Introducción a Python y a la programación gráfica con Python Arcade

Héctor Costa Guzmán (hektorprofe.net)

Índice

Índice **2**
Preparando el terreno **5**
Presentación personal 5
Explicación del temario 5
Entorno de aprendizaje 5
Preguntas frecuentes 6
Función de impresión **7**
Programas de texto 7
Funciones 7
Ejecución en la terminal 7
Ejecución en PyCharm 9
Errores y avisos 10
Impresión múltiple 13
Códigos de escape 13
Resumen del tema 14
Experimentos gráficos **15**
Por qué Python Arcade 15
Comentarios de código 15
Módulos estándar y públicos 16
Importando un módulo 16
Generando la ventana 17
Colores predefinidos 18
Proceso de renderizado 19
Colores RGB 20
Bits y bytes 23
Coordenadas cartesianas 25
Dibujando rectángulos 25
El algoritmo del pintor 27
Dibujando otras formas 28
Leyendo documentación 31
Resumen del tema 32
Lenguajes de programación **33**
Cuestiones previas 33
CPU: Unidad Central de Procesamiento 33
GPU: Unidad de Procesamiento Gráfico 33
Lenguajes de programación 34
El lenguaje Python 35
Resumen del tema 36
Variables y expresiones **37**
La información es poder 37

Datos literales 37
Memoria y variables 39
Nombres de variables 41
Operadores y expresiones 42
Precedencia y asociación 43
Operadores en asignación 44
Expresiones en funciones 45
Impresión de variables 46
Resumen del tema 47
Funciones de código **48**
Utilidad de las funciones 48
Funciones simples 48
Función principal 50
Parámetros de entrada 51
Ámbito de los parámetros 52
Parámetros por defecto 52
Parámetros inmutables 53
Valores de retorno 55
Anotaciones de tipado 56
Funciones estándar 57
Función de lectura 57
Dibujos con funciones 58
Animaciones con funciones 61
Animaciones con ChatGPT 66
Resumen del tema 70
Fundamentos lógicos **71**
El tipo lógico 71
Operadores relacionales 72
Operadores lógicos 73
Resumen del tema 74
Condiciones **75**
Controlar el flujo 75
Bloques condicionales 75
Diagramas de flujo 76
Instrucción else 77
Instrucción elif 78
Instrucción match-case 80
Condiciones anidadas 80
Animaciones y condiciones 81
Resumen del tema 84
Bucle Mientras **86**
Potencial ilimitado 86

Mientras infinito 86
Mientras condicionado 87
Mientras contador 87
Mientras continuar 88
Resumen del tema 88
Bucle For **89**
Para en una colección 89
Para en un rango 89
Para continuar 91
Paras anidados 92
Dibujando tablas 94
Dibujando un tablero 97
Resumen del tema 104
Errores y excepciones **105**
Errar es humano 105
Manejando excepciones 105
Múltiples excepciones 106
Invocando una excepción 107
Resumen del tema 107
Tuplas **108**
Definición 108
Índices e inmutabilidad 109
Técnica del slicing 109
Tuplas anidadas 110
Acceso secuencial 112
Métodos internos 113
Valor en colección 113
Resumen del tema 114
Listas **115**
Definición 115
Añadir elementos 116
Borrar elementos 117
Modificar elementos 118
Función numeradora 119
Mutabilidad y copia 120
Repaso de los métodos 122
Comprensión de listas 122
Cuadrícula animada 123
Resumen del tema 128

Preparando el terreno

Presentación personal

Me llamo Héctor, soy de Barcelona (España) y seré vuestro instructor a lo largo de esta aventura. Empecé a programar hace 17 años y a parte de mi trabajo también es una de mis aficiones. Me atrae mucho la parte de la programación enfocada a los gráficos, muy especialmente en los videojuegos. Una de las cosas que más me satisface es analizar cómo funcionan los videojuegos de mi infancia e intentar recrear sus mecánicas. Desde pequeño soy muy curioso y me gusta saber cómo funcionan las cosas, por eso aprender y entender la magia detrás de los videojuegos me resulta muy gratificante.

Cuando conocí Python en 2008 este lenguaje distaba mucho de ser tan utilizado como lo es hoy. Según el índice TIOBE actualmente se encuentra en el top, pero en aquel momento no solo era poco utilizado sino también vilipendiado y tachado de poco eficiente. Pese a las críticas decidí aprenderlo y eso me ha llevado a donde estoy, porque la realidad es que Python es un lenguaje excelente para aprender a programar, su sintaxis sencilla permite enfocarse en la lógica de programación. No es necesario invertir horas para entender la estructura de un programa, solo escribir el código y ejecutarlo.

Espero que aprendáis muchísimo y sobretodo que disfrutéis descubriendo la magia detrás de la alquimia de los videojuegos.

Explicación del temario

Python desde CERO programando videojuegos es un curso práctico donde aprenderemos el ABCD de la programación. Estos conocimientos son los pilares que sustentan todo:

1. Introducción a la programación.
2. Control de flujo.
3. Colecciones de datos.
4. Programación orientada a objetos.
5. Módulos y ficheros.

Aparte de introducir todo tipo de conceptos tecnológicos para programadores, algunas lecciones intercalan experimentos gráficos haciendo uso del módulo *Python Arcade*. Esta biblioteca nos acompañará durante todo el curso y no solo apoyará nuestro aprendizaje sino que nos servirá para programar videojuegos.

Entorno de aprendizaje

Para programar en Python es necesario instalar el intérprete de Python y contar con un editor de código. Más adelante hablaremos sobre los lenguajes de programación y qué significa un lenguaje interpretado. Por ahora voy a indicar el minuto exacto del video donde os explico como instalar Python en Windows, MAC y GNU/Linux porque el proceso es un poco distinto:

- Windows: https://api.arcade.academy/en/latest/install/windows.html *[+extensiones]*
- Mac: https://api.arcade.academy/en/latest/install/mac.html
- Linux: https://api.arcade.academy/en/latest/install/linux.html

Instalaremos Python Arcade: *python -m pip install arcade* o *python3 -m pip install arcade*

Crearemos un directorio para trabajar a lo largo del curso desde *PyCharm*: *python-mola* *(explicar que significa mola por si alguien no lo sabe)*.

Instalaremos PyCharm Community Edition.

Preguntas frecuentes

Función de impresión

Programas de texto

En esta lección vamos a escribir nuestro primer programa gracias a la herramienta más fundamental del programador, la función de impresión (*print function*). Su utilidad es, como puedes suponer, imprimir texto en pantalla.

Aunque hoy en día muchos programas, por no decir la gran mayoría, se basan en gráficos, todavía existen muchos programas textuales (de texto). Suelen utilizarlos los administradores de computadoras y analistas de datos. Y no solo programas, también existen los videojuegos basados en texto. De hecho los primeros videojuegos eran precisamente eso, aventuras de texto, resaltando entre ellas el famoso género *MUD*, del inglés *Multi-User Dungeon*, juegos RPG multijugador textuales, precursores de los actuales MMORPG. Desde los principios de la programación millones de personas han trabajado y se han divertido de lo lindo gracias a la función *print*.

La parte interesante de imprimir es que no es algo limitado a una pantalla, es posible imprimir en ficheros para guardar contenido en el disco duro e incluso hacerlo a través de una red, siendo esto la semilla del funcionamiento del Internet moderno.

Funciones

Así que vamos a utilizar una función para imprimir texto en la pantalla. ¿Pero qué es una función? Cada vez que utilizamos la calculadora ejecutamos funciones. Todos esos botones como seno, coseno, tangente, logaritmo, potencia… ¿Recuerdas que antes de presionarlos es necesario escribir un valor? El valor que escribimos antes de presionar el botón se denomina ***valor de entrada*** o en programación ***argumento de entrada***. Lo que aparece después de presionarlo es el ***valor de salida*** o en programación ***valor de retorno***. Lo que sucede entre medio son diferentes operaciones que transforman la entrada en la salida, a eso se le denomina función.

La función ***print*** es lo mismo pero aplicado a la programación, junto con muchas otras funciones se nos provee para utilizarla sin necesidad de saber cómo funciona internamente. Simplemente toma una entrada como un texto, y devuelve una salida, que en este caso se basa en imprimir ese texto en la pantalla. La sintaxis es simple, el nombre de la función seguido de un paréntesis con los argumentos de entrada en su interior. Para indicar que el valor de entrada es un texto, en Python solemos escribirlo entre comillas dobles:

```
print("Hola mundo")
```

En la programación hay muchos tipos de información, se conocen como datos y tienen sus propios tipos. *“Hola mundo”* es una cadena de caracteres y no se escribe igual que un número o un valor booleano. Aprenderemos poco a poco sobre esos tipos de datos, por ahora veamos como decirle a Python que ejecute este código.

Ejecución en la terminal

Hay varias formas de ejecutar código, la más directa es accediendo al modo interactivo de Python ejecutándolo directamente en una terminal: *python*

El modo interactivo es un entorno donde se ejecuta Python, sabemos que estamos en él porque al principio de una línea siempre se mostrarán tres flechas `>>>` indicando que se espera alguna función:

```
>>> print("Hola mundo")
Hola mundo
>>>
```

Al ejecutarlo tenemos el resultado e inmediatamente después otras tres flechas esperando para ejecutar otra función. Como podéis suponer, programar de esta forma no es muy cómodo. Llamemos a la función *exit()* para abandonar el modo interactivo. En este caso es una función que no recibe argumentos de entrada, solo ejecuta una única tarea que es salir del programa actual.

```
>>> exit()
```

La verdadera forma de ejecutar código Python es mediante ***Scripts***, del inglés *manuscript*, y que son ficheros con código fuente. Si creamos un fichero vacío con algun editor de texto como *Notepad*, *TextEdit* o el que tengáis en vuestro sistema y le escribimos una línea con la misma función, podemos ejecutarlo si le pasamos la ruta al ejecutable de Python (podemos arrastrarlo a la terminal):

```
C:\Users\hcost>python C:\Users\hcost\Desktop\hola.txt
Hola mundo
```

Fijaros como independientemente de la extensión del fichero se ha ejecutado la función sin ningún problema. El problema es que tener cualquier extensión para un código específico no es práctico, por eso en Python a los scripts se les pone la extensión *.py*. Eso permite además que el propio Python descubra otros ficheros *.py* en el sistema y pueda cargarlos en la memoria como módulos, aunque de esto hablaremos más adelante.

Por tanto la forma correcta de ejecutar nuestro script sería préviamente cambiándole la extensión:

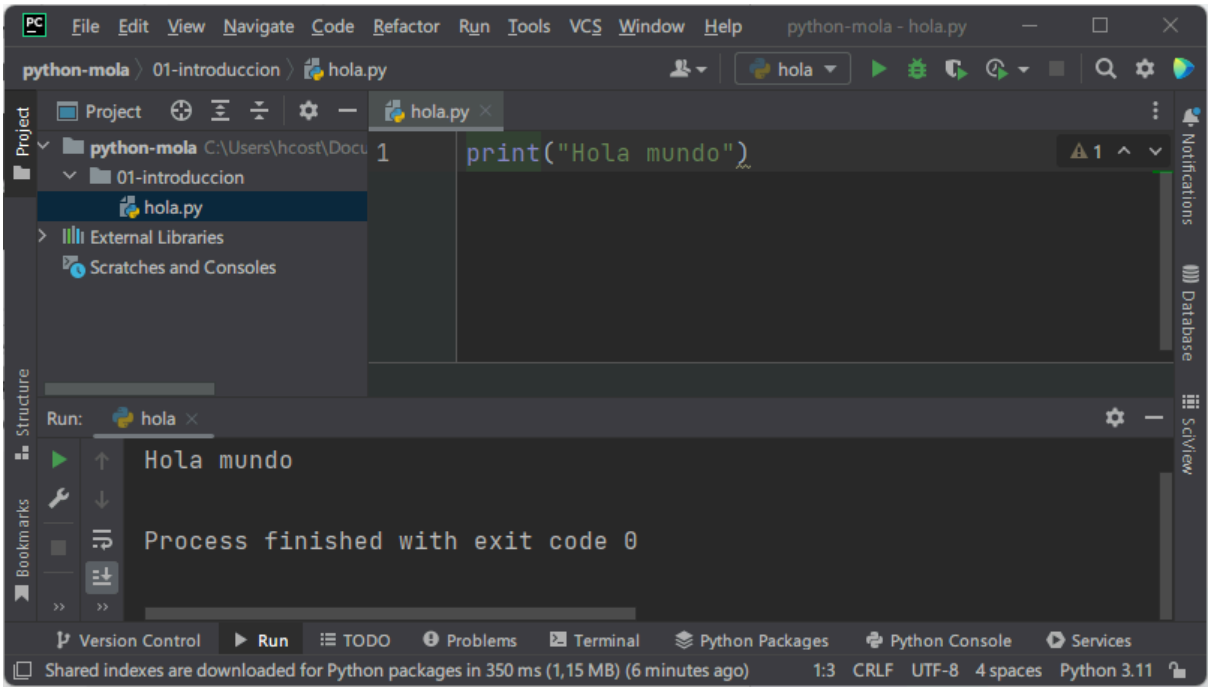
```
C:\Users\hcost>python C:\Users\hcost\Desktop\hola.py
Hola mundo
```

Si tenéis algunos conocimientos del manejo de vuestro sistema, sabréis que en la parte delantera de la terminal se indica una ruta. Esa es la ruta del directorio actual, podemos navegar entre directorios escribiendo el comando *cd* y el nombre de otra carpeta (se puede autocompletar con tabulador). También podemos movernos al directorio anterior con dos puntos *cd ..* Esto es interesante porque si nos ubicamos en el directorio que contiene un script, no hace falta pasarle toda la ruta:

```
C:\Users\hcost>cd Desktop
C:\Users\hcost\Desktop>python hola.py
Hola mundo
```

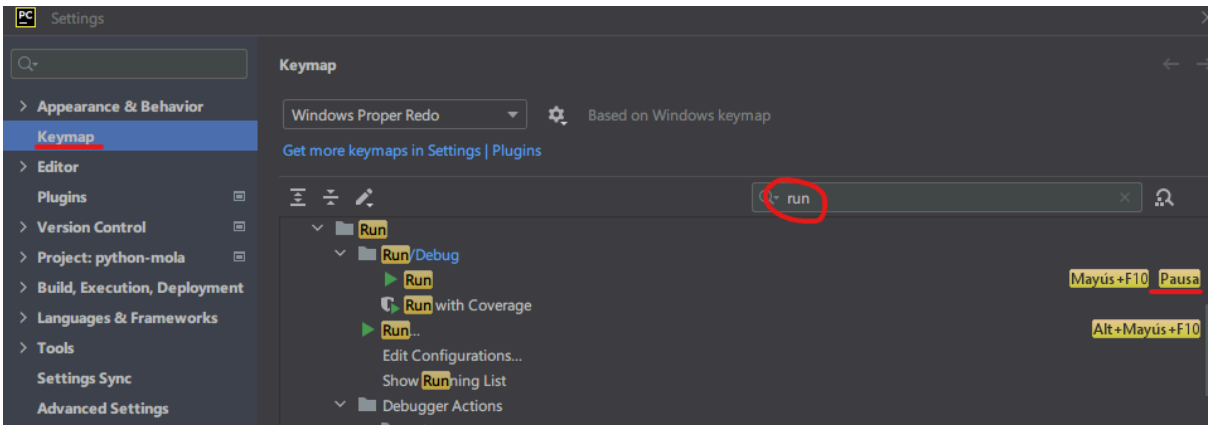
Ejecución en PyCharm

Ejecutar scripts en la terminal puede ser un salvavidas si no tenemos entorno gráfico, pero por suerte para nosotros hemos instalado *PyCharm*, así que mejor abrir el directorio *python-mola*, crear una carpeta *01-introduccion*, mover a ella el script *01-hola.py* y abrirlo en PyCharm. Ahora para ejecutarlo hacemos clic derecho sobre el fichero y buscamos la opción ***"Run 01-hola.py"***:



Al ejecutar un script en PyCharm el editor guarda los cambios automáticamente, pero no todos los editores lo hacen. Si utilizáis otro editor recordad guardar siempre las modificaciones antes de volver a ejecutar el script. En cualquier caso el resultado aparecerá en la terminal integrada en la parte inferior y en una pestaña específica para él, algo bastante cómodo para ejecutar diferentes scripts de forma ágil.

Otra forma de ejecutar el script es con la combinación *Mayus+F10*, pero si la encontráis demasiado compleja os sugiero crear un nuevo acceso con alguna tecla que nos se utiliza, como la de *PAUSE*:



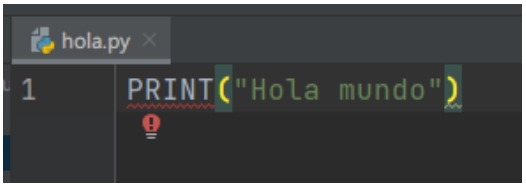
***Nota**: Si la fuente es demasiado pequeña para vosotros podéis aumentar y reducir el tamaño mediante Mayus+Alt+. y Mayus+Alt+,.*

Errores y avisos

Python es un lenguaje denominado *case-sensitive*, eso significa que es sensible a mayúsculas y minúsculas. Vamos a modificar *print* por *PRINT* y lo ejecutamos:

```
PRINT("Hola mundo")
```

Lo primero es que el propio editor nos indicará mediante un subrayado que algo no anda bien con esa porción de código



El problema es que no existe ninguna función llamada *PRINT*, por lo que Python no es capaz de encontrar nada con ese nombre y el editor nos avisa.

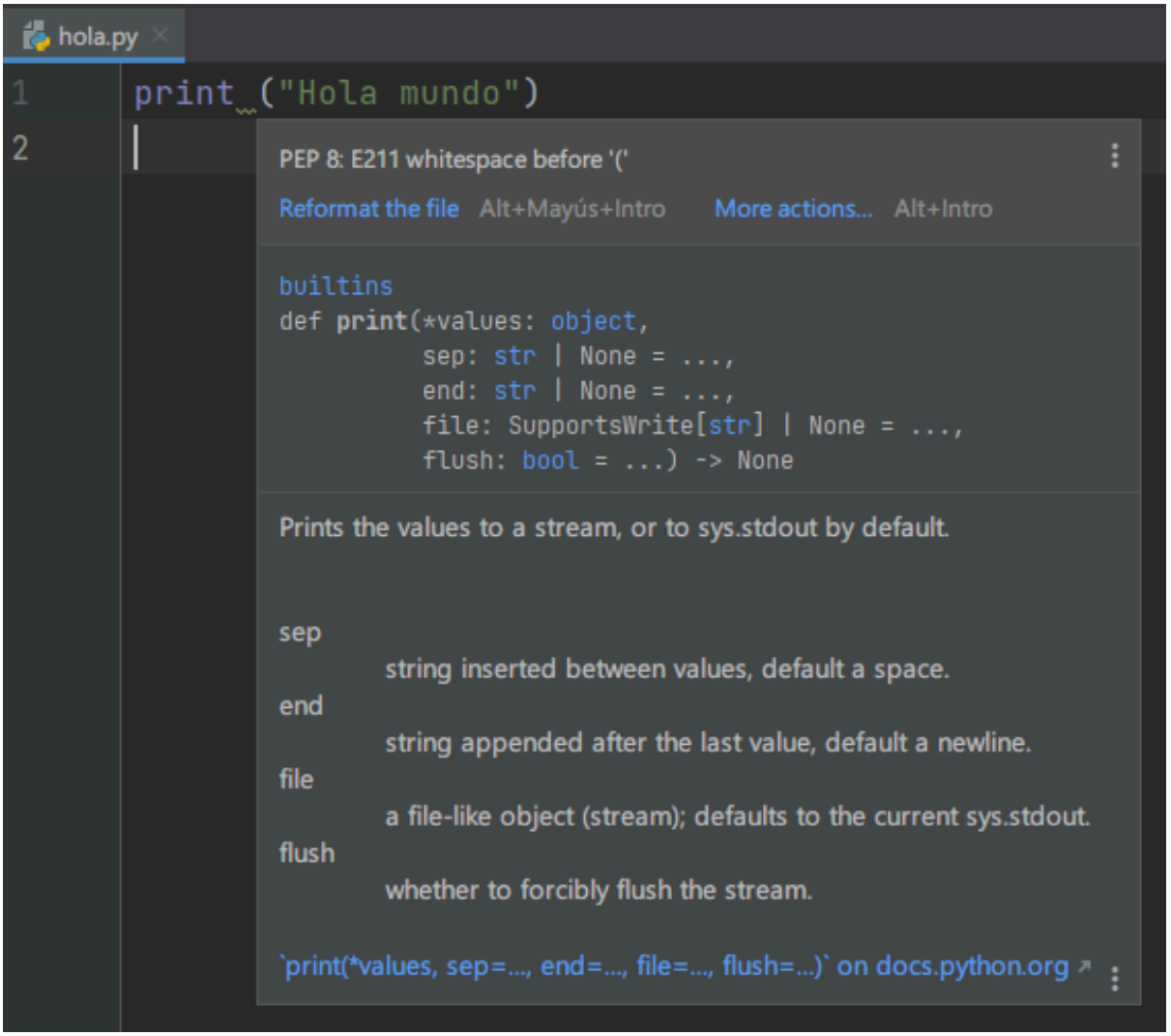
Si igualmente intentamos ejecutar el programa, saltará un error en la terminal:

```
Traceback (most recent call last):
  File "C:\Users\hcost\...\hola.py", line 1, in <module>
    PRINT("Hola mundo")
    ^^^^^
NameError: name 'PRINT' is not defined

Process finished with exit code 1
```

Debo inculcaros la práctica de analizar detenidamente los errores. Se requiere un poco de inglés pero generalmente podemos solucionar nuestros errores leyendo detenidamente la descripción del error, es una habilidad imprescindible para cualquier programador. En este caso hacia el final podemos leer ***"NameError: name 'PRINT' is not defined"***, debemos repasar esa parte.

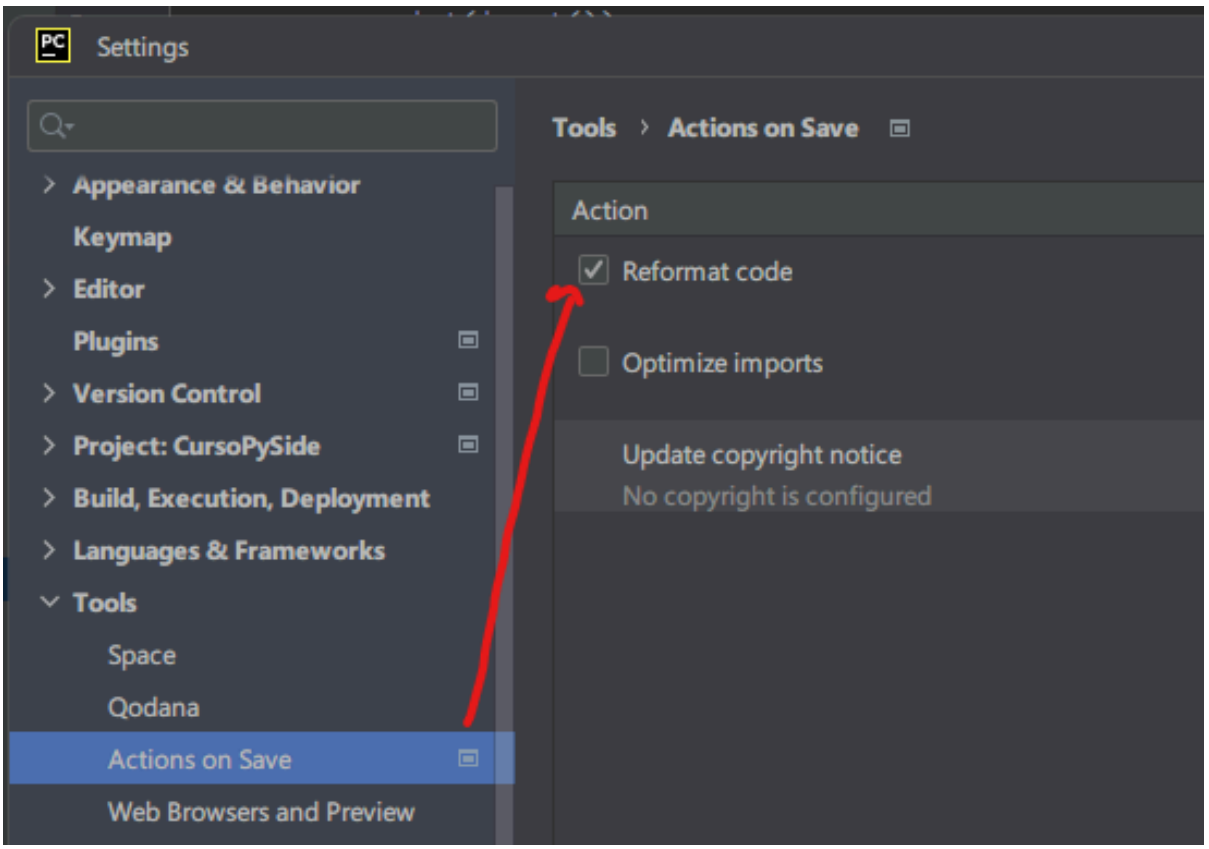
Un error significa que el programa finaliza abruptamente y deja de ejecutarse, pero no tiene que ser siempre así. Aprenderemos a lidiar con los errores más adelante, a capturarlos y a evitar que los programas se detengan automáticamente.

Otro tipo de avisos que nos marcará PyCharm son los de buenas prácticas. Por ejemplo si escribimos lo siguiente nos mostrará:

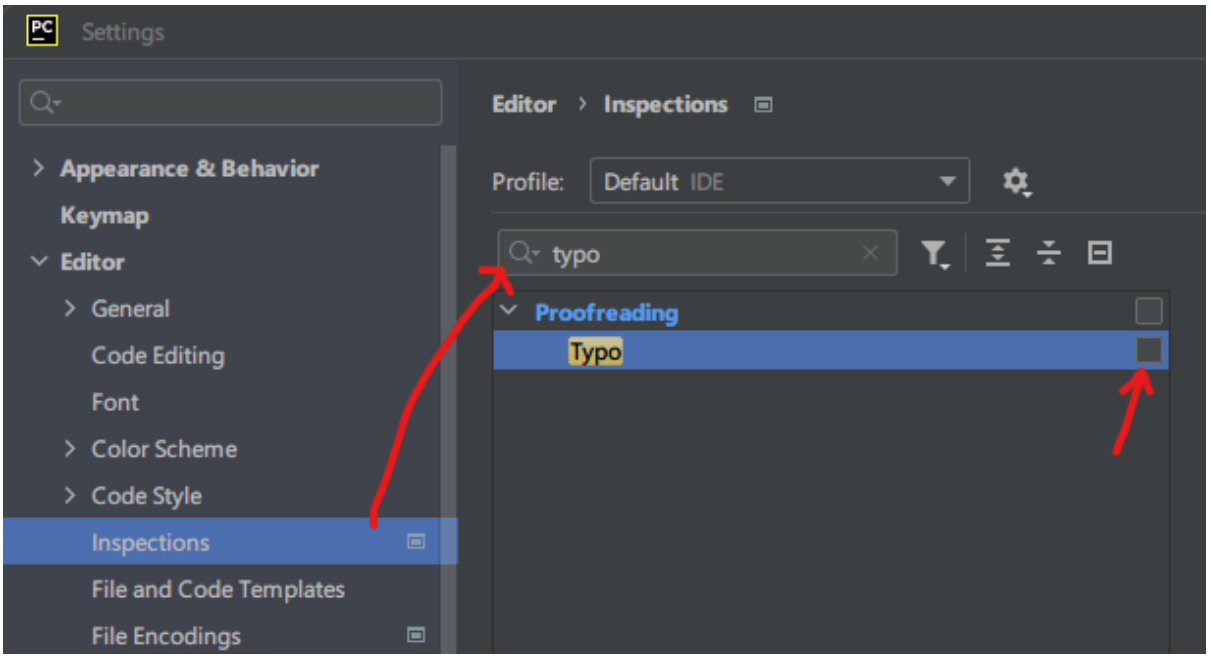


La PEP-8 es una ***Python Enhancement Proposal*** o en español *Propuesta de mejora de Python*, hecha por el creador de Python (*Guido Van Rossum*) y otros dos importantes programadores del núcleo de Python (*Barry Warsaw* y *Nick Coghlan*). Define una guía de estilo para escribir mejor el código y aunque no es obligatorio utilizarla es tan común que editores como PyCharm la integran por defecto.

Si le indicamos a PyCharm la opción “Reformat the file”, el propio editor formateará el código acordemente para cumplir las guías de estilo. También se puede configurar para que formatee automáticamente el código al guardar el fichero, algo que os recomiendo hacer:



También os recomiendo desactivar la corrección de gramática, por defecto está en lengua inglesa y es molesto que todo lo marque como fallo:



Os dejo el enlace a la PEP-8 original por si queréis echarle una ojeada, aunque todavía es pronto para que la entendáis porque recién estamos empezando con los fundamentos de la programación.

Impresión múltiple

Como es de esperar algo que podemos hacer es llamar a la función print varias veces, dando como resultado en una impresión múltiple:

```
print("Hola mundo")
print("Python mola mucho")
print("Así imprimimos varias líneas")
```

Otra forma de conseguir este resultado es mediante triple comillas, una forma de indicar que se interprete una cadena de caracteres multilínea:

```
print("""Hola mundo
Python mola mucho
Así imprimimos varias líneas""")
```

Esta forma de imprimir texto conserva los espacios y tabulaciones:

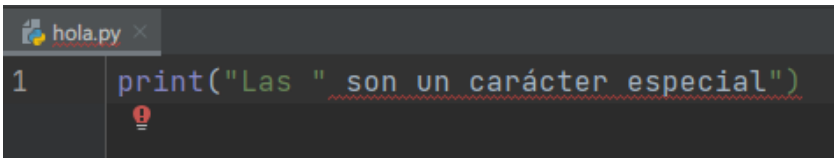
```
print("""Hola mundo
    Python mola mucho
        Así imprimimos varias líneas""")
```

Códigos de escape

No podemos terminar esta lección sin hablar de los códigos de escape. Supongamos que necesitas mostrar una comilla doble en el texto que vamos a imprimir, algo asi:

```
print("Las " son un carácter especial")
```

Como veremos el propio editor nos indica que algo falla ahí:



Fijaros como la segunda parte del texto sale en blanco, eso significa que Python no sabe interpretar esa parte. El problema es que al definir una doble comilla hemos cerrado el espacio que define una cadena de texto y eso es debido a que las comillas son caracteres especiales para Python, si queremos imprimirlas en un texto debemos hacérselo saber a Python y eso se consigue escapándolas mediante el carácter \:

```
print("Las \" son un carácter especial")
```

La barra invertida se utiliza para escapar otros caracteres, algunas combinaciones permiten interpretar caracteres especiales:

\'	Comilla simple
\"	Comilla doble
\t	Tabulación
\n	Salto de línea

Volviendo a nuestra primera cadena multilínea podemos hacer lo siguiente de una vez:

```
print("Hola mundo\n\tPython mola mucho\n\t\tAsí imprimimos varias líneas")
```

Mismo resultado pero sin impresión multilínea y todo de una vez.

Resumen del tema

En esta primera lección hemos aprendido a ejecutar código en Python, tanto en la terminal como utilizando el editor PyCharm. Hemos explicado qué son las funciones y hemos aprendido una de las más esenciales, ***print***, que sirve para imprimir por pantalla. Hemos experimentado sobre los errores y avisos, también qué es la PEP-8 y cómo PyCharm la integra para hacer nuestro código más legible. Y por último aprendimos a imprimir texto en múltiples líneas, haciendo uso de triple comillas y también mediante caracteres especiales escapados con la barra invertida.

En la próxima lección empezaremos a hacer algunos experimentos gráficos y a dibujar formas en la pantalla, os estaré esperando.

Experimentos gráficos

Por qué Python Arcade

Menos de un tercio de los usuarios matriculados en un curso de programación llega a empezarlo y de ese tercio solo un 3% completa el temario. Que el otro 97% abandone antes de acabar es un gran fracaso y me entristece. ¿Habrá alguna forma de hacer más atractiva la programación? ¿Podría transmitir estos conceptos sin que tantos alumnos se frustren y abandonen?

He reflexionado mucho sobre esto y al final encontré una respuesta: como una imagen vale más que mil palabras debería enfocarme en la programación gráfica. ¿El problema? Para programar gráficos primero hay que aprender a programar, es un pez que se muerde la cola. ¿Habrá alguna herramienta que permita hacerlo? Me resigné durante años hasta que no hace mucho y por azares del destino descubrí un módulo llamado Python Arcade. Creado por el profesor *Paul Vincent,* Arcade está ideado para enseñar ciencias de la computación a sus alumnos. Contiene una gran colección de funciones gráficas que simplifican el proceso de dibujo para centrarse en la lógica de programación.

Es su trabajo lo que me ha inspirado a crear este curso.

Comentarios de código

Cuando configuramos el entorno de aprendizaje ya instalamos el módulo arcade así que veamos como empezar a utilizarlo. Lo primero será crear un nuevo script *02-dibujos.py*.

Cuando los programadores escriben código, a menudo necesitan escribir notas explicando el propósito y funcionamiento del mismo. Para conseguirlo se utilizan los comentarios.

Los comentarios multilínea entre triple comilla que suelen ponerse en la cabecera del documentos:

```
"""
Los comentarios multilínea para realizar
explicaciones más extensas del funcionamiento
"""
```

Una práctica común es documentar diferentes metadatos, como una descripción, la autoría, fecha…

```
"""
Fichero: 01-introduccion/02-dibujos.py

En este script se realizarán diferentes dibujos
para explorar el funcionamiento de Python Arcade

Autor: Hektor Profe
Fecha: 22/03/2023
"""
```

Los otros comentarios son de una sola línea y sirven para anotaciones sobre el funcionamiento del código, generalmente de la línea que hay por debajo y empiezan con almohadilla #:

```
# Imprimimos un mensaje por pantalla
```

```
print("Hola mundo")
```

También se pueden poner al final de la línea, aunque no es tan legible:

```
print("Hola mundo")  # Imprimimos un mensaje por pantalla
```

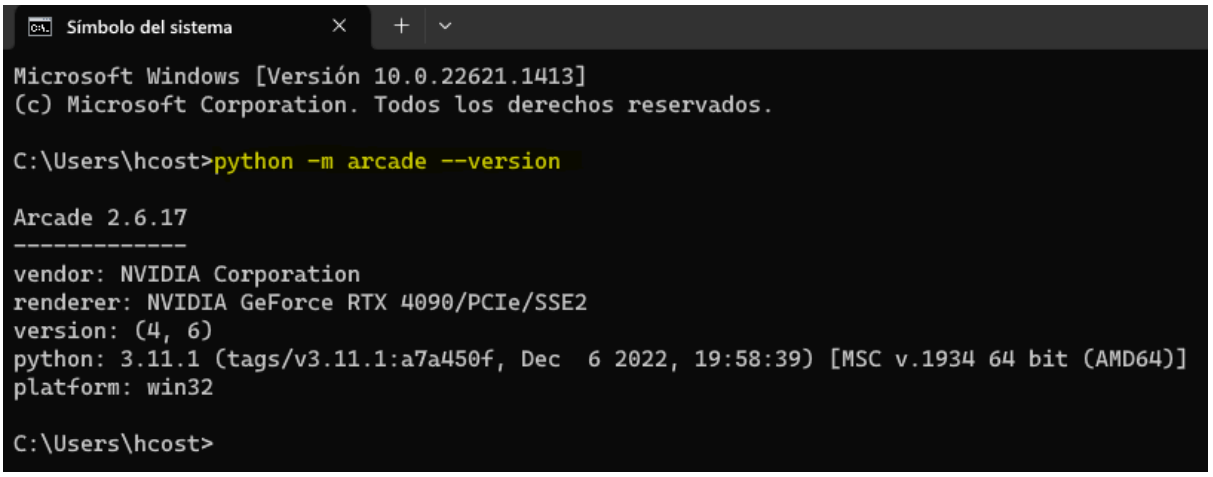
La *PEP-8* especifica que se guarden dos espacios de margen en los comentarios después del código.

Módulos estándar y públicos

El lenguaje Python incorpora todo tipo de módulos que conforman la conocida biblioteca estándar. Los módulos son colecciones de funciones y otras definiciones que podemos utilizar en nuestros programas. Hay módulos con funciones matemáticas, otros para controlar ficheros, bases de datos y todo tipo de tareas. Sin embargo el módulo *Python Arcade* no forma parte de la biblioteca estándar, está creado por el profesor Paul. Estos módulos se conocen como un módulo de terceros y existen cientos de miles creados por la comunidad, todos ellos publicados en el repositorio público PyPI, el índice de paquetes de Python. Cuando utilizamos el comando *pip install arcade* lo que hicimos fue buscar en el repositorio PyPI un módulo con ese nombre, descargarlo e instalarlo. En mi ***Curso Maestro de Python*** (disponible en Udemy) trato el tema en profundidad y enseño como publicar paquetes en el repositorio.

Importando un módulo

Así pues tenemos el módulo arcade instalado, o como mínimo deberíamos. Podemos comprobarlo mediante el siguiente comando en una terminal:



Para utilizar el módulo en un script debemos utilizar la instrucción *import* en la parte superior, antes de utilizar sus funciones:

```
"""
En este script se realizarán diferentes dibujos
para explorar el funcionamiento de Python Arcade

Autor: Hektor Profe
Fecha: 22/03/2023
```

```
"""

import arcade
```

Generando la ventana

Una vez cargadas en la memoria todas las funciones del módulo podemos empezar a dibujar y para ello primero debemos crear un lienzo que no es más ni menos que una ventana. La biblioteca Arcade tiene una función para la ventana a partir de diferentes argumentos llamada *open_window*:

```
# Creamos una ventana para dibujar
# arcade = biblioteca
# open_window = función
# argumentos = píxeles de ancho, alto y título de la ventana
arcade.open_window(600, 400, "Dibujos gráficos")
```

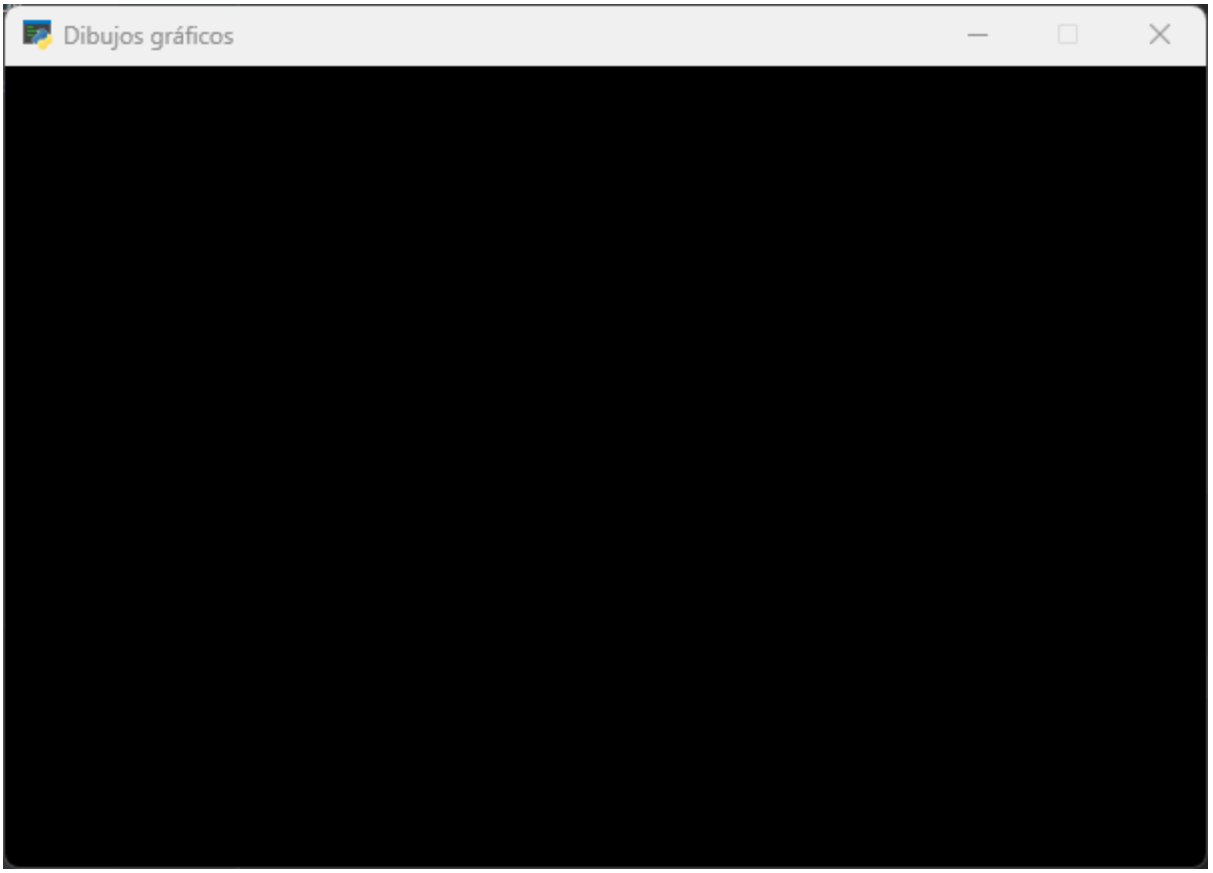
Fijaros que para establecer una cantidad numérica se pone sin las comillas dobles, en este caso *600* y *400* hacen referencia a dos números de tipo entero (*int*) porque no tienen parte decimal.

¿Cómo sabemos qué parámetros requiere una función? Pues leyendo la documentación de un módulo. ¡Leer documentación es una habilidad imprescindible para cualquier programador!

Con esto ya lo tenemos, si ejecutamos el script veremos que se abre y cierra inmediatamente una ventana. Debemos indicarle que se mantenga en marcha hasta que el usuario la cierre y para ello debemos ejecutar una segundo función llamada *run*:

```
# Mantenemos la ventana abierta hasta cerrarla
arcade.run()
```

Si volvemos a ejecutar sí veremos la ventana:

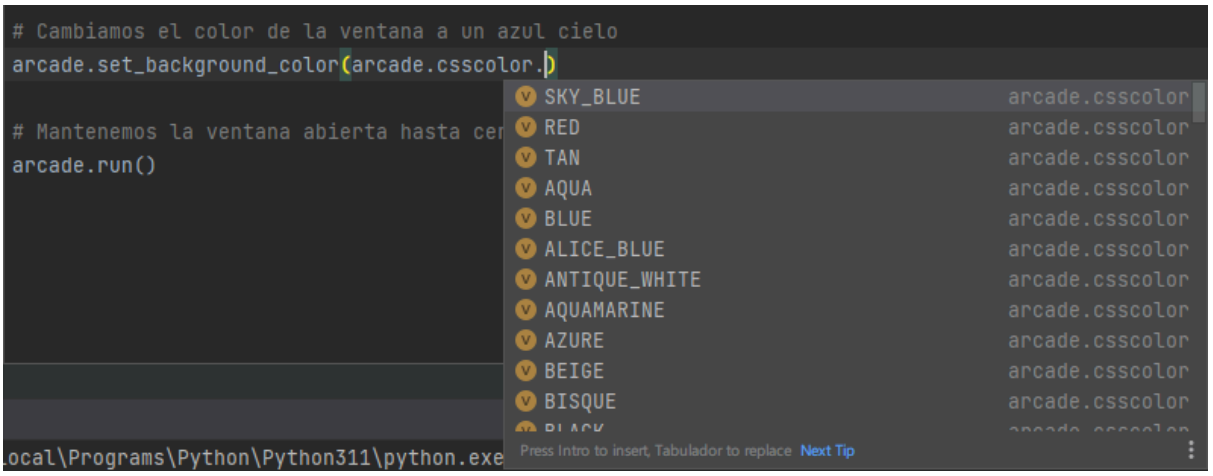


Colores predefinidos

La ventana está vacía, pero esa es la gracia, es como un lienzo sobre el que podemos dibujar!

Dependiendo del tema de vuestro sistema operativo quizá aparece el fondo negro o blanco, a mí me aparece negro. Hay otra función que nos permite cambiar el color. Se llama *set_background_color* y requiere que le pasamos un color como argumento. ¿Un color? ¿Qué es un color? Recordáis cuando os expliqué que existían muchos tipos de datos, pues un color es uno de esos tipos.

En Arcade hay dos formas de establecer un color, la más sencilla es utilizar los colores predeterminados del módulo. PyCharm generará una lista de los colores disponibles al hacer referencia a la colección *arcade.csscolor*:



```
# Configuramos el color de la ventana a un azul cielo
arcade.set_background_color(arcade.csscolor.SKY_BLUE)
```

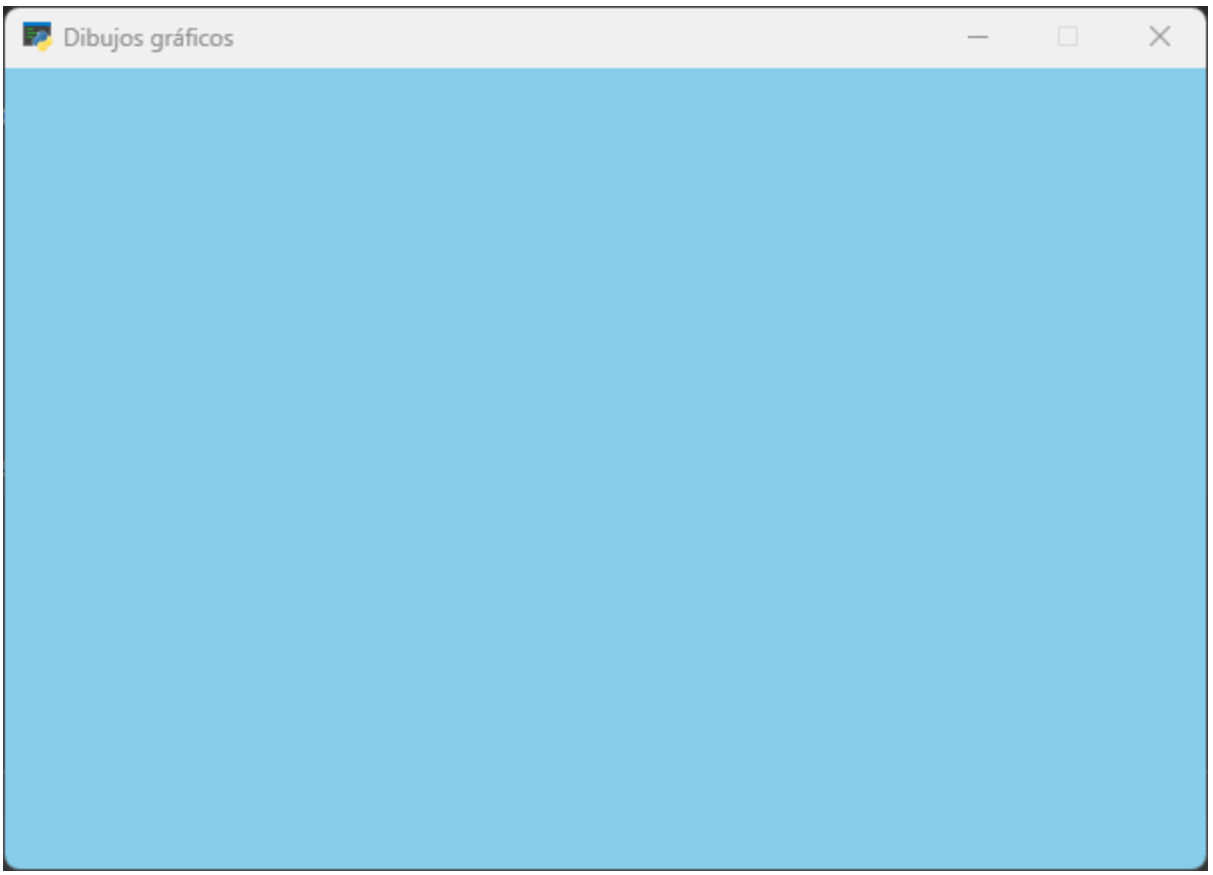
Proceso de renderizado

Si ejecutamos de nuevo el script podremos apreciar que el color no ha cambiado. ¿Por qué? Por una sencilla razón, no estamos dibujando el color de fondo, ¿cómo lo hacemos? Pues iniciando un proceso de dibujado, también llamado vulgarmente *renderizado*, del inglés *rendering*.

Este proceso inicia después de la configuración de la ventana y termina justo antes de finalizarla, por tanto requiere dos funciones que son *start_render* y *finish_render*:

```
# Creamos la ventana
arcade.open_window(600, 400, "Dibujos gráficos")

# Configuramos el color de la ventana
arcade.set_background_color(arcade.csscolor.SKY_BLUE)

# Iniciamos el dibujado
arcade.start_render()

# Finalizamos el dibujado
arcade.finish_render()

# Mantenemos la ventana abierta hasta cerrarla
arcade.run()
```

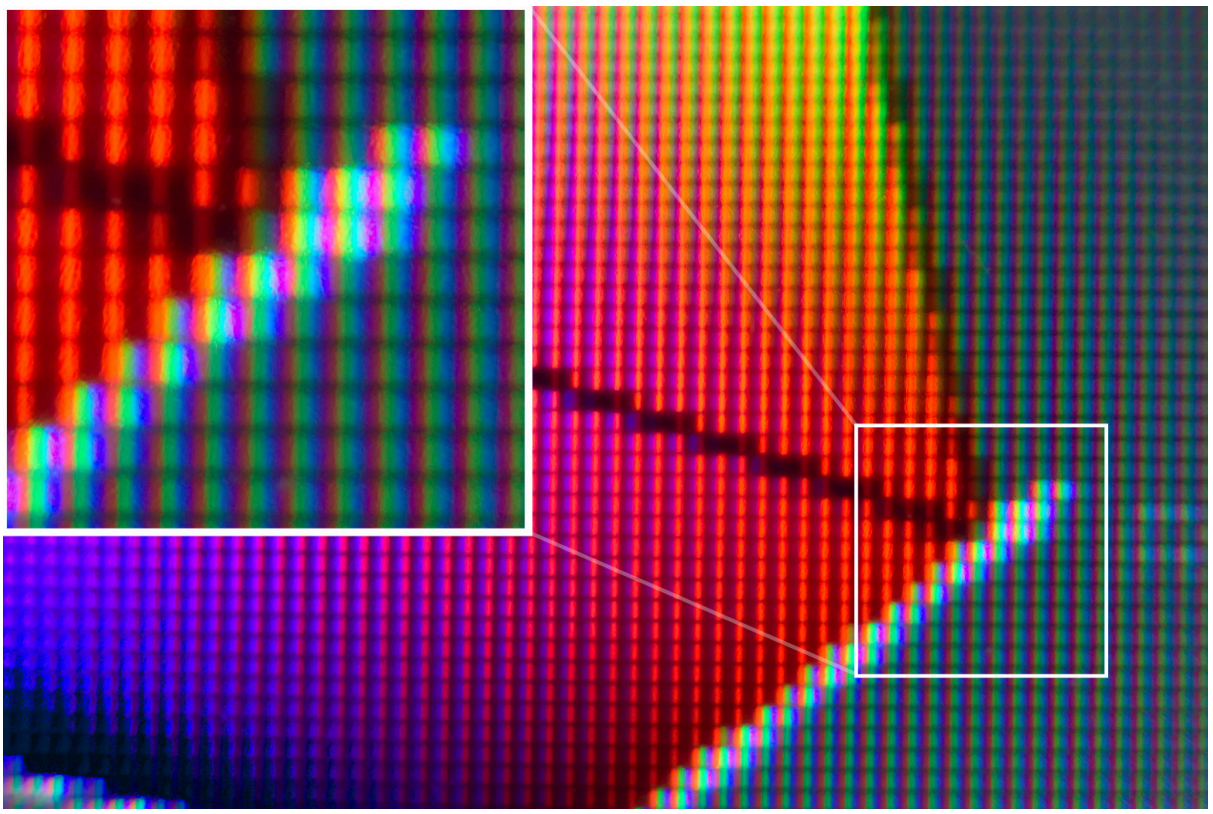
Con esto ya debería cambiar el color:



Colores RGB

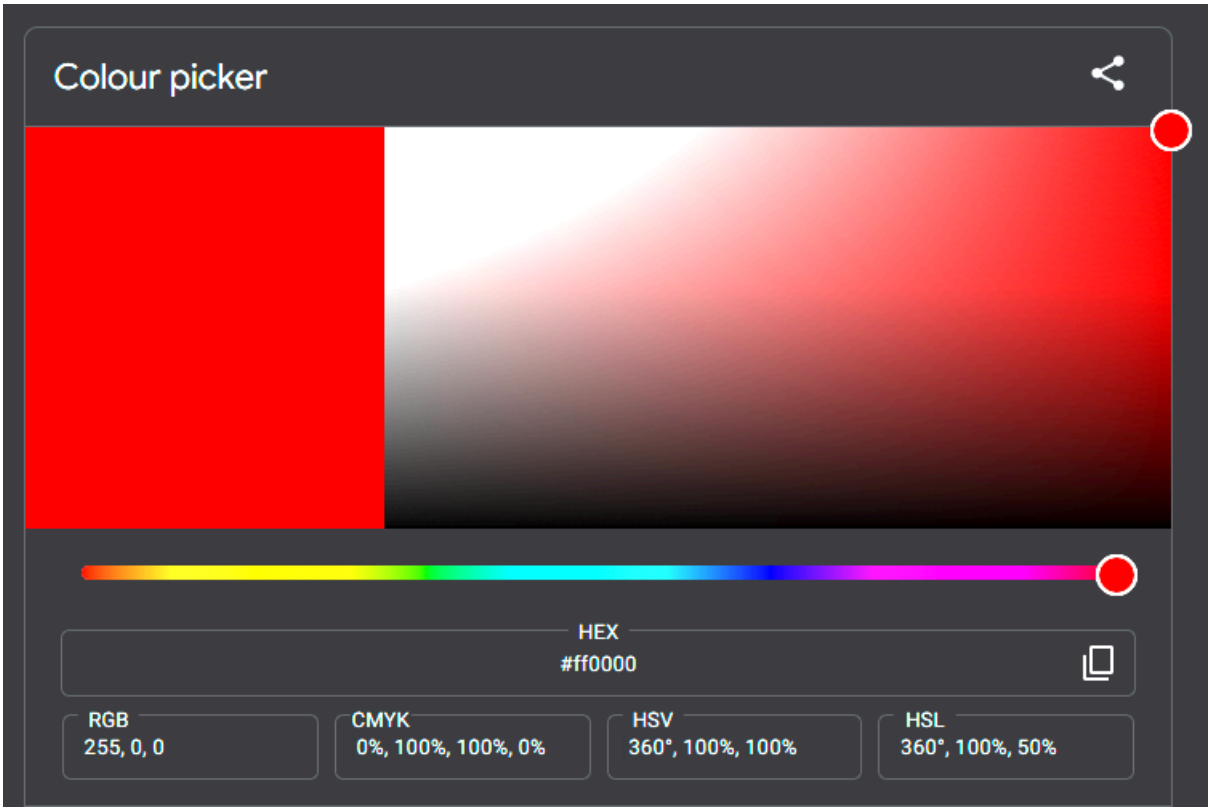
Ya vimos que podemos utilizar colores predefinidos, pero ¿y si queremos un color que no se encuentra en la lista? En ese caso deberemos generarlo a partir de los tres colores primarios **Red** (rojo), **Green** (verde) y **Blue** (azul).

Los monitores de computadora, los televisores, las pantallas del smartphone, las luces de colores LED… Todos estos sistemas tienen algo en común, y es que funcionan con tres pequeñas luces de color rojo, verde y azul:

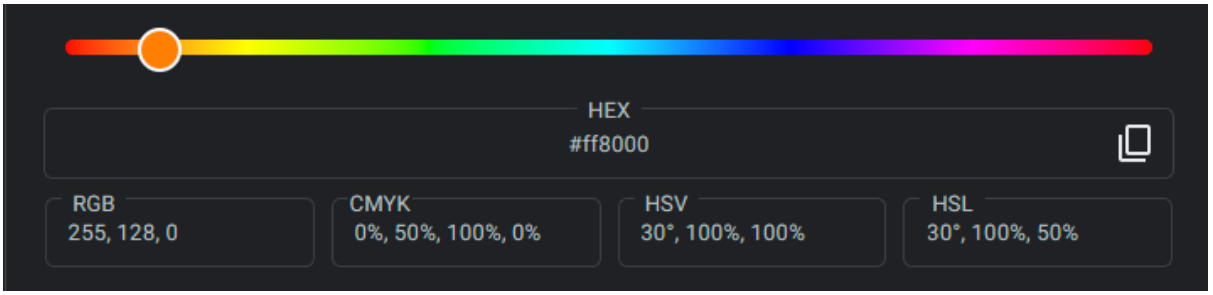


Apagar las tres luces se percibirá como negro, mientras que encender las tres a la vez se percibirá como blanco. Encender únicamente el rojo se verá rojo, el azul azul y el verde verde. Lo interesante es que al combinar las tres luces con diferente intensidad podemos generar una gran gama de colores. Esto se conoce como el modelo de adición RGB y es la forma cómo vamos a crear colores en la computadora. Por cierto, también existe el modelo RGBA, donde A hace referencia al canal *Alpha* y permite establecer una transparencia. El módulo Arcade permite tanto colores RGB como RGBA.

Si buscamos ***"color picker"*** en Google nos aparecerá una herramienta de generación de colores en diferentes modelos de adición: **RGB**, CMYK, HSV y HSL. Al seleccionar cualquier color del gradiente veremos el código proporcional a la intensidad de los tres colores RGB. La intensidad mínima de un color RGB es 0 y la máxima 255. Siendo por ejemplo *(255, 0, 0)* el código del rojo, *(0, 255, 0)* el verde y el *(0, 0, 255)* el azul:

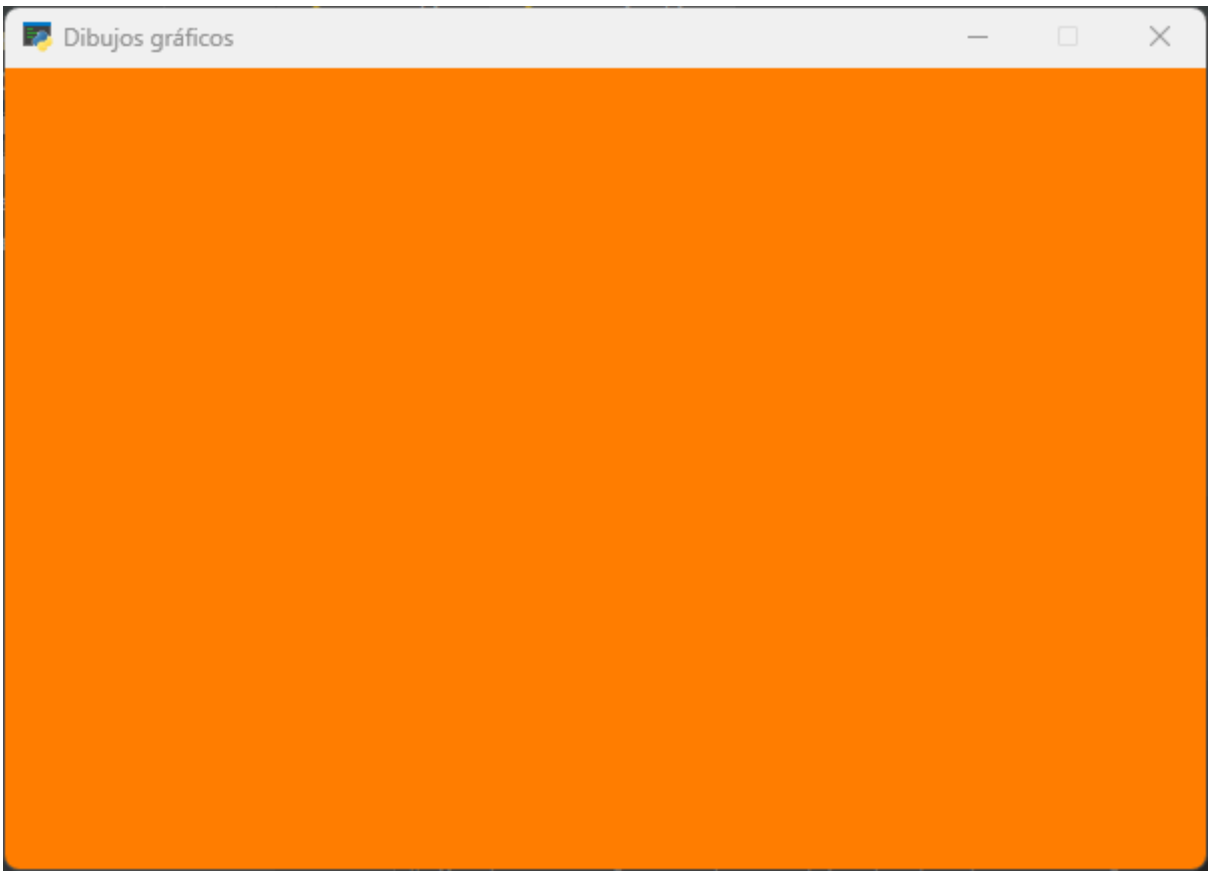


Un color como el naranja se conseguiría mezclando el rojo y el verde a una proporción de 1 a 0.5:



Podemos indicarle a Arcade que establezca el color de fondo en naranja en base a esos tres números (255, 128, 0):

```
arcade.set_background_color((255, 128, 0))
```



¿Pero por qué se utilizan números que van de 0 a 255 y no de 0 a 100 o 1000? Para resolver esta cuestión debemos empezar hablando sobre los bits y los bytes.

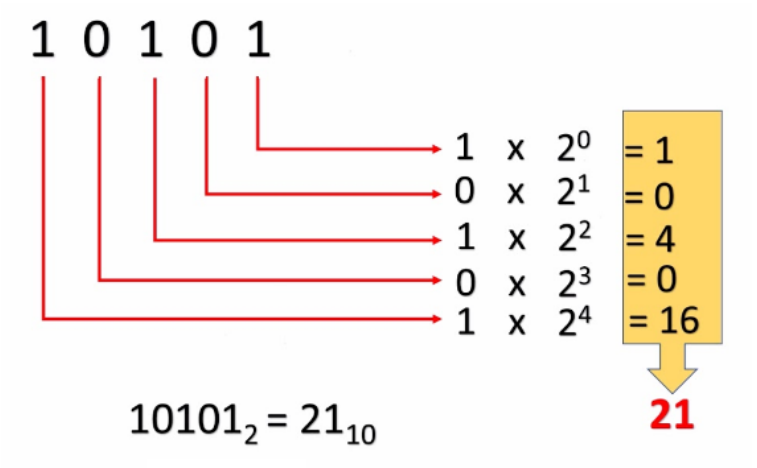
Bits y bytes

Es obvio que las computadoras funcionan con electricidad, por eso suele decirse que sólo comprenden dos estados: apagado y encendido. Para una computadora todo son interruptores que controlan circuitos por donde pasa o no la electricidad. Todo en las ciencias de la computación parte de esa idea.

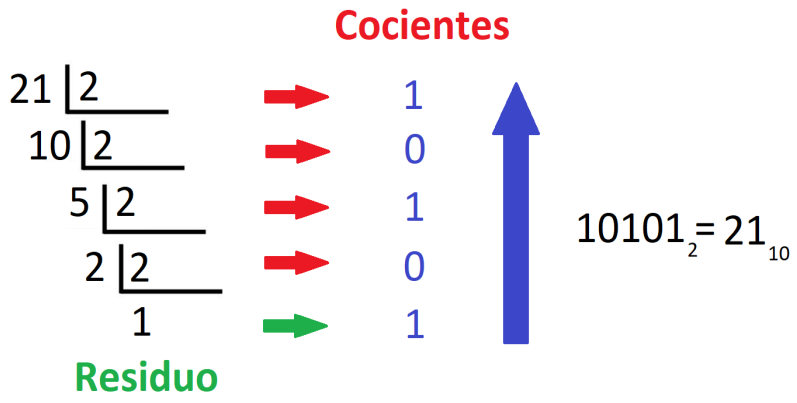
A las dos opciones del interruptor se les conoce como un ***bit***, del inglés "*binary digit",* un dígito binario cuyas opciones representan únicamente ***0*** = apagado y ***1*** = encendido. Lo interesante es que, de forma similar a como los humanos trabajamos en la base decimal con números que van del 0 al 9, es posible realizar combinaciones de 0 y 1 para expresar cualquier cantidad, estamos hablando de los números en sistema binario:

0	0000	**5**	0101
1	0001	**6**	0110
2	0010	**7**	0111
3	0011	**8**	1000
4	0100	**9**	1001

Matemáticamente podemos determinar el número decimal de un binario sumando las potencias de dos de cada número respecto a su posición empezando en cero:



De forma inversa podemos calcular el binario de un decimal agrupando a la inversa los cocientes de la división entre 2 y tomando el último residuo como última cifra:



Para almacenar los datos en la memoria las computadoras han basado la arquitectura de sus procesadores en diferentes cantidades de bits. Por ejemplo, el ***Intel 4004*** fue una CPU lanzada en 1971 cuya memoria trabajaba a 4 bits, pudiendo almacenar *2^4* = 16 valores decimales, ya fueran sin signo (de 0 a 15) o con signo (de -8 a 7), el cero contaría como positivo.

Para hacernos una idea de lo que ha evolucionado la tecnología, actualmente una CPU moderna basada en la arquitectura de 64 bits puede manejar *2^64* valores decimales, lo que son exactamente 18,446,744,073,709,551,616 números, 18 quintillones de nada. Si contamos el cero, sería esa cantidad menos 1 para números sin signo, mientras que con signo tendríamos la mitad negativos y la otra positivos (de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807).

Pero los humanos necesitamos algo más que números para comunicarnos, necesitamos el lenguaje para expresarnos y por eso era necesario otro tipo de dato que nos permitiese almacenar símbolos de escritura. Como históricamente la cantidad de bits necesaria para codificar un carácter de texto fue de 8 bits, se consideró que esa fuese la cantidad mínima de memoria que se podía almacenar en

una dirección de memoria. A ese conjunto de 8 bits se le denominó deliberadamente ***byte*** siendo esta palabra una reortografía deliberada del verbo "bite" que significa "morder", todo para poder diferenciarlo de "bit". Otra forma más técnica de referirse a este conjunto es "octeto", introducida en 1983 en el Protocolo de Internet RFC 791.

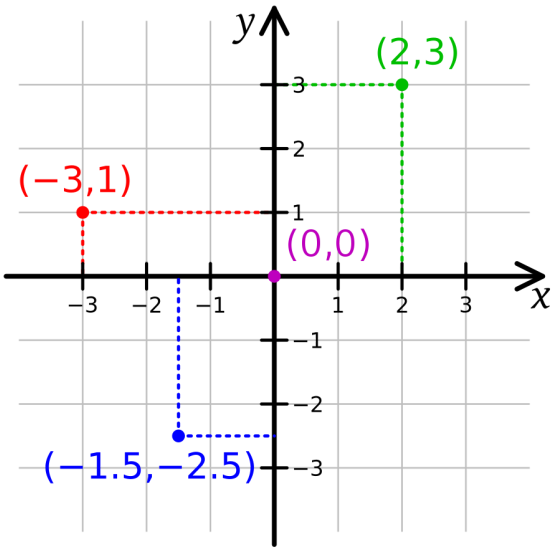
Dejando la historia de lado, la parte que nos interesa es qué cantidad máxima puede representar un byte. Ese número sería el 1111111, que en decimal corresponde a 2^8 y que es ni más ni menos que el 256. ¿Os suena? Si restamos un espacio para el 0, entonces el conjunto almacenable en un byte es cualquier número entre 0 y 255, los mismos rangos que se utilizan para definir la intensidad de un color RGB, correspondiendo exactamente al espacio de 1 byte u 8 bits en la memoria.

Coordenadas cartesianas

Un lienzo vacío es muy aburrido, ¡vamos a dibujar cosas! Para hacerlo necesitamos entender el funcionamiento del sistema de coordenadas y cómo utilizarlo para dibujar exactamente donde queramos.

Un sistema de coordenadas es una referencia que utiliza uno o más números llamados coordenadas para determinar la posición de un punto u objeto geométrico. Existen muchos tipos de sistemas pero a nosotros nos interesa el sistema de coordenadas cartesianas en el plano, que define dos ejes ***(x, y)*** para un sistema de dos dimensiones.

Se parte de un punto llamado origen cuya coordenada es ***(0, 0)*** para el ancho y alto. A partir de ahí se trazan dos líneas que simbolizan las dimensiones y se divide el espacio en partes proporcionales dándole un aspecto de cuadrícula. A partir de ahí podemos dibujar cualquier punto sobre el plano sabiendo sus dos coordenadas ***(x, y)***:



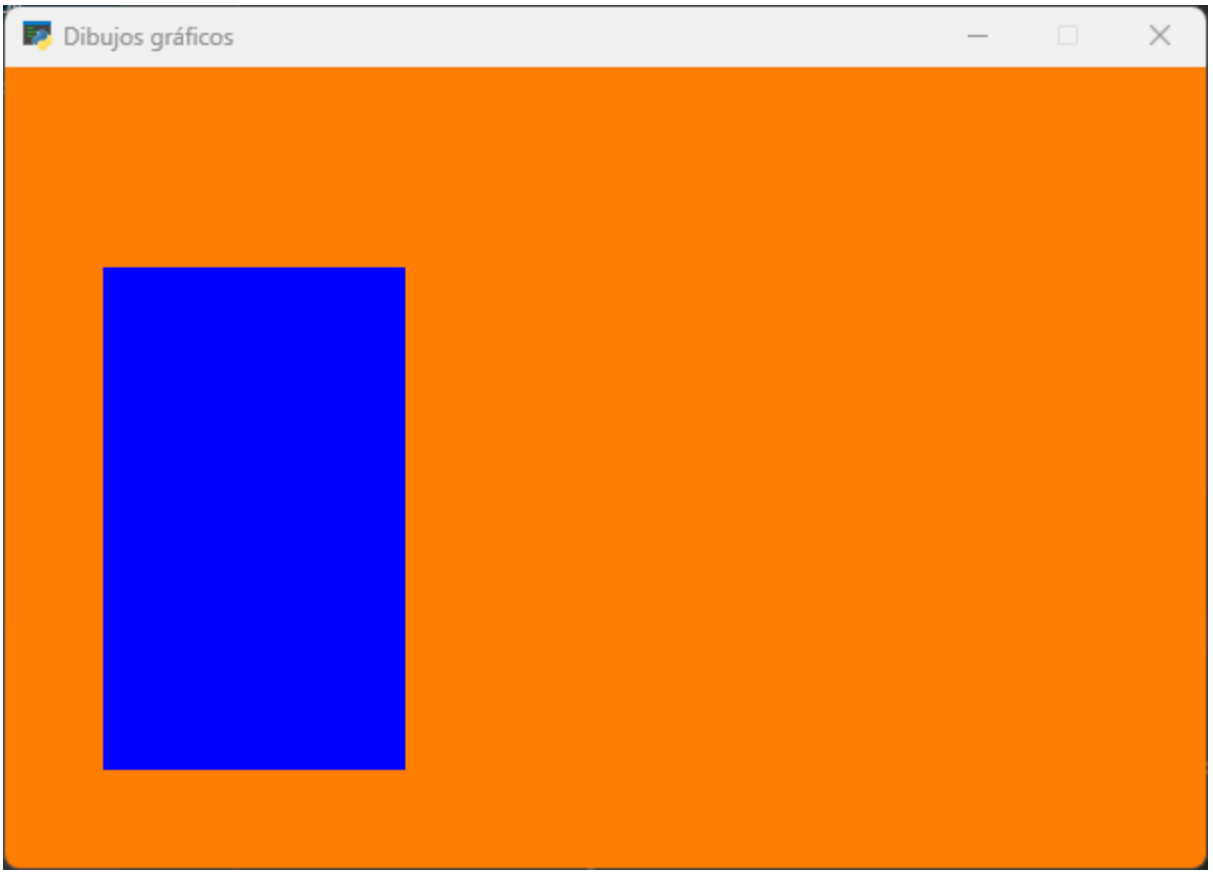
Este es el sistema que utilizaremos para dibujar sobre el lienzo, con la diferencia de que la cuadrícula la forman los píxeles de alto y ancho de la ventana y no tendremos parte negativa, será como utilizar solo el cuadrante superior derecho.

Dibujando rectángulos

Vamos a empezar dibujando un rectángulo con la función *draw_lrtb_rectangle_filled*:

```
# Dibujamos un rectángulo: left, right, top, bottom, color
arcade.draw_lrtb_rectangle_filled(50, 200, 300, 50, arcade.csscolor.BLUE)
```

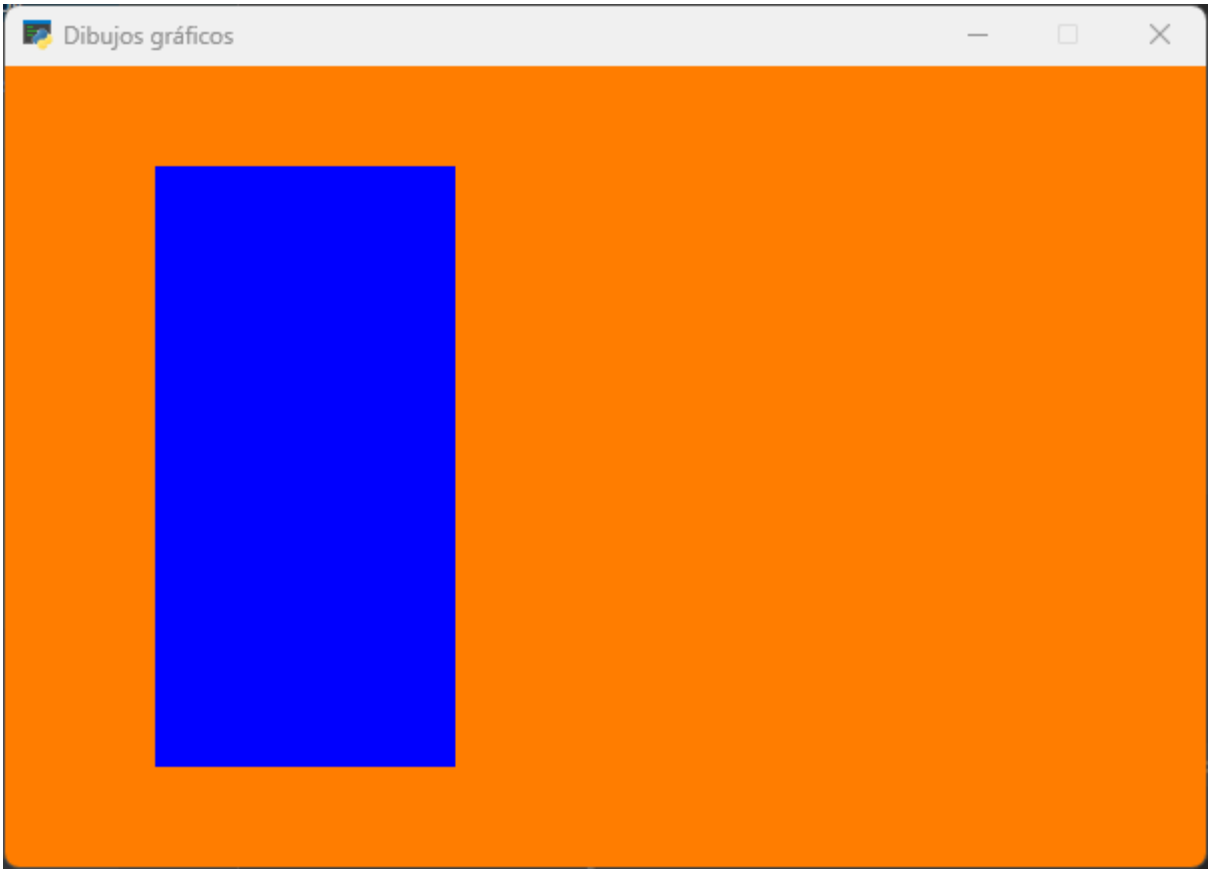
En esta función los argumentos indican la posición en ancho de los lados izquierdo y derecho; y la posición en alto de los lados superior e inferior:



La parte de *"filled"* significa que es un rectángulo con color de relleno. Existe la variante *draw_lrtb_rectangle_outline* para dibujar solo el contorno, luego veremos un ejemplo.

Otra manera de dibujar un rectángulo es a partir de su punto central, su ancho y alto. Se utiliza la misma función sin la parte de *lrtb*:

```
# Dibujamos un rectángulo: x, y, ancho, alto
arcade.draw_rectangle_filled(150, 200, 150, 300, arcade.csscolor.BLUE)
```



El algoritmo del pintor

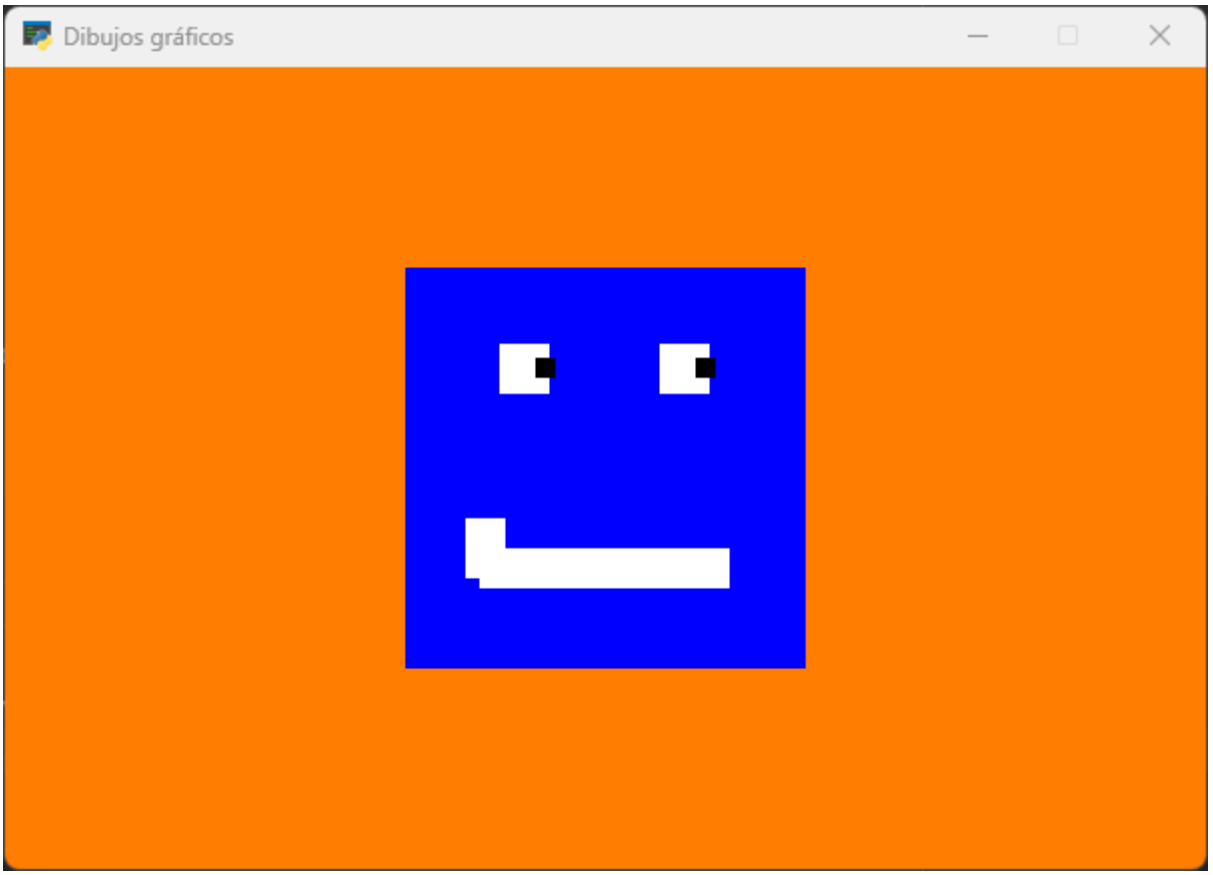
Al pintar un cuadro generalmente pintamos primero los elementos que se encuentran más lejanos y por encima los más cercanos. En los gráficos computacionales esta técnica recibe el nombre de algoritmo del pintor y consiste en lo mismo, ordenar los polígonos de una escena en función de su profundidad para pintarlos en ese orden.

La palabra ***algoritmo*** es extraña pero no deja de referirse a un conjunto de operaciones ordenadas para hallar la solución a un problema. Cosas como una receta de cocina, un manual de instrucciones o un formulario administrativo son ejemplos de algoritmos.

En cualquier caso cuando dibujamos sobre el lienzo podemos utilizar la técnica del algoritmo del pintor para superponer dibujos:

```
# Iniciamos el dibujado
arcade.start_render()

# Empezamos con un cuadrado en el centro para una cara
arcade.draw_rectangle_filled(300, 200, 200, 200, arcade.csscolor.BLUE)
# Un par de cuadrados blancos para hacer de ojos
arcade.draw_rectangle_filled(260, 250, 25, 25, arcade.csscolor.WHITE)
arcade.draw_rectangle_filled(340, 250, 25, 25, arcade.csscolor.WHITE)
# Sobre los ojos unas pupilas de color negro mirando a su izquierda
arcade.draw_rectangle_filled(270, 250, 10, 10, arcade.csscolor.BLACK)
arcade.draw_rectangle_filled(350, 250, 10, 10, arcade.csscolor.BLACK)
```

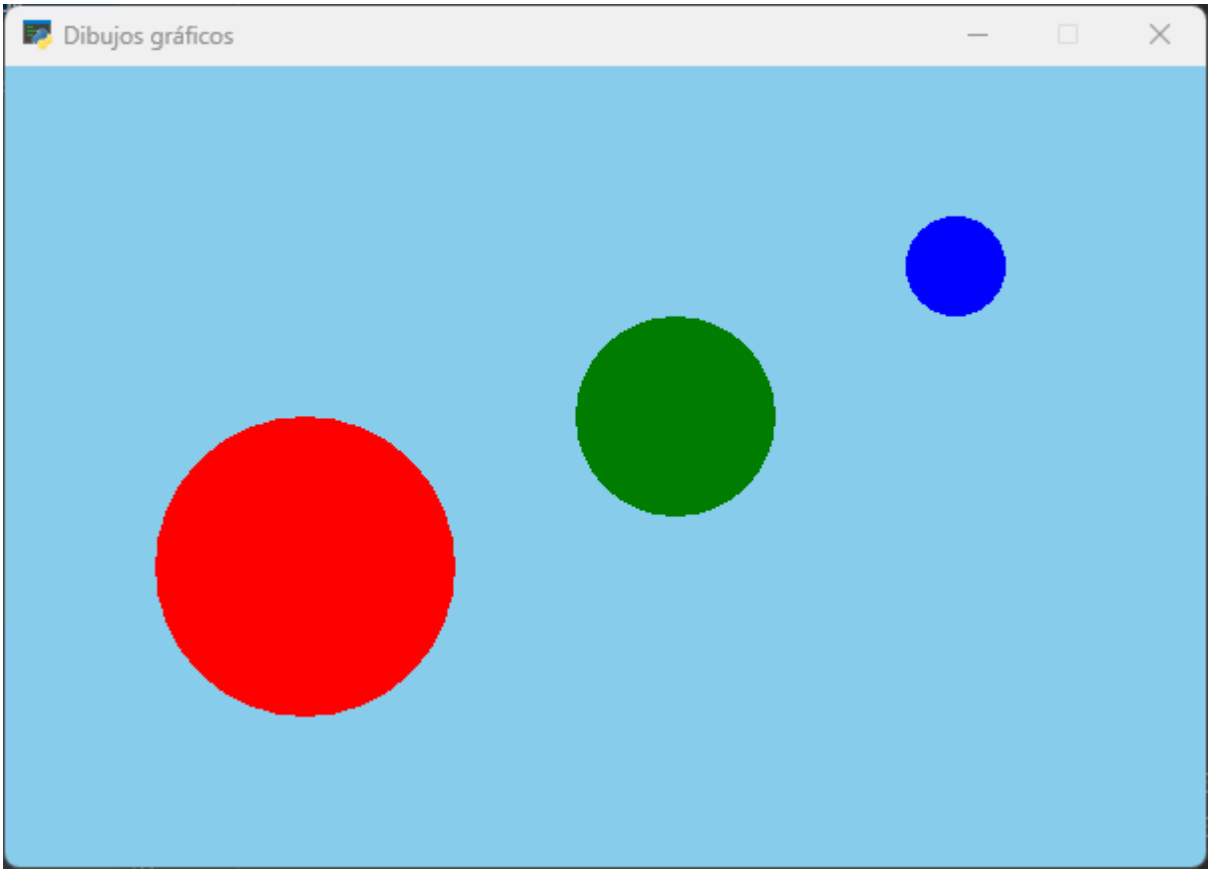
```
# Dos rectángulos para crear un efecto de boca sonriendo
arcade.draw_rectangle_filled(300, 150, 125, 20, arcade.csscolor.WHITE)
arcade.draw_rectangle_filled(240, 160, 20, 30, arcade.csscolor.WHITE)

# Finalizamos el dibujado
arcade.finish_render()
```



Dibujando otras formas

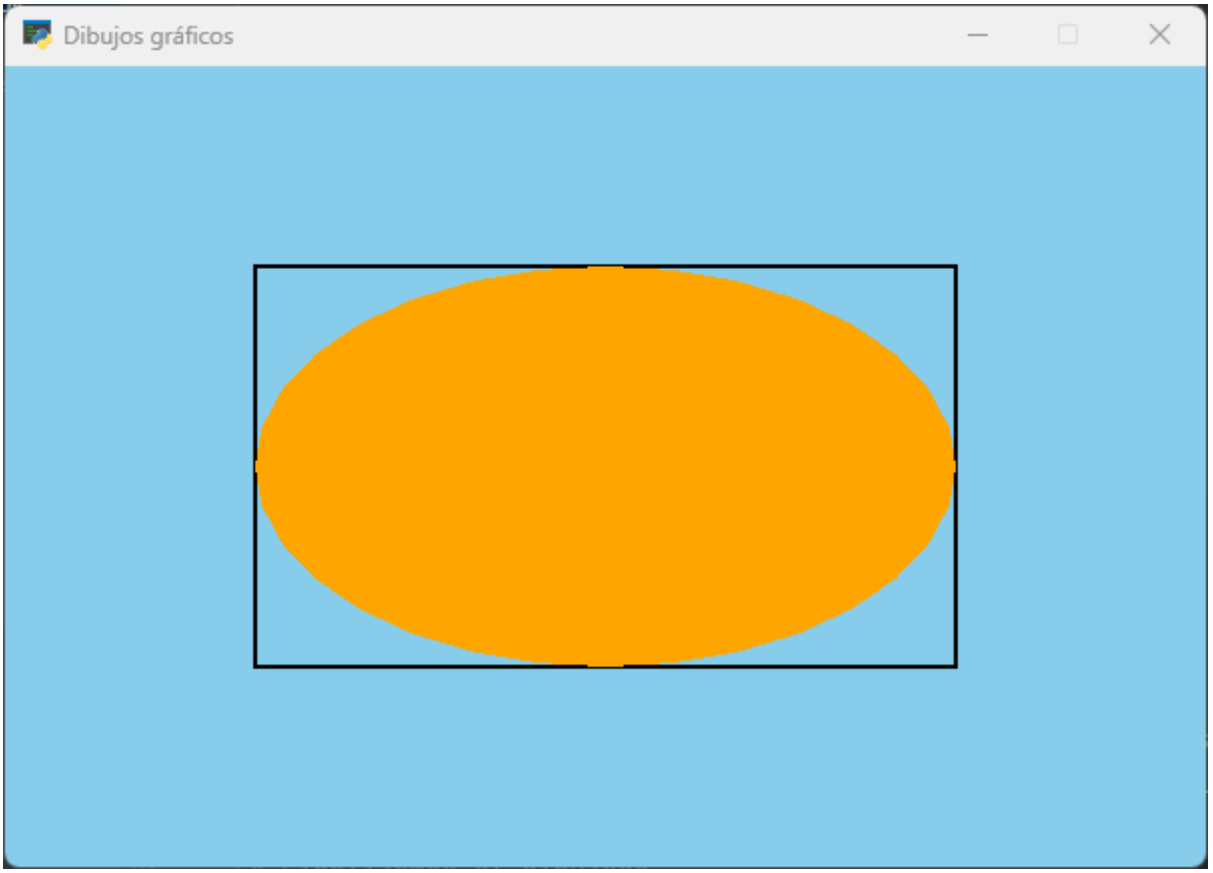
Hay muchas otras formas geométricas que podemos dibujar, por ejemplo círculos en un punto con un radio en píxeles:

```
# Configuramos el color de la ventana
arcade.set_background_color(arcade.csscolor.SKY_BLUE)

# Iniciamos el dibujado
arcade.start_render()

# Dibujamos una composición
arcade.draw_circle_filled(150, 150, 75, arcade.csscolor.RED)
arcade.draw_circle_filled(335, 225, 50, arcade.csscolor.GREEN)
arcade.draw_circle_filled(475, 300, 25, arcade.csscolor.BLUE)

# Finalizamos el dibujado
arcade.finish_render()
```



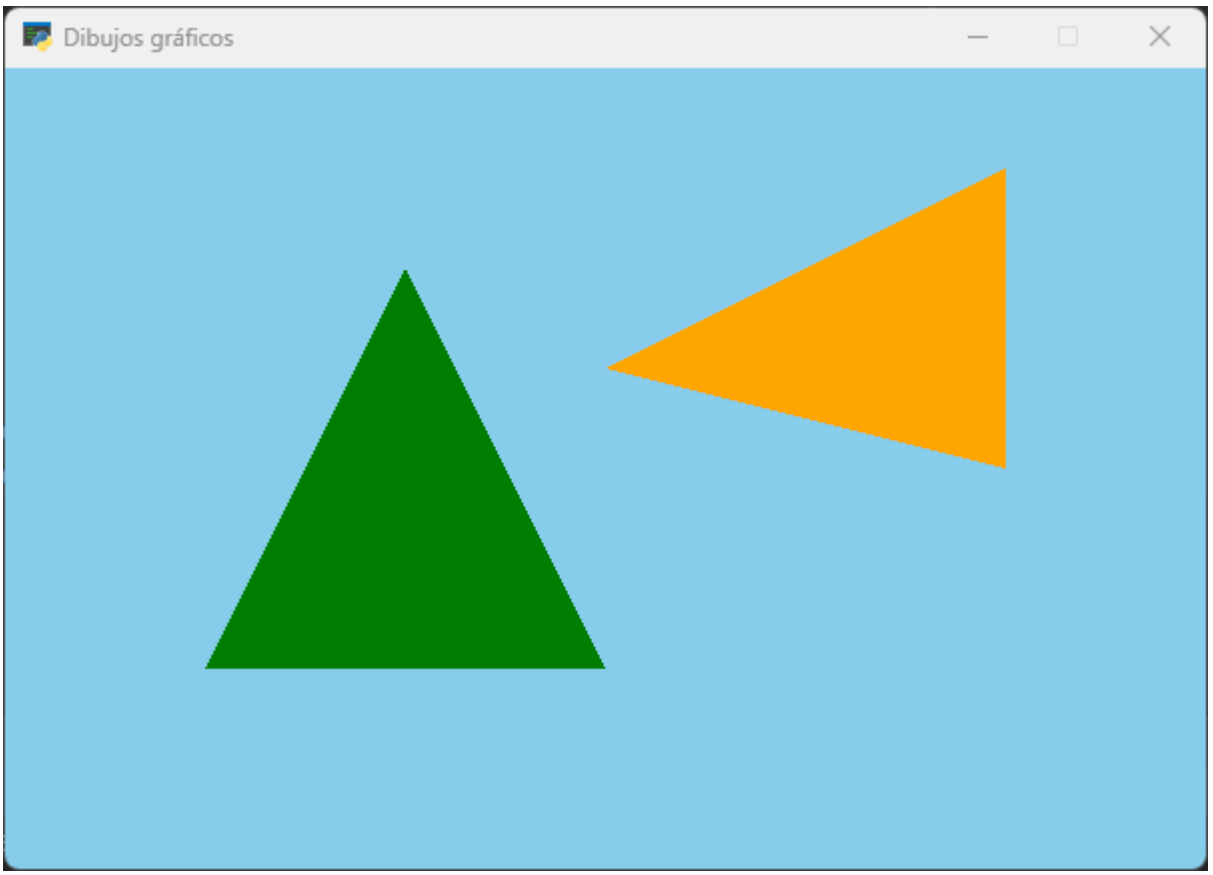
También tenemos elipses, que son como círculos no perfectos contenidos en formas rectangulares. La función es prácticamente la misma que para dibujar un rectángulo:

```
# Dibujamos una composición
arcade.draw_rectangle_outline(300, 200, 350, 200, arcade.csscolor.BLACK, 2)
arcade.draw_ellipse_filled(300, 200, 350, 200, arcade.csscolor.ORANGE)
```



Para dibujar triángulos la función requiere tres puntos, uno para cada vértice:

```
arcade.draw_triangle_filled(100, 100, 200, 300, 300, 100, arcade.csscolor.GREEN)
arcade.draw_triangle_filled(300, 250, 500, 350, 500, 200, arcade.csscolor.ORANGE)
```



Leyendo documentación

La documentación de software se define como la información enfocada en la descripción del sistema o producto para quienes se encargan de desarrollarlo, implementarlo y utilizarlo.

La mayoría de programadores no pierden el tiempo escribiendo documentación al gusto sino que la generan automáticamente a partir de los propios comentarios en su código. Esa es la razón por la que las páginas de documentación suelen estar muy referenciadas, son poco atractivas y contienen descripciones muy técnicas.

Vamos a echar un vistazo a la documentación de la función *draw_text* de Arcade, que es una de las más completas que tiene el módulo y sirve para dibujar texto:

arcade.Text

```
class arcade.Text(
      text: str,
      start_x: float,
      start_y: float,
      color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]] =
    (255, 255, 255),
      font_size: float = 12,
      width: int = 0,
      align: str = 'left',
      font_name: Union[str, Tuple[str, ...]] = ('calibri', 'arial'),
      bold: bool = False,
      italic: bool = False,
      anchor_x: str = 'left',
      anchor_y: str = 'baseline',
      multiline: bool = False,
      rotation: float = 0)                                          [source]
```

Cuando consultamos la documentación de una función siempre encontramos un listado de los argumentos que aceptan y el tipo de dato que se espera para cada uno de ellos. Estos argumentos, que también reciben el nombre de parámetros, pueden definirse por orden o por nombre, pero no os preocupéis por ahora. Más adelante en el curso aprenderemos a crear nuestras propias funciones y profundizaremos en el tema.

Arcade tiene muchas otras funciones para dibujar líneas y arcos. Os dejo la documentación sobre ellas y os reto a echar un vistazo para aprender cómo funcionan, consideradlo un ejercicio para practicar:

- Documentación para dibujar líneas
- Documentación para dibujar arcos

Resumen del tema

¡En esta segunda lección hemos aprendido un montón de cosas! Empezamos viendo cómo utilizar comentarios de código, qué son los módulos y la diferencia entre módulos estándar y de terceros, así como la forma de utilizarlos en nuestros scripts. También hemos empezado a trabajar con *Python Arcade*, creando la ventana, pintando el fondo de colores y dibujando todo tipo de formas geométricas. Además hemos introducido conceptos más técnicos como los bits y los bytes, el sistema de coordenadas cartesianas, el algoritmo del pintor y la documentación de código.

En la próxima lección nos tomaremos un rato para aprender el funcionamiento de los ordenadores, los lenguajes de programación, los orígenes y peculiaridades de Python y otros conceptos imprescindibles para entender este mundillo.

Lenguajes de programación

Cuestiones previas

Hasta ahora hemos instalado Python, el editor PyCharm y hemos empezado a crear scripts que hacen cosas, desde imprimir mensajes a dibujar formas en una ventana. Estamos programando, pero realmente no sabemos qué sucede en segundo plano. ¿Qué es realmente un lenguaje de programación?, ¿Cómo funcionan?, ¿Cuáles hay? En esta lección daremos respuesta a todas esas preguntas y muchas otras.

CPU: Unidad Central de Procesamiento

¿Recordáis que para crear un color RGB definimos tres valores entre 0 y 255? Vimos que eso tiene que ver con el número de bits que tienen un byte y el valor máximo que puede expresar. También hablamos de que la arquitectura de un sistema (4 bits, 16 bits, 64 bits…) es lo que determina el número máximo que puede manejar la máquina.

Pero cuando hablamos de manejar, realmente nos referimos a operar, pues dentro de la computadora todo se basa en realizar operaciones matemáticas y la parte encargada de realizarlas es el chip conocido como CPU (*Unidad Central de Procesamiento)*. Este chip repleto de millones de microtransistores es capaz de realizar ingentes cantidades de operaciones por segundo, por eso a la CPU también se la conoce como el "cerebro" de la máquina.

El caso es que si queremos que la CPU realice alguna tarea no es tan fácil como pedírselo en lenguaje natural. Este chip solo entiende números binarios y si queremos que realice alguna tarea debemos traducirla a ese "lenguaje".

GPU: Unidad de Procesamiento Gráfico

Antes de continuar aprendiendo sobre esos "lenguajes", tomémonos unos minutos para hablar de las GPU, un componente que para mí es igual o incluso más interesante que la CPU. A fin de cuentas representa la parte más "gráfica" del sistema.

Si os pido que penséis en una tarjeta gráfica seguro que recordaréis esos enormes trastos con ventiladores y lucecitas. Todos sabemos que se utilizan para jugar a videojuegos, trabajar con programas de edición de imágenes, vídeos y ese tipo de cosas. Pero lo que quizá no todos sabéis es que al principio las computadoras no tenían GPU, sino que era la propia CPU quien manejaba la pantalla.

Las operaciones que puede realizar una CPU son muchas y variadas, están directamente relacionadas con los programas que le pedimos que ejecute. Pero algunas operaciones como dibujar la pantalla se repiten constantemente y son siempre las mismas. Los ingenieros de antaño lo sabían y se les ocurrió una brillante idea. Si dibujar la pantalla no requería una potencia de cálculo masiva podían externalizar el trabajo a otro chip con un procesador menos potente pero más especializado, así podían ahorrar trabajo a la CPU y de paso crear una nueva línea de producción.

Dicho y hecho, al nuevo componente le llamaron GPU (Unidad de Procesamiento Gráfico) y en 1981 IBM lanzó la primera gráfica al mercado, la *MDA* (*Monochrome Display Adapter*), una placa capaz de representar 25 líneas de 80 caracteres en pantalla y con una RAM de 4 kB capaz de almacenar una

página en la memoria. Esto puede parecer poco pero fue todo un avance porque permitía interacciones mucho más fluidas con el sistema.

Separar las operaciones generales de las operaciones gráficas fue una decisión tan importante que marcó el futuro para siempre. Los ingenieros empezaron a añadir más y más procesadores especializados a las GPU para realizar cálculos gráficos en paralelo. A estos procesadores en paralelo se les conoce como núcleos y mientras una CPU como la *Intel i9 13900k* puede tener 24 núcleos de uso general, una gráfica de gama alta como la *Nvidia RTX 4090* tiene 16.384 núcleos especializados. Esto es brutal, tienen tanta potencia que dibujar gráficos es solo una parte de lo que pueden hacer. Los fabricantes como Nvidia y AMD no han desaprovechado la oportunidad de modificar sus gráficas con nuevas operaciones para potenciar la minería de criptomonedas y el machine learning en el ámbito de la inteligencia artificial. Son una parte vital del desarrollo tecnológico moderno y no es posible imaginar un futuro sin ellas.

Lenguajes de programación

Volviendo a los procesadores, como podéis suponer, crear programas informáticos en un lenguaje binario es muy difícil, pero de hecho así es como se hacía al principio y todas las computadoras siguen funcionando así tras bambalinas. Por suerte con el tiempo la cosa se ha sofisticado y han aparecido nuevos lenguajes que suelen agruparse en tres generaciones:

1. ***Primera generación, "lenguajes de código máquina":*** Creados entre las décadas de 1940 y 1950, los lenguajes de primera generación son los de más bajo nivel, eso significa que sus instrucciones son las más cercanas a lo que entiende el procesador porque se escriben en secuencias binarias de ceros y unos. No hay una lista universal de lenguajes de primera generación ya que cada fabricante desarrolló su propio conjunto de instrucciones para sus computadoras. Por nombrar un par tenemos la *"Harvard Mark I"* creada por *IBM* en 1944 y que ocupaba una habitación entera o la "*UNIVAC I"* de la empresa *Remington Rand* en 1952, considerada la primera computadora comercial fabricada en EE UU y cuyo tamaño era como un escritorio.
2. **Segunda generación, "lenguajes ensamblador"**: En 1949, una científica de la computación llamada Kathleen Booth del departamento de matemáticas de la Universidad Birkbeck en Londres inventó algo llamado lenguaje ensamblador como una mejora a los lenguajes de código máquina. En lugar de usar códigos numéricos, estos lenguajes utilizaban mnemónicos (códigos abreviados) para representar las instrucciones. Para funcionar requieren de un programa llamado compilador que toma el código en lenguaje ensamblador y lo traduce a código máquina. Cada fabricante de procesadores suele tener su propia variante y en la actualidad los más utilizados son "*ARM"*, "*MIPS*" y "*x86*". Aunque es más sencillo que el código máquina sigue siendo difícil programar en ensamblador y también se considera un lenguaje de bajo nivel.
3. **Tercera generación, "lenguajes de alto nivel"**: La tercera generación empieza en 1955 cuando el informático teórico John Backus crea FORTRAN y a la vez la científica de la computación Grace Hopper crea FLOW-MATIC, que posteriormente evolucionaría a COBOL en 1959. Son lenguajes cuyas instrucciones se asemejan al lenguaje humano, de ahí que se les denomine de alto nivel y suelen dividirse en dos grupos dependiendo de su funcionamiento:

a. **Lenguajes compilados**: Igual que los lenguajes ensamblador, estos se compilan y se traducen a código máquina, lo que los hace muy rápidos. Sirven para programar cualquier cosa, desde programas a videojuegos de última generación. Los más utilizados son la familia C, C++ y C#, lanzados en 1972, 1985 y 2000 respectivamente y que según el índice TIOBE, suman un 35% de la cuota de mercado. Otros dignos de mención son Java y Visual Basic.
b. **Lenguajes interpretados**: Estos lenguajes son más recientes y en lugar de compilarse para generar un código máquina, la computadora examina el código fuente línea a línea y lo ejecuta en tiempo real. Son mucho más lentos que los lenguajes compilados, pero a su favor tienen un nivel de abstracción todavía mayor, con una sintaxis más intuitiva y fácil de aprender. Además son muy flexibles, tened en cuenta que la compilación de un programa o videojuego puede llevar muchas horas. En su lugar, los lenguajes interpretados permiten modificar su funcionamiento en vivo, haciéndolos más versátiles. Python, JavaScript, SQL y MatLab son algunos de los más utilizados.

Evidentemente hay muchos otros lenguajes, basta ver el índice TIOBE para hacerse una idea. Los más modernos permiten programar de forma visual y otros hacen uso de la inteligencia artificial para solucionar problemas. Relacionado con esto, un tema que da mucho de que hablar es si la IA acabará suplantando a los programadores. Aplicaciones como ChatGPT son capaces de crear funciones con unas pocas indicaciones. Yo mismo le he pedido que me ayude a comentar algunos de los scripts del curso y lo hace bastante bien, pero a veces se equivoca y por eso creo que más que suplantarnos pueden apoyarnos. O por lo menos eso creo actualmente, en el futuro ya veremos.

El lenguaje Python

Hablemos un poco del lenguaje Python, que como vimos anteriormente se trata de un lenguaje interpretado, con mayor abstracción y flexibilidad que un lenguaje compilado.

Según el índice TIOBE, Python es el lenguaje más utilizado en 2023 por encima de C, Java y C++. Nunca antes en la historia un lenguaje interpretado había sido tan popular, al contrario, tradicionalmente los programadores han criticado este tipo de lenguajes por ser poco eficientes. “Un verdadero programador utiliza Java o C++, nunca Python”, decían antaño. ¿Entonces cómo ha conseguido Python posicionarse tan alto? Como programador que lleva 15 años trabajando con él he tenido la suerte de vivir su auge en primera persona, por eso lo tengo muy claro.

Seamos sinceros, la programación, al igual que todo lo relacionado con la informática siempre se ha considerado como una habilidad para *nerds*, algo de ingenieros y técnicos con pocas habilidades sociales aislados en sus oficinas “haciendo” cosas que solo ellos entienden. Pero nuestro mundo se ha tecnificado masivamente en los últimos años. Ahora no solo estamos rodeados de pequeñas computadoras todo el tiempo: en el bolsillo, en la muñeca, en el coche… Sino que nos hemos vuelto adictos a ellas. Para bien o para mal esta nueva industria requiere de mano de obra y la programación se ha vuelto una habilidad imprescindible, no sólo para los que desarrollan toda esa tecnología, sino también para cualquiera que se proponga comprender su funcionamiento.

En el momento del boom el éxito de Python fue casi inevitable, estaba destinado a ser el lenguaje que todo el mundo necesitaba esperando ser descubierto. Pero no fue solo un golpe de suerte, sino

décadas de trabajo y esfuerzo por parte de su fiel comunidad. Python fue publicado en 1991 por el neerlandés Guido van Rossum aspirando a ser un lenguaje fácil de utilizar y aprender. Si bien no logró destacar en el ámbito empresarial gozó de buena salud entre el público general y los administradores de sistemas por su facilidad para automatizar tareas. A lo largo de los años sus usuarios desarrollaron una colección de módulos impresionante que les permitía realizar casi cualquier tarea sin reinventar la rueda. Las semillas estaban plantadas, los brotes bien cuidados y el empujón final llegó con su adopción por parte de la comunidad educativa y el auge del análisis de datos. En 5 años pasó de una cuota del 3% a su máximo histórico del 17% a finales de 2022.

Resumen del tema

Esta ha sido una lección teórica repleta de historias sobre las computadoras, los procesadores, las tarjetas gráficas y los lenguajes de programación.

Variables y expresiones

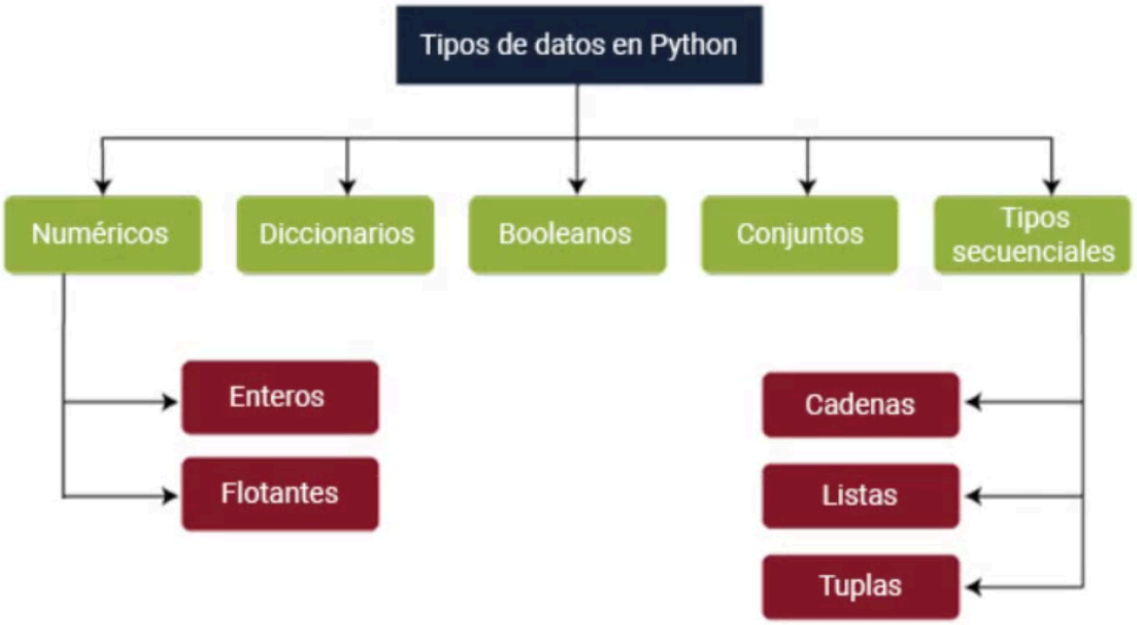
La información es poder

Desde épocas inmemoriales la información ha sido la fuente del poder. Uno de mis libros favoritos, *El Arte de La Guerra* de *Sun-Tzu*, es un tratado que explica las tácticas de un famoso estratega militar chino del siglo V antes de Cristo. *Sun-Tzu* dijo que el arte supremo de la guerra consiste en ganar la batalla sin luchar, de ahí que gran parte de sus estrategias tengan que ver con el arte del subterfugio, engañar al enemigo utilizando espías que divulguen información falsa para confundirlo mientras lo tienes controlado. No seré yo quien venga a despertar mentes, pero pensad por un momento cuál debe ser la razón de que vivamos en un bombardeo constante de información que en lugar de resolver dudas tenga el efecto contrario.

Os cuento esto porque al final la programación informática es una herramienta que sirve para manipular información, procesarla con alguna finalidad. Evidentemente a una computadora no podemos explicarle una historia, tenemos que presentarle la información de manera que pueda entenderla y eso significa descomponerla en datos.

Datos literales

En la programación existen diferentes tipos de datos dependiendo de la información que necesitamos representar. Por regla general hay unos tipos comunes para todos los lenguajes, aunque cada uno tiene sus peculiaridades. Sin entrar en detalle, en Python encontramos los siguientes:



Es necesario diferenciar entre tipos de datos porque cada uno tiene sus peculiaridades. Por ejemplo, los números se pueden operar entre ellos pero como podemos suponer sumar un número y un texto no funcionará porque no es lógico.

Lo bueno que tiene Python es que es un lenguaje de tipado dinámico orientado a objetos capaz de manejar la memoria automáticamente. Más adelante aprenderemos qué son los objetos, por ahora aprendamos más sobre ese concepto llamado tipado dinámico haciendo un pequeño experimento.

En un nuevo script *03-datos.py*, vamos a imprimir diferentes tipos de datos. No os preocupéis si no entendéis algo, el objetivo es dejar claro que Python saben diferenciarlos automáticamente:

```
"""
Fichero: 03-datos.py
"""

print("Hola mundo")
print(123)
print(-123)
print(32.64)
print(True)
print([1, 2, 3])
print((1, 2, 3))
print({1, 2, 3})
print({'n': 123})
```

Cada uno de estos *print* contiene un tipo de dato diferente. Son datos definidos y utilizados en la misma línea, como no se almacenan en la memoria se les llaman datos literales.

El primero ya lo conocemos, es una cadena de caracteres. ¿Y el segundo? Está claro que se trata de un número. ¿Pero qué tipo de número? Debe ser entero porque no tiene parte decimal, ¿no?. ¿Y el -123? Este también es un número pero es negativo. ¿Será un entero o tendrá otro tipo? ¿Y el 32.64? ¿Por qué lleva un punto en medio¿ ¿Será un número decimal? ¿En ese caso porque lleva un punto y no una coma? ¡Madre mía que lío!

Tranquilidad, se supone que Python es un lenguaje capaz de determinar automáticamente el tipo de un dato. Veamos si es cierto ayudándonos de la función *type*:

```
print(type("Hola mundo"))  # <class 'str'>
print(type(123))           # <class 'int'>
print(type(-123))          # <class 'int'>
print(type(32.64))         # <class 'float'>
print(type(True))          # <class 'bool'>
print(type([1, 2, 3]))     # <class 'list'>
print(type((1, 2, 3)))     # <class 'tuple'>
print(type({1, 2, 3}))     # <class 'set'>
print(type({'n': 123}))    # <class 'dict'>
```

Ahí los tenemos, todos los tipos de datos en Python: *string, integer, float, bool, list, tuple, set y dict*.

Ya que estamos podemos comprobar cuánto ocupa cada tipo en la memoria. Para ello importaremos el módulo de sistema *sys*, éste contiene una función llamada *getsizeof* que nos lo dirá en bytes:

```
import sys

print(type("Hola mundo"), sys.getsizeof("Hola mundo"))  # 59
print(type(123), sys.getsizeof(123))                    # 28
print(type(-123), sys.getsizeof(-123))                  # 28
print(type(32.64), sys.getsizeof(32.64))                # 24
```

```
print(type(True), sys.getsizeof(True))                  # 28
print(type([1, 2, 3]), sys.getsizeof([1, 2, 3]))        # 88
print(type((1, 2, 3)), sys.getsizeof((1, 2, 3)))        # 64
print(type({1, 2, 3}), sys.getsizeof({1, 2, 3}))        # 216
print(type({'n': 123}), sys.getsizeof({'n': 123}))      # 184
```

Dependiendo de la complejidad del tipo de dato ocupará más o menos en la memoria. Pero un momento… Si habéis programado en otros lenguajes notaréis algo raro. ¿Como es que el número *123* ocupa 28 bytes? ¿No es demasiado? *123* en binario es *0111 1011*, son 8 bits y por tanto ocupa 1 byte en la memoria. ¿Por qué se necesitan 28? ¿Y por qué un número flotante como el 32.64 que parece más complejo ocupa solo 24? Todo esto es debido a lo que os expliqué al principio de que Python es un lenguaje orientado a objetos. Los objetos son un tipo de dato sofisticado capaz de manejar la memoria automáticamente y albergar sus propias funcionalidades únicas.

Por ejemplo, los objetos de tipo ***string*** tienen una función interna especial llamada ***__len__*** que es capaz de contar el número de caracteres que la forman:

```
print("Hola mundo".__len__())  # 10
```

Llamarla de esta forma es poco práctico, por eso Python tiene una función global llamada ***len*** para simplificar la llamada:

```
print("Hola mundo".__len__(), len("Hola mundo"))  # 10 10
```

Pero ya llegaremos a ello, no debemos adelantarnos a los fundamentos. Poco a poco aprenderemos a utilizar la mayoría de estos tipos y sus funciones internas, por ahora vamos a enfocarnos en trabajar con los números y las cadenas de caracteres.

Memoria y variables

Si queremos que la computadora manipule datos es imprescindible que los almacene en algún lugar. Existen dos sitios donde puede guardarlos: la memoria RAM y el disco duro. La memoria RAM es una memoria volátil de rápida transferencia que almacena datos para que el procesador los utilice en tiempo real, mientras que un disco duro es más lento pero permite guardar la información de forma persistente. Es decir, los datos de la RAM se borran al apagar la computadora pero el disco duro no.

En el año 2000 una memoria RAM DDR-400 ofrecía un ancho de banda de hasta 3.2 GB/s y un disco duro HDD de 5.400rpm con duras penas llegaba a tasas de 0.1 GB/s. Eso hacía inviable al procesador trabajar en tiempo real con el disco porque hacía cuello de botella. Sin embargo los discos han evolucionado tanto que el procesador ya puede trabajar con ellos, aunque eso solo ocurre si ocupamos toda la memoria RAM del sistema. Por dar un dato, los discos SSD más rápidos pueden llegar a tasas de hasta 8 GB/s, eso es casi el triple que una memoria RAM de primera generación pero una cuarta parte de una memoria RAM DDR5-4000 de quinta generación. Como siempre me voy por las ramas, pero me parece información interesante que vale la pena conocer.

Así que necesitamos almacenar datos en la memoria RAM, ¿cómo lo hacemos? Pues el procedimiento es fácil, necesitamos decirle a la computadora que reserve un hueco en la memoria

para que podamos guardar los datos ahí. El problema es que la memoria RAM se maneja en pequeños sectores identificados con unos códigos numéricos muy raros. Podemos consultar el lugar donde se almacena un dato mediante la función ***id*** de Python:

```
"""
Fichero: 04-variables.py
"""

print(id("Hola mundo"))  # 2405230674096
```

Una forma común de visualizar la dirección de memoria de un dato es en formato hexadecimal, podemos transformar el decimal que nos devuelve ***id*** a hexadecimal con la función ***hex***:

```
print(hex(id("Hola mundo")))  # 0x25834b380b0
```

Este código es lo que se conoce como una referencia a la memoria. ¿Os imagináis que para reservar un espacio en la memoria y guardar un dato tuviésemos que memorizar esta referencia tan compleja? No, lo que se hace es más fácil. Simplemente daremos un nombre a ese espacio en la memoria y le asignaremos el dato que deseemos almacenar:

```
dato = "Hola mundo"
```

Una vez hemos almacenado la información en ese nombre podemos acceder a ella cómodamente usándolo en lugar del dato literal:

```
dato = "Hola mundo"
print(dato, type(dato), hex(id(dato)))  # Hola mundo <class 'str'> # 0x25834b380b0
```

Una vez tenemos el espacio reservado podemos modificar el contenido, incluso cambiando de tipo:

```
dato = "Hola mundo"
print(dato, type(dato), hex(id(dato)))  # Hola mundo <class 'str'> # 0x25834b380b0

dato = -185
print(dato, type(dato), hex(id(dato)))  # -185  <class 'int'>  0x2865914d290

dato = 2.748
print(dato, type(dato), hex(id(dato)))  # 2.748 <class 'float'> 0x2865913bb10
```

El nombre técnico que recibe este espacio en la memoria con un nombre es ***variable***, haciendo referencia precisamente a que su valor puede modificarse en cualquier momento. Son el pináculo de la programación informática y es imposible crear cualquier programa sin utilizarlas.

Por cierto, en muchos lenguajes de programación encontramos una contraparte a las variables llamadas constantes. Como su nombre indica se basan en el mismo concepto pero una vez definidas

su valor es inalterable. En Python no existen las constantes pero se puede dar a entender que una variable no debe modificarse escribiendo su nombre en mayúsculas.

Hablando de nombres, no todos sirven para definir una variable, hablemos un poco de eso.

Nombres de variables

Según las reglas de la *PEP-8*, en cuanto al nombre de una variable o una función:

- Debe comenzar con una letra o guión bajo (nunca con un número).
- Solo puede contener caracteres alfanuméricos y guiones bajos (A-z, 0-9 y _).
- Debemos evitar el uso de tildes y símbolos especiales como la ñ.
- No deben sobreescribir el nombre de una funcionalidad del lenguaje.

Además hay que tener en cuenta que:

- Se distinguen mayúsculas y minúsculas, ***Dato***, ***dato*** y ***DATO*** serían tres variables diferentes.
- Se recomienda utilizar nombres autoexplicativos, claros y concisos.

Ejemplos de buenos nombres podrían ser:

- velocidad_media
- numero_de_plazas
- producto_1
- producto_2
- precio_final
- metros_de_alto

Otros nombres que funcionan pero no se recomiendan:

- VelocidadMedia
- numerodeplazaslibres
- Precio_Final
- campaña_navideña
- metros_por_segundo_que_cae_la_pelota_al_soltarla_desde_una_altura

Nombres que definitivamente no funcionarán y hay que evitar:

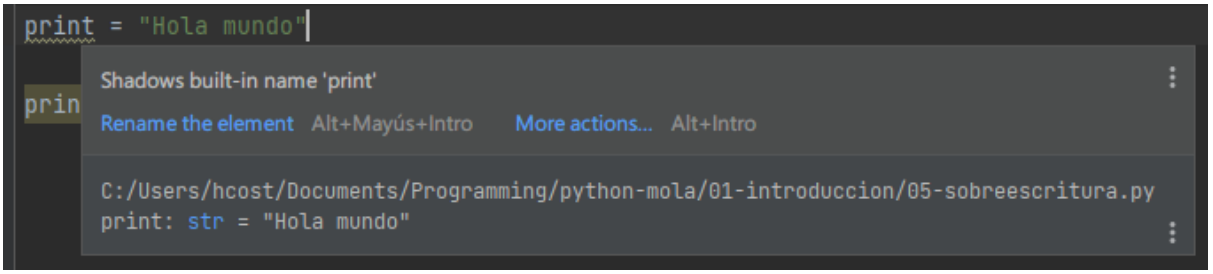
- velocidad media
- 1producto
- 2producto
- print
- type

A diferencia de otros lenguajes, en Python se da libertad al programador para sobreescribir las definiciones del lenguaje. Si las sobreescribimos perderemos su funcionalidad en el resto de la ejecución. Creo que es importante experimentar este problema:

```
"""
Fichero: 05-sobreescritura.py
```

```
"""

print = "Hola mundo"

print("Hola mundo")    # TypeError: 'str' object is not callable
```

Pycharm nos avisará si estamos sobreescribiendo una definición del lenguaje, pero otros entornos podrían no hacerlo:



Algo que sí es posible es copiar el acceso de una función existente, por ejemplo:

```
imprimir = print
imprimir("Hola mundo")
```

Podemos ir un paso más allá y eliminar la referencia a la función *print* asignándole un valor vacío:

```
imprimir = print
print = None
imprimir("Hola mundo")
```

None es un tipo especial de dato que no almacena información en su interior, sería equivalente a una dirección de memoria nula en otros lenguajes.

Esto ha sido solo un experimento para saber los efectos de sobrescribir accidentalmente una definición, en la práctica no tiene mucho sentido.

Operadores y expresiones

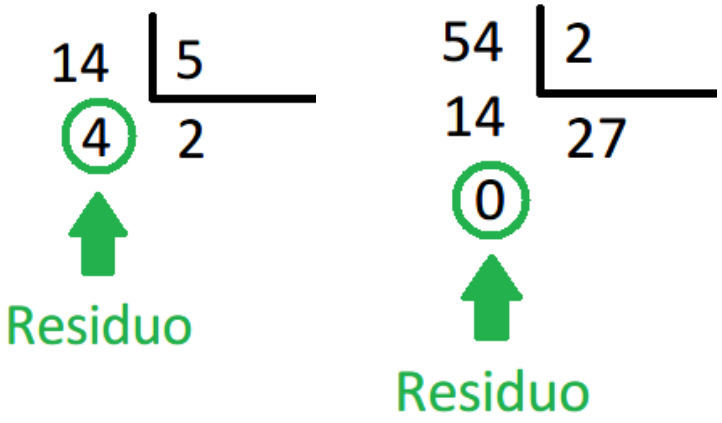
Guardar datos en la memoria está muy bien pero lo suyo es manipularlos con alguna finalidad y para conseguirlo se utilizan los operadores.

Los operadores más simples que se nos pueden ocurrir son los aritméticos: suma, resta, producto y división. Utilizarlos es muy fácil, solo tenemos que realizar la operación que necesitemos sobre los literales y variables numéricas. El resultado lo podemos simplemente mostrar en un *print* o almacenarlo en una variable para seguir trabajando:

```
"""
Fichero: 01-introduccion/06-operadores.py
"""
```

```
print("Suma =", 5 + 6)
print("Resta =", 7.5 - 13)
print("Producto =", 13 * 2.5)
print("Divisón =", 18 / 3)

numero_1 = 25
numero_2 = 47
resultado_suma = numero_1 + numero_2
print("Resultado suma =", resultado_suma)
```

También consideramos operaciones aritméticas la división entera `//`, la potencia `**` y el módulo `%`, una operación clásica de la programación que en lugar del resultado devuelve el residuo de una división entera:




```
print("División entera =", 20 // 3)
print("Potencia =", 2 ** 4)
print("Módulo =", 14 % 5)
print("Módulo =", 54 % 2)
```

Las combinaciones que forman los literales, variables, operadores y funciones en programación se conocen como expresiones. Una expresión se puede evaluar como un único valor y se interpreta de acuerdo a las normas de precedencia y asociación del lenguaje.

Precedencia y asociación

En aritmética, de izquierda a derecha las potencias y las raíces tienen prioridad, luego el producto y la división y finalmente la suma y la resta:

```
"""
Fichero: 01-introduccion/07-precedencia.py
"""

print(2 + 5**2 * 10 - 5)  # 247
```

El valor evaluado de esta expresión es 247, se calcula por este orden:

1. *Potencia*: 2 + **25** * 10 - 5
2. *Producto*: 2 + **250** - 5
3. *Suma y resta*: **247**

En algunas ocasiones quizá necesitamos modificar el orden de precedencia, por ejemplo para calcular un valor medio, basado en sumar **N** valor y dividir el resultado entre el mismo número **N**:

```
valor_1 = 10
valor_2 = 15
valor_3 = 20
valor_medio = valor_1 + valor_2 + valor_3 / 3

print("Valor medio =", valor_medio)  # Valor medio = 31.666666666666668
```

Fijaros que el problema es que la división tiene prioridad pero nosotros necesitamos sumar primero los valores. Igual que las matemáticas utilizaremos paréntesis para indicar precedencia sobre todos los demás operadores:

```
valor_1 = 10
valor_2 = 15
valor_3 = 20
valor_medio = (valor_1 + valor_2 + valor_3) / 3

print("Valor medio =", valor_medio)  # Valor medio = 15.0
```

En la programación no existe el concepto de yuxtaposición como en las matemáticas, no podemos simplemente omitir operaciones. Expresiones como *5x* y *3(2+5)* no funcionarán, debemos escribir el operador explícitamente: *5*x* y *3 * (2+5)*.

Operadores en asignación

En algunas ocasiones quizá nos interesa realizar un cálculo sobre la propia variable sin tener que definir otra para almacenar el resultado. En estos casos utilizaremos la propia variable como operando de la operación deseada, por ejemplo para realizar un incremento de un valor:

```
"""
Fichero: 01-introduccion/08-asignacion.py
"""

resultado = 0

resultado = resultado + 5
print("+5 =", resultado)

resultado = resultado * 2
print("*2 =", resultado)

resultado = resultado // 3
print("//3 =", resultado)
```

Cuando hacemos una operación sobre la propia variable podemos utilizar operadores en asignación para ahorrarnos tiempo, el resultado es el mismo:

```
"""
Fichero: 01-introduccion/08-asignacion.py
"""

resultado = 0

resultado += 5
print("+5 =", resultado)

resultado *= 2
print("*2 =", resultado)

resultado //= 3
print("//3 =", resultado)
```

Recordad que en programación el signo `=` no es un operador algebráico sino un operador de asignación para establecer un valor en una variable. Algo como `2 * x = 3 + 5` nunca funcionará porque el intérprete espera el nombre de la variable a la izquierda de la asignación.

Expresiones en funciones

Cuando una expresión se evalúa con el mismo tipo que el argumento requerido por una función podemos utilizarla directamente. Esto es muy conveniente por ejemplo al realizar configuraciones, ya que podemos definir una serie de valores al principio del programa y utilizarlos de forma flexible.

Veamos cómo adaptar un ejemplo de dibujado básico:

```
"""
Fichero: 01-introduccion/09-ejemplo.py
"""

import arcade

# Constantes con valores de referencia
VENTANA_ANCHO = 600
VENTANA_ALTO = 400
VENTANA_TITULO = "Dibujos gráficos"
VENTANA_COLOR = arcade.csscolor.SKY_BLUE

# Creamos la ventana
arcade.open_window(VENTANA_ANCHO, VENTANA_ALTO, VENTANA_TITULO)

# Configuramos el color de la ventana
arcade.set_background_color(VENTANA_COLOR)

# Iniciamos el dibujado
arcade.start_render()
```

```
# Dibujamos un círculo en el centro usando expresiones
arcade.draw_circle_filled(
    VENTANA_ANCHO / 2, VENTANA_ALTO / 2, 100, arcade.color.DARK_ORANGE)

# Finalizamos el dibujado
arcade.finish_render()

# Mantenemos la ventana abierta hasta cerrarla
arcade.run()
```

Impresión de variables

Sobre la marcha hemos visto que la función *print* permite imprimir literales y variables en una misma línea separando los datos con comas:

```
"""
Fichero: 01-introduccion/10-impresion.py
"""

nombre = "Manolo"
print("Buenos días", nombre)  # Buenos días Manolo
```

Esta forma de imprimir tiene un pequeño problema, siempre se imprime un espacio como separador y si quisiéramos utilizar un signo de puntuación al final no se vería bien:

```
print("Buenos días", nombre, ".")  # Buenos días Manolo .
```

Para solucionarlo lo que se solía hacer es sumar cadenas entre ellas. ¿Sumas cadenas? Así es, para Python sumar cadenas equivale a tener una sola cadena con los caracteres de todas ellas. De esa forma se evalúa un único dato en la función *print*:

```
print("Buenos días" + nombre + ".")  # Buenos díasManolo.
```

El problema como vemos es que ahora los espacios debemos escribirlos nosotros:

```
print("Buenos días " + nombre + ".")  # Buenos días Manolo.
```

Y otro problema que tiene esta forma es que no podemos sumar cadenas y números:

```
nombre = "Manolo"
edad = 47
print(nombre + " nació en " + (2022 - edad))  # Error
```

Es necesario transformar los números a cadenas creando un nuevo tipo de dato *string* haciendo uso de la función *str*:

```
nombre = "Manolo"
```

```
edad = 47
print(nombre + " nació en " + str(2022 - edad))  # Manolo nació en 1975
```

Pero vosotros no tenéis que saber todo esto porque en 2016 Python añadió una nueva funcionalidad llamada ***format Strings*** que simplifica el proceso de generar cadenas.

Solo debemos indicar que la cadena contiene código interpretado en su interior poniendo una *f* delante. A partir de ese momento todos los valores entre llaves *{}* serán interpretados y sumados a la cadena automáticamente:

```
nombre = "Manolo"
edad = 47
print(f"{nombre} nació en {2022 - edad}")  # Manolo nació en 1975
```

Las ***f-Strings*** facilitan mucho el trabajo con cadenas y estoy seguro que las utilizaréis todo el tiempo.

Resumen del tema

A lo largo de esta lección hemos aprendido a trabajar con datos en la programación. Hemos introducido los datos, sus tipos esenciales y como se almacenan en la memoria utilizando las variables. También hemos introducido el uso de los operadores aritméticos para realizar cálculos y sus formas en asignación para operar directamente en las propias variables.

Funciones de código

Utilidad de las funciones

En la lección anterior aprendimos a crear variables y usarlas en expresiones. En ésta vamos a utilizar ese conocimiento para crear nuestras propias funciones, grupos de instrucciones identificadas bajo un nombre y que la computadora ejecuta de forma ordenada.

El uso de las funciones es imprescindible en la programación porque permite dividir los problemas complejos en otros más simples de forma natural, mejora la legibilidad y mantenimiento del código, abstrae los procesos complejos y quizá lo más importante, permite crear código reutilizable.

Funciones simples

Si bien las reglas para definir el nombre de una función son las mismas que en una variable, la forma difiere porque la función contiene múltiples instrucciones en lugar de una sola asignación. Para hacerlo se utiliza la palabra reservada *def* seguida del nombre de la función, unos paréntesis *()* y dos puntos:

```
"""
Fichero: 01-introduccion/11-nueva-funcion.py
"""

def hola():
```

Los dos puntos *(:)* de la función indican que a partir de ahí viene un bloque de código.

A diferencia de otros lenguajes donde los bloques se definen entre llaves *{}*, en Python un bloque de código se diferencia del resto por su nivel de indentación o tabulación. Es decir, su contenido empezará siempre a una distancia homogénea del resto. Esta distancia puede ir ser desde un único espacio a varias tabulaciones, siempre que todas las instrucciones respeten la misma distancia:

```
def hola():
        """ Esta función imprime hola """
        print("Hola a todos")
```

La *PEP-8* recomienda 4 espacios o una tabulación por nivel de indentación, razón por la cuál si tenemos el auto formateador activado, al guardar se establecerá esa distancia automáticamente:

```
def hola():
    """ Esta función imprime hola """
    print("Hola a todos")
```

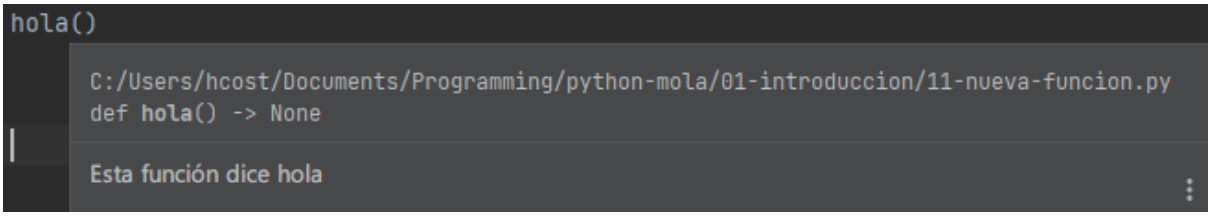
Por cierto, fijaros que hemos escrito como primera instrucción un comentario multilínea, Python es capaz de generar la documentación de una función a partir del comentario de su primera línea, esta funcionalidad se conoce como *docstings* (cadenas de documentación).

Muy bien, con esto hemos definido nuestra primera función. ¿Cómo la ejecutamos? Pues igual que *print* o cualquier otra, a través de su nombre y con unos paréntesis:

```python
def hola():
    """ Esta función imprime hola """
    print("Hola a todos")

hola()  # Hola a todos
```

Es muy importante volver al bloque de código principal para ejecutarla, sin ningún espacio delante, al mismo nivel de tabulación que la definición de la función.

Como curiosidad fijaros cómo PyCharm muestra el contenido del docstring al poner el cursor sobre el nombre de la función:



Esto nos puede servir para consultar también el tipo de argumentos o de retorno de una función. En este caso el retorno se indica mediante la flecha y el tipo *None*, haciendo referencia a que no se devuelve nada.

En cualquier caso hemos dicho que una de las ventajas de las funciones es su reusabilidad, nada nos impediría ejecutar múltiples veces la misma función:

```python
hola()  # Hola a todos
hola()  # Hola a todos
hola()  # Hola a todos
```

¿Y qué ocurrirá si no indicamos los paréntesis? Pues simplemente haremos referencia a ella sin ejecutarla. De hecho fijaros qué aparece en la pantalla si pasamos la función hola a un *print*:

```python
print(hola)  # <function hola at 0x0000027683F704A0>
```

¿Os suena de algo? Nos está diciendo en qué lugar de la memoria se encuentra almacenada la definición de la función. Por eso si asignamos a una nueva variable la función *hola* en lugar de su ejecución mediante los paréntesis estaremos creando otro acceso a ese lugar de la memoria:

```python
decir_hola = hola
decir_hola()
```

Luego volveremos a este concepto, por ahora termines la lección creando una nueva función:

```python
"""
Fichero: 01-introduccion/11-nueva-funcion.py
"""
```

```
def hola():
    """ Esta función dice hola """
    print("Hola a todos ")


def adios():
    """ Esta función dice adiós """
    print("Adiós a todos ")


hola()   # Hola a todos
adios()  # Adiós a todos
```

Como véis podemos definir múltiples funciones y hacer uso de ellas de forma muy simple.

Por cierto, si necesitáis definir una función a modo de recordatorio sin programarla en ese momento, podemos utilizar la instrucción *pass* para dejarla sin nada. Los bloques en Python deben tener como mínimo una instrucción y ésta sirve precisamente para pasar de largo :

```
def funcion():
    pass
```

Función principal

Es muy común en la programación tener una función principal desde donde empezar a ejecutar el código de un programa. Podríamos adaptar el ejemplo anterior a este nuevo concepto:

```
"""
Fichero: 01-introduccion/12-funcion-principal.py
"""


def hola():
    """ Esta función dice hola """
    print("Hola a todos ")


def adios():
    """ Esta función dice adiós """
    print("Adiós a todos ")


def main():
    """ Esta es la función principal """
    hola()
    adios()


main()
```

La mayoría de lenguajes de programación busca dentro del código fuente una función llamada *main* para empezar la ejecución desde ahí. En lugar de una función, Python considera como fichero principal del programa el script que empieza la ejecución, ese que se pasa al intérprete.

Podemos comprobar el nombre del fichero en la memoria del programa imprimiendo su atributo especial *__name__*:

```
print(__name__)  # __main__
```

Para garantizar que el código de un fichero solo se pueda ejecutar desde el propio fichero a menudo encontraréis una cláusula como la siguiente:

```
# Ejecutar la función principal sólo desde el fichero principal
if __name__ == "__main__":
    main()
```

Dentro de un par de unidades aprenderemos qué es y cómo funciona esa instrucción llamada *if*, por ahora quedaros con la idea de la importancia de tener un fichero principal en vuestros programas.

Parámetros de entrada

Las funciones dan mucho más juego si las alimentamos con información. Por ejemplo, la función *print* recibe uno o varios datos para imprimirlos. Esos datos se conocen como argumentos durante el envío y parámetros durante la definición.

Los parámetros se definen entre los paréntesis y equivalen a variables que toman su valor de los argumentos al llamarse la función:

```
"""
Fichero: 01-introduccion/13-parametros.py
"""

def describir(nombre, edad):
   print(f"{nombre} tiene {edad} años.")

describir("Ana", 31)     # Ana tiene 31 años.
describir("Víctor", 46)  # Víctor tiene 46 años.
```

Debemos tener en cuenta un que el valor de los parámetros se toma por orden, si cambiamos su orden también debemos cambiar el orden de los argumentos:

```
"""
Fichero: 01-introduccion/13-parametros.py
"""

def describir(edad, nombre):
   print(f"{nombre} tiene {edad} años.")
```

```
describir(31, "Ana")     # Ana tiene 31 años.
describir(46, "Víctor")  # Víctor tiene 46 años.
```

Sin embargo Python permite especificar el nombre de los argumentos durante la llamada, de esa forma podemos pasar los datos en el orden que nos convenga:

```
describir(nombre="Ana", edad=31)     # Ana tiene 31 años.
describir(edad=46, nombre="Víctor")  # Víctor tiene 46 años.
```

Ámbito de los parámetros

Otro concepto importante es el del ámbito, este nos dice que los parámetros y variables abarcan únicamente su bloque de código. Algo como lo siguiente no funcionará:

```
def prueba_de_ambito():
    texto = "Python mola"

print(texto)  # NameError: name 'texto' is not defined.
```

Incluso si tenemos previamente definida la variable, la que se utiliza dentro del bloque, pese a tener el mismo nombre, existe únicamente dentro de la función:

```
def prueba_de_ambito():
    texto = "Python mola"

texto = "Hola mundo"
prueba_de_ambito()
print(texto)  # Hola mundo
```

Sin embargo sí sería posible utilizar una variable definida en un ámbito superior

```
texto = "Hola mundo"

def prueba_de_ambito():
    print(texto)

prueba_de_ambito()  # Hola mundo
```

Aquí Python comprueba primero si la variable *texto* existe en la función y si no es así busca en un ámbito superior, como la encuentra en el bloque principal utiliza su valor.

Esta práctica provoca código acoplado y rompe el objetivo de una función de ser independiente y escalable. Su uso es desaconsejable y os recomiendo evitarla siempre.

Parámetros por defecto

Python permite establecer valores por defecto en los parámetros en caso de que estos no se envíen a la función. Para conseguirlo debemos asignarles el valor deseado:

```
"""
Fichero: 01-introduccion/15-parametros-defecto.py
"""


def sumar(a=5, b=3):
   print(f"La suma de {a} y {b} es {a + b}")


sumar(15, 7)  # La suma de 15 y 7 es 22
```

Esto nos permitirá ejecutar cosas como pasar el primer parámetro o el segundo mediante el nombre para conseguir diferentes resultados:

```
sumar(11)    # La suma de 11 y 3 es 18
sumar(b=12)  # La suma de 5 y 12 es 17
```

Esta funcionalidad es interesante porque deja abierta la puerta a enviar parámetros opcionales y condicionar el código, algo que aprenderemos al introducir el control de flujo.

Parámetros inmutables

Hemos visto que los parámetros de una función existen únicamente en su ámbito, eso significa que si por ejemplo pasamos un número en una variable a una función y lo modificamos, el número del bloque principal seguirá siendo el mismo:

```
"""
Fichero: 01-introduccion/16-inmutabilidad.py
"""

def modificar_numero(a):
   numero *= 2
   print(f"numero = {numero }")

a = 25
modificar_numero(a)  # 50
print(a)          # a sigue siendo 25
```

Y así es, el número se copia y se dobla en el interior de la función y en el exterior no se modifica.

En Python los argumentos se envían a las funciones por asignación de referencia, eso significa que tanto la dirección a la memoria del dato enviado como del dato recibido es la misma. Podemos comprobarlo imprimiendo su posición en la memoria tanto fuera como dentro de la función:

```
def posicion_memoria(parametro):
   print(f"{parametro} DENTRO de la función se encuentra en {hex(id(parametro))}")


numero = 1234
posicion_memoria(numero)
```

```
print(f"{numero} FUERA de la función se encuentra en {hex(id(numero))}")

# 1234 DENTRO de la función se encuentra en 0x227cfd9bb50
# 1234 FUERA de la función se encuentra en  0x227cfd9bb50
```

Pero un momento… ¿Si ambos datos apuntan al mismo lugar por qué no se modifica el dato externo al doblarlo? Esto es debido a una propiedad llamada ***inmutabilidad*** y que tienen únicamente los números, las cadenas, los booleanos y las tuplas.

Esta propiedad impide cambiar su contenido porque sus referencias a la memoria se encuentran blindadas y al modificar el valor se crea en el ámbito de la función una nueva variable con el mismo nombre pero en un lugar distinto de la memoria.

Todos los demás tipos son mutables, por lo que si intentamos modificar sus valores el cambio debería afectar a la variable externa. Probemos por ejemplo con una lista, una colección de datos que veremos próximamente. Fijémonos en lo que ocurre al doblar sus elementos:

```
def modificar_lista(lista):
   lista *= 2
   posicion_memoria(lista)


numeros = [1, 2, 3]
modificar_lista(numeros)
print(f"{numeros} FUERA de la función se encuentra en {hex(id(numeros))}")

# [1, 2, 3, 1, 2, 3] DENTRO de la función se encuentra en 0x1ab8b6a3d40
# [1, 2, 3, 1, 2, 3] FUERA de la función se encuentra en  0x1ab8b6a3d40
```

Y efectivamente, la lista que inicialmente era *[1,2,3]* pasa a ser *[1,2,3,1,2,3]* tanto dentro como fuera de la función, además podemos observar que su dirección en la memoria es la misma.

Como apunte final no hay que confundir modificar un valor con redefinirlo. Estos experimentos funcionan porque estamos sobreescribiendo una variable sobre sí misma con un operador en asignación, pero una asignación directa con el operador igual siempre supondrá la creación de una nueva variable en un lugar distinto de la memoria, sea del tipo que sea:

```
def redefinir_lista(lista):
   lista = [4, 5, 6]
   posicion_memoria(lista)


numeros = [1, 2, 3]
redefinir_lista(numeros)
print(f"{numeros} FUERA de la función se encuentra en {hex(id(numeros))}")

# [4, 5, 6] DENTRO de la función se encuentra en 0x1ff1f713500
# [1, 2, 3] FUERA de la función se encuentra en  0x1ff1da73d40
```

Quizá no lo parezca pero esta lección es muy importante y da luz a algunas cuestiones clave del tratamiento de la información en Python. No dudéis en volver a echarle un vistazo siempre que lo necesitéis.

Valores de retorno

La clave de las funciones no es sólo recibir datos sino también devolverlos para ser utilizados por otros procesos. Sin ir más lejos, hemos utilizado funciones como *type*, *id* y *hex* y hemos imprimido sus valores mediante la función *print*.

Debe quedar muy clara la diferencia entre una función con retorno y sin retorno. Por ejemplo, una función que toma dos números e imprime la suma no está devolviendo nada:

```
"""
Fichero: 01-introduccion/17-retornos.py
"""


def suma_sin_retorno(a, b):
    """ Esta suma no retorna nada """
    print(a + b)


suma_sin_retorno(10, 5)  # 15
```

Asignar el resultado de llamar una función sin retorno siempre será igual a nada, *None* en Python:

```
resultado = suma_sin_retorno(10, 5)
print("El resultado es", resultado)  # El resultado es None
```

Para devolver un dato debemos utilizar la declaración *return* seguida de un espacio y el dato a devolver.

En el caso de la suma podemos hacerlo de dos formas, por ejemplo guardando la suma en una variable para devolverla:

```
def suma_con_retorno(a, b):
    """ Esta suma retorna un valor """
    resultado = a + b
    return resultado


resultado = suma_con_retorno(12, 9)
print("El resultado es", resultado)  # El resultado es 21
```

La otra manera es devolver directamente la expresión de la suma:

```
def suma_con_retorno(a, b):
```

```
    """ Esta suma retorna un valor """
    return a + b

resultado = suma_con_retorno(12, 9)
print("El resultado es", resultado)  # El resultado es 21
```

Ambas son correctas pero la segunda está más optimizada, ahorra cálculos y uso de la memoria.

Sobra decir que el tipo de la variable que almacena el valor será el mismo que el de la variable o expresión retornada por la función.

Anotaciones de tipado

En la versión de Python 3.5 se añadió la posibilidad de establecer anotaciones de tipo tanto para los parámetros de entrada como para los valores de retorno de una función. En la documentación se deja muy claro que Python no impone estas anotaciones de tipado, pero entornos como *PyCharm* pueden hacer uso de ellas para avisarnos en caso de no cumplirlas. Veamos como se aplican a los ejemplos de *suma_sin_retorno* y *suma_con_retorno*.

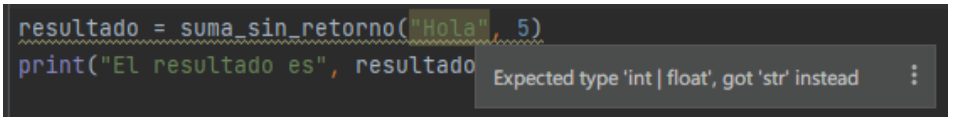
```
def suma_sin_retorno(a: int, b: int) -> None:
    """ Esta suma no retorna nada """
    print(a + b)
```

Como véis la sugerencia para el tipo de los parámetros se escribe después de dos puntos haciendo referencia al nombre del tipo. Recordad que podéis comprobar el tipo de un dato con la función *type*. En cuanto al retorno se indica con una flecha antes de los dos puntos, como no devolvemos nada el tipo *None*.

Si queremos anotar múltiples tipos, en versiones de Python 3.10 o mayor podemos usar una tubería:

```
def suma_sin_retorno(a: int | float, b: int | float) -> None:
    """ Esta suma no retorna nada """
    print(a + b)
```

Pasar un dato distinto a estos nos mostrará un aviso en el editor, pero no nos impedirá ejecutarlo:



La otra función con retorno podría quedar así:

```
def suma_con_retorno(a: int | float, b: int | float) -> int | float:
    """ Esta suma retorna un valor """
    return a + b
```

Las variables también permiten la anotación de tipo, yo no estoy acostumbrado a utilizarla:

```
resultado: int | float = suma_con_retorno(12, 9)
```

En esta tabla tenéis los identificadores de los tipos más comunes:

int	Número entero
float	Número decimal
str	Cadena de caracteres
bool	Booleano
list	Lista
tuple	Tupla
dict	Diccionario
None	Nada

Funciones estándar

Python incorpora una colección de funciones llamadas *Built-in*. De ella forman parte las funciones que ya conocemos como *print*, *id*, *hex* y *type*. También incluye constructores de datos básicos como *int*, *float*, *str*, *dict*… Accesores de métodos como *len* y funciones matemáticas para redondeos, sumatorios, potencias, filtros y muchas más.

La lista completa y su funcionamiento se puede encontrar en la documentación oficial de Python. Nosotros utilizaremos algunas más a lo largo del curso pero hay una que vale la pena aprender lo antes posible, me refiero a la función de lectura *input*.

Función de lectura

La función *input* es la contraparte de *print*. Una sirve para imprimir datos en la pantalla y la otra nos permite leerlos del teclado. Es por así decirlo la forma más básica de interactuar con un programa informático. Funciona almacenando un *buffer* o *memoria* de caracteres tomados del teclado que termina con el carácter de retorno:

```
"""
Fichero: 01-introduccion/18-input.py
"""

texto_del_teclado = input()
print("El texto escrito es:", texto_del_teclado)

>>> Hola buenos días
>>> El texto escrito es: Hola buenos días
```

Esta función puede tomar una cadena para imprimirla antes del buffer de lectura:

```
nombre = input("Escribe tu nombre: ")
print("Hola", nombre)

>>> Escribe tu nombre: Héctor
>>> Hola Héctor
```

Dado que el tipo de captura es siempre una cadena de caracteres no es posible operar aritméticamente los datos tomados del teclado:

```
ano_nacimiento = input("¿En qué año naciste? ")
print(f"Entonces debes tener unos {2023 - ano_nacimiento} años")

>>> ¿En qué año naciste? 1989
TypeError: unsupported operand type(s) for -: 'int' and 'str
```

Aquí entran en juego las funciones constructoras de datos que os comenté anteriormente. Con ellas es posible realizar conversiones de texto a número mediante *int* y *float* o viceversa, de número a texto con *str*:

```
ano_nacimiento = input("¿En qué año naciste? ")
ano_nacimiento = int(ano_nacimiento )
print(f"Entonces debes tener unos {2023 - ano_nacimiento} años")

>>> ¿En qué año naciste? 1989
>>> Entonces debes tener unos 34 años
```

El problema de estas funciones que si no reciben el dato esperado, en este caso un número, dan error al realizar la conversión de tipo:

```
>>> ¿En qué año naciste? No me acuerdo
ValueError: invalid literal for int() with base 10: 'No me acuerdo'
```

Esto se puede resolver mediante la captura y tratamiento de los errores de código, algo que aprenderemos al final de la próxima unidad.

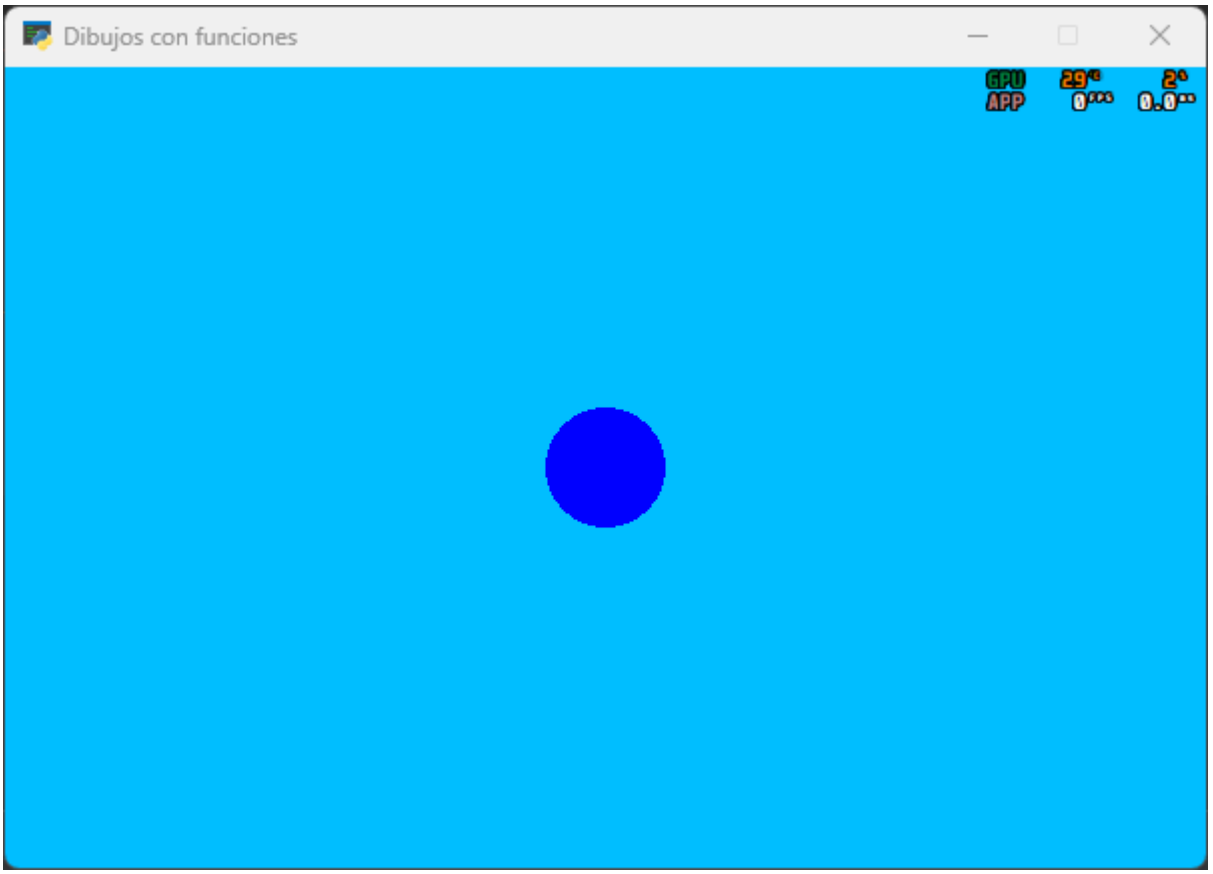
Dibujos con funciones

Para acabar el tema veamos como aplicar lo aprendido en los dibujos gráficos, donde el uso de las funciones nos servirá para organizar mejor el código y reutilizarlo.

Partiremos de una estructura básica con un círculo en el centro utilizando las funciones que ya conocemos del módulo arcade:

```
"""
Fichero: 01-introduccion/19-dibujos-funciones.py
"""

import arcade
```

```
SCREEN_WIDTH = 600
SCREEN_HEIGHT = 400

# Configurar la ventana e iniciar el renderizado
arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Dibujos con funciones")
arcade.set_background_color(arcade.csscolor.DEEP_SKY_BLUE)
arcade.start_render()

# Funciones de dibujado
arcade.draw_circle_filled(300, 200, 30, arcade.csscolor.BLUE)

# Finalizar el renderizado y ejecutar el programa
arcade.finish_render()
arcade.run()
```



Lo primero que podemos hacer es crear una función principal y a ejecutarla con la cláusula que aprendimos, esto es por buenas prácticas así que os recomiendo hacerlo en vuestros programas:

```
import arcade

SCREEN_WIDTH = 600
SCREEN_HEIGHT = 400

def main():
```

```
    # Configurar la ventana e iniciar el renderizado
    arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Dibujos con funciones")
    rcade.set_background_color(arcade.csscolor.DEEP_SKY_BLUE)
    arcade.start_render()

    # Funciones de dibujado
    arcade.draw_circle_filled(300, 200, 30, arcade.csscolor.BLUE))

    # Finalizar el renderizado y ejecutar el programa
    arcade.finish_render()
    arcade.run()


if __name__ == "__main__":
    main()
```

El siguiente paso podría ser extraer la función que dibuja el círculo en nuestra propia función:

```
def dibujar_circulo():
    """ Esta función dibuja un círculo personalizado """
    arcade.draw_circle_filled(300, 200, 30, arcade.csscolor.BLUE)
```

La ventaja de hacer esto es que podemos programar unos parámetros en nuestra función para dibujar de forma simplificada los círculos en un lugar de la ventana:

```
def dibujar_circulo(x, y, radio, color):
    """ Esta función dibuja un círculo personalizado """
    arcade.draw_circle_filled(x, y, radio, color)

~~~~
dibujar_circulo(300, 200, 30, arcade.csscolor.BLUE)
```

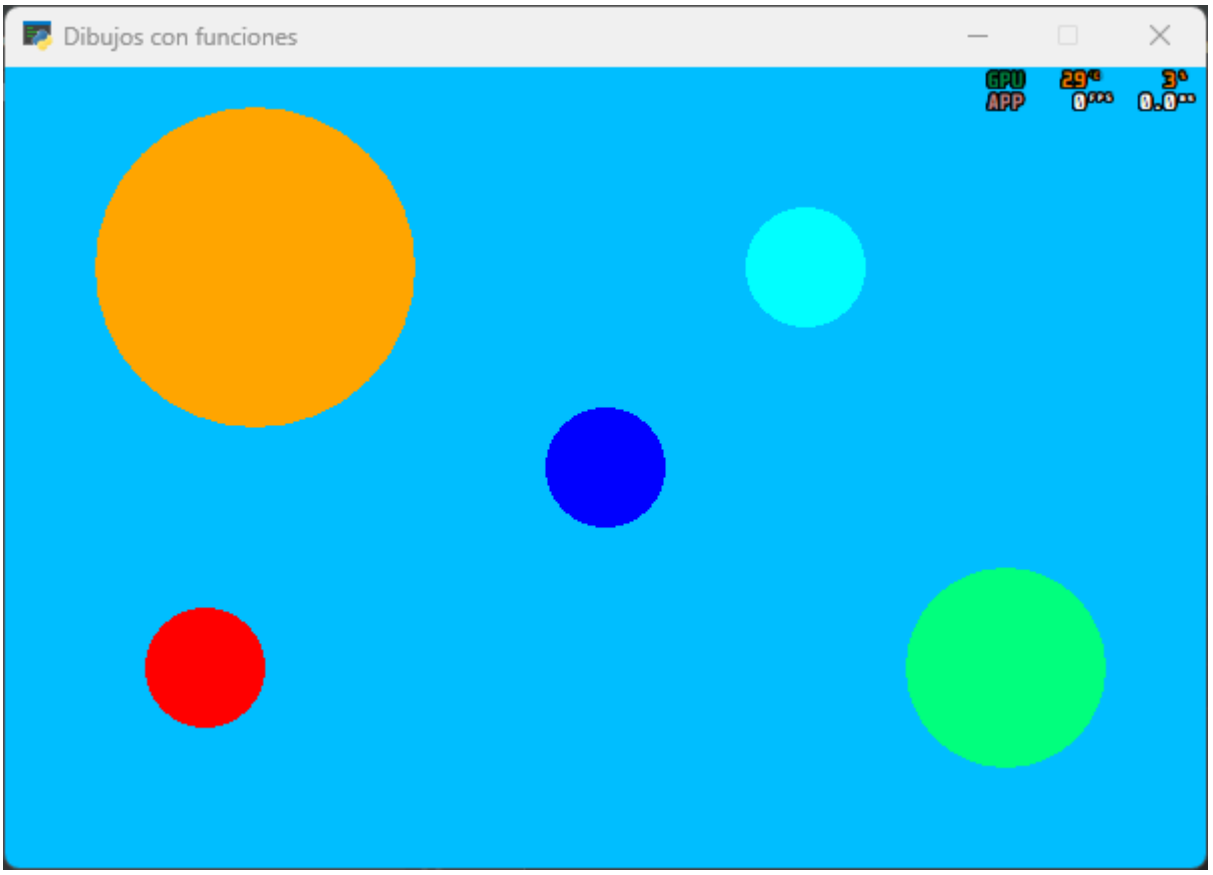
Sabiendo que podemos utilizar valores por defecto, el radio y el color podrían ser optativos:

```
def dibujar_circulo(x, y, radio=30, color=arcade.csscolor.BLUE):
    """ Esta función dibuja un círculo personalizado """
    arcade.draw_circle_filled(x, y, radio, color)

~~~~
dibujar_circulo(300, 200)
```

Así hemos simplificado el proceso de dibujar círculos, pudiendo establecer o no el radio y el color:

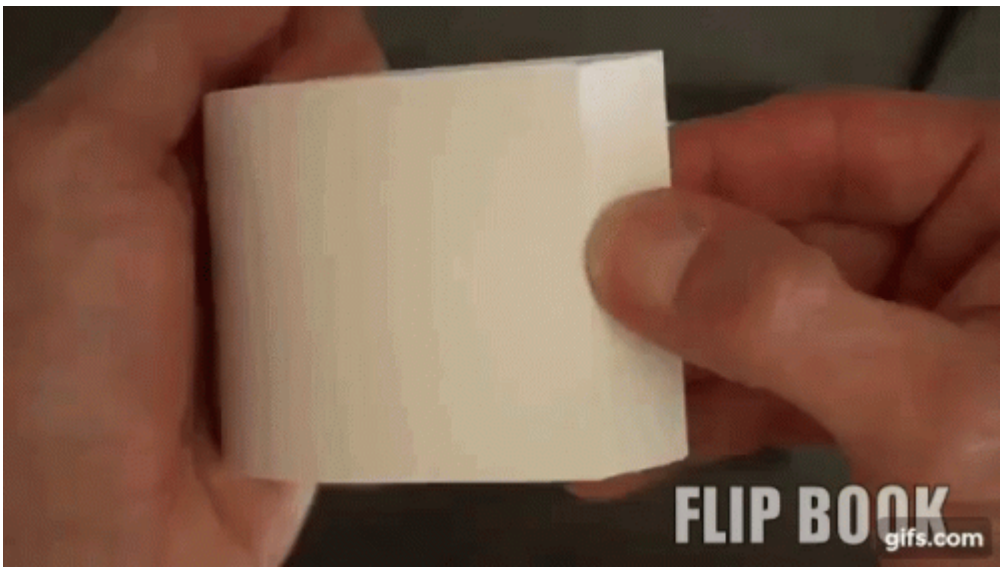
```
def main():
    # Configurar la ventana e iniciar el renderizado
    arcade.open_window(600, 400, "Dibujos con funciones")
    arcade.set_background_color(arcade.csscolor.DEEP_SKY_BLUE)
    arcade.start_render()
```

```python
# Funciones de dibujado
dibujar_circulo(300, 200)
dibujar_circulo(125, 300, color=arcade.csscolor.ORANGE, radio=80)
dibujar_circulo(100, 100, color=(255, 0, 0))
dibujar_circulo(400, 300, color=(0, 255, 255))
dibujar_circulo(500, 100, color=(0, 255, 125), radio=50)

# Finalizar el renderizado y ejecutar el programa
arcade.finish_render()
arcade.run()
```



Animaciones con funciones

Ahora que sabemos trabajar con funciones podemos extender la funcionalidad de arcade pensada para realizar animaciones. ¿Sabéis como funcionan las animaciones? Se trata de un truco para generar ilusión de movimiento en nuestro cerebro. Si una sucesión de imágenes con un patrón común se hace lo suficientemente rápido la percibimos como algo contínuo:



Arcade tiene una función llamada schedule que podríamos traducir como “cronometrar” o “programar” y que permite ejecutar en bucle otra función un número determinado de veces por segundo. Veamos cómo utilizarla a partir del código anterior:

```
"""
Fichero: 01-introduccion/20-primera-animacion.py
"""

import arcade

SCREEN_WIDTH = 600
SCREEN_HEIGHT = 400


def dibujar_circulo(x, y, radio=30, color=arcade.csscolor.BLUE):
    """ Esta función dibuja un círculo personalizado """
    arcade.draw_circle_filled(x, y, radio, color)


def dibujar():
    # Funciones de dibujado
    arcade.start_render()
    dibujar_circulo(300, 200)


def main():
    # Configurar la ventana e iniciar el renderizado
    arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Primera animación")
    arcade.set_background_color(arcade.csscolor.DEEP_SKY_BLUE)

    # Programar el renderizado y ejecutar el programa
    arcade.schedule(dibujar)
    arcade.run()
```

```
if __name__ == "__main__":
    main()
```

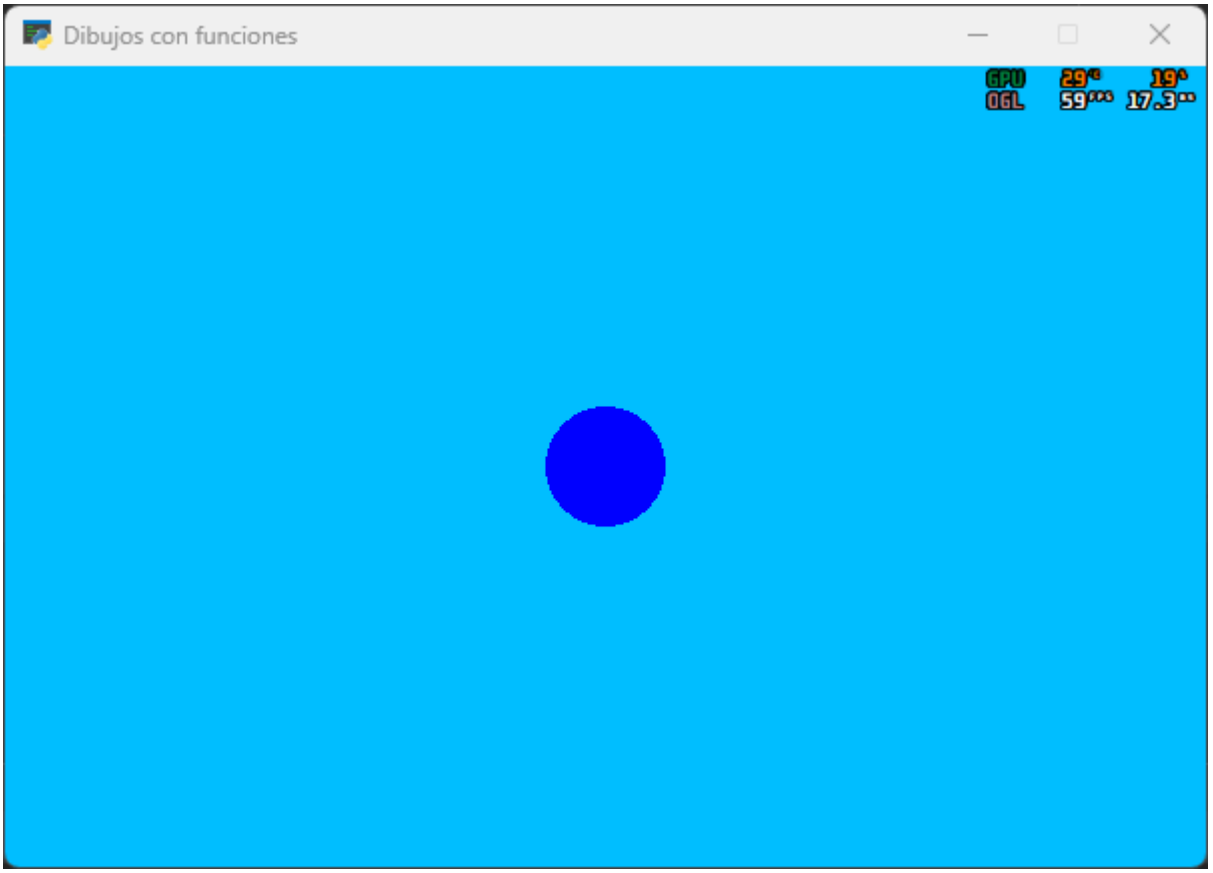
Como vemos, en lugar de finalizar el dibujado con *finish_render*, lo que hacemos es volver a ejecutar todo el proceso de dibujado en bucle. Pero si ejecutamos esto nos dará un error:

```
TypeError: schedule() missing 1 required positional argument: 'interval'
```

Nos está pidiendo un argumento llamado intervalo en la función *schedule*. Este intervalo es el tiempo en segundos entre cada proceso de renderizado. Por ejemplo, suponiendo que deseamos dibujar 60 veces nuestro lienzo en un segundo, debemos calcular la fracción de 1/60. Esos 0,016 segundos es el tiempo que deseamos esperar antes de redibujar todo el lienzo. Así que pasaremos ese intervalo de 1/60 segundos y también lo recibiremos en la función dibujar porque Arcade lo necesita para realizar algunas tareas internas:

```
def dibujar(intervalo):
    # Funciones de dibujado
    arcade.start_render()
    dibujar_circulo(300, 200)


def main():
    # Configurar la ventana e iniciar el renderizado
    arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Primera animación")
    arcade.set_background_color(arcade.csscolor.DEEP_SKY_BLUE)

    # Programar el renderizado cada 1/60 segundos y ejecutar el programa
    arcade.schedule(dibujar, 1/60)
    arcade.run()


if __name__ == "__main__":
    main()
```

Si ejecutamos de nuevo el programa aparecerá lo mismo en la ventana:



La gran diferencia es que ahora estamos dibujando la ventana en bucle de 60 ciclos por segundo, o lo que es lo mismo, estamos generando 60 fotogramas por segundo.

¿Qué podemos hacer ahora? Pues añadir dinamismo al proceso, por ejemplo, cada vez que se ejecuta la función dibujar podríamos incrementar el tamaño del radio de nuestro círculo. Para ello necesitamos una variable donde guardar un número que podamos ir incrementando en cada ciclo. ¿Dónde podemos definirla? Python nos permite definir unas variables especiales para una función llamadas atributos. Estos atributos no se reinician cada vez que ejecutamos la función, sino que son comunes a las distintas ejecuciones y lo mejor es que se pueden definir desde el exterior de la función. Un ejemplo vale más que mil palabras, veamos como se hace:

```
def dibujar(intervalo):
    # Funciones de dibujado
    arcade.start_render()
    dibujar_circulo(300, 200)

# Creamos un atributo para la función después de crearla
dibujar.radio_circulo = 1
```

Ahora el truco consiste en decirle a la función que utilice el atributo en lugar del radio por defecto:

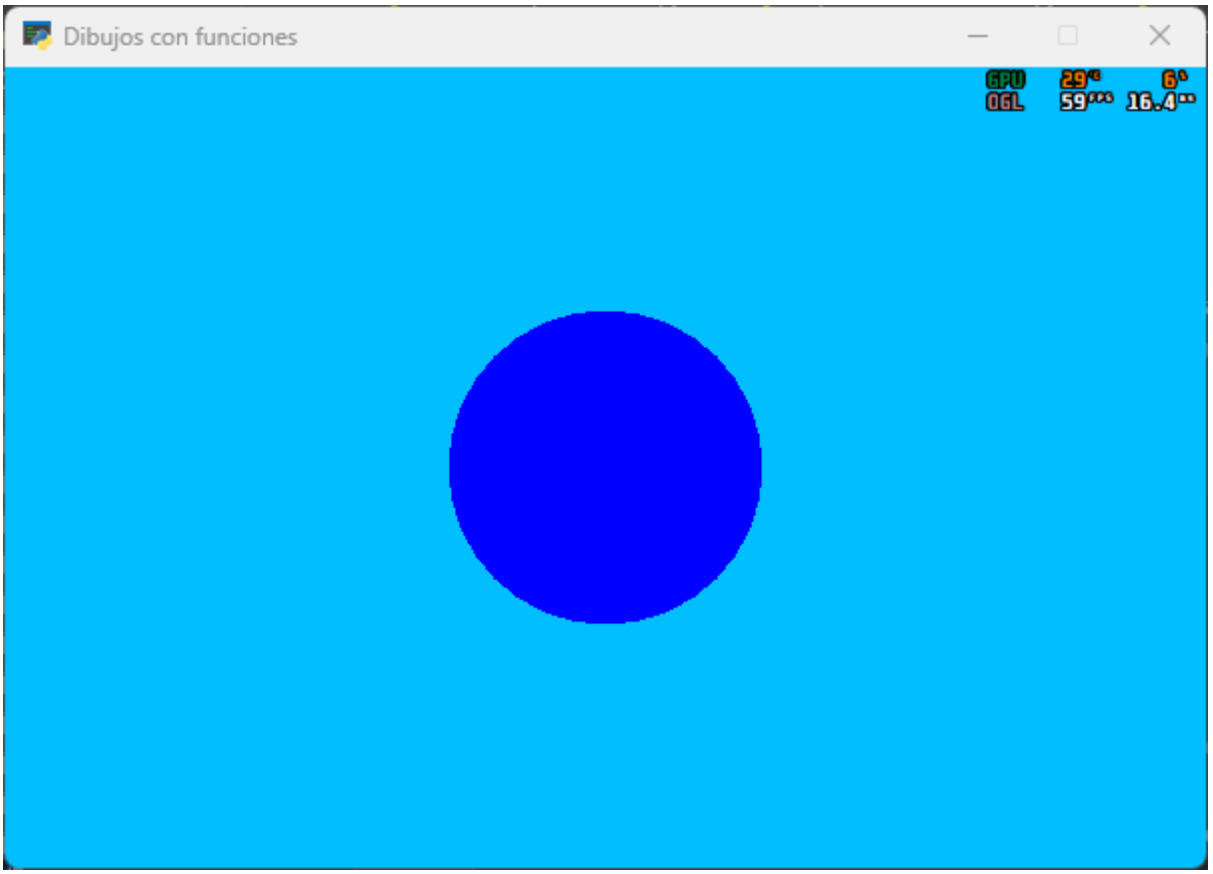
```
def dibujar(intervalo):
    # Funciones de dibujado
    arcade.start_render()
```

```
    dibujar_circulo(300, 200, radio=dibujar.radio_circulo)

# Creamos un atributo para la función después de crearla
dibujar.radio_circulo = 1
```

Solo falta decirle a que incremente brevemente el tamaño del radio después de cada ciclo de dibujo:

```
def dibujar(intervalo):
    # Funciones de dibujado
    arcade.start_render()
    dibujar_circulo(300, 200, radio=dibujar.radio_circulo)
    # Incrementamos ligeramente el radio después de cada ciclo de dibujo
    dibujar.radio_circulo += 0.5
```

¡Y eureka!



¡Felicidades, hemos creado nuestra primera animación haciendo crecer un círculo!

No os sintáis abrumados, lo último que pretendo es que memoricéis todas estas funciones. Lo que debéis entender es cómo hemos pasado de dibujar un círculo estático a obtener una animación gracias a repetir el proceso de dibujado y alterando ligeramente alguna de las propiedades del objeto del dibujo, como sería el radio del círculo.

No hay más misterio, este es el funcionamiento detrás de los videojuegos. A partir de aquí es cuestión de sofisticar todo.

Animaciones con ChatGPT

En esta lección vamos a realizar otro ejemplo de animación, pero para hacerlo más divertido vamos a ayudarnos de inteligencia artificial. Y es que se me ha ocurrido que podía ser interesante mover nuestro círculo por la ventana en lugar de incrementar su radio.

Para ello vamos a rectificar el código del ejemplo anterior para dibujar un círculo un poco más pequeño en el centro de la ventana:

```
"""
Fichero: 01-introduccion/21-animacion-chatgpt.py
"""

import arcade

SCREEN_WIDTH = 600
SCREEN_HEIGHT = 400


def dibujar_circulo(x, y, radio=30, color=arcade.csscolor.BLUE):
    """ Esta función dibuja un círculo personalizado """
    arcade.draw_circle_filled(x, y, radio, color)


def dibujar(intervalo):
    # Funciones de dibujado
    arcade.start_render()
    dibujar_circulo(300, 200, 15)


def main():
    # Configurar la ventana e iniciar el renderizado
    arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Animaciones con ChatGPT")
    arcade.set_background_color(arcade.csscolor.DEEP_SKY_BLUE)

    # Programar el renderizado cada 1/60 segundos y ejecutar el programa
    arcade.schedule(dibujar, 1 / 60)
    arcade.run()


if __name__ == "__main__":
    main()
```

Ahora vamos a crear dos atributos para controlar la posición x e y del círculo:

```
# Creamos dos atributos para controlar la posición del círculo
dibujar.posicion_x = 0
dibujar.posicion_y = 0
```

Y los sumamos al centro de la ventana al dibujar el círculo para compensar la distancia de dibujado:

```
def dibujar(intervalo):
    # Funciones de dibujado
    arcade.start_render()
    dibujar_circulo(300 + dibujar.posicion_x, 200 + dibujar.posicion_y, 15)
```

Si modificamos ligeramente las posiciones incrementando o disminuyendo su valor, el círculo debería dibujarse en lugares distintos y crear una ilusión de movimiento. Disminuir o incrementar la *posicion_x* debería mover el círculo horizontalmente, mientras que hacer lo propio con *posicion_y* lo moverá verticalmente:

```
def dibujar(intervalo):
    # Funciones de dibujado
    arcade.start_render()
    dibujar_circulo(300 + dibujar.posicion_x, 200 + dibujar.posicion_y, 15)
    # Modificamos ligeramente las posiciones del círculo en cada dibujado
    dibujar.posicion_x += 0.5
    dibujar.posicion_y += 0.25
```

Aquí es cuando las cosas se ponen interesantes, pues podemos aplicar todo tipo de funciones matemáticas para determinar las posiciones *x* e *y* de diferentes formas geométricas. No temáis, yo tampoco soy muy bueno en matemáticas, por eso vamos a pedirle ayuda a una IA como *ChatGPT* (en caso de no tener acceso utilizar el *Chat de Bing*).

Por ejemplo, supongamos que nos gustaría que el círculo en lugar de moverse en una dirección, hiciera una especie de órbita alrededor del centro de la ventana. Lo que le voy a pedir es lo siguiente:

"Hazme una función en Python para determinar la posición X e Y en una órbita alrededor de un punto del eje de coordenadas."

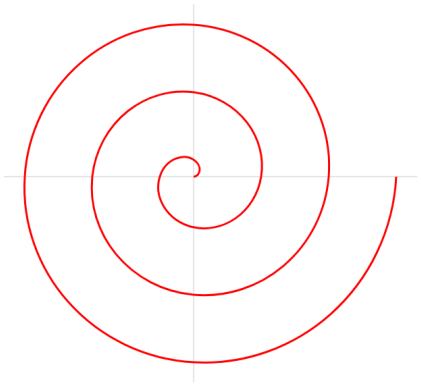
Bendita inteligencia artificial, no sólo nos genera la función sino que nos explica el funcionamiento:

```
import math

def posicion_orbita(x_c, y_c, r, angulo):
    """
    Calcula la posición (X, Y) de un objeto en una órbita alrededor de un punto.

    Parámetros:
    - x_c (float): coordenada en X del centro de la órbita.
    - y_c (float): coordenada en Y del centro de la órbita.
    - r (float): radio de la órbita.
    - angulo (float): ángulo en radianes que forma el objeto con respecto al centro.

    Retorna:
    - x (float): posición en X del objeto en la órbita.
    - y (float): posición en Y del objeto en la órbita.
    """
    x = x_c + r * math.cos(angulo)
    y = y_c + r * math.sin(angulo)
    return x, y
```

La verdad es que no nos interesa toda la función, solo necesitamos la parte que sirve para calcular las posiciones *x* e *y*. Vamos a intentar aplicarla inventándonos un radio fijo para la órbita e incrementando el ángulo ligeramente en cada fotograma:

```
def dibujar(intervalo):
    # Funciones de dibujado
    arcade.start_render()
    dibujar_circulo(300 + dibujar.posicion_x, 200 + dibujar.posicion_y, 15)

    # Incrementamos el ángulo de la órbita
    dibujar.angulo_orbita += 0.05

    # Calculamos las posiciones x e y donde dibujar el círculo
    dibujar.posicion_x = 100 * math.cos(dibujar.angulo_orbita)
    dibujar.posicion_y = 100 * math.sin(dibujar.angulo_orbita)


# Creamos unos atributos para controlar la posición y ángulo de órbita
dibujar.posicion_x = 0
dibujar.posicion_y = 0
dibujar.angulo_orbita = 0
```

¿Qué os parece? Con un poco de ingenio y la ayuda de una inteligencia artificial podemos generar cualquier animación. ¿Intentamos con una espiral de Arquímedes?



Sinceramente no tengo ni idea de como aplicar la función, ChatGPT, ayúdanos un poco anda…

> *"Genérame una función en Python para encontrar la posición X e Y en el eje de coordenadas cartesianas para una Espiral de Arquímedes."*

Bendita inteligencia artificial, no sólo nos genera la función sino que nos explica el funcionamiento:

```
import math

def posicion_espiral_arquimedes(t, a, b):
    """
    Calcula la posición (X, Y) de la Espiral de Arquímedes para un valor de parámetro t
```

```
dado.

    Parámetros:
    - t (float): valor del parámetro t para el cual se quiere conocer la posición X, Y.
    - a (float): parámetro que determina la separación entre las vueltas de la espiral.
    - b (float): parámetro que determina el grado de apertura de la espiral.

    Retorna:
    - x (float): posición en X correspondiente al valor de t dado.
    - y (float): posición en Y correspondiente al valor de t dado.
    """
    x = a * t * math.cos(t)
    y = b * t * math.sin(t)
    return x, y
```

En este caso tengo una duda, entiendo los parámetros *a* y *b*, ¿pero qué es *t*?

*"¿Puedes explicarme qué es el parámetro **t** de la función?"*

*En el caso de la Espiral de Arquímedes, el parámetro **t** suele representar el ángulo que forma la recta que une el centro de la espiral con un punto cualquiera de la misma, respecto a una recta fija. En otras palabras, **t** indica la cantidad de vueltas completas que ha dado la espiral y el ángulo adicional que se ha recorrido desde el inicio de la última vuelta.*

Ahora que sabemos que *t* es un número que representa el número de vuelta, podemos suponer que si vale *0.5* sería media vuelta y *3.25* serían *3* vueltas y cuarto. No sé vosotros pero tengo claro que *t* es el valor que debemos ir modificando en cada fotograma… A ver si es verdad:

```
def dibujar(intervalo):
   # Funciones de dibujado
   arcade.start_render()
   dibujar_circulo(300 + dibujar.posicion_x, 200 + dibujar.posicion_y, 15)

   # Incrementamos la cantidad de la vuelta actual de la espiral
   dibujar.vuelta += 0.05

   # Calculamos las posiciones x e y donde dibujar el círculo
   dibujar.posicion_x = 3 * dibujar.vuelta * math.cos(dibujar.vuelta)
   dibujar.posicion_y = 3 * dibujar.vuelta * math.sin(dibujar.vuelta)


# Creamos unos atributos para la posición y la vuelta actual de la espiral
dibujar.posicion_x = 0
dibujar.posicion_y = 0
dibujar.vuelta = 0
```

Y ahí lo tenemos… ¿No es alucinante cómo aplicando herramientas a nuestra disposición estamos generando animaciones super interesantes? ¿Qué se os pasa por la mente? ¿Se os ocurre alguna idea para crear vuestras propias animaciones? Pues preparaos porque esto es solo la punta del iceberg, a partir de la próxima unidad váis a descubrir el verdadero potencial de la programación y si hasta ahora os ha gustado vais a disfrutar como niños.

Resumen del tema

Esta ha sido una lección muy importante, hemos dado un paso adelante enorme respecto a lo que sabíamos al comienzo. Mediante la práctica hemos aprendido paso a paso la importancia de las funciones. Ahora sabemos por qué son claves a la hora de organizar y reutilizar código. Hemos aprendido a manejar parámetros y valores de retorno en sus distintas formas, incluso hemos hablado de las anotaciones de tipado. También hemos echado un vistazo a las funciones integradas de Python y hemos introducido la función de lectura *input*. Con todo ese conocimiento hemos culminado aplicando lo aprendido en el dibujo de gráficos e incluso hemos empezado a generar nuestras propias animaciones con la ayuda de una IA como ChatGPT.

Fundamentos lógicos

El tipo lógico

A lo largo de esta unidad iremos descubriendo como hacer que la programación se comporte como queramos. Aprenderemos a valorar posibilidades para actuar en consecuencia, a repetir bloques de código y a tratar los errores para que la ejecución de un programa no se detenga bruscamente. En definitiva vamos a aprender lo que se conoce como controlar el flujo.

Para controlar el flujo es necesario aprender antes sobre un tipo de dato que llevamos comentando todo el curso, el tipo lógico, también conocido por el barbarismo inglés «booleano». Este dato puede representar los valores de la lógica binaria, es decir, dos valores que representan únicamente falso o verdadero, traducidos numéricamente como 0 y 1:

```
"""
Fichero: 02-control-de-flujo/01-tipo-logico.py
"""

falso = False
verdadero = True
print(f"falso es {falso}, verdadero es {verdadero}")
# falso es False, verdadero es True
```

El tipo de dato, según nos lo devuelve la función *type* es *bool*:

```
print(f"El tipo de dato lógico es {type(True)}")
# El tipo de dato lógico es <class 'bool'>
```

Estos valores permiten el uso de un operador lógico especial llamado negación y que como su nombre indica niega el valor lógico que le sigue. Junto con el dato forma una expresión lógica:

```
falso_negado = not False
verdadero_negado = not True

print(f"falso_negado es {falso_negado}, verdadero_negado es {verdadero_negado}")
# falso_negado es True, verdadero_negado es False
```

Como decíamos antes, su representación numérica es 0 y 1 y en conjunto con operadores lógicos y relacionales darán como resultado este tipo:

```
print(f"0 negado es {not 0}, 1 negado es {not 1}")
# 0 negado es True, 1 negado es False
```

Veamos qué es eso de los operadores relacionales.

Operadores relacionales

Los operadores relacionales son unos símbolos que se usan para comparar dos valores. Si el resultado de la comparación es correcto la expresión considerada es verdadera, en caso contrario es falsa. Son muy fáciles de utilizar:

<	menor que	*a < b*	a es menor que b
>	mayor que	*a > b*	a es mayor que b
==	igual a	*a == b*	a es igual a b
!=	no igual a	*a != b*	a no es igual a b
<=	menor que o igual a	*a <= b*	a es menor que o igual a b
>=	mayor que o igual a	*a >= b*	a es menor que o igual a b

Podemos datos de los mismos tipos o con representaciones equivalentes sin problema:

```
"""
Fichero: 02-control-de-flujo/02-relacionales.py
"""

a, b = 5, 7

print(f"{a} < {b} es {a < b}")                  # 5 < 7 es True
print(f"{a} > {b} es {a > b}")                  # 5 > 7 es False
print(f"{a} == {b} es {a == b}")                # 5 == 7 es False
print(f"{a} != {b} es {a != b}")                # 5 != 7 es True
print(f"{a} <= {b} es {a <= b}")                # 5 <= 7 es True
print(f"{a} >= {b} es {a >= b}")                # 5 >= 7 es False
print()
print(f"Hola == Mola es {'Hola' == 'Mola'}")    # Hola == Hola es False
print(f"Hola != Mola es {'Hola' != 'Mola'}")    # Hola != Mola es True
print()
print(f"True == False es {True == False}")      # True == False es False
print(f"True > False es {True > False}")        # True > False es True
print(f"False == 0 es {False == 0}")            # False == 0 es True
print(f"True == 1 es {True == 1}")              # True == 1 es True
print(f"True == 999 es {True == 999}")          # True == 999 es False
```

Si bien no podemos comparar un número y una cadena:

```
"hola" > 10  # TypeError: '>' not supported between instances of 'str' and 'int'
```

Sí podemos comparar el valor de retorno de una función si concuerda con el tipo comparado:

```
print("¿Longitud de 'HOLA' mayor que 3?", len("HOLA") > 3)
# ¿Longitud de 'HOLA' mayor que 3? True
```

Recordad no confundir el operador asignación con el operador relacional igual a. Las asignaciones se realizan con un `=` y las comparaciones con dos `==`.

Operadores lógicos

Nos falta aprender un último concepto antes de lanzarnos a controlar el flujo, sin él no seríamos capaces de evaluar múltiples relaciones. Vamos a ilustrar esto con un ejemplo real.

Supongamos que una empresa quiere hacer una promoción para jóvenes de entre 18 y 32 años, por eso necesita ponerse en contacto con sus clientes de esa edad. Como programadores tenemos acceso a la base de datos de clientes pero necesitamos filtrar los que cumplen la edad de los que no. ¿Con lo que sabemos sobre operadores relacionales podemos hacerlo? Pues no porque no nos basta con un operador relacional, necesitamos evaluar dos condiciones en conjunto: la edad debe ser igual o mayor que 18 y a la vez igual o menor 32.

Fijaros que textualmente hemos utilizado una **Y** para descubrir el conjunto formado por las dos condiciones. La **Y** es una conjunción lógica que implica que ambas sentencias deben ser verdad para que el conjunto se evalúe como verdadero:

Falso	*Y*	*Falso*	**Falso**
Falso	*Y*	*Verdadero*	**Falso**
Verdadero	*Y*	*Falso*	**Falso**
Verdadero	*Y*	*Verdadero*	**Verdadero**

Si todavía os cuesta entenderlo os pondré un ejemplo más fácil. Si os digo: *"Me llamo Héctor y vengo de Saturno"*, ¿estoy diciendo la verdad o miento? Pues aunque la mitad es verdad, la mentira determina la veracidad del conjunto y por tanto estoy mintiendo.

En Python podemos usar el operador *and* o el símbolo `&` para evaluar dos relaciones en conjunción:

```
"""
Fichero: 02-control-de-flujo/03-operadores-logicos.py
"""

print(False and False)  # False
print(False and True)   # False
print(True and False)   # False
print(True and True)    # True
```

Sabiendo eso podemos comprobar fácilmente si se cumple la promoción para cualquier edad:

```
# Comprobación de descuento
print("¿Hay promoción con 16 años?", 16 >= 18 and 16 <= 32)  # False
print("¿Hay promoción con 24 años?", 24 >= 18 and 24 <= 32)  # True
print("¿Hay promoción con 41 años?", 41 >= 18 and 41 <= 32)  # False
```

Ahora, volviendo a nuestra discusión lógica, supongamos que en lugar de deciros *"Me llamo Héctor y vengo de Saturno"* os digo *"Me llamo Héctor o vengo de Saturno".* ¿Esto es verdad o mentira? Desde el punto de vista lógico, una **O** no es una conjunción sino una disyunción, eso implica que solo si ambas sentencias son falsas el conjunto se evalúa como falso:

Falso	*O*	*Falso*	**Falso**
Falso	*O*	*Verdadero*	**Verdadero**
Verdadero	*O*	*Falso*	**Verdadero**
Verdadero	*O*	*Verdadero*	**Verdadero**

En la frase *"Me llamo Héctor o vengo de Saturno"*, como una de las dos sentencias es verdad, la sentencia resultando se evalúa como verdadera.

En Python podemos usar el operador `or` o el símbolo `|` para evaluar dos relaciones en disyunción:

```
"""
Fichero: 02-control-de-flujo/03-operadores-logicos.py
"""

print(False or False)  # False
print(False or True)   # True
print(True or False)   # True
print(True or True)    # True
```

La programación es muy flexible y podemos conseguir el mismo resultado con ambas operaciones, sólo necesitamos adaptar las relaciones y negar el resultado. Por ejemplo, volviendo al problema de determinar la promoción, esto sería equivalente:

- **Sí** hay descuento cuando edad >= 18 Y edad <= 32
- **No** hay descuento cuando edad < 18 O edad > 32

Vamos a comprobarlo:

```
print("¿Hay promoción con 16 años?", not (16 < 18 or 16 > 32))  # False
print("¿Hay promoción con 24 años?", not (24 < 18 or 24 > 32))  # True
print("¿Hay promoción con 41 años?", not (41 < 18 or 41 > 32))  # False
```

Si no acabáis de pillarle el truco repasad un poco los ejemplos y no os preocupéis. Estos operadores se utilizan todo el tiempo y al final os saldrá como algo natural.

Resumen del tema

Condiciones

Controlar el flujo

Hasta ahora hemos aprendido a usar variables, crear expresiones y hacer nuestras propias funciones, pero todavía nos falta aprender cómo controlar el flujo. Controlar el flujo es añadir dinamismo a un programa, hacer que pase de ser algo estático a actuar en función de diferentes necesidades e incluso repetirse para conseguir un objetivo.

Hay dos formas de controlar el flujo: dividir su ejecución mediante condiciones o repetir su ejecución mediante bucles. En este tema vamos a aprender a hacer lo primero.

Bloques condicionales

Una condición parte de una sentencia *if* (si) para evaluar una expresión lógica. Si la expresión se evalúa como verdadera entonces se ejecuta un bloque de código condicional, parecido al bloque de código de una función:

```
"""
Fichero: 02-control-de-flujo/04-condiciones.py
"""

# Se entra al bloque condicional (True / not False)
if True:
    print("Se ejecuta el bloque condicional")
```

Si la expresión se evalúa como falsa, no se entra al bloque:

```
# Bloque condicional (False / not True)
if False:
    print("Se ejecuta el bloque condicional")
```

Los valores *0* y *None* también se evalúan como falsos. Cualquier otro valor, independientemente del tipo, se evalúa como verdadero:

```
# Bloque condicional (False / not True)
if 0:
    print("No se ejecuta el bloque condicional")

if None:
    print("No se ejecuta el bloque condicional")

if "Hola":
    print("Se ejecuta el bloque condicional")

if -230:
    print("Se ejecuta el bloque condicional")
```

Recuperando el ejemplo de la lección anterior donde estábamos comprobando una promoción en

función de la edad, podemos incorporar una lectura por teclado y controlar el flujo del código para mostrar un mensaje distinto en caso de que se cumpla o no:

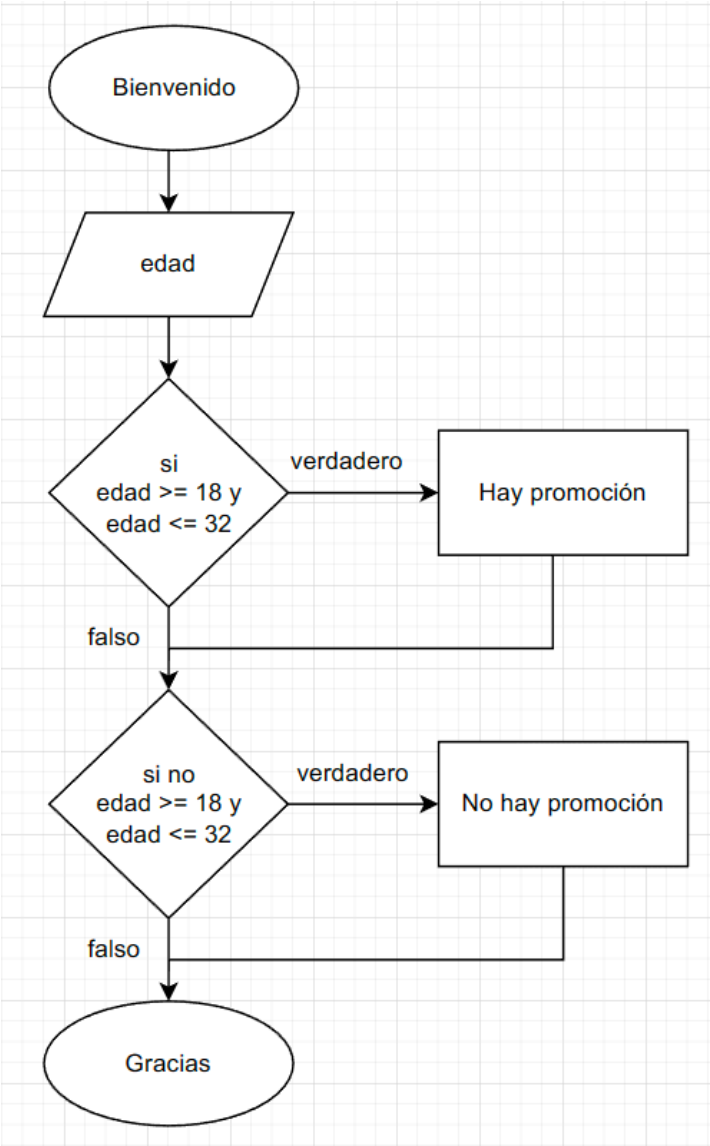
```
print("¡Bienvenido al comprobador de promociones!")
edad = int(input("Introduce una edad: "))
promocion = edad >= 18 and edad <= 32

if promocion:
   print("Puedes acceder a la promoción.")
if not promocion:
   print("No puedes acceder a la promoción.")

print("Gracias por utilizar nuestro comprobador.")
```

Como véis, cualquier expresión que se pueda evaluar como *True* o *False* nos servirá para iniciar el bloque condicional.

Diagramas de flujo

Una forma de ilustrar el flujo de un programa informático es mediante un diagrama de flujo. En estos diagramas los rectángulos son sentencias con una entrada y una salida y las condiciones se dibujan en un diamante que divide los caminos en función de la respuesta.

Existen muchas herramientas para crear diagramas, algunas de ellas son online. Vamos a crear el diagrama de flujo de nuestro ejemplo con la app gratuita de https://diagrams.net:



Lo bueno de los diagramas de flujo es que personas sin conocimientos de programación pueden hacerse una idea del funcionamiento de un programa. En este curso no vamos a profundizar en ellos pero si queréis aprender más tenéis la simbología explicada en la Wikipedia.

Instrucción else

Volviendo al código de la promoción, ¿no creéis que es bastante incómodo comprobar la misma condición dos veces para ejecutar un código cuando se cumple y cuando no?

Por suerte para nosotros existe una instrucción que va ligada a un bloque *if* y que permite ejecutar otro bloque de código en caso de que la condición no se cumpla, se trata del *else* (sino):

```
"""
Fichero: 02-control-de-flujo/05-bloque-else.py
"""

print("¡Bienvenido al comprobador de promociones!")
edad = int(input("Introduce una edad: "))
promocion = edad >= 18 and edad <= 32
```

```
if promocion:
   print("Puedes acceder a la promoción.")
else:
   print("No puedes acceder a la promoción.")
print("Gracias por utilizar nuestro comprobador.")
```

Este bloque se ejecuta en contraposición al *if* y como no necesita una comprobación explícita podemos ahorrarnos guardar la condición:

```
if edad >= 18 and edad <= 32:
   print("Puedes acceder a la promoción.")
else:
   print("No puedes acceder a la promoción.")
```

Con la excusa de introducir un problema muy común vamos a realizar otro ejemplo.

Supongamos que necesitamos comprobar si un número es par o impar. Para detectar si un número es múltiplo de otro necesitamos recuperar aquella operación llamada módulo que realizaba una división entera y devolvía el residuo. Si el residuo de dividir algo entre 2 es 0, ese número será par y en caso contrario será impar. Vamos a probarlo:

```
"""
Fichero: 02-control-de-flujo/06-par-o-impar.py
"""

numero = int(input("Introduce un número entero: "))
if numero % 2 == 0:
   print(f"El número {numero} es par.")
else:
   print(f"El número {numero} es impar.")
```

Cuando el módulo de un número es cero, el cero se puede tomar como *False* y si lo negamos con un operador *not* obtendremos el mismo resultado directamente:

```
if not numero % 2:
   print(f"El número {numero} es par.")
```

La programación es una herramienta muy flexible y se puede llegar a la misma respuesta siguiendo caminos distintos. Hacerlo de forma más o menos eficiente depende de la voluntad de cada uno de seguir aprendiendo.

Instrucción elif

En muchas ocasiones nos encontraremos que dos posibilidades no son suficientes para dividir el flujo de un programa. Supongamos que en lugar de ofrecer una promoción a un rango determinado de edad tengamos varios rangos, cada uno con diferentes promociones. Sin ir más lejos, pongamos de ejemplo un parque de atracciones con estos precios:

- Entrada Júnior (4-10 años): 35€

- Entrada Adulto (11-59 años): 40€
- Entrada Sénior (60 años o más): 30€

Está claro que un bloque *if-else* no es suficiente, así que debemos utilizar múltiples *if*:

```
"""
Fichero: 02-control-de-flujo/07-entrada-parque.py
"""

edad = int(input("Edad del visitante: "))

if edad < 4:
   print("No tiene la edad suficiente")
if edad >= 4 and edad <= 10:
   print("Entrada Júnior: 35€")
if edad >= 11 and edad <= 59:
   print("Entrada Adulto: 40€")
if edad >= 60:
   print("Entrada Sénior: 30€")
```

¿Pero realmente estamos dividiendo el flujo? Pues no, todas las condiciones se ejecutan sin falta. Para establecer múltiples condiciones específicas y dividir correctamente el código se utiliza la instrucción *elif* (sino si):

```
if edad < 4:
   print("No tiene la edad suficiente")
elif edad >= 4 and edad <= 10:
   print("Entrada Júnior: 35€")
elif edad >= 11 and edad <= 59:
   print("Entrada Adulto: 40€")
elif edad >= 60:
   print("Entrada Sénior: 30€")
```

Al final de los bloques else-if se puede definir un último *else* como caso genérico en caso de que ninguno de los demás se ejecute. Si estamos seguros de haber cubierto todas las posibilidades, podemos obviar la última condición:

```
if edad < 4:
   print("No tiene la edad suficiente")
elif edad >= 4 and edad <= 10:
   print("Entrada Júnior: 35€")
elif edad >= 11 and edad <= 59:
   print("Entrada Adulto: 40€")
else:
   print("Entrada Sénior: 30€")
```

Lo bueno de utilizar elif y dividir el flujo, es que ahora las condiciones se excluyen entre ellas. Es decir, no necesitamos comprobar si un júnior tiene más de 4 años porque la primera condición ya lo

ha excluido, solo debemos comprobar si tiene 11 o menos y lo mismo ocurre para el adulto. Esto simplifica la programación del conjunto condicional si se sigue un orden lógico:

```
if edad < 4:
   print("No tiene la edad suficiente")
elif edad <= 10:
   print("Entrada Júnior: 35€")
elif edad <= 59:
   print("Entrada Adulto: 40€")
else:
   print("Entrada Sénior: 30€")
```

Un bloque *if-elif-else* debe facilitaros la vida, si os la está complicando quizá no sea la mejor opción.

Instrucción match-case

Desde Python 3.10 tenemos a nuestra disposición otra forma de dividir el flujo. Ésta no se basa en evaluar una expresión lógica sino un valor directo. Hablamos de la instrucción *match-case* y es muy útil para crear menús y máquinas de estado donde las posibilidades son siempre estáticas. Una máquina de estados es un patrón aplicado en los videojuegos para controlar el funcionamiento de una entidad. Por ejemplo, un personaje puede tener diferentes estados como parado, caminando, corriendo, saltando… La máquina de estados es la que controla toda esa lógica interna.

Quizá implementemos alguna más adelante en el curso, por ahora veamos esta instrucción en acción con un sencillo menú textual:

```
"""
Fichero: 02-control-de-flujo/08-match-case.py
"""

opcion = input("Ingresa una opción del 1 al 4: ")

match opcion:
   case "1":
       print("Seleccionaste la opción 1")
   case "2":
       print("Seleccionaste la opción 2")
   case "3":
       print("Seleccionaste la opción 3")
   case "4":
       print("Seleccionaste la opción 4")
   case _:
       print("Opción inválida")
```

Fijaros como los casos son estáticos, son siempre datos literales. Si necesitamos evaluar variables o expresiones esta instrucción no nos servirá, deberemos utilizar condiciones *if-elif-else*.

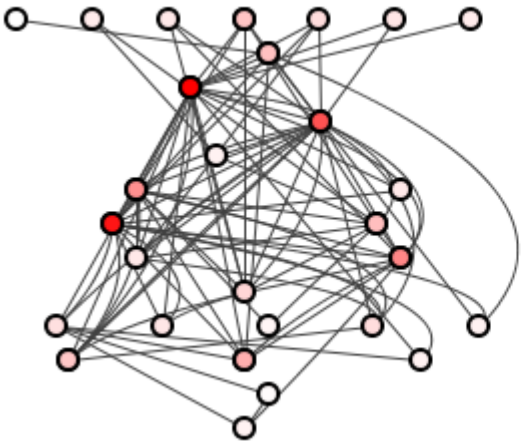
Condiciones anidadas

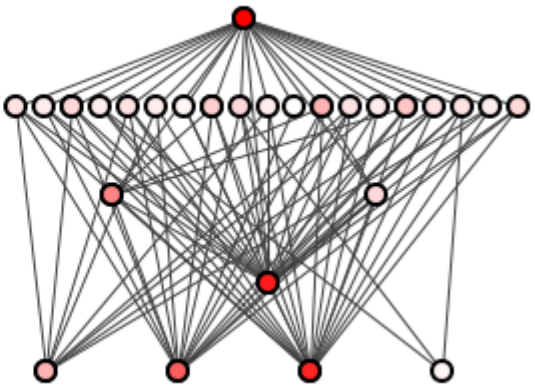
Algo que también es posible es utilizar condiciones dentro de condiciones, conocidas como condiciones anidadas. Podemos hacerlo sin problema siempre y cuando respetemos la indentación

de los bloques. Cada bloque debe encontrarse acordemente separado, el primer nivel de condición a una distancia del bloque principal, el segundo al doble de esa distancia y así sucesivamente:

```
"""
Fichero: 02-control-de-flujo/09-condiciones-anidadas.py
"""

nota = int(input("Introduce tu nota: "))

if nota >= 0 and nota <= 10:
    if nota < 5:
        print("Suspenso")
    elif nota < 7:
        print("Bien")
    elif nota < 9:
        print("Notable")
    else:
        print("Sobresaliente")
else:
    print("La nota introducida es incorrecta")
```

Como consejo os diría que no abuséis de anidar condiciones, no es una mala práctica pero un uso excesivo puede generar fácilmente código espagueti. Se le dice así a los programas que tienen una estructura de control de flujo compleja e incomprensible, parecida a un plato de espaguetis:





Animaciones y condiciones

En esta lección vamos a recuperar el código para dibujar un círculo animado:

```
"""
Fichero: 02-control-de-flujo/10-animacion-condicion.py
"""

import arcade

SCREEN_WIDTH = 600
SCREEN_HEIGHT = 400
```

```
def dibujar_circulo(x, y, radio=30, color=arcade.csscolor.BLUE):
    arcade.draw_circle_filled(x, y, radio, color)


def dibujar(intervalo):
    arcade.start_render()
    dibujar_circulo(300 + dibujar.posicion_x, 200 + dibujar.posicion_y, 15)

dibujar.posicion_x = 0
dibujar.posicion_y = 0


def main():
    arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Primera animación")
    arcade.set_background_color(arcade.csscolor.DEEP_SKY_BLUE)
    arcade.schedule(dibujar, 1 / 60)
    arcade.run()

if __name__ == "__main__":
    main()
```

En esta ocasión vamos a aplicar lo aprendido sobre condiciones para realizar una animación dinámica donde nuestro círculo se mueva rebotando contra los bordes de la pantalla.

Podemos empezar con algo sencillo, incrementar la posición X en cada ciclo pero solo si es menos de 300. Si es 300 significa que hemos llegado al margen derecho (el ancho es 600 y partimos del centro):

```
def dibujar(intervalo):
    arcade.start_render()
    dibujar_circulo(300 + dibujar.posicion_x, 200 + dibujar.posicion_y, 15)

    if dibujar.posicion_x < 300:
        dibujar.posicion_x += 3
```

Ahora vamos a plantearnos cómo hacer que en lugar de quedarse quieto se mueva en dirección contraria. ¿Podríamos utilizar un else para hacer lo contrario y restar a la posición horizontal? Vamos a intentarlo:

```
if dibujar.posicion_x < 300:
    dibujar.posicion_x += 3
else:
    dibujar.posicion_x -= 3
```

Veréis que no acaba de funcionar, pero si os fijáis detenidamente se nota que el círculo se mueve brevemente a la izquierda después de tocar el margen derecho. Cuando llegamos al margen derecho la posición X es 300 o más, entonces se ejecuta el *else* y se mueve a la izquierda, pero inmediatamente en el siguiente fotograma la posición vuelve a ser menor de 300 y se mueve otra vez a la derecha. ¿Se os ocurre alguna forma de solucionar este problema? Hay varias formas de

abordarlo, según mi experiencia mí la más sencilla es plantear un atributo para controlar la dirección de movimiento. Como estamos trabajando con la dirección en el eje X lo llamaremos *direccion_x* y digamos que puede tener dos valores, *1* para la derecha y *-1* para la izquierda:

```
dibujar.direccion_x = 1
```

Simplemente moveremos a derecha o izquierda el círculo dependiendo de esta dirección. En caso de superar los 300 píxeles cambiaremos el valor a -1 para moverlo a la izquierda:

```
def dibujar(intervalo):
    arcade.start_render()
    dibujar_circulo(300 + dibujar.posicion_x, 200 + dibujar.posicion_y, 15)

    # Movemos el círculo horizontalmente
    if dibujar.direccion_x == 1:
        dibujar.posicion_x += 3
    else:
        dibujar.posicion_x -= 3

    # Cambiamos la dirección horizontal al rebotar
    if dibujar.posicion_x > 300:
        dibujar.direccion_x = -1
```

Y haremos lo propio para cambiar la dirección a la derecha cuando la posición X sea menor de -300:

```
# Cambiamos la dirección horizontal al rebotar
if dibujar.posicion_x > 300:
    dibujar.direccion_x = -1
elif dibujar.posicion_x < -300:
    dibujar.direccion_x = 1
```

Esto que hemos hecho se puede simplificar muchísimo jugando el el *1* y el *-1*. Al final lo que queremos es sumar o restar un número, pues podemos multiplicarlo por la dirección para que sea positivo o negativo, de esa forma nos ahorramos toda la comprobación:

```
# Movemos el círculo horizontalmente
dibujar.posicion_x += 3 * dibujar.direccion_x
```

En cuanto a la dirección, podemos utilizar un único if con un operador *OR* para agrupar ambas condiciones. Multiplicando por (-1) para negar la dirección: 1 * (-1) será (-1) y (-1) * (-1) será 1:

```
# Cambiamos la dirección horizontal al rebotar
if dibujar.posicion_x > 300 or dibujar.posicion_x < -300:
    dibujar.direccion_x *= -1
```

¡Genial! Incluso diría que podemos dejarlo aún mejor creando un atributo *velocidad_x*:

```
# Movemos el círculo horizontalmente
dibujar.posicion_x += dibujar.velocidad_x * dibujar.direccion_x

# ~~~
dibujar.velocidad_x = 3
```

Esta es una forma clásica de manejar el movimiento en los videojuegos 2D. Ahora podemos extender la lógica para hacer exactamente lo mismo en el eje vertical. Os daré las indicaciones a ver si sois capaces de extender la lógica:

1. Crea dos atributos *direccion_y = 1* y *velocidad_y = 2.5* para controlar el movimiento vertical.
2. Cambia la dirección vertical utilizando la misma lógica que horizontalmente. El círculo rebotará en el techo si su posición Y es mayor que 200 y en el suelo si es menor que -200.
3. Mueve el círculo verticalmente de la misma forma que lo movemos en horizontal.

```
def dibujar(intervalo):
    arcade.start_render()
    dibujar_circulo(300 + dibujar.posicion_x, 200 + dibujar.posicion_y, 15)

    # Movemos el círculo horizontalmente
    dibujar.posicion_x += dibujar.velocidad_x * dibujar.direccion_x

    # Movemos el círculo verticalmente
    dibujar.posicion_y += dibujar.velocidad_y * dibujar.direccion_y

    # Cambiamos la dirección horizontal al rebotar
    if dibujar.posicion_x > 300 or dibujar.posicion_x < -300:
        dibujar.direccion_x *= -1

    # Cambiamos la dirección vertical al rebotar
    if dibujar.posicion_y > 200 or dibujar.posicion_y < -200:
        dibujar.direccion_y *= -1


dibujar.posicion_x, dibujar.posicion_y = 0, 0
dibujar.direccion_x, dibujar.velocidad_x = 1, 3
dibujar.direccion_y, dibujar.velocidad_y = 1, 2.5
```

¿Qué os parece? Recién estamos empezando pero ya hemos conseguido recrear el movimiento de la pelota igual que funciona en algunos famosos clásicos del arcade como Pong y Breakout.

Resumen del tema

En este tema hemos aprendido a trabajar con las condiciones y su capacidad para alterar el flujo del código que se puede representar de forma más visual utilizando figuras en los diagramas de flujo. También hemos visto cómo crear múltiples estructuras condicionales mediante bloques *if-elif-else* y la instrucción *match-case* introducida en Python 3.10 para condicionar a partir de valores literales en lugar de expresiones lógicas. Por último experimentamos con las condiciones anidadas y aplicamos todo lo aprendido animando un círculo que rebota por la ventana de forma muy similar a como funciona la pelota en algunos videojuegos arcade clásicos.

Bucle Mientras

Potencial ilimitado

Como explicamos anteriormente hay dos formas de controlar el flujo, una es condicionar el código y la otra es repetirlo. Es a partir de esta segunda forma cuando vamos a ser conscientes de la diferencia que existe entre el potencial de una máquina y una persona.

Veréis, todo lo que hemos programado hasta ahora han sido conceptos lógicos bastante simples. Almacenar datos en la memoria, operarlos, compararlos… Son programas interesantes pero no hacen nada que una persona no pueda hacer. Sin embargo, a partir de ahora vamos a incorporar los bucles al código y aquí es cuando las máquinas dejan atrás a las personas.

Los bucles son instrucciones que permiten repetir bloques y en Python tenemos dos: Los que se repiten mientras se cumple una expresión lógica y los que se repiten un número determinado de veces. Vamos a empezar hablando de los primeros.

Mientras infinito

Tal como indica el nombre de la lección, el primer bucle que vamos a ver es el mientras, en inglés *while*. Por naturaleza este bucle se ejecuta de forma indefinida y por eso es fácil que se descontrole, lo que acaba provocando inestabilidad en el sistema por monopolizar los recursos.

Una forma de prevenir estos problemas es añadir deliberadamente una pausa o una interacción con el usuario en cada ciclo, veamos algún ejemplo:

```
"""
Fichero: 02-control-de-flujo/11-bucle-mientras.py
"""

print("Cada ciclo de repetición se conoce como iteración.")

while True:
   input("Presiona enter para continuar con la siguiente iteración.")

print("En este punto el bucle mientras habrá finalizado.")
```

Como véis este bucle se ejecuta para siempre pero estamos controlando la frecuencia iteración al interrumpirlo para que el usuario presione ***Enter***. El problema es que aun así no termina nunca, para finalizar el programa debemos forzar la detención, ya sea cerrando la terminal o con algún comando como ***Control+C*** para matar el proceso, cosa que no siempre funciona.

Lo que sí es posible es finalizar un bucle desde su propio código rompiendo la ejecución, para ello se utiliza la instrucción ***break***. Ya que estamos utilizando *input* podemos leer algún texto y romper el bucle si el usuario lo indica:

```
while True:
   texto = input("Presiona enter para continuar o [q] para salir.")
   # Si escribimos la letra 'q' de QUIT rompemos el while desde dentro
   if texto == "q":
```

```
        break
```

Esta estructura es sobre la que se fundamentan todas las aplicaciones y videojuegos. La diferencia es que en lugar de interrumpir el bucle con una lectura se establece un límite de ciclos por segundo para evitar la sobrecarga del procesador y la tarjeta gráfica.

Mientras condicionado

Otra forma de romper la ejecución es cambiar la expresión lógica del bucle para que no se cumpla:

```
"""
Fichero: 02-control-de-flujo/12-mientras-condicion.py
"""

print("Cada ciclo de repetición se conoce como iteración.")

run = True

while run:
    texto = input("Presiona enter para continuar o [q] para salir.")
    # Si escribimos una letra 'q' finalizamos la condición
    if texto == "q":
        run = False

print("En este punto el bucle mientras habrá finalizado.")
```

Personalmente no me gusta controlar el bucle de esta forma, creo que es más cómodo utilizar la instrucción *break*, es más legible y nos ahorra una variable.

A parte de controlar menús, programas y videojuegos este bucle se utiliza cuando necesitamos repetir algo un número indeterminado de veces. Esto puede estar relacionado con factores externos como sensores y temporizadores.

Mientras contador

Si bien el bucle mientras tiene esa naturaleza de ejecución indefinida también es posible utilizarlo para repetir código un número de veces determinado. El problema es que por sí mismo no puede hacerlo, así que requiere la ayuda de una variable que le sirva como contador:

```
"""
Fichero: 02-control-de-flujo/13-mientras-contador.py
"""

repeticiones = 5

while repeticiones > 0:
    print("Repitiendo código...", repeticiones)
    repeticiones -= 1

print("El bucle mientras ha finalizado.")
```

Es imprescindible que el contador se modifique para garantizar que eventualmente la expresión deja de cumplirse y el bucle finaliza.

Mientras continuar

Terminando el tema vamos a ver una última instrucción llamada *continue*. Es parecida al *break* pero en lugar de romper el bucle rompe la iteración actual.

Empecemos con un ejemplo que utiliza *while* para sumar todos los números del 0 a 10:

```
"""
Fichero: 02-control-de-flujo/14-mientras-continuar.py
"""

suma = 0
contador = 1

while contador <= 10:
    print(f"Sumando {suma} + {contador}...")
    suma += contador
    contador += 1

print(f"Sumatorio = {suma}")
```

Ahora queremos cambiar el funcionamiento y no sumar los múltiplos de 3. Podemos utilizar el módulo para determinarlo y en ese caso utilizar *continue* para evitar la suma:

```
while contador <= 10:
    if contador % 3 == 0:
        print(f"{contador} es múltiplo de 3, nos saltamos la suma...")
        contador += 1
        continue

    print(f"Sumando {suma} + {contador}...")
    suma += contador
    contador += 1
```

Recordad que es imprescindible modificar el contador antes de realizar la continuación, en caso contrario vamos a generar un bucle infinito.

Resumen del tema

En este tema hemos aprendido a repetir código utilizando el bucle mientras, cuya finalidad es manejar repeticiones infinitas e indeterminadas en menús, programas y videojuegos. También se puede utilizar para repeticiones determinadas, pero por su naturaleza no puede hacerlo solo, necesita ayuda de variables auxiliares a modo de contadores. Por último no podemos olvidar la utilidad de las instrucciones *break* y *continue* para romper la ejecución de los bucles y las iteraciones.

Bucle For

Para en una colección

El segundo bucle que vamos a aprender y que nos permitirá realizar repeticiones determinadas es el para, en inglés *for*. En Python esta instrucción va ligada al hecho de recorrer una colección iterable de elementos. Ejemplos de colecciones iterables son las listas, las tuplas, los diccionarios y los conjuntos. Vamos a ilustrar el funcionamiento del bucle recorriendo una sencilla lista:

```
"""
Fichero: 02-control-de-flujo/15-bucle-para.py
"""

# Lista con tres cadenas de texto
cadenas = ["Hola", "buenos", "días"]

# Bucle for para recorrer las cadenas
for cadena in cadenas:
   print(cadena)
```

Como véis una lista se define entre corchetes y contiene múltiples datos separados por comas, es muy intuitiva. Por su parte el bucle *for* también es muy intuitivo, su sintaxis se puede leer como *"Para cada elemento en una colección de elementos"* y recorre de principio a fin cada elemento de la colección guardándolo en la variable definida entre el *for* y el *in* . A continuación se genera un bloque de código que es el que se irá repitiendo hasta llegar al último elemento.

Por tanto no hay una condición específica para mantener el bucle en marcha, la condición es automática y el número de repeticiones viene determinado por la cantidad de elementos de la colección que recorremos.

Esta forma de funcionar tiene algunos inconvenientes. ¿Qué pasa si queremos repetir un código 1000 veces? ¿Necesitamos entonces definir una colección con 1000 elementos? Pues como podéis suponer no. Y es que el bucle para va ligado íntimamente a una función llamada *range*.

Para en un rango

Un rango es una colección de números generada sobre la marcha. Esta propiedad de generarse mientras se recorre es importante porque no requiere almacenar todo el espacio de la colección. En otras palabras, un rango puede tener 1 millón de números pero internamente solo necesita uno para utilizarse a modo de contador:

```
"""
Fichero: 02-control-de-flujo/16-para-rango.py
"""

diez_numeros = range(10)
print(diez_numeros)  # range(0, 10)
```

Si imprimimos el rango en lugar de todos sus elementos se muestra una representación, que en este

caso va de 0 a 10, pero es muy importante recordar que el último número siempre se excluye, por lo que el rango realmente abarca los números del 0 al 9.

¿Cuánto ocupa en la memoria este rango de 10 números?

```
import sys

print(sys.getsizeof(diez_numeros))  # 48 bytes
```

¿En cambio cuánto ocupa una lista que almacena la misma cantidad de números?

```
lista_diez_numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(sys.getsizeof(lista_diez_numeros ))  # 136 bytes
```

Como vemos ocupa bastante más porque los datos deben almacenarse en la memoria, sin embargo un rango aunque tenga 1.000.000 números seguirá ocupando lo mismo:

```
print(sys.getsizeof(range(1000000)))  # 48 bytes
```

La parte interesante es que podemos recorrer un rango utilizando el bucle *for*:

```
for repeticion in range(10):
   print("Repetición", repeticion)
```

De esta forma podemos repetir el código las veces que queramos sin necesidad de ocupar memoria.

Por influencia del álgebra lineal, al recorrer rangos a la variable le damos el nombre *i* de índice:

```
for i in range(10):
   print(i)
```

Por cierto, un rango puede empezar en el número que queramos:

```
# Rango de 6 a 14
for i in range(6, 14):
   print(i)
```

También se le puede pasar un tercer argumento con un salto:

```
# Rango de 0 a 10 cada 3 números
for i in range(0, 10, 3):
   print(i)
```

Incluso el rango puede ir de más a menos, solo hay que cambiarle el orden y establecer un paso de 1 negativo:

```
# Rango inverso de 10 a 0
for i in range(10, 0, -1):
   print(i)
```

Si lo necesitamos podemos cambiar la cantidad del salto negativo:

```
# Rango inverso de 10 a 0 cada 3 números
for i in range(10, 0, -3):
   print(i)
```

Como curiosidad para terminar, es posible convertir un rango a una lista utilizando la función de tipo *list*, pero debemos tener presente que eso hará que el espacio de todos los números del rango se almacene en la memoria. ¿Cuanto ocupan 1 millón de números enteros en la memoria?

```
un_millon_de_numeros = list(range(1000000))
print(sys.getsizeof(un_millon_de_numeros))  # 8000056 bytes
```

Como véis son 8.000.056 Bytes o lo que es lo mismo 8 MBytes. Si suponemos que un entero ocupa 8 Bytes entonces 1 millón de enteros son 8 millones de Bytes, tiene bastante sentido.

Para continuar

Al igual que el bucle while, el *for* permite romper la ejecución o continuar con la siguiente iteración:

```
"""
Fichero: 02-control-de-flujo/17-para-continuar.py
"""

for i in range(10):
   # Ruptura del bucle
   if i == 5:
       break
   print("Repetición", i)
```

También podemos continuar con la siguiente iteración, la ventaja respecto al *while* es que no debemos preocuparnos de incrementar ningún contador:

```
suma = 0
for i in range(10):
   # Continuar si es múltiplo de 3
   if i % 3 == 0:
       print(f"{i} es múltiplo de 3, nos saltamos la suma...")
       continue
   print(f"Sumando {suma} + {i}...")
   suma += i

print(f"Sumatorio = {suma}")
```

Insisto en recordaros que un rango siempre excluye el último número, si necesitáis incluirlo en el bucle tendréis que alargarlo un número más: *range(11)*.

Paras anidados

Llegamos a una de las particularidades más importantes de la programación, la anidación de bucles, un concepto que puede ser difícil de entender a simple vista, por eso vamos a desarrollarlo desde el principio.

Empecemos con dos bucles no anidados que imprimen diferentes caracteres y antes de ejecutarlo vamos a intentar deducir qué aparecerá en la terminal:

```
"""
Fichero: 02-control-de-flujo/18-bucles-anidados.py
"""

# Al primer índice le llamaremos i
for i in range(3):
    print("A")

# Al segundo índice le llamaremos j
for j in range(3):
    print("B")
```

En este caso la salida es fácil de adivinar, son dos bucles separados y por tanto primero se imprime la A en tres líneas diferentes y luego la B en otras tres líneas, nada nuevo bajo el sol.

Muy bien, ahora vamos a incorporar un pequeño cambio, la función *print* por defecto siempre escribe un salto de línea en el último carácter, pero podemos cambiarlo por otro, por ejemplo un espacio. ¿Qué se imprimirá si sustituímos los saltos de línea por espacios?

```
# Al primer índice le llamaremos i
for i in range(3):
    print("A", end=" ")

# Al segundo índice le llamaremos j
for j in range(3):
    print("B", end=" ")

# A A A B B B
```

Lo que se mostrará es una sola línea con todos los caracteres separados por espacios. Aún así estos caracteres se siguen imprimiendo en orden, primero las tres A y luego las tres B.

Ahora viene la parte interesante, ¿qué se imprimirá por pantalla si incorporamos el segundo bucle dentro del primero?

```
for i in range(3):
    print("A", end=" ")
```

```
    for j in range(3):
        print("B", end=" ")
```

¿Ahora no es tan fácil hacernos una idea verdad? Tenemos que analizarlo detenidamente:

- Se ejecuta el primer bucle con el índice i=0
    - Se imprime la letra A
    - Se ejecuta el segundo bucle con el índice j=0
        - Se imprime la letra B
    - Se ejecuta el segundo bucle con el índice j=1
        - Se imprime la letra B
    - Se ejecuta el segundo bucle con el índice j=2
        - Se imprime la letra B
- Se ejecuta el primer bucle con el índice i=1
    - Se imprime la letra A
    - Se ejecuta el segundo bucle con el índice j=0
        - Se imprime la letra B
    - Se ejecuta el segundo bucle con el índice j=1
        - Se imprime la letra B
    - Se ejecuta el segundo bucle con el índice j=2
        - Se imprime la letra B
- Se ejecuta el primer bucle con el índice i=2
    - Se imprime la letra A
    - Se ejecuta el segundo bucle con el índice j=0
        - Se imprime la letra B
    - Se ejecuta el segundo bucle con el índice j=1
        - Se imprime la letra B
    - Se ejecuta el segundo bucle con el índice j=2
        - Se imprime la letra B

Según nuestro análisis, para cada vez que se ejecuta el primer bucle se ejecutará tres veces el segundo. Es decir, para cada A se imprimirán tres B y eso sucederá tres veces, por tanto en la pantalla se imprimirá el texto A B B B A B B B A B B B, ¿será cierto?

```
A B B B A B B B A B B B
Process finished with exit code 0
```

Estaréis de acuerdo conmigo que esta forma de visualizar dos bucles anidados es un poco confusa, vamos a imprimir un salto de línea justo al final del primer bucle:

```
for i in range(3):
    print("A", end=" ")

    for j in range(3):
        print("B", end=" ")
```

```
    print("")
```

Con esta pequeña variación se puede interpretar una estructura. ¿No os recuerda a algún tipo de figura? ¿Una tabla? ¿O quizá una matriz?

```
A B B B
A B B B
A B B B
```

¿Notáis alguna relación entre las filas y columnas de esta tabla o matriz con el primer bucle y el segundo? Pues ya os la digo yo, el primer bucle determina el número de filas y el segundo el número de columnas. Si imprimimos ambos índices juntos podemos saber la posición exacta de cada celda:

```
for i in range(3):
    for j in range(3):
        print(f"{i}{j}", end=" ")
    print("")
```

El primer índice indica la fila y el segundo la columna, pero empezando por cero. Por ejemplo 11 es la celda de la segunda fila y segunda columna, 20 es la tercera fila y primera columna:

```
00 01 02
10 11 12
20 21 22
```

Esto explica por qué se dice que anidar un bucle dentro de otro expande el procesamiento una dimensión. Mientras que un bucle se puede interpretar como una sola dimensión, dos bucles anidados se pueden interpretar como dos dimensiones. Y así sucesivamente para cada bucle anidado extra. Tres bucles para recorrer tres dimensiones, cuatro para cuatro dimensiones, etc… ¿Suena difícil? Puede ser, pero no os preocupéis antes de tiempo, al final este curso se enfoca en la programación de videojuegos 2D y por esa misma razón todo lo que vamos a necesitar lo podemos resolver anidando dos bucles.

Dibujando tablas

Ahora que sabemos dibujar tablas con bucles anidados vamos a realizar diferentes experimentos gráficos para la terminal. Veamos si con lo explicado sois capaces de dibujar en nuevo script llamado *19-dibujos-tablas.py*, una tabla 4x6 con 4 filas y 6 columnas cuyo contenido sean solo asteriscos *:

```
"""
Fichero: 02-control-de-flujo/19-dibujos-tablas.py
"""

for i in range(4):
    for j in range(6):
        print(f"*", end=" ")
```

```
    print()
```

```
* * * * * *
* * * * * *
* * * * * *
* * * * * *
```

Ahora una tabla de 5x8 donde cada celda muestre solo el número de fila pero empezando en 1:

```
for i in range(5):
    for j in range(8):
        print(f"{i+1}", end=" ")
    print()
```

```
1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5
```

Ahora viene una más difícil, prestad atención. Tenéis que dibujar una tabla de 7x7 pero en lugar de imprimir todos los valores, en cada fila debéis imprimir tantos números como el número de la fila. Por ejemplo, en la primera fila solo tenéis que imprimir el 1, en la segunda fila 1 2 y así hasta la última donde estarán todos los números 1 2 3 4 5 6 7. Una pista, en lugar de utilizar un rango fijo para el segundo bucle probad con algo variable:

```
for i in range(7):
    for j in range(i + 1):
        print(f"{j + 1}", end=" ")
    print()
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
```

¿Y si os reto a hacer lo mismo pero en lugar de menos a más, de más a menos? Dibujando en la primera fila todos los números 1 2 3 4 5 6 7 y en la última solo el 1? Una pista, prueba a restar algún valor en el segundo bucle:

```python
for i in range(7):
    for j in range(7 - i):
        print(f"{j + 1}", end=" ")
    print()
```

```
1 2 3 4 5 6 7
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

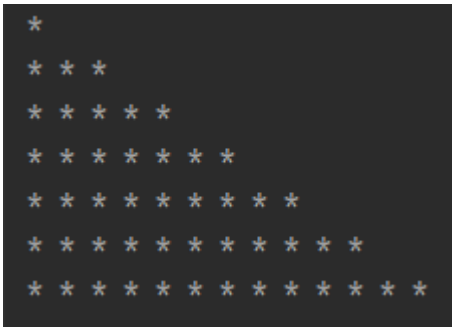
El último ejercicio consiste en dibujar una pirámide de asteriscos en una tabla de 7x13 como la siguiente, para ello os daré algunas pistas y os enseñaré el resultado de cada paso:

```
            *
          * * *
        * * * * *
      * * * * * * *
    * * * * * * * * *
  * * * * * * * * * * *
* * * * * * * * * * * * *
```

En lugar de pensar cómo dibujar partiendo de un rectángulo intentad dibujar primero la mitad derecha de la pirámide incluyendo la columna del centro:

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
```

A continuación debéis dibujar delante la mitad izquierda, ésta no debe tener en cuenta la columna del centro así que tendrá 1 asterisco menos. Deberéis lograr un resultado como el siguiente, con todos los asteriscos de la pirámide agrupados al principio de cada fila:



En este punto solo debéis pensar en una forma de añadir al principio de cada fila un número determinado de espacios para desplazar todos los asteriscos y recrear la pirámide. Os daré una pista, anotad el número de espacios que necesitáis al principio de cada línea, seguro que notáis algún patrón que podéis evaluar automáticamente:

```
for i in range(7):
    # Desplazamos los asteriscos con espacios
    for j in range(7 - i - 1):
        print(" ", end=" ")
    # Mitad izquierda de la pirámide
    for j in range(i):
        print("*", end=" ")
    # Mitad derecha de la pirámide
    for j in range(i + 1):
        print("*", end=" ")
    print()
```

Estos ejercicios son una forma excelente y divertida de practicar los bucles anidados. Todos los videojuegos basados en escenarios cuadriculados aplican estos conocimientos y para demostrarlo dibujaremos un tablero dinámicamente.

Dibujando un tablero

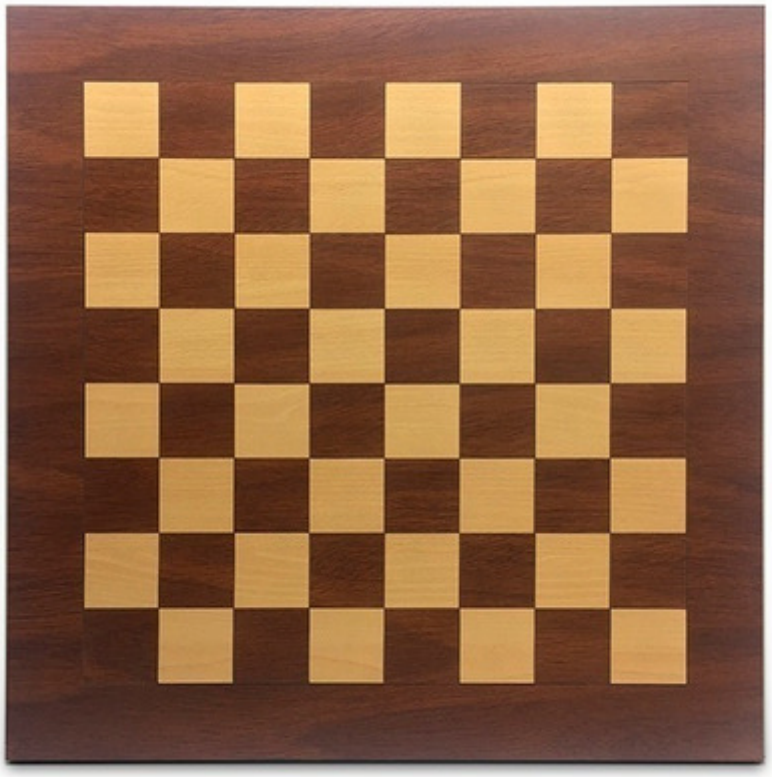
En esta lección vamos a dibujar una cuadrícula dinámica a modo de tablero de ajedrez. No solo haremos el dibujo sino que podremos modificar el color de las casillas utilizando índices, va a ser una práctica muy interesante.

Partiremos de una ventana cuadrada de 500x500 píxeles:

```
"""
Fichero: 02-control-de-flujo/20-dibujo-tablero.py
"""
import arcade

SCREEN_WIDTH = 500
SCREEN_HEIGHT = 500


def dibujar(intervalo):
    arcade.start_render()
```

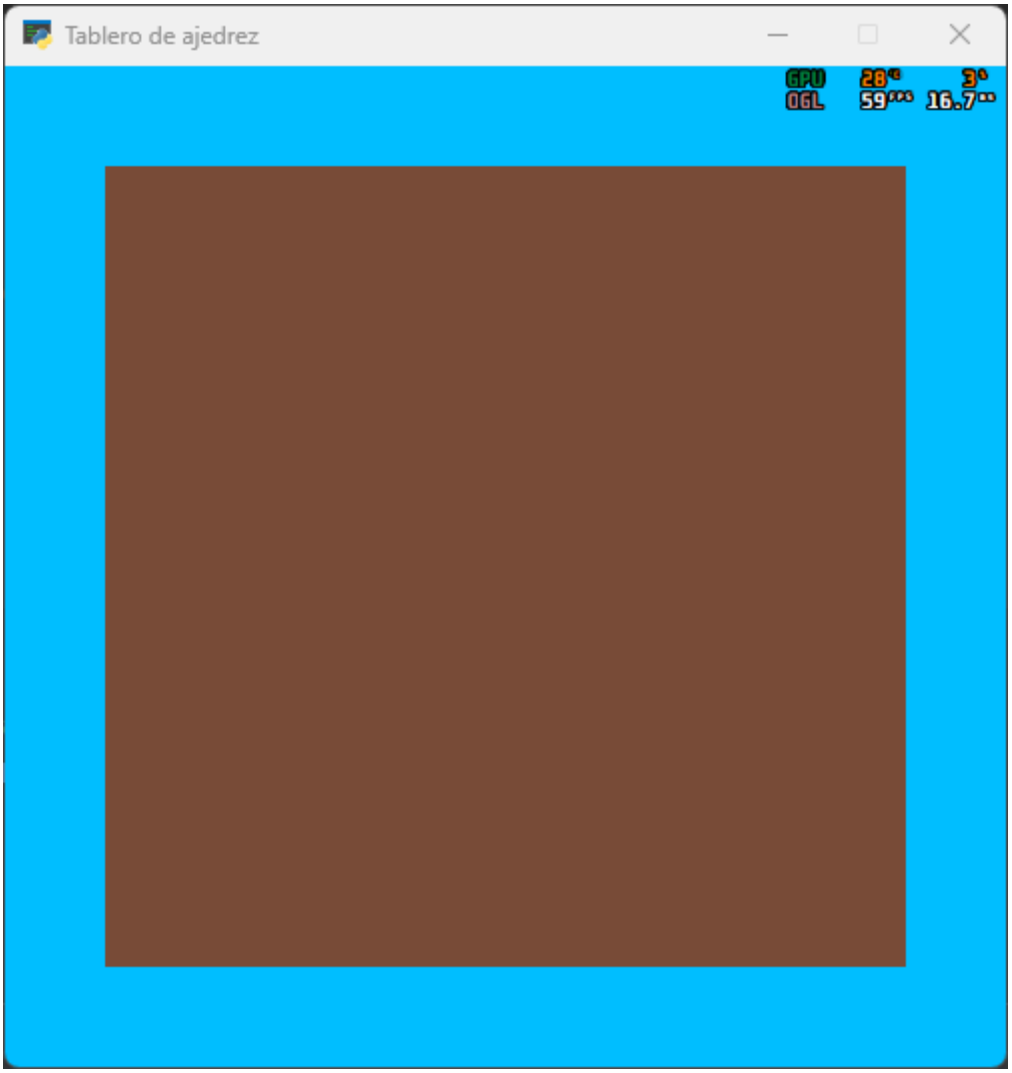
```
def main():
    arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Tablero de ajedrez")
    arcade.set_background_color(arcade.csscolor.DEEP_SKY_BLUE)
    arcade.schedule(dibujar, 1 / 60)
    arcade.run()


if __name__ == "__main__":
    main()
```

Para el dibujo del tablero vamos a intentar replicar una imagen de un tablero de madera cualquiera:



Este tablero es muy interesante porque parte de un color de fondo que concuerda con las casillas para las fichas negras, las blancas son otros cuadrados de color más claro que podemos pintar por encima. Así que empecemos con un cuadrado marrón de 400x400 para el fondo:

```
def dibujar_tablero():
    # Primero dibujamos el fondo marrón
    arcade.draw_rectangle_filled(250, 250, 400, 400, color=(120, 75, 59))


def dibujar(intervalo):
    arcade.start_render()
    dibujar_tablero()
```



Ahora tenemos que dibujar las casillas blancas por encima dando forma al tablero. Como tenemos 400x400 píxeles y el tablero tiene 8x8 casillas podemos dividir 400/8 para saber que cada casilla debe ocupar exactamente 50x50 píxeles.

Dado que el tablero consta de muchas casillas podemos crear una función para dibujar casillas, por defecto el color podría ser el amarillo de las casillas blancas. Creo que sería mejor idea dibujar las casillas de forma absoluta en relación a la esquina inferior izquierda de la ventana así que esta vez utilizaremos la función *draw_lrtb_rectangle_filled*:

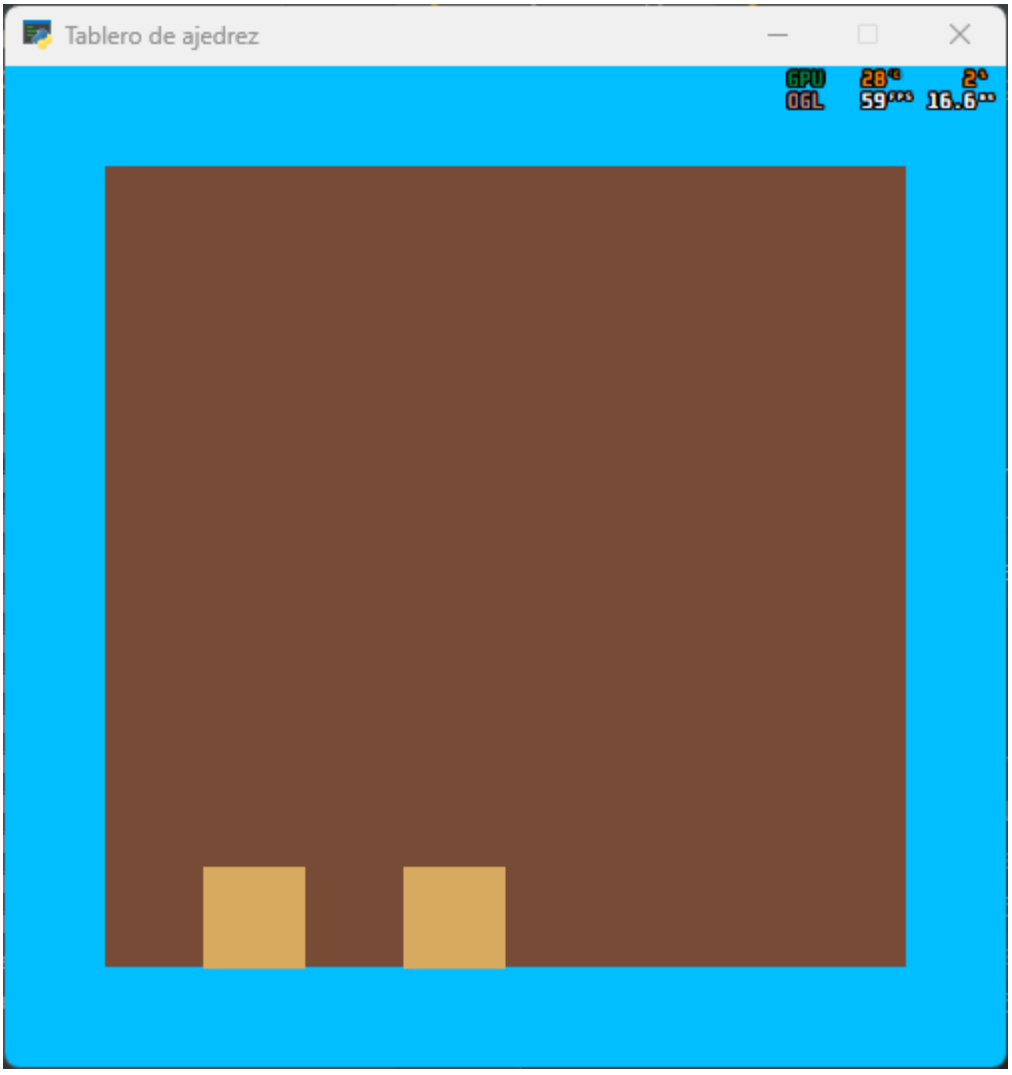
```
def dibujar_casilla(x, y, ancho, alto, color=(217, 171, 99)):
    """Función para dibujar una casilla con una posición, tamaño y color"""
    arcade.draw_lrtb_rectangle_filled(x, x + ancho, y + alto, y, color)
```

Vamos a probar nuestra función dibujando la primera casilla blanca, que según la foto se encuentra a 100 píxeles de ancho y 50 píxeles de alto desde la esquina izquierda. Recordad que el origen *(0,0)* en el módulo *Arcade* corresponde la esquina inferior izquierda:

```
def dibujar_tablero():
    # Primero dibujamos el fondo marrón
    arcade.draw_rectangle_filled(250, 250, 400, 400, color=(120, 75, 59))
    # Dibujamos la primera casilla blanca
    dibujar_casilla(100, 50, 50, 50)
```

La segunda casilla blanca la deberíamos dibujar 50 píxeles a la derecha de la primera, teniendo en cuenta que la primera casilla ya tiene un ancho de 50, serían 100 + 50 + 50 = 200 píxeles en ancho y 50 píxeles de alto respecto a la esquina inferior izquierda:

```
# Dibujamos la primera casilla blanca
dibujar_casilla(100, 50, 50, 50)
# Dibujamos la segunda casilla blanca
dibujar_casilla(200, 50, 50, 50)
```



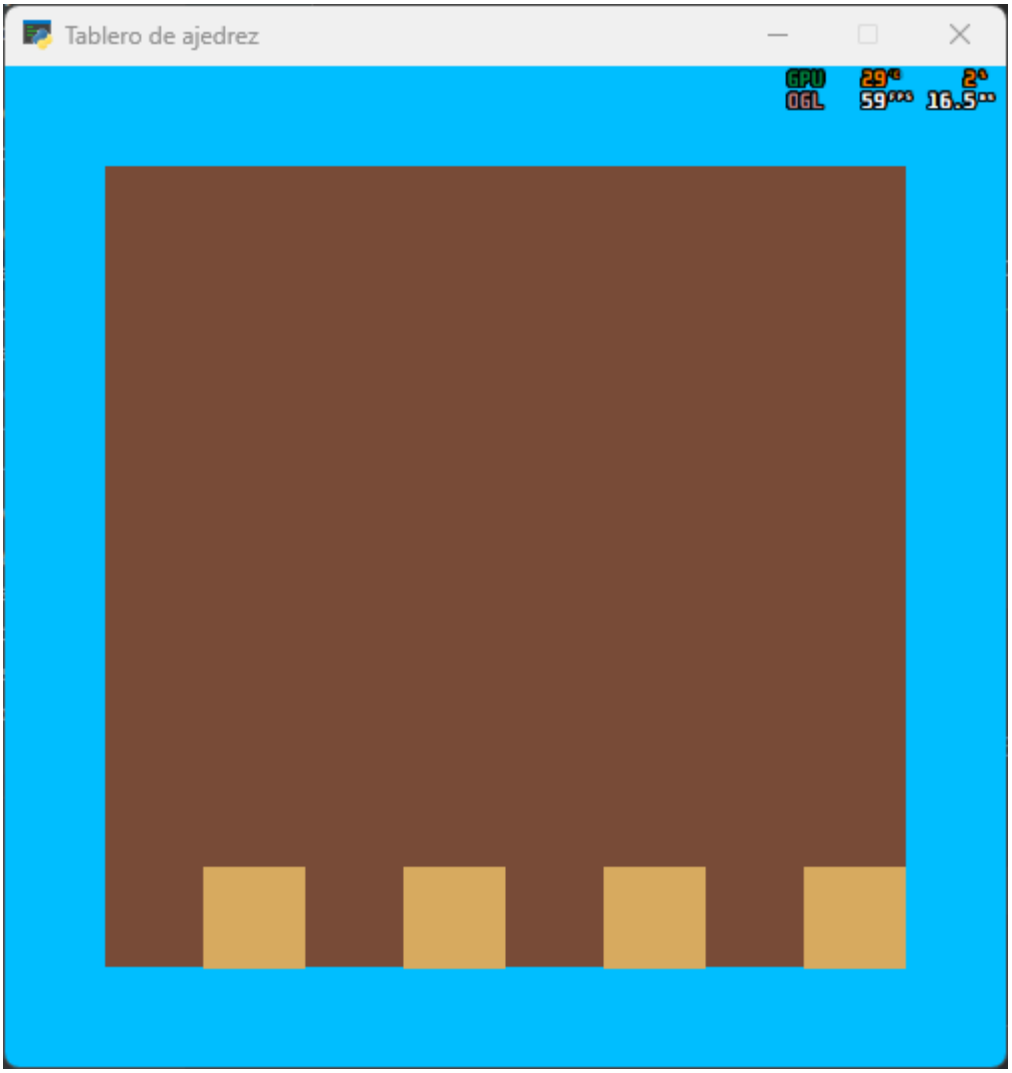
Otra forma de hacer lo mismo sería simplemente sumar la distancia entre ambas casillas:

```
# Dibujamos la primera casilla blanca
```

```
dibujar_casilla(100, 50, 50, 50)
# Dibujamos la segunda casilla blanca
dibujar_casilla(100 + 100, 50, 50, 50)
```

¿Con lo que hemos visto se os ocurre alguna forma de automatizar esto con algún bucle? ¿Seríais capaces de dibujar las casillas blancas de la primera fila automáticamente? Una pista, con un for es muy fácil, son 4 casillas blancas separadas 100 píxeles horizontalmente entre ellas:

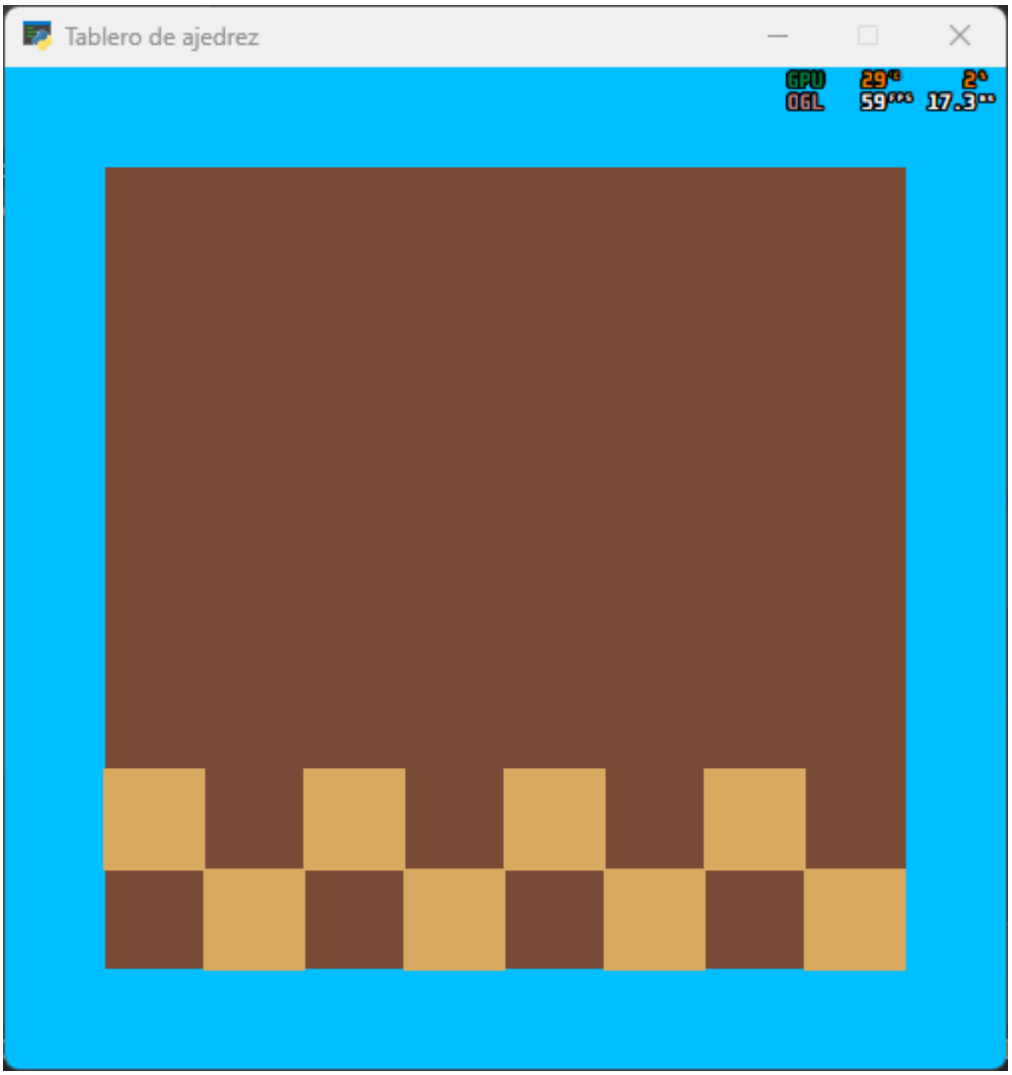
```
# Dibujamos las casillas blancas de la primera fila
for i in range(1, 5):
   dibujar_casilla(100 * i, 50, 50, 50)
```



Vamos a por la segunda fila, en esta ocasión la altura es 100 píxeles más arriba y se empieza a 50 píxeles del lado izquierdo. ¿Cómo quedaría el bucle de la segunda fila? ¿No sería lo mismo pero restando 50 píxeles a la posición horizontal?

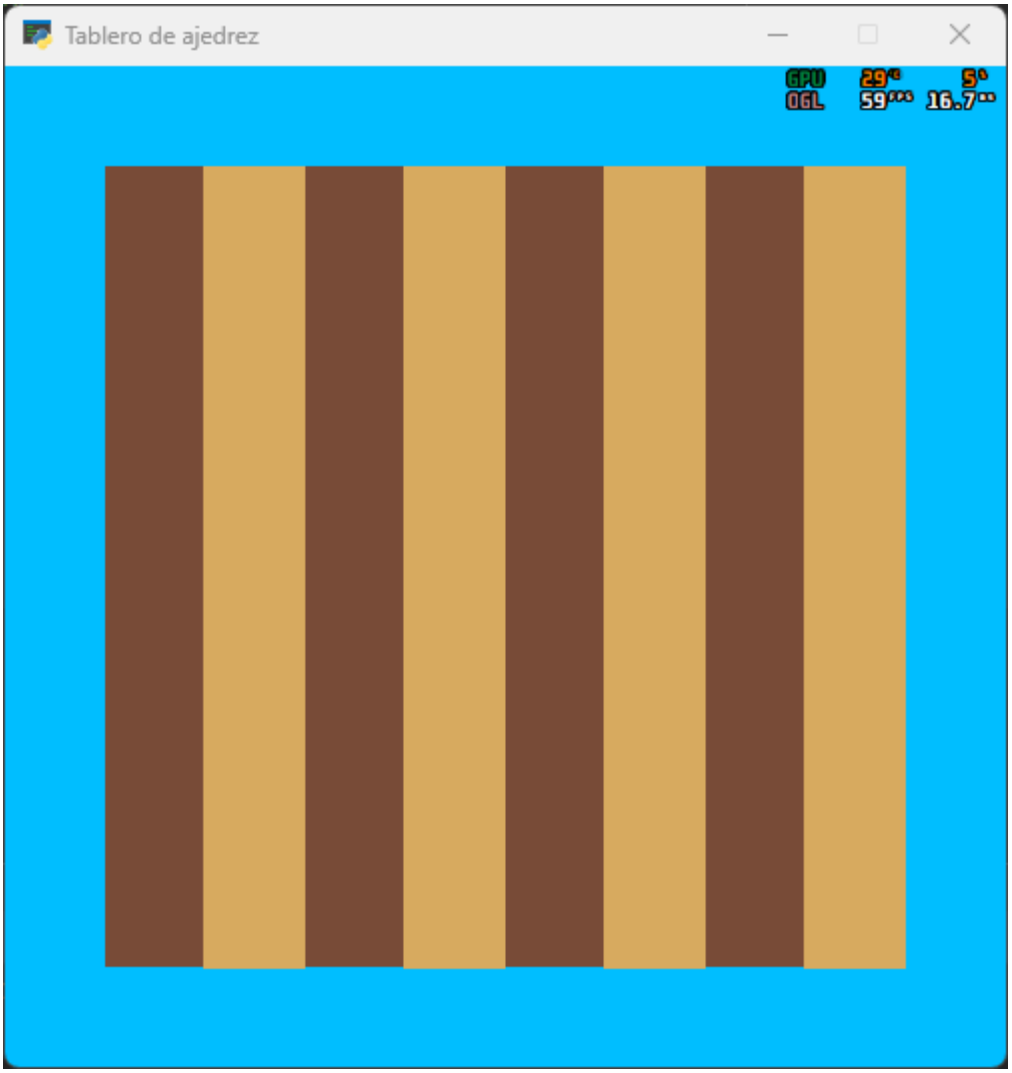
```
# Dibujamos las casillas blancas de la primera fila

for i in range(1, 5):
```

```
    dibujar_casilla(100 * i, 50, 50, 50)

# Dibujamos las casillas blancas de la segunda fila
for i in range(1, 5):
    dibujar_casilla(100 * i - 50, 100, 50, 50)
```



Con esto ya lo tenemos todo, solo debemos encontrar alguna forma de automatizar el dibujo de las 8 filas. ¿Pistas? ¡Bucles *for* anidados modificando la altura! Vamos a intentar por ahora dibujar la primera fila 8 veces:

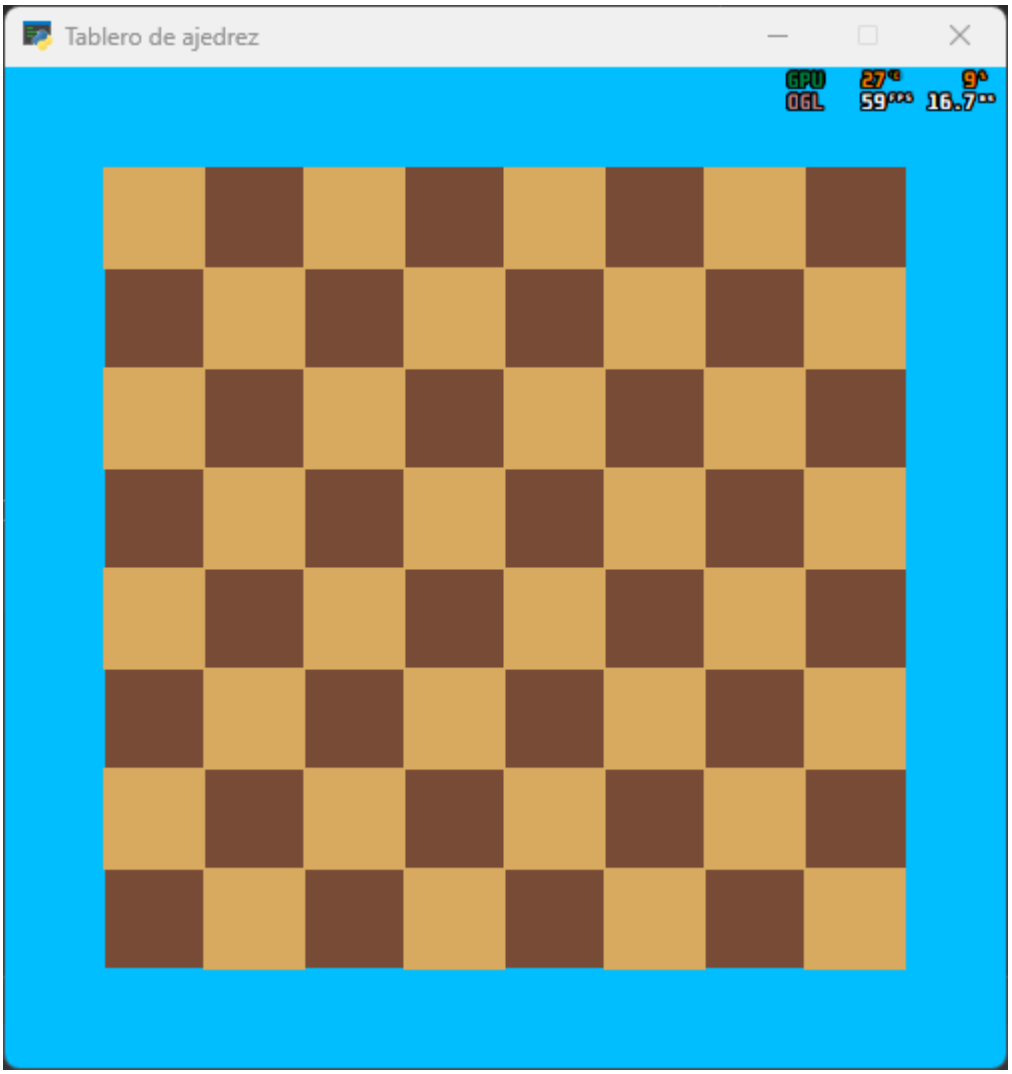
```
# Generamos un bucle para dibujar las 8 filas
for j in range(1, 9):
    # Cada fila estará 100 píxeles más arriba
    for i in range(1, 5):
        dibujar_casilla(100 * i, 50 * j, 50, 50)
```



¡Perfecto! Ahora viene la última parte, cada dos filas debemos rectificar la posición 50 píxeles a la izquierda. ¿Podemos reutilizar el código de alguna forma? ¡Claro! Os daré una pista, utilizad una condición para comprobar el índice de la fila. La primera fila no necesita rectificación, pero la segunda sí… Buscad algún patrón que podáis automatizar:

```
def dibujar_tablero():
    # Primero dibujamos el fondo marrón
    arcade.draw_rectangle_filled(250, 250, 400, 400, color=(120, 75, 59))
    # Generamos un bucle para las 8 filas a dibujar
    for j in range(1, 9):
        # Cada fila estará 50 píxeles más arriba
        for i in range(1, 5):
            # Si la fila es impar hay que rectificarla (restar 1 al índice j)
            # si j-1 == 0 es par, j
            if (j - 1) % 2 == 0:
                dibujar_casilla(100 * i, 50 * j, 50, 50)
            # Pero si es impar la rectificamos a la izquierda
            else:
                dibujar_casilla(100 * i - 50, 50 * j, 50, 50)
```

¡Listo! He tenido que restar 1 al índice de las filas porque mi rango empieza en 1 y la idea es empezar con el 0 porque es par. Si tenéis algún problema revisad el rango para adaptar vuestro índice:



Y ya está, así es como se dibuja un tablero de ajedrez utilizando un bucle anidado y una condición. ¿Qué os parece? ¿Os ha resultado fácil o difícil? Más adelante en el curso introduciremos una forma más sencilla de manejar las cuadrículas que tiene el módulo Arcade.

Resumen del tema

En este tema hemos aprendido a utilizar el bucle *for* cuyo propósito es realizar iteraciones un número determinado de veces en función de los elementos de una colección o de un rango, que por si no lo recordáis, se diferencia de una colección en que no se almacena en la memoria, sino que se genera sobre la marcha mediante la función *range*.

También vimos que el *for* al igual que el *while* permite el uso de las instrucciones *break* y *continue* para controlar las rupturas del bucle y de la iteración. Y por último dedicamos el resto de la unidad a los bucles anidados realizando diferentes dibujos en la terminal y un tablero de ajedrez con gráficos primitivos.

Errores y excepciones

Errar es humano

Es natural cometer errores en nuestro trabajo y la programación no es una excepción. El problema de un error en un programa informático es que el sistema no sabe cómo proceder y por eso detendrá abruptamente el flujo del código.

Para evitar esta situación la mayoría de lenguajes de programación proveen de una técnica conocida como manejo de excepciones. Tal como su nombre indica, una excepción a un error es un caso excepcional en el que incluso habiendo ocurrido un fallo el programa podrá seguir funcionando si así lo habremos previsto.

No hay mejor forma de entenderlo que con la práctica.

Manejando excepciones

Un error común es el de leer un texto mediante la función *input* y tratar de convertirlo en un número. Esta función necesita que la cadena de entrada sea un número, sino ocurrirá un error:

```
"""
Fichero: 02-control-de-flujo/21-manejar-excepciones.py
"""

numero = int("Hola")
# ValueError: invalid literal for int() with base 10: 'Hola'
```

Todos los errores tienen un nombre y una descripción, en este caso el nombre es ***ValueError*** y la descripción nos dice que se está usando un literal inválido en la función *int()* en lugar de un decimal.

El nombre del error es importante porque nos permite configurar una excepción en caso de que ocurra. ¿Cómo se hace? Pues envolviendo la parte problemática en un bloque de excepción *try*:

```
try:
    numero = int("Hola")
```

De esta manera le decimos al código “prueba a realizar lo siguiente”. Si funciona bien pero en caso de que falle debemos definir un contra bloque llamado *except* de forma similar al bloque *if-else*:

```
try:
    numero = int("Hola")
except:
    print("Ha ocurrido un error al convertir el texto a número")
```

Esto funcionará como una excepción genérica y no es una buena práctica, el propio PyCharm nos indica que es una forma demasiado “vaga” de hacerlo. La forma correcta consiste en indicar el nombre del error específico:

```
except ValueError:
```

```
    print("Ha ocurrido un error al convertir el texto a número")
```

Incluso podemos ir más allá y guardar el error en una variable para consultar su información:

```
except ValueError as error:
    print(type(error), error)
```

Con lo que sabemos estamos capacitados para programar una función que lea un número por teclado y que en caso de fallar informe al usuario para que vuelva a intentarlo:

```
def leer_numero():
    # Creamos una variable vacía para un número
    numero = None
    # Mientras la variable no se lea correctamente
    while not numero:
        # Intentamos leer y transformar el texto a número
        try:
            numero = int(input("Introduce un número: "))
            # Si funciona lo devolvemos
            return numero
        # Si ocurre un error avisamos y empezamos de nuevo
        except ValueError:
            print("Valor incorrecto, vuelve a intentarlo.")


edad = leer_numero()
print("La edad es", edad)
```

A partir de ahora ya no tenéis excusa para obligar a un usuario a introducir correctamente un número con el teclado.

Múltiples excepciones

En una misma pieza de código pueden ocurrir muchos errores distintos y quizá nos interese actuar de forma diferente en cada caso. En ese caso podemos definir múltiples excepciones como de bloques *elif* se tratasen. Veamos un ejemplo con más de un posible error:

```
"""
Fichero: 02-control-de-flujo/22-multiples-excepciones.py
"""

try:
    n = float(input("Introduce un número divisor: "))
    print("El resultado de la división es ", 5/n)
except ValueError:
    print("Debes introducir una cadena que sea un número")
except ZeroDivisionError:
    print("No se puede dividir por cero, prueba otro número")
```

Por último podemos definir una excepción genérica para otros posibles errores no identificados:

```
except Exception as e:
   print("Ha ocurrido un error no previsto", type(e).__name__ )
```

Sin más, como veis es un ejemplo muy sencillo y no tiene más misterio.

Invocando una excepción

Como curiosidad podemos invocar una excepción de forma intencional incluso si el sistema no falla por sí mismo. Los errores pueden dejar trazas en los registros, por lo que en algunas ocasiones quizá nos interesa provocar uno en lugar de mostrar un *print* poco elegante:

```
"""
Fichero: 02-control-de-flujo/23-invocar-excepcion.py
"""

def funcion(algo=None):
    if algo is None:
        print("Error! No se permite un valor nulo (con un print)")

funcion()
```

Como siempre debemos envolver el código tendiente a errores en un *try-except* y dentro haremos la invocación del error deseado mediante la palabra reservada *raise*:

```
def funcion(algo=None):
    try:
        if algo is None:
            raise ValueError("Error! No se permite un valor nulo")
    except ValueError:
        print("Error! No se permite un valor nulo (desde la excepción)")

funcion()
```

La lista completa de errores se puede encontrar en la documentación oficial de Python. Incluso es posible crear tus propios errores pero no es algo frecuente así que no vamos a profundizar en ello.

Resumen del tema

En esta pequeña unidad hemos aprendido sobre los errores y cómo manejarlos gracias a las excepciones, unos bloques de código que permiten definir casos excepcionales. Hemos visto como aplicarlas en algunos ejemplos, incluso de forma múltiple y también a invocarlas utilizando sus códigos predeterminados.

Tuplas

Definición

Empezamos nuestro repaso por las colecciones con las tuplas, unas estructuras de datos no homogéneas, ordenadas e inmutables. Esto significa que pueden contener datos de diferentes tipos, los datos conservan el orden en el que se definen y su contenido no puede modificarse una vez creadas.

Una tupla se define como una enumeración de datos entre paréntesis separados por comas:

```
"""
Fichero: 03-colecciones/01-tuplas.py
"""

tupla = (100, "Hola", False, 3.14)
print(type(tupla))
print(tupla)
```

Si asignamos múltiples datos separados por comas a una variable también se definirá una tupla:

```
tupla = 100, "Hola", False, 3.14
```

Una de las peculiaridades de las tuplas es que sus valores pueden asignarse a múltiples variables en la misma línea tomando como referencia su orden:

```
entero, cadena, booleano, flotante = (100, "Hola", False, 3.14)
print(entero, cadena, booleano, flotante)
```

Esto también funcionará si lo escribimos son paréntesis:

```
entero, cadena, booleano, flotante = 100, "Hola", False, 3.14
```

Al permitir este tipo de asignación, una función puede devolver múltiples valores separados por comas. Estos se pueden asignar a una tupla o directamente a múltiples variables:

```
def funcion():
    return 100, "Hola", False, 3.14

tupla = funcion()
print(tupla)

entero, cadena, booleano, flotante = funcion()
print(entero, cadena, booleano, flotante)
```

Como véis las tuplas no solo sirven para almacenar múltiples datos sino que también se utilizan indirectamente en asignaciones y retornos múltiples.

Índices e inmutabilidad

Las tuplas al ser colecciones ordenadas permiten acceder a sus elementos a partir de su posición, correspondiendo el primer elemento al índice cero. Para hacer referencia a una posición pasaremos el índice entre corchetes:

```
"""
Fichero: 03-colecciones/02-indices-inmutables.py
"""

tupla = 100, "Hola", False, 3.14

print(tupla[0])
print(tupla[1])
print(tupla[2])
print(tupla[3])
```

Acceder a un índice fuera del tamaño de la tupla dará un error de fuera de rango:

```
print(tupla[100])  # IndexError: tuple index out of range
```

También podemos usar índices negativos, correspondiendo el *-1* al último elemento de la tupla y así sucesivamente:

```
print(tupla[-1])
print(tupla[-2])
print(tupla[-3])
print(tupla[-4])
```

Pero quizá el rasgo más identificativo de una tupla es su inmutabilidad. Eso implica que una vez creada sus elementos no pueden modificarse, si intentamos una asignación a una posición de la tupla debería saltarnos un error:

```
tupla[0] = 99999  # TypeError: 'tuple' object does not support item assignment
```

Esta propiedad es la que diferencia una lista y una tupla, dándole sentido a que sean tipos distintos.

Técnica del slicing

Tanto las tuplas como las listas permiten una técnica para recuperar subconjuntos de sus elementos, se trata del slicing. El slicing no es más que un segundo índice referido en el acceso a la tupla separado del primero con dos puntos. Funciona de forma parecida a la función range. El primer índice indica la posición de inicio del subconjunto y el segundo la del fin pero excluyéndose:

```
"""
Fichero: 03-colecciones/03-técnica-slicing.py
"""

tupla = 100, "Hola", False, 3.14, "Adiós", 999, True, 2.72
```

```
subtupla = tupla[2:7]
print(subtupla)  # (False, 3.14, 'Adiós', 999, True)
```

Dejar vacío el primer índice significa desde el principio:

```
print(tupla[:5])  # (100, 'Hola', False, 3.14, 'Adiós')
```

Dejar vacío el segundo índice significa hasta el final:

```
print(tupla[5:])  # (999, True, 2.72)
```

Dejar ambos vacíos significa de principio a fin, lo que generará una copia completa de la tupla:

```
print(tupla[:])  # (100, "Hola", False, 3.14, "Adiós", 999, True, 2.72)
```

También es posible utilizar índices negativos, por ejemplo para indicar que recupere una subtupla con los últimos 4 elementos, en lugar de indicar un segundo índice 5 lo pasamos el primero en -4:

```
print(tupla[-4:])  # ('Adiós', 999, True, 2.72)
```

Para rematar también se puede utilizar un tercer índice a modo de salto, igual que con la función *range*. Por ejemplo podemos recuperar una subtupla de otra cada dos elementos con un paso de 2:

```
print(tupla[::2])  # (100, False, 'Adiós', True)
```

Si queremos que empiece en el segundo elemento podemos indicar un primer índice con el valor 1:

```
print(tupla[1::2])  # ('Hola', 3.14, 999, 2.72)
```

Por último algo muy interesante es que si el tercer índice tiene un paso de -1, lo que estamos indicando es que el conjunto se devuelva desde el final hasta el principio:

```
print(tupla[::-1])  # (2.72, True, 999, 'Adiós', 3.14, False, 'Hola', 100)
```

Este es un truco muy común en Python para voltear tanto una tupla como una lista.

Tuplas anidadas

Todas las colecciones en Python son no homogéneas, eso significa que permiten almacenar cualquier tipo de dato en su interior y cuando digo cualquiera es literalmente cualquiera, incluso otras tuplas.

Esto es más común de lo que parece porque en una tupla podemos almacenar datos de una entidad común y luego en otra tupla varias de esas entidades, lo váis a ver claro con un ejemplo.

Supongamos unas tuplas para almacenar la información de varias personas. En la primera posición tendremos el nombre de una persona, luego el apellido, la edad y finalmente la altura en metros:

```
"""
Fichero: 03-colecciones/05-tuplas-anidadas.py
"""

persona1 = ("María", "Pérez", 23, 1.58)
persona2 = ("Fernando", "López", 38, 1.71)
persona3 = ("Victoria", "Escribano", 33, 1.66)
```

Dado que esta información está almacenada siguiendo un orden común nos interesa agrupar los datos para que en lugar de tres variables tengamos una única colección. En ese caso podemos simplemente hacer lo siguiente:

```
personas = (
   ("María", "Pérez", 23, 1.58),
   ("Fernando", "López", 38, 1.71),
   ("Victoria", "Escribano", 33, 1.66)
)
```

Esto es bastante práctico porque nos sirve como pequeña base de datos. Si queremos consultar la información de la primera persona podemos acceder mediante el índice 0 y así sucesivamente:

```
print(personas[0])  # ('María', 'Pérez', 23, 1.58)
```

Pero todavía podemos ir más allá, sabiendo que en la primera posición de esas tuplas se encuentra el nombre, como estamos recuperando otra tupla, podemos añadir otro índice para acceder a él:

```
print(personas[0][0])  # "María"
```

Esto es un poco confuso, pero mediante unas constantes representando índices textuales sería mucho más cómodo trabajar con estas personas y sus propiedades:

```
MARIA, FERNANDO, VICTORIA  = 0, 1, 2
NOMBRE, APELLIDO, EDAD, ALTURA = 0, 1, 2, 3

print(personas[MARIA][APELLIDO])  # "María
print(personas[FERNANDO][EDAD])  # 38
print(personas[VICTORIA][ALTURA])  # 1.66
```

Esto es solo un ejemplo para empezar a entrever la utilidad de las colecciones como bases de datos simples en la memoria.

Acceso secuencial

Cuando aprendimos a trabajar con los bucles for explicamos que están pensados para trabajar en conjunto con las colecciones y la función *range*. Las colecciones tienen naturaleza iterable, eso significa que se pueden recorrer sus elementos.

Recuperando nuestra tupla de personas podemos recorrerlas secuencialmente para acceder a su información de forma muy sencilla:

```
"""
Fichero: 03-colecciones/06-acceso-secuencial.py
"""

personas = (
    ("María", "Pérez", 23, 1.58),
    ("Fernando", "López", 38, 1.71),
    ("Victoria", "Escribano", 33, 1.66)
)

for persona in personas:
    print(type(persona), persona)
```

En cada iteración la variable *persona* va almacenando cada subtupla de la tupla *personas*. Lo interesante es que podemos anidar otro bucle *for* para recorrer sus campos:

```
for persona in personas:
    print(type(persona), persona)
    for campo in persona:
        print("\t", type(campo), campo)
```

La posibilidad de recorrer de forma automatizada un conjunto nos abre la puerta a realizar tareas como búsquedas comparando alguno de los campos. Por ejemplo, si necesitamos consultar únicamente la información de una persona en concreto entonces podemos recorrer todos los registros e irlos comparando con ese nombre:

```
def buscar_por_nombre(nombre):
    print("Buscando a", nombre)
    for persona in personas:
        print("... Recorriendo", persona[0])
        # Buscamos la información de la persona a partir del nombre
        if persona[0] == nombre:
            # Si la encontramos imprimirmos sus datos
            print("...... Nombre: ", persona[0])
            print("...... Apellido: ", persona[1])
            print("...... Edad: ", persona[2])
            print("...... Altura: ", persona[3])
            # Justo después rompemos el bucle para optimizar memoria
            break
```

```
buscar_por_nombre("Victoria")
buscar_por_nombre("María")
```

En una base de datos real en lugar de un nombre podríamos utilizar un campo único como un documento de identidad o un identificador del registro, pero con esto ya os hacéis una idea.

Métodos internos

Como todas las colecciones las tuplas tienen el método especial *__len__* que lleva una cuenta del número de elementos que contiene. Al ser un método especial no está pensado para ejecutarlo directamente sino a través de la función *len* que hace de accesor externo:

```
"""
Fichero: 03-colecciones/07-metodos-tuplas.py
"""

numeros = 2, 3, 6, 7, 8, 9, 0, 5, 3, 2, 34, 65, 23, 112, 545
print("Longitud de la tupla:", len(numeros))
```

Aparte tienen dos métodos internos que nos proporcionan algunas funcionalidades extendidas:

- *count*: Devuelve el número de veces que se encuentra un valor en la tupla.
- *index*: Busca en la tupla un valor especificado y devuelve su posición.

```
print("Conteo del valor 2: ", numeros.count(2))
print("Conteo del valor 10: ", numeros.count(10))
print("Índice del valor 2: ", numeros.index(2))
print("Índice del valor 65: ", numeros.index(65))
print("Índice del valor 999: ", numeros.index(999))  # error
```

Las tuplas son las colecciones más sencillas y por eso son una buena forma de empezar a trabajar, además comparten casi todas sus propiedades con las listas así que todo eso que nos ahorramos.

Valor en colección

Por último quiero compartir con vosotros un pequeño truco para realizar una búsqueda *in situ* de un valor en cualquier colección. Se aplica una lógica parecida a la de iterar con el for hasta encontrar un valor pero lo hace de forma automática y mejor optimizada, resulta extremadamente útil:

```
"""
Fichero: 03-colecciones/08-valor-en-coleccion.py
"""

numeros = 2, 3, 6, 7, 8, 9, 0, 5, 3, 2, 34, 65, 23, 112, 545

print(112 in numeros)  # True
print(999 in numeros)  # False

def buscar(valor):
    if valor in numeros:
```

```
        print(f"{valor} se encuentra en la lista")
    else:
        print(f"{valor} no se encuentra en la lista")


buscar(34)
buscar(999)
```

El problema sucede al trabajar con colecciones anidadas, pues en esos casos necesitamos una referencia exacta del valor que buscamos, o en el caso único de las tuplas, al ser inmutables, una copia exacta de la misma tupla que buscamos:

```
personas = (
    ("María", "Pérez", 23, 1.58),
    ("Fernando", "López", 38, 1.71),
    ("Victoria", "Escribano", 33, 1.66)
)


def buscar_persona(persona):
    if persona in personas:
        print(f"{persona[0]} se encuentra en la lista")
    else:
        print(f"{persona[0]} no se encuentra en la lista")


buscar_persona("Victoria")  # Esto no funcionará
buscar_persona(personas[2])  # La misma referencia también funcionará
buscar_persona(("María", "Pérez", 23, 1.58))  # Una copia de la tupla también
```

Como veis se trata de una funcionalidad muy práctica.

Resumen del tema

En esta unidad hemos introducido el uso de las tuplas, las colecciones más básicas. Hemos aprendido a definirlas y a acceder a sus valores con índices y slicing. También hemos visto la utilidad de anidar tuplas dentro de tuplas para replicar estructuras de datos lógicas y cómo acceder a su información secuencialmente utilizando bucles *for*. Finalmente hemos visto sus dos métodos internos y como utilizar el operador *in* para saber fácilmente si un elemento se encuentra en la colección.

Listas

Definición

Si las tuplas eran las colecciones básicas de Python, las listas son las colecciones esenciales. Como ya vengo diciendo se trata de tuplas mutables que se pueden modificar una vez definidas y ese pequeño detalle extiende muchísimo las posibilidades de este tipo.

Vamos a empezar repasando brevemente todo lo que podemos hacer con una lista que también se puede hacer con una tupla y luego nos enfocaremos en la parte dinámica de manipular su contenido.

Una lista se define como una tupla pero utilizando corchetes para delimitar su contenido:

```
"""
Fichero: 03-colecciones/08-listas.py
"""

# Definición
lista = [100, "Hola", False, 3.14, "Adiós", 999, True, 2.72]
print(type(lista))
print(lista)
```

Toda la parte de índices y slicing funciona exactamente igual, la gran diferencia es que mientras el slicing antes devolvía una subtupla inmutable, ahora devuelve una sublista mutable:

```
# Índices y slicing
print(lista[:4])     # Últimos 4 elementos
print(lista[3:])     # Primeros 3 elementos
print(lista[-3:])    # Últimos 3 elementos
print(lista[:])      # Copia con todos los elementos
print(lista[::2])    # Sublista cada 2 elementos
print(lista[1::-2])  # Sublista cada 2 elementos empezando por el segundo
print(lista[::-1])   # Sublista completa volteada
```

Por supuesto podemos utilizar un bucle *for* para recorrer los elementos de la lista secuencialmente:

```
# Lectura secuencial
for elemento in lista:
   print(elemento, end=" ")
```

Y podemos representar bases de datos con anidación de listas, de la misma forma que con las tuplas:

```python
# Listas anidadas
personas = [
   ["María", "Pérez", 23, 1.58],
   ["Fernando", "López", 38, 1.71],
   ["Victoria", "Escribano", 33, 1.66]
]

for persona in personas:
   for campo in persona:
       print(campo, end=" ")
   print()
```

Perfecto, hasta aquí hemos visto la parte común y ahora toca aprender las diferencias.

Añadir elementos

Tenemos varias maneras de añadir un elemento a una lista, vamos a partir de una lista vacía:

```python
"""
Fichero: 03-colecciones/09-anadir-borrar.py
"""

lista = []
```

La primera forma de añadir elementos es mediante la suma de listas:

```python
# Añadir elementos sumando listas, de 1 a N elementos
lista += ["Hola", 1234]
print(lista)  # ['Hola', 1234]
```

Sumar un único elemento no funcionará, es necesario siempre que forme parte de otra lista:

```python
lista += True  # Error
lista += [True]  # Correcto
```

Pero si lo que intentamos sumar es una estructura iterable, como por ejemplo una cadena, entonces se interpretará como que se deben sumar todos los caracteres uno por uno:

```python
lista += "Adiós"  # Funciona, pero no es lo que queremos
print(lista)  # ['Hola', 1234, True, 'A', 'd', 'i', 'ó', 's']
```

Esto sucede porque las cadenas son en realidad secuencias de caracteres, que de hecho se pueden iterar uno a uno en *for*:

```python
# Una cadena es una secuencia iterable por un for
for caracter in "Esto es un texto cualquiera":
   print(caracter, end=" ")
   # E s t o   e s   u n   t e x t o   c u a l q u i e r a
```

La otra forma de añadir un elemento es mediante el método *append*:

```
lista.append("Adiós")
lista.append(999)
print(lista)  # ['Hola', 1234, True, 'Adiós', 999]
```

Tanto sumar listas como anexar elementos añadirá los elementos al final, pero existe una tercera forma de añadir un elemento, insertarlo en una posición determinada mediante el método *insert*:

```
# Insertar elementos con insert
lista.insert(0, "Python")  # Al principio, índice 0
print(lista)  # ['Python', 'Hola', 1234, True, 'Adiós', 999]
lista.insert(-2, "Mola")  # En la antepenúltima posición, índice -2
print(lista)  # ['Python', 'Hola', 1234, True, 'Mola', 'Adiós', 999]
```

En definitiva estas son las tres formas de añadir elementos:

- Sumar listas con el operador *+*.
- Anexar elementos al final con el método *append*.
- Insertar elementos en una posición con el método *insert*.

Borrar elementos

La contra parte de añadir es borrar y también tenemos a nuestra disposición diferentes formas:

```
"""
Fichero: 03-colecciones/09-borrar-elementos.py
"""

lista = ['Python', 'Hola', 1234, True, 'Mola', 'Adiós', 999]
```

Podemos borrar un elemento a partir de su valor con el método *remove*. En caso de que el elemento este repetido solo se borra la primera concordancia:

```
# Borrar elementos usando remove
lista.remove("Mola")  # Borramos el valor "Mola"
print(lista)  # ['Hola', 1234, True, 'Mola', 'Adiós', 999]
```

También podemos extraer un elemento de una posición con el método *pop*, esta forma permite guardar el elemento en lugar de borrarlo directamente:

```
# Extraer elementos usando pop y un índice
elemento = lista.pop(0)  # Borramos el índice 0 que es "Python"
print(lista, "->", elemento)  # ['Hola', 1234, True, 'Adiós', 999] -> Python
```

Alternativamente existe la posibilidad de destruir la referencia de un elemento usando la función *del*.

```
# Destruir la referencia utilizando del y un índice
del lista[-2]  # Borramos el penúltimo elemento
print(lista)  # ['Hola', 1234, True, 999]
```

Como la función *del* directamente destruye la referencia en la memoria también sirve para borrar la existencia de cualquier variable. No es que vacíe la memoria como asignar *None*, es que directamente destruye todo rastro de existencia de la variable:

```
del lista
print(lista)
# NameError: name 'lista' is not defined.
```

Dada su agresiva naturaleza yo os recomiendo utilizar solo *remove* y *pop* para borrar los elementos de una lista y evitar destrucciones no intencionadas.

Por último es posible borrar todos los elementos de una vez utilizando el método *clear*:

```
lista = [1, 2, 3, 4, 5]
lista.clear()
print(lista)  # []
```

Modificar elementos

Ya sabemos añadir y borrar elementos, solo nos falta aprender a modificarlos. Para conseguirlo debemos volver a los índices, pues para sobreescribir un valor debemos hacer referencia a su posición en la lista:

```
"""
Fichero: 03-colecciones/12-modificar-elementos.py
"""

lista = ['Hola', 1234, True, 3.14]

lista[0] = "Adiós"
lista[1] -= 234
lista[2] = not lista[2]
lista[3] **= 2

print(lista)
```

La verdad es que es bastante simple, pero ¿y para modificar los valores secuencialmente? Vamos a probar con un bucle *for* a ver si somos capaces de anular todos los valores de la lista, es decir, darles el valor *None*:

```
for elemento in lista:
   elemento = None

print(lista)
```

Pues no funciona y con lo que hemos aprendido hasta ahora deberíais ser capaces de determinar la razón. ¿No se os ocurre nada? ¿Qué contiene esta lista? Un texto, un par de números y un booleano. ¿Son estos tipos mutables o inmutables? Efectivamente, son datos inmutables, eso significa que no podemos modificarlos. Si los datos de una lista son inmutables y queremos cambiar sus valores la única forma es hacerlo mediante los índices.

El truco consiste en utilizar una variable con un número a modo de índice, si la vamos incrementando 1 en cada iteración podemos simular la posición de cada elemento en la lista:

```
i = 0
for elemento in lista:
   lista[i] = None
   i += 1
print(lista)
```

Esta es la lógica básica para realizar modificaciones secuenciales en listas, pero como ir definiendo un contador manual es tedioso existe una función que nos facilitará la vida, la vemos en la siguiente lección.

Función numeradora

Nos hemos quedado en que para modificar secuencialmente datos inmutables de una lista es imprescindible hacerlo a través de los índices:

```
"""
Fichero: 03-colecciones/13-funcion-numeradora.py
"""

lista = ['Hola', 1234, True, 3]

i = 0
for elemento in lista:
   lista[i] = None
   i += 1
print(lista)
```

¿Y si existiera una función que pudiera devolvernos en cada iteración del for tanto el índice como el elemento que estamos recorriendo? Pues esa es la función *enumerate* y se utiliza así:

```
for indice, elemento in enumerate(lista):
   lista[indice] = None
print(lista)
```

¿Cómo funciona? Para entenderlo vamos a empezar imprimiendo qué devuelve:

```
lista = ['Hola', 1234, True, 3]
lista_numerada = enumerate(lista)
print(type(lista_numerada))
```

```
print(lista_numerada)
```

Como vemos *enumerate* es literalmente un dato de ese mismo tipo y es muy parecido a un rango en el sentido de que se va generando sobre la marcha. Para entender mejor qué hace podemos transformarlo en una lista:

```
print(list(lista_numerada))  # [(0, 'Hola'), (1, 1234), (2, True), (3, 3)]
```

Esto es interesante, la función toma los elementos de una lista y genera otra lista con tuplas con el índice y el valor del elemento por separado. ¿Y si la recuperamos en el *for*?

```
for tupla in enumerate(lista):
   print(type(tupla), tupla)
```

Como véis no hay ningún problema y podemos recorrer cada tupla secuencialmente. Pero como las tuplas permiten la asignación múltiple sería una pena no aprovecharnos de ello, así que al final simplificamos el proceso:

```
for indice, elemento in enumerate(lista):
   print(i, " -> ", e)
```

En caso de solo necesitar recuperar el índice de lo elementos se suele poner la variable del elemento con una barra baja, tal que así:

```
for i, _ in enumerate(lista):
   pass
```

¿Qué os ha parecido la función *enumerate*? La vamos a utilizar mucho a partir de ahora.

Mutabilidad y copia

En esta lección vamos a recuperar el concepto de inmutabilidad que vimos durante el tema de las funciones. Por si no lo recordáis aprendimos que en Python solo los números, las cadenas, los booleanos y las tuplas inmutables, todos los demás datos son mutables.

En Python cuando se realiza una asignación de una variable a otra variable se enlaza a partir de su referencia en la memoria, si ambas variables comparten la misma dirección de memoria significa que no se está malgastando memoria para representar datos duplicados:

```
"""
Fichero: 03-colecciones/12-mutabilidad-copia.py
"""

def comparar_referencias(dato1, dato2):
   return id(dato1) == id(dato2)

numero = 123
```

```
copia_numero = numero
print("¿numero y copia_numero comparten referencia?", comparar_referencias(numero,
copia_numero))

cadena = "Hola mundo"
copia_cadena = cadena
print("¿cadena y copia_cadena comparten referencia?", comparar_referencias(cadena,
copia_cadena))

lista = [1, 2, 3]
copia_lista = lista
print("¿lista y copia_lista comparten referencia?", comparar_referencias(lista,
copia_lista))
```

La gran diferencia entre los tipos inmutables y mutables no sucede en la asignación sino en la modificación del valor. La naturaleza de un tipo inmutable impide que el valor almacenado en la memoria pueda cambiar, por esa razón se crea un nuevo espacio en la memoria y se guarda el dato modificado en otra referencia:

```
numero = 123
copia_numero = numero
copia_numero += 456  # Modificamos la copia
print("¿numero y copia_numero comparten referencia?", comparar_referencias(numero,
copia_numero))

cadena = "Hola mundo"
copia_cadena = cadena
copia_cadena += ", adiós mundo"  # Modificamos la copia
print("¿cadena y copia_cadena comparten referencia?", comparar_referencias(cadena,
copia_cadena))

lista = [1, 2, 3]
copia_lista = lista
copia_lista += [4, 5, 6]  # Modificamos la copia
print("¿lista y copia_lista comparten referencia?", comparar_referencias(lista,
copia_lista))
```

¿Entonces cómo podemos realizar una verdadera copia de un dato mutable? Pues depende, en las listas es tan fácil como recuperar una sublista de todos los elementos utilizando slicing:

```
lista = [1, 2, 3]
copia_lista = lista[:]
copia_lista += [4, 5, 6]  # Modificamos la copia
print("¿lista y copia_lista comparten referencia?", comparar_referencias(lista,
copia_lista))
```

O mediante el método *copy* para crear una “copia profunda” en una nueva dirección de la memoria:

```
copia_lista = lista.copy()  # Realizamos una copia profunda
```

Lo que hemos aprendido en esta lección es muy importante. Distinguir cómo funcionan los tipos inmutables y mutables es básico para el correcto uso del lenguaje Python y eso es algo que se hace patente al trabajar con clases y objetos, un tema que trataremos en breve.

Repaso de los métodos

A lo largo de la unidad hemos visto en acción casi todos los métodos de las listas, repasémoslos:

- *index*: Busca un valor especificado y devuelve su posición.
- *append*: Anexa un elemento al final de la lista.
- *insert*: Inserta un elemento en una posición de la lista.
- *remove*: Borra el primer elemento con el valor indicado.
- *pop*: Borra un elemento de una posición de la lista.
- *clear*: Vacía todos los elementos de una lista.
- *copy*: Crea una copia profunda de la lista en una referencia distinta.
- *count*: Devuelve el número de veces que se encuentra un valor.
- *extend*: Permite unir dos listas, es el equivalente a sumar dos listas.
- *reverse*: Permite voltear una lista, es equivalente al tercer índice negativo *[::-1]*

Comprensión de listas

La comprensión de listas, del inglés *list comprehensions*, es una funcionalidad que nos permite crear listas en una sola línea de código. Esto se ve mucho mejor en la práctica, así que vamos a trabajar distintos ejemplos.

1) Empecemos creando una lista con las letras de una palabra:

```
# Método tradicional
lista = []
for letra in 'casa':
   lista.append(letra)
print(lista)

# Con comprensión de listas
lista = [letra for letra in 'casa']
print(lista)
```

Como vemos, gracias a la comprensión de listas podemos indicar directamente cada elemento que va a formar la lista, en este caso la letra, a la vez que definimos el *for*.

2) Ahora vamos a crear una lista con las potencias de 2 de los primeros 10 números:

```
# Método tradicional
lista = []
for numero in range(11):
   lista.append(numero ** 2)
print(lista)

# Con comprensión de listas
lista = [numero ** 2 for numero in range(11)]
```

```
print(lista)
```

De este ejemplo podemos aprender que es posible modificar al vuelo los elementos que van a formar la lista.

3) Probemos a crear una lista con los todos los múltiplos de 2 entre 0 y 10, ambos incluidos:

```
# Método tradicional
lista = []
for numero in range(0, 11):
   if numero % 2 == 0:
       lista.append(numero)
print(lista)

# Con comprensión de listas
lista = [numero for numero in range(0, 11) if numero % 2 == 0]
print(lista)
```

En este caso podemos observar que incluso podemos marcar una condición justo al final para añadir o no el elemento en la lista.

4) Para acabar un ejemplo con listas anidadas, vamos a crear una lista de pares a partir de otra lista creada con las potencias de 2 de los primeros 10 números:

```
# Método tradicional
lista = []
for numero in range(0, 11):
   lista.append(numero ** 2)

pares = []
for numero in lista:
   if numero % 2 == 0:
       pares.append(numero)

print(pares)

# Con comprensión de listas
lista = [numero for numero in
        [numero ** 2 for numero in range(0, 11)]
        if numero % 2 == 0]
print(lista)
```

Crear listas a partir de listas anidadas nos permite llevar la comprensión de listas al siguiente nivel y además no hay un límite.

Cuadrícula animada

Para sintetizar lo aprendido vamos a hacer una divertida práctica. En lugar de dibujar un tablero, en esta ocasión vamos a dibujar una cuadrícula cuyas celdas tendrán una referencia a un elemento de

una lista donde almacenaremos un color. El objetivo será modificar el color de las celdas dinámicamente cada cierto tiempo. Para aprovechar código partiremos del ejemplo del tablero:

```
"""
Fichero: 03-colecciones/16-cuadricula-dinamica.py
"""

import arcade
import arcade.gui

SCREEN_WIDTH = 500
SCREEN_HEIGHT = 500


def dibujar_casilla(x, y, ancho, alto, color):
    """Función para dibujar una casilla con una posición, tamaño y color"""
    arcade.draw_lrtb_rectangle_filled(x, x + ancho, y + alto, y, color)


def dibujar_cuadricula():
    pass


def dibujar(intervalo):
    arcade.start_render()
    dibujar_cuadricula()


def main():
    arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Cuadrícula dinámica")
    arcade.set_background_color(arcade.csscolor.BLACK)
    arcade.schedule(dibujar, 1 / 60)
    arcade.run()


if __name__ == "__main__":
    main()
```

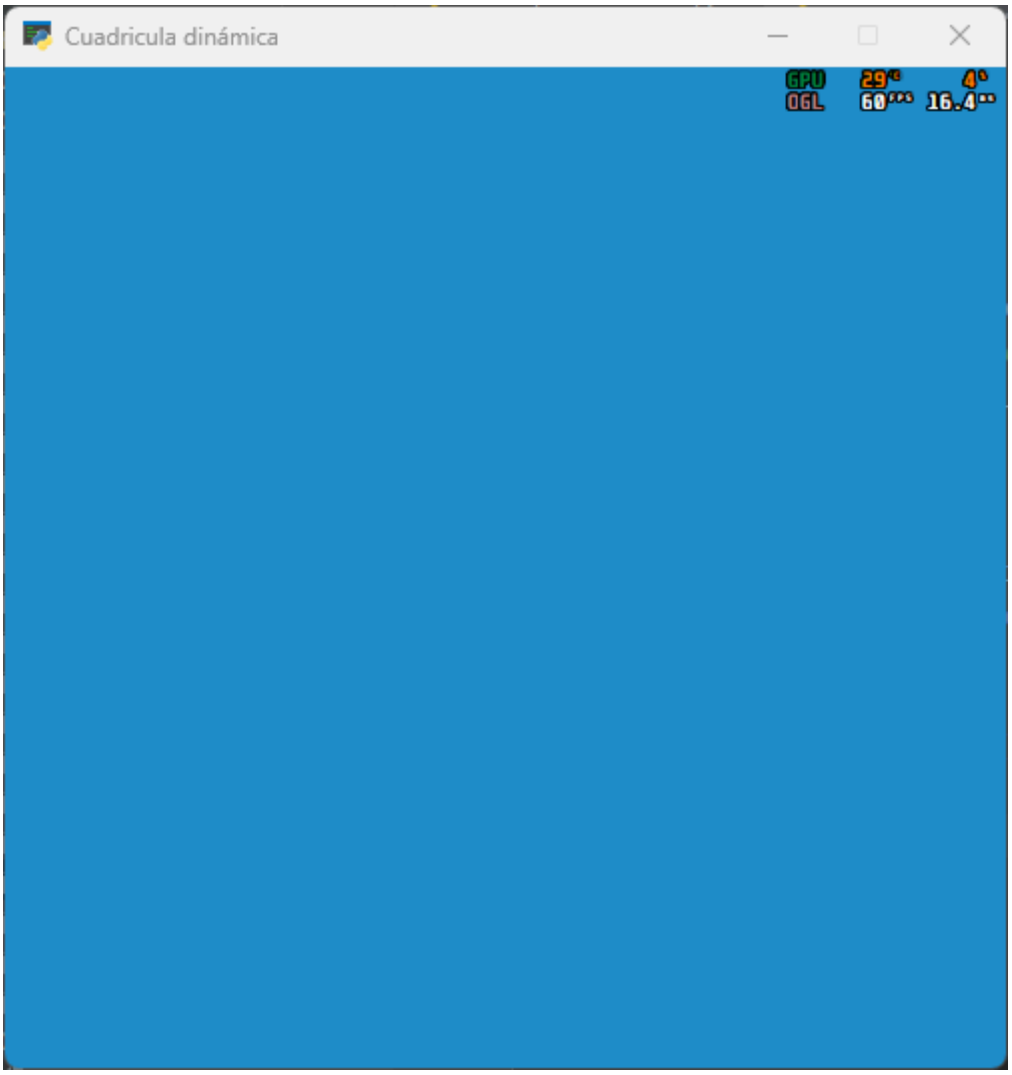
Lo primero que vamos a hacer es definir un atributo como una lista anidada para representar las filas y columnas de la cuadrícula, lo haremos dinámicamente añadiendo un color cualquiera por defecto:

```
dibujar.cuadricula = [
    [[(30, 140, 200) for j in range(10)] for i in range(10)]
```

A continuación vamos a dibujar todas las casillas de la cuadrícula con el color que tiene almacenado:

```
def dibujar_cuadricula():
    for i, fila in enumerate(dibujar.cuadricula):
        for j, columna in enumerate(fila):
            dibujar_casilla(50 * i, 50 * j, 50, 50, dibujar.cuadricula[i][j])
```

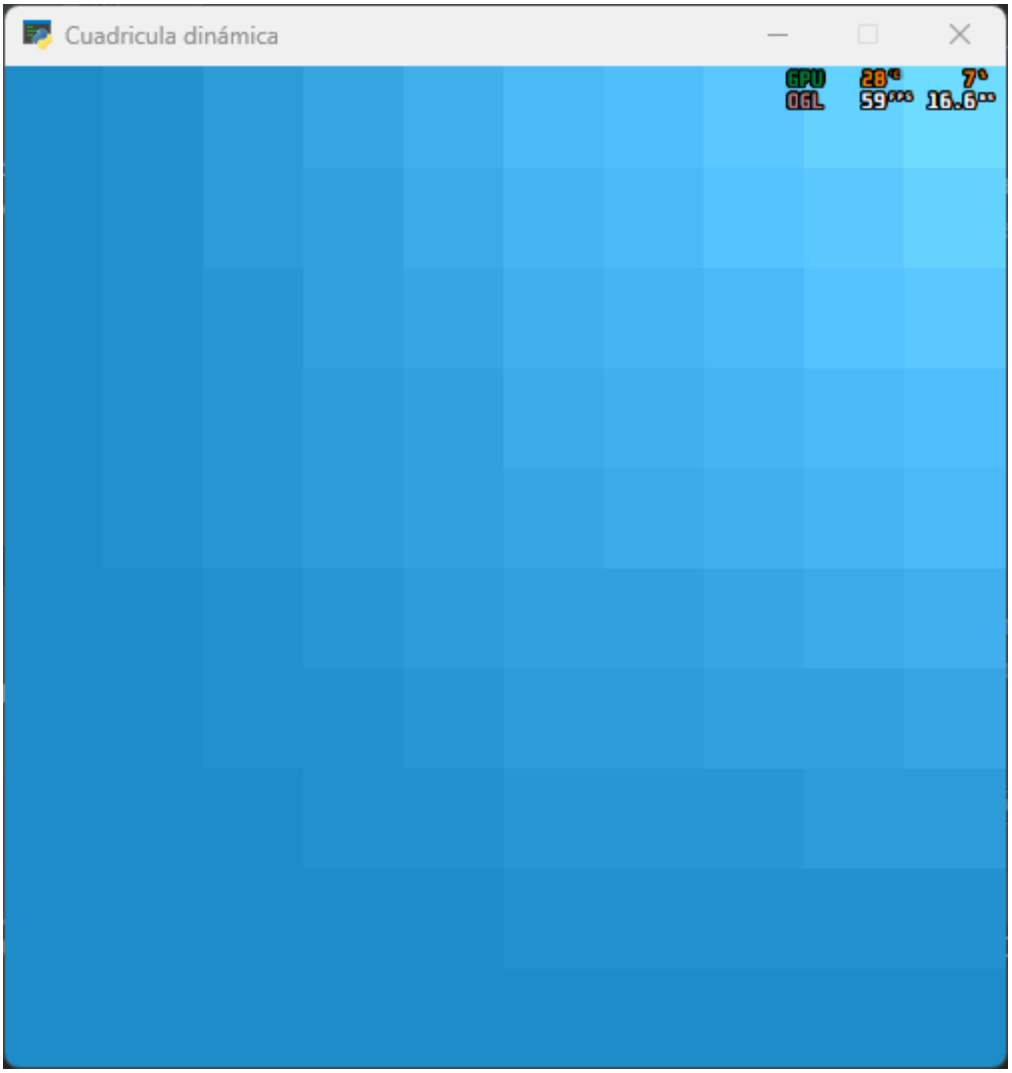
Si el fondo negro se pinta de azul es que se están dibujando por encima todas las casillas:



Vamos a hacer que el dibujo sea un poco más interesante. Justo después de crear la cuadrícula vamos a recorrer todas las celdas y a darles a los colores unos valores dinámicos incrementando brevemente su color RGB:

```
# Definir una cuadrícula de un color determinado
dibujar.cuadricula = [[(30, 140, 200) for j in range(10)] for i in range(10)]
# Vamos a recorrer la cuadrícula y a cambiar el color de los márgenes
for i, fila in enumerate(dibujar.cuadricula):
    for j, columna in enumerate(fila):
        dibujar.cuadricula[i][j] = (30 + i * j, 140 + i * j, 200 + i * j)
```

¡Menudo degradado nos hemos sacado de la manga!



Pero no nos detengamos todavía que ahora viene lo mejor. En cada fotograma vamos a modificar el color de cada casilla incrementando ligeramente los valores RGB. Para ayudarnos crearemos una función que en caso de llegar a 256 los reiniciaremos a 0:

```
def incrementar_color(codigo):
    if codigo + 1 > 255:
        return 0
    return codigo + 1
```

Vamos a probarla…

```
# Al final del fotograma incrementamos ligeramente los colores
for i, fila in enumerate(dibujar.cuadricula):
    for j, columna in enumerate(fila):
        rojo, verde, azul = dibujar.cuadricula[i][j]
        dibujar.cuadricula[i][j] = (
            incrementar_color(rojo),
            incrementar_color(verde),
```

```
            incrementar_color(azul))
```

¡Simplemente hermoso! ¿Y si creamos otra función llamada *decrementar_color* que reste 1 al color y si llega a -1 lo reinicie a 255?

```
def decrementar_color(codigo):
    if codigo - 1 < 0:
        return 255
    return codigo - 1
```

Vamos a modificar el código para que el color rojo lo deje igual, el verde lo decremente y el azul lo incremente:

```
# Al final del fotograma incrementamos ligeramente los colores
for i, fila in enumerate(dibujar.cuadricula):
    for j, columna in enumerate(fila):
        rojo, verde, azul = dibujar.cuadricula[i][j]
        dibujar.cuadricula[i][j] = (
            rojo, decrementar_color(verde), incrementar_color(azul))
```

¿Qué os parece?

Espero que os haya gustado este experimento, en él hemos puesto a prueba todo lo aprendido sobre listas, bucles, condiciones y funciones de forma visual. Ahora es vuestro turno de experimentar y modificar los colores a vuestro gusto, el tamaño de la cuadrícula y las casillas, la forma de incrementar o decrementar los valores RGB… ¡Jugad con el código a ver qué podéis generar!

Resumen del tema

En este tema hemos aprendido todo sobre las listas y sus diferencias respecto a las tuplas. Hemos visto cómo añadir, borrar y modificar elementos utilizando índices. También hemos aprendido a utilizar la función *enumerate* para recuperar los índices de cada elemento en conjunto con los bucles *for*. Finalmente explicamos cómo copiar listas, vimos algunos ejemplos de la técnica de comprensión de listas para generarlas en una sola línea y cerramos con un interesante experimento visual manejando los colores de una cuadrícula animada.