo entiendes? La **escena** tiene un atributo, **self.estado**, que podemos modificar cuando queramos llamando al método **cambiar_estado()** con el estado que deseemos como argumento. Muy bien, sólo queda, en este sentido, definir los diferentes estados que queremos contemplar. Hemos elegido los siguientes:

- **Iniciando**. Nos servirá para ajustar los datos iniciales de cada nivel e implementar esas pequeñas cosas que ocurren al comienzo, justo antes de que el control pase a manos del jugador.
- **Jugando**. Aquí tendremos codificado el estado general del jugador durante el juego. Nos servirá para indicar el nivel en el que se encuentra (al crearse) y cambiar de nivel cuando se eliminen todos los asteroides (¡Sí! Los diferentes niveles dentro de la misma escena los podemos gestionar con los estados).
- **PierdeVida**. Hace lo propio :-). Y además mira, como es lógico, si se han perdido todas las vidas para acabar el juego.
- **PierdeTodoElJuego**. Aquí realizaremos las tereas necesarias cuando el jugador acabe con todas sus vidas.

¡Muy bien! ¡A implementar el código! Empecemos analizando el estado **Iniciando**:

```
class Iniciando(Estado):
    "Estado que indica que el juego ha comenzado."

    def __init__(self, juego, nivel):
        self.texto = pilas.actores.Texto("Iniciando el nivel %d" % (nivel))
        self.nivel = nivel
        self.texto.color = pilas.colores.blanco
        self.contador_de_segundos = 0
        self.juego = juego

        # Cada un segundo le avisa al estado que cuente.
        pilas.mundo.agregar_tarea(1, self.actualizar)

    def actualizar(self):
        self.contador_de_segundos += 1

        if self.contador_de_segundos > 2:
            self.juego.cambiar_estado(Jugando(self.juego, self.nivel))
            self.texto.eliminar()
            return False

        return True    # para indicarle al contador que siga trabajado.
```

Observa su método **__init__()**. Cuando creemos el estado **Iniciando**, lo inicializaremos con dos argumentos, **juego** y **nivel**. El primero de ellos, **juego**, contendrá una referencia a la escena actual (con el fin de poder acceder a sus objetos) y, el segundo, obviamente, el **nivel** en el que se desea iniciar. Obsérvalo en la siguiente línea, extraida del método **iniciar()** de la escena:

```
        self.cambiar_estado(Iniciando(self, 1))
```

El número de nivel se muestra con un actor de tipo Texto, en color blanco. Y, a continuación se pone **contador_de_segundos** a **0**. ¿Por qué se necesita este contador? ¿Recuerdas que hemos dicho que, en muchos juegos, habrás visto que al iniciar cada nivel hay unos segundos en los que se muestran datos, sin que el jugador pueda interaccionar, y luego se le pasa el control? ¡Aquí está el quid!. Fíjate bien. Con la siguiente línea,

```
        pilas.mundo.agregar_tarea(1, self.actualizar)
```

conseguimos que se ejecute el método **actualizar()** del estado cada segundo. Y en dicho método, aumentamos el contador en **1** y si han pasado más de **2** segundos se cambia el estado a **Jugando** (en cuyo caso hay que eliminar el texto y devolver **False** para que no se vuelva a repetir la tarea). Es más difícil de explicar que de entender; a estas alturas, esperamos sinceramente que intuyas el proceso sin mayores dificultades.

Pasemos al estado **Jugando**:

```
class Jugando(Estado):
    "Representa el estado de juego."

    def __init__(self, juego, nivel):
        self.nivel = nivel
        self.juego = juego
        self.juego.crear_piedras(cantidad=nivel * 3)

        # Cada segundo le avisa al estado que cuente.
        pilas.mundo.agregar_tarea(1, self.actualizar)

    def actualizar(self):
        if self.juego.ha_eliminado_todas_las_piedras():
            self.juego.cambiar_estado(Iniciando(self.juego, self.nivel + 1))
            return False

        return True
```

Como es de imaginar, en el método **__init__()** debemos crear los enemigos con los que el jugador va a tener que enfrentarse. También, lógicamente, si el nivel es mayor, mayor dificultad hemos de tener y eso se traduce en un mayor número de enemigos. Eso es lo que conseguimos con el método **crear_piedras()** de la escena que veremos más adelante.

A continuación y, del mismo modo que antes, se agrega la tarea de ejecutar el método **actualizar()** cada segundo. En él, y usando otro método de la escena, se mira si se han eliminado todos los asteorides. En caso afirmativo, se da por terminada la tarea y se aumenta de nivel.

¡Esto va sobre ruedas! El turno ahora es para el estado **PierdeVida**, ya sin mostrar explícitamente su código (puedes verlo más arriba). Léelo con atención y síguelo como si fuese una receta. Para cocinar el plato, verás que, análogamente al inicio de nivel, se dedican **2** segundos al proceso de perder una vida (observa el **contador_de_segundos**). Allí, en diferentes lugares, verás cómo se crea la nave del jugador y se mira previamente, tras eliminar la que se ha perdido, si quedan todavía disponibles. Todo ello con diferentes funciones que veremos enseguida. En caso de que se hayan agotado, se cambia el estado a **PierdeTodoElJuego**.

Y vamos a este último estado. Éste es un estado terminal, pues se ha acabado el juego al ser derrotado el jugador. Ello hace que no nos molestemos en implementar el método **actualizar()**; simplemente mostramos un texto de notificación de la derrota y lo movemos con una animación simple.

¡Ya estamos en condiciones de analizar la implementación de la escena propiamente dicha!

Primero veamos, a vista de pájaro, los diferentes métodos que la componen:

```
class Juego(pilas.escena.Base):
    "Es la escena que permite controlar la nave y jugar"

    def __init__(self):
        ...

    def iniciar(self):
        ...

    def cambiar_estado(self, estado):
        ...

    def crear_nave(self):
        ...

    def cuando_explota_asteroide(self):
        ...

    def crear_contador_de_vidas(self):
        ...

    def cuando_pulsa_tecla_escape(self, *k, **kw):
        ...

    def explotar_y_terminar(self, nave, piedra):
        ...

    def crear_piedras(self, cantidad):
        ...

    def ha_eliminado_todas_las_piedras(self):
        ...
```

Bien. Sin perder de vista su código, vamos método a método a ver qué hacen:

- **__init__()**
  Simplemente llama al método correspondiente de la clase padre.
- **iniciar()**
  Se encarga de inicializar todas las variables del juego, además de colocar el fondo y conectar la pulsación de la tecla **escape** con el método que se encarga de gestionarlo; crea la lista (inicialmente vacía) de asteroides, que denominamos **piedras**, crea el marcador, crea la nave del jugador y el contador de vidas con otros tantos métodos (ver más abajo) y cambia el estado a **Iniciando**.
- **cambiar_estado()**
  Ya hemos hablado antes de este método, que hace lo que indica su nombre actualizando el atributo **estado** de la escena al valor que se le proporciona.
- **crear_nave()**
  Crea la nave del jugador usando un actor predefinido de Pilas, impidiendo que se salga de la pantalla. Además, define **piedras** como los enemigos e indica qué función ejecutar cuando se impacte en uno de ellos (justo la siguiente). Finalmente se añade la colisión entre la nave y los asteroides que supondrán la pérdida de una vida.

- **cuando_explota_asteroide()**
  Como es lógico, lo que se hace es aumentar en **1** el valor del marcador.
- **crear_contador_de_vidas()**
  Como el contador de vidas es más complicado (y más dado a ser modificado en el futuro), su definición la tenemos encapsulada en otro módulo; de esta manera, aquí lo cargamos en memoria con el **import** correspondiente y se utiliza un méodo de dicho módulo para establecer que son **3** las vidas de las que dispone el jugador.
- **cuando_pulsa_tecla_escape()**
  Aquí lo que queremos es que se vuelva al menú principal y esto es lo que se realiza importando la escena correspondiente y llamando a **cambiar_escena()**.
- **explotar_y_terminar()**
  Éste es el método que se lanza cuando un asteroide colisiona con la nave del jugador. ¿Qué es lo que hacemos? Venga, que lo imaginas... Sí; eliminamos el actor **nave** que ha sido alcanzado y cambiamos el estado al de **PierdeVida**. El nombre del método hace alusión a que, de paso, también se mira si el juego ha acabado, ya que está implementado en el estado anterior.
- **crear_piedras()**
  También hemos aludido antes a este método. Lo que hace es crear tantos asteroides como indica el argumento que se le pasa. La manera de hacerlo ya nos es familiar, pues lo vimos en la escena **EscenaMenu**. El matiz es que no solo usamos una posición aleatoria para cada uno de los asteroides, si no que su tamaño también lo seleccionamos al azar (de entre los que se definen en la implementación de **PiedraEspacial**).
- **ha_eliminado_todas_las_piedras()**
  Y llega el último método, que devuelve **True** o **False** en función de que se haya llegado no a eliminar todos los asteroides (lo que se hace mirando si la longitud de la lista que los contiene es **0**).

¡Ufff....!

Respira, respira hondo.

Son muchas cosas pero, si has seguido el hilo del razonamiento, verás que en el fondo son todo tareas muy lógicas. Simplemente nos tenemos que hacer con la manera de trabajar del **engine** y estructurar nuestro pensamiento en función de los objetivos que nos marquemos.

Con esto completamos el análisis de las diferentes escenas y pasamos a completar los flecos, es decir, aquellos elementos cuyo código lo hemos modularizado en archivos a parte. Son, como vimos al principio de este tutorial, dos: **contador_de_vidas.py** y **piedra_espacial.py**.

Vamos con el primero.

contador_de_vidas.py

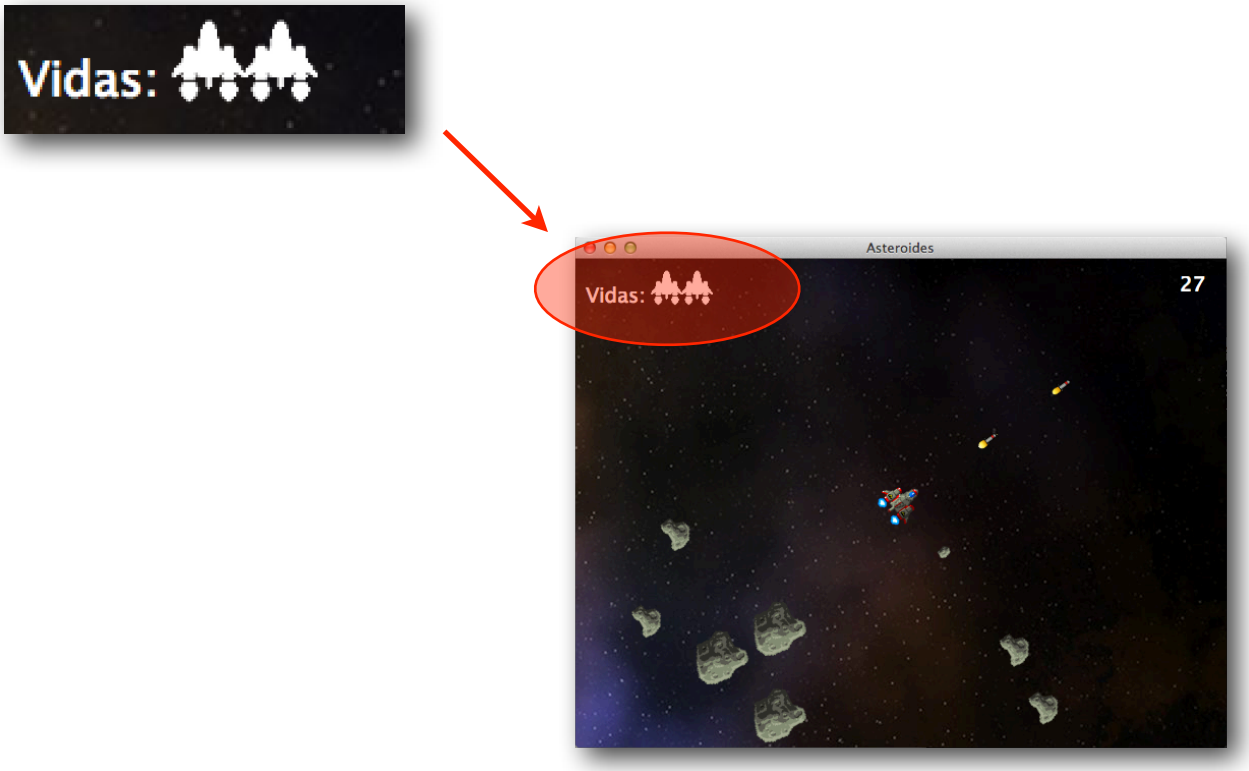
Tenemos que definir aquí el aspecto y el comportamiento del marcador que muestra las vidas de las que dispone el jugador. De paso, veremos ciertas características de **Python** que no habíamos tenido ocasión, previamente, de comentar.

```
# -*- coding: utf-8 -*-

import pilas


class ContadorDeVidas:

    def __init__(self, cantidad_de_vidas):
        self.crear_texto()
        self.cantidad_de_vidas = cantidad_de_vidas
        self.vidas = [pilas.actores.Actor("data/vida.png") for x
                                    in range(cantidad_de_vidas)]

        for indice, vida in enumerate(self.vidas):
            vida.x = -230 + indice * 30
            vida.arriba = 230

    def crear_texto(self):
        "Genera el texto que dice 'vidas'"
        self.texto = pilas.actores.Texto("Vidas:")
        self.texto.color = pilas.colores.blanco
        self.texto.magnitud = 20
        self.texto.izquierda = -310
        self.texto.arriba = 220

    def le_quedan_vidas(self):
        return self.cantidad_de_vidas > 0

    def quitar_una_vida(self):
        self.cantidad_de_vidas -= 1
        vida = self.vidas.pop()
        vida.eliminar()
```

Lo único que hace el módulo es definir la clase de objeto **ContadorDeVidas**. Los ajustes se realizan, como siempre, en su método **__init__()** y se apoya en la definición de otros tres métodos, **crear_texto()**, **le_quedan_vidas()** y **quitar_una_vida()**.

De los tres, el más sencillo es **le_quedan_vidas()**, pues devuelve simplemente un valor **booleano**: **True** si el jugador tiene vidas disponibles (mirando el valor del atributo **cantidad_de_vidas**) y **False** en caso contrario.

Tampoco le anda a la zaga **crear_texto()**, pues simplemente crea el texto del contador de vidas, de color **pilas.colores.blanco** y tamaño **20**, y lo coloca en la posición adecuada.



Sin embargo, para entender **quitar_una_vida()** debemos comprender **__init__()**:

```
def __init__(self, cantidad_de_vidas):
    self.crear_texto()
    self.cantidad_de_vidas = cantidad_de_vidas
    self.vidas = [pilas.actores.Actor("data/vida.png") for x
                                    in range(cantidad_de_vidas)]

    for indice, vida in enumerate(self.vidas):
        vida.x = -230 + indice * 30
        vida.arriba = 230
```

Te llamará la atención la manera en la se crea la lista de imágenes de las vidas de las que dispone el jugador. La técnica se conoce como **listas por comprensión** (mira el cuadro de la próxima página).

```
self.vidas = [pilas.actores.Actor("data/vida.png") for x in range(cantidad_de_vidas)]
```

Una vez creada esa lista de imágenes de las naves, **vidas**, para colocarlas en las posiciones adecuadas usamos un bucle **for** y la función **enumerate()**, de manera que para cada una de ellas, su coordenada **x** se desplace **30** píxeles a la derecha.

Listas por Comprensión

Se trata de una técnica que permite construir listas de elementos de una manera compacta, pudiendo incluso añadir condiciones que han de cumplir. De forma genérica, son de la forma

```
[ ① for ② in ③ if ④ ]
```

y generan una lista con elementos calculados con ①, en donde los parámetros que se usan, indicados en ②, se eligen de entre los de ③ siempre que cumplan la condición ④.

Por ejemplo, el resultado de evaluar la expresión

```
[ x*x for x in range(10) if x % 2 == 1 ]
```

sería

```
[1, 9, 25, 49, 81]
```

Consulta la documentación de **Python** para más información..

Ahora ya podemos terminar con el método **quitar_una_vida()**. Allí no solo disminuimos en **1** a **cantidad_de_vidas**, si no que usamos el método **pop()** de las listas, el cual nos extrae su último elemento y nos lo devuelve, listo para que usemos **eliminar()** con el objetivo de borrarlo de la memoria.

¡Listo!

Nos queda la última modularización...

piedra_espacial.py

Te gustan nuestros asteroides, ¿eh?. Al dispararles y dar en el blanco, se dividen en trozos más pequeños...

La forma de implementarlo es muy ilustrativa, así que no lo demoremos más:

```
# -*- coding: utf-8 -*-

import pilas
import random


class PiedraEspacial(pilas.actores.Piedra):
    "Representa una piedra espacial que se puede romper con un disparo."

    def __init__(self, piedras, x=0, y=0, tamano="grande"):
        # Obtiene una velocidad de movimiento aleatoria.
        posibles_velocidades = range(-10, -2) + range(2, 10)

        dx = random.choice(posibles_velocidades) / 10.0
        dy = random.choice(posibles_velocidades) / 10.0

        pilas.actores.Piedra.__init__(self, x=x, y=y,
                                      dx=dx, dy=dy, tamano=tamano)
        self.tamano = tamano
        self.piedras = piedras

    def eliminar(self):
        "Este metodo se invoca cuando el disparo colisiona con la nave."
        pilas.actores.Explosion(self.x, self.y)
        pilas.actores.Piedra.eliminar(self)

        if self.tamano == "grande":
            self.crear_dos_piedras_mas_pequenas(self.x, self.y, "media")
```

```
    elif self.tamano == "media":
        self.crear_dos_piedras_mas_pequenas(self.x, self.y, "chica")

def crear_dos_piedras_mas_pequenas(self, x, y, tamano):
    "Genera dos piedras mas chicas, enemigas de la nave."
    piedra_1 = PiedraEspacial(self.piedras, x=x, y=y, tamano=tamano)
    piedra_2 = PiedraEspacial(self.piedras, x=x, y=y, tamano=tamano)
    self.piedras.append(piedra_1)
    self.piedras.append(piedra_2)
```

Los argumentos del método **__init__()** de la clase **PiedraEspacial** son la posición en la que se va a crear la correspondiente instancia, el tamaño y una lista que representa al grupo de asteroides al que va a pertenecer. Fíjate, también, que en el código se usa otra lista para el abanico posible de velocidades (excluyendo que quede parado) y que se elige al azar de entre ellas. Todo ello, excepto **piedras**, se usa al invocar a la clase padre, **pilas.actores.Piedra**.

¿Qué tiene de especial nuestro asteroide, entonces? Lo que marca la diferencia es la forma en la que es eliminado. Observa:

```
def eliminar(self):
    "Este metodo se invoca cuando el disparo colisiona con la nave."
    pilas.actores.Explosion(self.x, self.y)
    pilas.actores.Piedra.eliminar(self)

    if self.tamano == "grande":
        self.crear_dos_piedras_mas_pequenas(self.x, self.y, "media")
    elif self.tamano == "media":
        self.crear_dos_piedras_mas_pequenas(self.x, self.y, "chica")
```

En efecto, además de crear una explosión y lanzar el método **eliminar()** de la clase padre, se crean dos piedras de tamaño más pequeño que la piedra original con la ayuda del método **crear_dos_piedras_mas_pequenas()** que tenemos definido más abajo. En ese método, por supuesto, las nuevas piedras son añadida a la lista **piedras**, con el objetivo de poder capturar más adelante las posibles colisiones con el jugador... Todo lo demás debería resultarte, ya, evidente.

Estupendo, estupendo... Todo lo que se refiere al código específico del juego ya está terminado. Solo nos quedan un par de archivos que tienen una misión más organizativa y clarificadora y que han sido nombrados al comienzo de este tutorial: **ejecutar.py** y **asteroides.py**.

asteroides.py y ejecutar.py

Entre tanto módulo y archivo, necesitaríamos clarificar cómo arrancamos el juego desde el principio. Nada mas sencillo. Éste es el turno de **ejecutar.py**:

```
# -*- coding: utf-8 -*-

import pilas

pilas.iniciar(titulo="Asteroides")

# Inicia la escena actual.
import escena_menu
pilas.cambiar_escena(escena_menu.EscenaMenu())
pilas.ejecutar()
```

Como ves, algo muy simple, al haber planificado bien la modularización de todo el material. Lo que hemos de hacer, simplemente, es inicializar Pilas, cargar la escena del menú, cambiar a ella y lanzar el engine.

Pero también, como indicamos al principio, hay que facilitarle al usuario la labor de encontrar el archivo sobre el que hacer doble click para jugar a nuestro videojuego. En ese sentido, en la raíz de la carpeta que lo contiene, encontramos a **asteroides.py**:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import os
import sys

directorio_actual = os.path.dirname(os.path.abspath(__file__))
os.chdir(os.path.join(directorio_actual, 'asteroides'))
sys.path.insert(0, os.path.join(directorio_actual, 'asteroides'))

import ejecutar
```

Esta tarea es más específica, por que fíjate que lo que hemos de decirle a **Python** (¿recuerdas la estructura de carpetas y archivos que hemos planteado para los materiales que componen el juego?) es que entre en la carpeta **asteroides** e importe nuestro archivo **ejecutar.py**, todo ello conservando las rutas de los archivos para que, al cargar los diferentes módulos, Pilas los encuentre. Para ello, nos hemos de apoyar en dos módulos fundamentales que permiten realizar la tarea independientemente del sistema operativo (Linux, Mac OS X, Windows) que usemos, **os** y **sys**.

Vamos línea por línea:

```
directorio_actual = os.path.dirname(os.path.abspath(__file__))
```

Aquí, **__file__** es un objeto predefinido en Python que contiene una referencia al propio archivo que se está ejecutando. Y **os.path.abspath()** es una función que nos devuelve la ruta absoluta al archivo que se le pasa. Dando un paso más, **os.path.dirname()** extrae el nombre de la carpeta que contiene al archivo indicado, con lo que al final de todo el proceso, la variable **directorio_actual** contendrá una referencia a la carpeta en la que está nuestro **asteroides.py**.

```
os.chdir(os.path.join(directorio_actual, 'asteroides'))
```

Una vez que sabemos dónde estamos, hemos de decirle a Python que cambie el directorio de trabajo a donde tenemos todo el material, es decir, a la carpeta **asteroides**. Para eso, primero usamos la función **os.path.join()** que de forma inteligente une la ruta de **directorio_actual** con el nombre de la carpeta a la que queremos cambiar para tener la ruta completa construida correctamente. Éste es un detalle importante, pues en Linux y Mac OS X el separador entre carpetas es "/" mientras que en Windows es "\" y Python, de la forma anterior, se encarga automáticamente de comprobarlo. Una vez conocida la ruta, la función **os.chdir()** nos cambia a la carpeta indicada.

```
sys.path.insert(0, os.path.join(directorio_actual, 'asteroides'))
```

Finalmente, una vez que tenemos el directorio fijado (lo que es necesario para que Pilas encuentre las imágenes y demás recursos que podamos usar en el juego), también tenemos que indicar a Python dónde debe mirar para ejecutar el código. El truco es usar la función **sys.path.insert()**, la cual inserta un directorio nuevo como primer elemento de la lista que el sistema operativo usa para buscar ejecutables.

¡Listo! Al importar, en la última línea, nuestro módulo **ejecutar.py** se lanzará el juego :-)

Bien, ya tienes un horizonte extendido para poder expresar esa creatividad que puja dentro de ti por salir... Adapta a tu imaginación los diferentes elementos para hacer tu propio juego, conservando este tipo de estructura de archivos, bien ordenada y que te permitirá organizarte mejor.

¡Hay un artista dentro en ti!