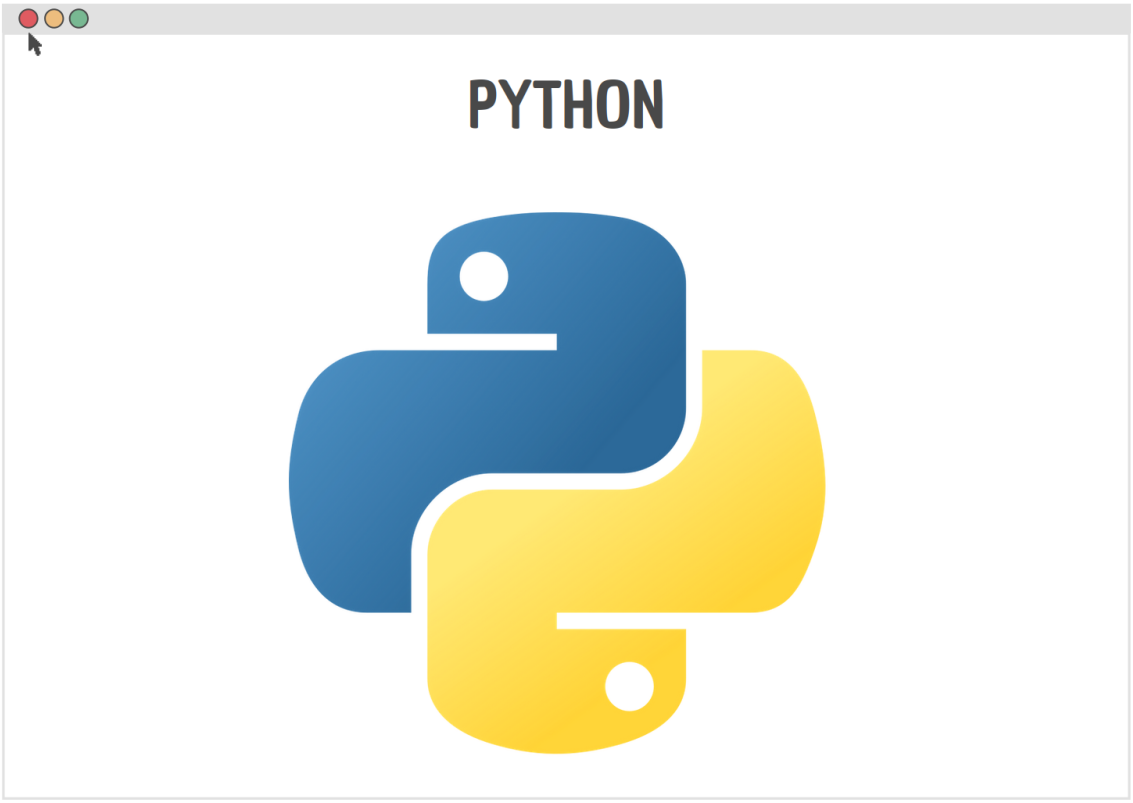
PROGRAMACIÓN WEB 2



Profesor(a):
Carlo Jose Luis Corrales Delgado

Estudiante:
Jorge Luis Mamani Huarsaya

Email:
jmamanihuars@unsa.edu.pe

Repositorio GitHub:
https://github.com/jorghee/chess-with-python

24 de mayo, 2024

Implementando los métodos de la clase Picture

Para entender toda la lógica que se explicará a continuación debemos tener claro los siguiente conceptos:

- La clase Picture tiene un campo denominado **img**, el cual es una lista de cadenas de texto (strings). Lo curioso es que esta lista de strings representa la figura que usando el paquete **pygame** y la lógica del archivo interpreter.py se transforma de caracteres a pixeles.
- Nosotros no debemos preocuparnos por la conversión de caracteres a pixeles, nuestra aventura es implementar la manipulación de los strings, por lo tanto vamos a trabajar con la lista de strings `self.img`

El método `horizontalMirror()`

Este método devuelve el espejo horizontal de la imagen, es decir, el giro se hace alrededor de una linea imaginaria horizontal trazada en medio de la imagen.

La lógica en mente es iterar sobre la lista de string del objeto Picture al cual se le esta aplicanco este método y pasar el últimstring de la lista al primer elemento de una nueva lista que denominaremos `horizontal`

- Primero creamos la lista `horizontal` vacía con el objetivo de que según la iteración se vayan agregando las los strings de `self.img`
- Ahora declaramos la estructura de control `for in` para iterar sobre `self.img`. Empezamos interando desde el último elemento de esta lista y dicho valor lo asignamos al primer elemento de la lista creada inicialmente vacía.

```
# Iterando desde el último elemento de la listas hacia el primer elmemento
def horizontalMirror(self):
  size = len(self.img)
  horizontal = []
  for value in range(size):
    horizontal.append(self.img[size - 1 - value])
  return Picture(horizontal)
```

El método `negative()`

Este método se encarga de cambiar algunos caracteres que se muestran en el script colors.py justamente en el diccionario `inverter`.

Para implementar esta funcionalidad rápidamente pensamoe en que debemos iterar por cada string de la lista `self.img` y luego iterar por cada caracter y aplicar el método `\_invColor()` que se muestra acontinuación:

```
def _invColor(self, color):
  if color not in inverter:
    return color
  return inverter[color]
```

Este método mostrado recibe el caracter como argumento y lo pasa como clave al diccionario mintpythoninverter el cual devuelve como valor el caracter de cambio.

Con esta información procedemos a codear las siguientes instrucciones.

- Creamos una variable `empty` que contiene un string vacío y creamos una lista `inverter` vacía que regresará la lista nueva.
- Utilizamos un `for in` para iterar sobre la imagen y luego en cada string aplicamos el concepto de **Comprensión de listas** que es sintaxis propia de python. Esta sintaxis nos permite crear una nueva lista de acuerdo a un elemento iterable, en este caso va a ser el string actualmente iterado por el `for in` externo.
- Aqui llamamos al método `\_invColor()` que va a recibir los caracteres del string en iteración y aplicará el cambio si es necesario.
- Por último, lo que se ha generado es una lista, pero nosotros deseamos un string. Por lo tanto aplicamos el método `join()` para concatenar todos los caracteres de esta nueva lista generada y ya como string usamos la función `append()` para agregar a la nueva lista que se retornará.

```
# Usamos el método _invColor() para cambiar el caracter
def negative(self):
  """ Devuelve un negativo de la imagen """
  empty = ""
  inverter = []
  for value in self.img:
    inverter.append(empty.join([self._invColor(caracter) for caracter in value]))
  return Picture(inverter)
```

El método `join()`

Este método debe agregar a la derecha la imagen que recibe como argumento, como nos imaginamos este métod es sencillo de implementar porque simplemente necesitamos concatenar a la derecha de la imagen actual.

- Creamos la lista `join` vacía que representará la nueva imagen formada.
- Iniciamos la intruccion de control `for in` para iterar en un rango igual a la longitud de `self.img`
- Luego con la variable **i** que recibe los numeros del rango generado, accedemos al string de la imagen actual y la imagen como argumento finalizando con la concatenación de estos.

```
# Usamos el operador '+' para concatenar los strings
def join(self, p):
  """ Devuelve una nueva figura poniendo la figura del argumento
      al lado derecho de la figura actual """
  join = []
  for i in range(len(self.img)):
    join.append(self.img[i] + p.img[i])
  return Picture(join)
```

El método `up()`

El método debe poner a `self.img` por encima de la imagen que recibe como argumento. Este se torna sencillo cuando usamos el **Operador de destructuración** propio de la sintaxis de python.

Lo que el operador de destructuración es literalmente destructurar por ejemplo una lista y pasar los elementos de esta como elementos sueltos, no como una lista como es común. Entonces esta sntaxis nos ahorra la necesidad de estar iterando por cada string y recien concatenar.

```python
# Usamos los operadores de destructuración  propios de la sintaxis de python
def up(self, p):
  up = [*self.img, *p.img]
  return Picture(up)
```

El método `under()`

Consiste en sobreponer la imagen que se recibe como argumentos sobre la imagen actual. Para esta lógica hemos decidido que el caracter vacio sea el punto de cambio. Es decir, que si algún caracater que conforma cada string de la lista que representa a la figura como argumeto es vacío, se toma en cuenta el caracter de la figura actual descartando el caracter de la figura como argumento.

Dificultad : La dificultad fue que el método funcione para sobreponer figuras de diferentes tamaños: cantidad de strings en cada lista y cantidad de caracateres en cada string.

Para solucionar este reto, se tenía la obligación de establecer el tamaño más grande tanto en la catidad de string como en la cantidad de caracateres. Y de acuerdo a estos limites realizamos las iteraciones y verificaciones para que al final se rellenen con los string y/o caracateres restantes de la figura más grande.

- Primero tenemos que encontrar a la figura con mayor tamaño, es decir con mayor número de strings. Además debemos encontrar el número de caracateres de la figura que contiene strings con mayores caracateres
- Luego, declaramos una lista vacía denominada `under` que representa la lista nueva que se va a generar.
- Empezamos iterando en un rango de la figura más grande. Aqui debemos verificar solo una cosa sencilla, que la variable de control no sobrepase el tamaño de la figura pequeña. En caso sobrepase, lo que queda son los string de la figura más grande, por lo tanto solo agregamos estos strings a la lista `under`.
- Hasta el momento ya hemos solucionado el problema de figuras con cantidad de strings diferentes, ahora necesitamos solucionar las figuras con cantidad de caracateres de cada strings diferentes. Para ello hemo creado el método `newString()` que recibe como argumentos al string de la figura actual, el string de la figura como argumento y el número que indica el número de caracateres de la figura con strings más grandes.

```python
# Implementación que funciona para cualquier longitud de las figuras
def under(self, p):
  """ Devuelve una nueva figura poniendo la figura p sobre la
      figura actual """

  # Verificamos qué figura tienes más cadenas de texto
  lower = self.img
  high = p.img
  highC = len(p.img[0])

  if len(lower) > len(high):
    lower = p.img
    high = self.img

  if len(self.img[0]) > highC:
    highC = len(self.img[0])
```

```
17
18    # Iteramos sobre las cadenas
19    under = []
20    for i in range(len(high)):
21      if i < len(lower):
22        under.append(self.newString(self.img[i], p.img[i], highC))
23      else:
24        under.append(high[i])
25    return Picture(under)
```

El método `newString()` usado por `under()`

Usamos la misma lógica de rellenar con los strings restantes de la figura más grande, solo que ahora son los caracteres del string más grande.

- Creamos una variable **`string`** que será la que se concatenará con los diferentes caracateres y se retornará en este método.
- Iniciamos el bucle en un rango de `h`, argumento que representa el tamaño del string de la figura más grande. Luego necesitamos realizar varias verificaciones comenzando con saber cúal es la figura que es más grande, parece absurdo hacer esto ahora pero es necesario ya que no se podia pasar directamente con a la figura con mayor número de caracateres porque de esta manera se perdia el conocimiento de qué figura es la que debe superponer a la otra. Por ejemplo nosotros sabemos que `p` deber superponer a `s` y eso no lo podemos perder.
- Ahora empezamos con las verificaciones de acuerdo al caracter vacío. Como sabemos que `p` es el string que debe superponer a `s` y sabiendo que `p` es el string con mayor tamaño, debemos verificar que la variable de control `i` no sea mayor al tamaño de string `s` y tambien verificar que `p[i]` sea igual al caracter vacio. De cumplirse estas 2 condiciones entonces se concatena dicho caracter en iteración, de lo contrario se contrario se contatenará el caracter `p[i]`.
- Si es el caso de que `p` no es el string de mayor tamaño entonces se asume que `s` lo es, por lo tanto debemos verificar que `i` sea menor que el tamaño que `p` que ahora es el string pequeño y tambien verificar que `p[i]` sea diferente del caracter vacío, ya que de lo contrario se concatena con `s[i]`.

```
1   # Funcion que itera por cada caracter
2   def newString(self, s, p, h):
3     string = ""
4     for i in range(h):
5       if len(p) == h:
6         if i < len(s) and p[i] == " ":
7           string += s[i]
8         else:
9           string += p[i]
10      elif i < len(p) and p[i] != " ":
11        string += p[i]
12      else:
13        string += s[i]
14    return string
```

El método `horizontalRepeat()`

En este método vamos a utilizar el concepto de **Operador de multiplicación** que se puede aplicar a listas y strings en python. Básicamente el método debe tener la funcionalidad de repetir horizontalmente n veces a la figura actual.

- Creamos una lista `horizontal` en la cual se le ira agregando cada string multiplicado.
- Empezamos iterando en la figura actual, esta lógica es sencilla porque simplemente usamos la función `append` a la cual pasamos como argumento el string actualmente iterado multiplicado n veces, lo que hace que dicho string se repita n veces.

```
# Usando el operador multiplicador '*' para generar el elemento cuantas veces queramos
def horizontalRepeat(self, n):
  """ Devuelve una nueva figura repitiendo la figura actual al costado
      la cantidad de veces que indique el valor de n """
  horizontal = []
  for value in self.img:
    horizontal.append(value * n)
    # print(value * n)
  return Picture(horizontal)
```

El método `verticalRepeat()`

Este método repite también n veces la figura actual pero ahora lo hacemos verticamente. En este caso necesitamos multiplicar el número de string en la lista, ya no los caracteres. El concepto de **Operador de destructuración** nuevamente nos simplifica enormemente esta funcionalidad.

- Creamos una lista `vertical`, luego iteramos en un rango de `n` que es el argumento que el usuario ingresa.
- En la iteración nosotros modificamos la `vertical` que recibe los elementos destructurados de las listas `vertical` y `self.img`. Aclarar que en la primera iteración `vertical` es una lista vacía y poco a poco se va modificando sumandos sus propios string con los strings de `self.img`.

```
# Usando destructuración de listas
def verticalRepeat(self, n):
  vertical = []
  for value in range(n):
    vertical = [*vertical, *self.img]
  return Picture(vertical)
```

El método `rotate()`

Este método a implementar es el segundo más dificil de esta sección seguido del método `under()` porque nos pide devolver la figura actual rotada 90 grados. Es necesario mencionar que esta implementación solo funciona para figuras con tamaños de strings iguales y cada strings con el número de caracteres iguales.

Lógica: Para iniciar esta lógica se ha dispuesto que la rotación será antihoraria. Entonces, la idea básica es que el el primer caracter del primer string de la figura se convierta en el primer caracter del último string de la nueva figura, luego que el segundo caracter del primer string de la figura se convierta en el primer caracter del penúltimo string de la nueva figura, y asi sucesivamente terminando con el último caracter del último string de la figura como el último caracter del primer string de la nueva figura.

- Comenzamos creando una lista `rotate` de un tamaño de `len(self.img)` que contiene strings vacios. Estos strings vacíos poco a poco se iran concatenando con los correspondientes caracteres según la lógica ya explicada.
- Ahora necesitamos empezar iterarando por los caracteres de cada string, por lo tanto se ha dispuesto usar un `for in` anidado. Entonces cada caracter que iteramos de la figura lo concatenamos con los strings vacíos que se ha creado. Recordemos que debemos ir concatenando el primer caracter en el último string de `rotate` e ir bajando hasta llegar al primer string de `rotate` donde concatenamos el último caracter del primer string de `self.img`, luego esto se repite con los strings restantes de `self.img`.
- Para que vaya disminuyendo el indice de `rotate` hasta llegar a `rotate[0]` debemos ayudarnos de una variable de control que la nombramos `i`, esta variable aumenta en 1 de acuerdo al `for in` interno y se reinicia a 0 cuando termina dicho bucle, asi logramos que no haya un desborde de indice al acceder a los strings de `rotate`

```
# Extra: Sólo para realmente viciosos
def rotate(self):
  """Devuelve una figura rotada en 90 grados, puede ser en sentido horario
  o antihorario"""
  rotate = []

  # Creamos la lista vacia pero con n elementos
  for i in range(len(self.img)):
    rotate.append("")

  i = 0
  for value in self.img:
    for caracter in value:
      rotate[len(self.img) - 1 - i] += caracter
      i += 1
    i = 0

  return Picture(rotate)
```

Resolución de los ejercicios

Ejercicio 2A

Para crear esta figura se ha hecho uso de los métodos `join()`, `negative()` y `up()`.

- Comenzamos usando el objeto `knight` y uniendo a la derecha el mismo objeto pero invirtiendo el color con el uso del método `negative()`.
- Luego, esta misma referencia `table` al nuevo objeto la ponemos encima de este mismo objeto pero invertido de color. La acción de ponerlo encima lo realizamos usando el método `up()`.

```python
from interpreter import draw
from chessPictures import *

table = knight.join(knight.negative())
table = table.up(table.negative())

draw(table)
```

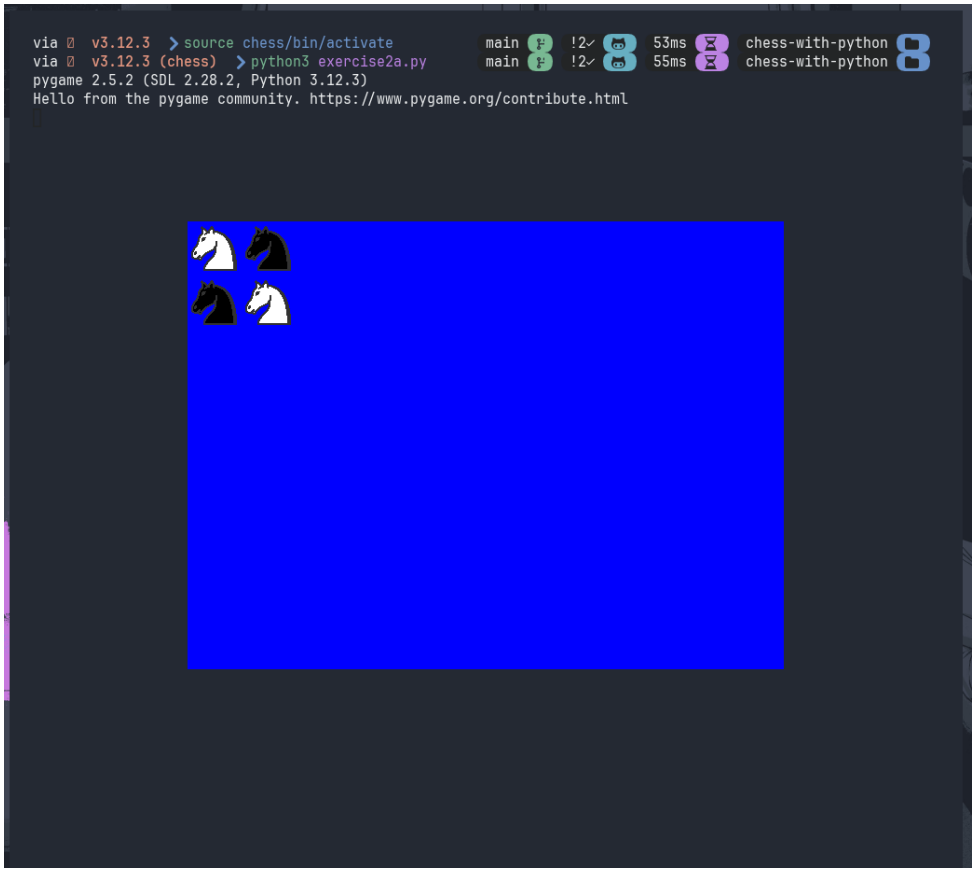


Figure 1: Ejercicio 2A

Ejercicio 2B

Lo que nos pide es parecido al Ejercicio 2A, con la única deferencia que la referencia `table` ahora la ponemos encima del mismo `table` pero ya no invertido de color, sino con su espejo vertical. La acción de crear el espejo vertical de la figura `table` la realizamos usando el método `verticalMirror()`

```python
from interpreter import draw
from chessPictures import *

table = knight.join(knight.negative())
table = table.up(table.verticalMirror())

draw(table)
```

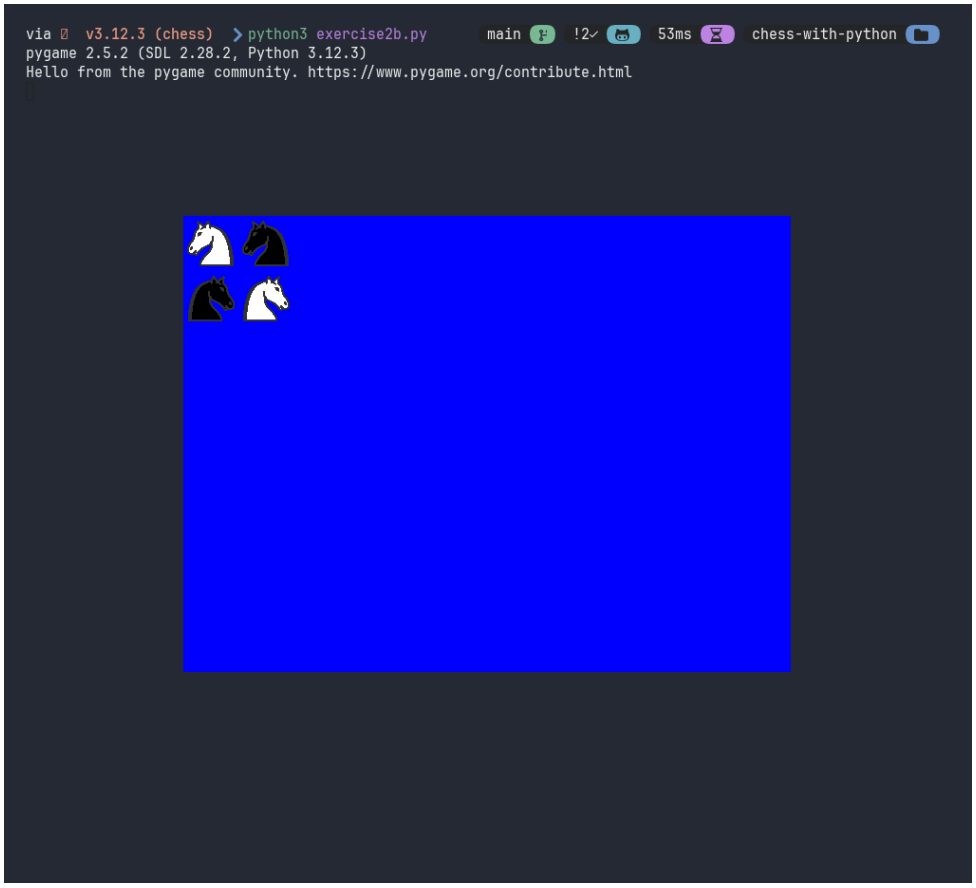


Figure 2: Ejercicio 2B

Ejercicio 2C

El problema nos pide generar 4 reinas consecutivas a la derecha. El único método necesario que se usará es el método `horizontalRepeat(n)` con el argumento de $n = 4$.

```
1 from interpreter import draw
2 from chessPictures import *
3
4 draw(queen.horizontalRepeat(4))
```

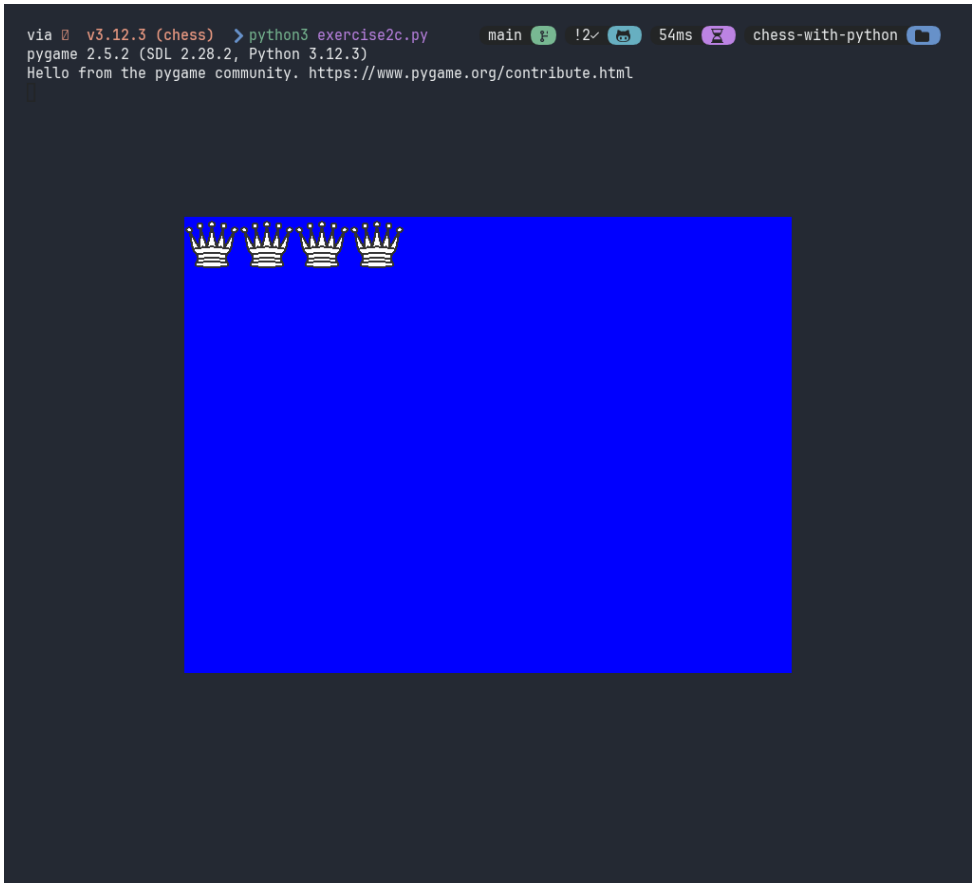


Figure 3: Ejercicio 2C

Ejercicio 2D

Desde este problema en adelante nuestra lógica se basa en crear un patron base y manipularlo, cuando sea necesario, con todas los métodos que hemos implementado.

El problema nos pide generar una fila de cuadrados intercalados de color, empezando con el cuadrado de color plomo o blanco. Rápidamente podemos imaginar que el patrón base es la figura `square.join(square.negative()` y en este caso esta figura la repetimo 4 veces usando el método `horizontalRepeat()`.

```
from interpreter import draw
from chessPictures import *

draw(square.join(square.negative()).horizontalRepeat(4))
```

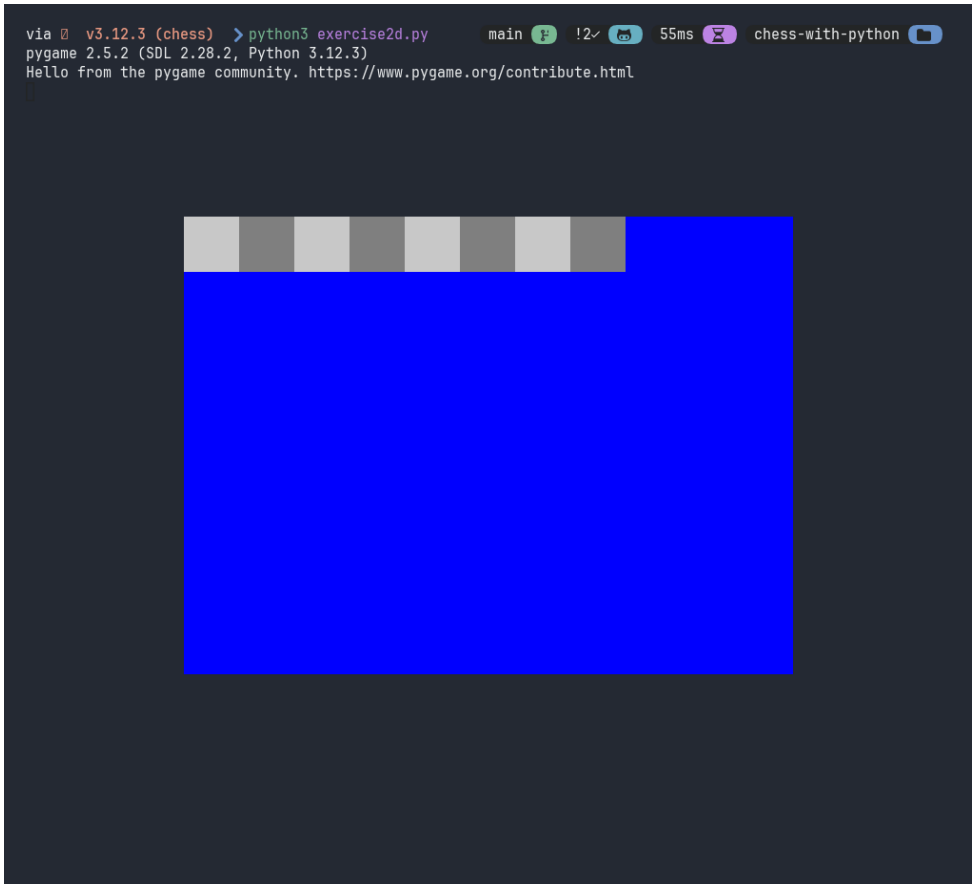


Figure 4: Ejercicio 2D

Ejercicio 2E

Este problema nos pide lo mismo que el Ejercicio 2D pero empezando con el cuadrado de color negro. Entonces el patrón base ahora es la figura `square.negative().join(square)` y esta la repetimos tambien 4 veces.

```
1 from interpreter import draw
2 from chessPictures import *
3
4 draw(square.negative().join(square).horizontalRepeat(4))
```

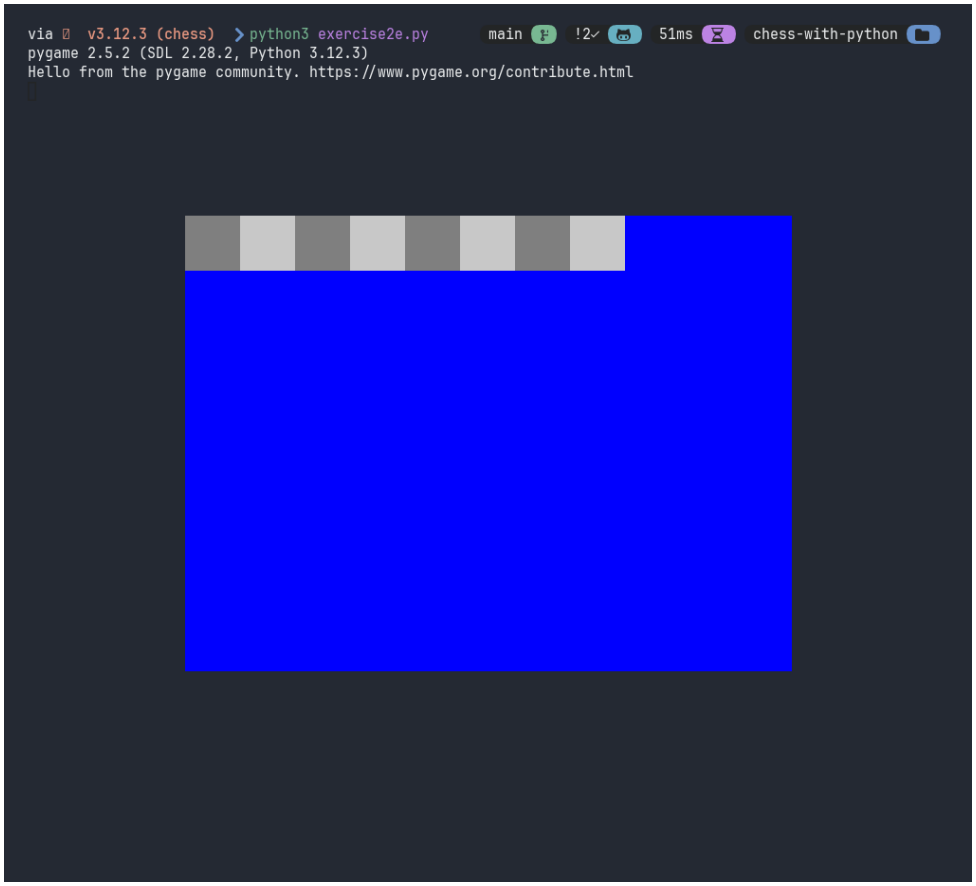


Figure 5: Ejercicio 2E

Ejercicio 2F

El problema nos dice crear un tablero de ajedrez de 8x4 y empezando obviamente con el cuadrado blanco y asi intercalar los colores de los siguientes cuadrados. Entonces ahora podemos decir que nuestro patrón base será la primera fila del tablero, referenciaremos a este patrón con el nombre de `table`.

Luego usamos este patrón para generar la segunda fila que simplemente es invertir los colores del patron o, tambien, es generar su espejo vertical. Hemos optado por usar `negative()`.

Finalmente hemos generado una figura de 2 filas con 8 columnas y solo nos queda repetirla verticalmente 2 veces. Para esta accion usamos el método `verticalRepeat()`.

```
1 from interpreter import draw
2 from chessPictures import *
3
4 table = square.join(square.negative()).horizontalRepeat(4)
5 table = table.up(table.negative()).verticalRepeat(2)
6
7 draw(table)
```

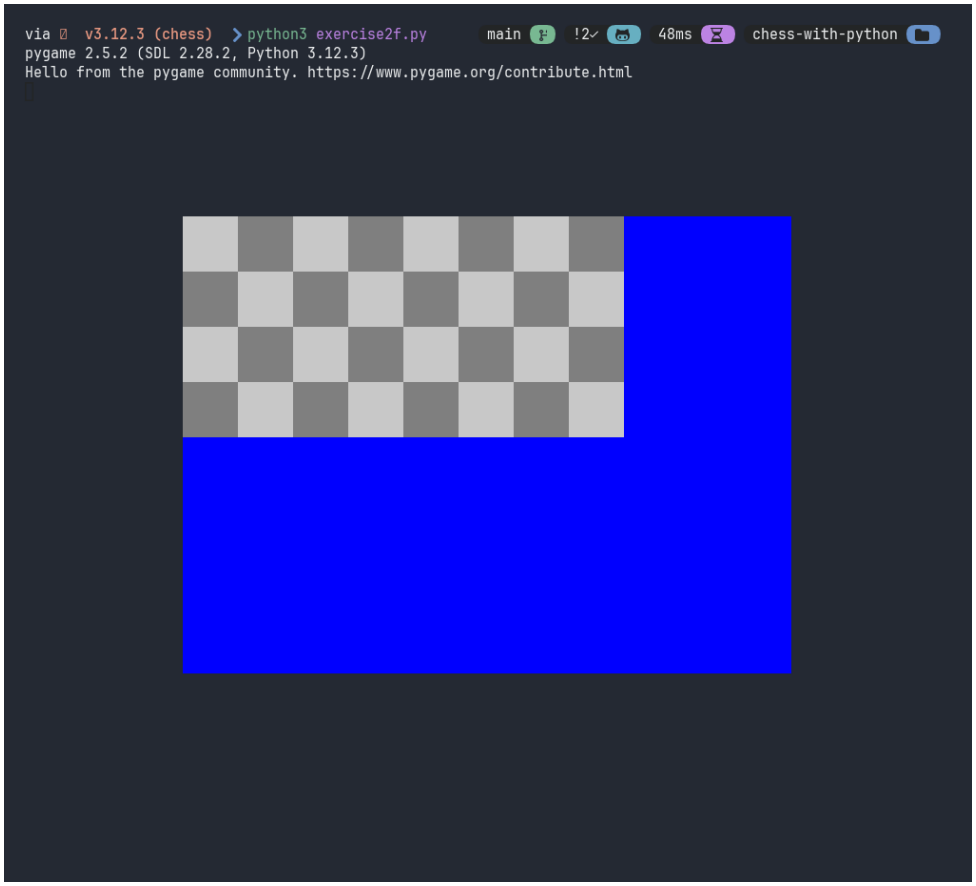


Figure 6: Ejercicio 2F

Ejercicio 2G

Finalmente llegamos al problema donde nos pide crear un tablero de ajedrez completo con las fichas ordenadas. Se ha dividido en 3 partes que al final se deben de unir para conforma el tablero completo:

- **Piezas negras:** Las piezas negras siempre van sobrepuestas sobre un tablero donde la pieza **torre** esta por encima del cuadrado blanco, y asi se intercalan los cuadrados. Esta parte la partimos en 2 partes más, pieces y paws:
  - **Pieces:** Para generar la base de las piezas negras es la misma lógica que el Ejercicio 2D y esta figura la estamos referenciando con el identificador `squares`. Ahora debemos generar la fila de las piezas, para ello hemos usado el método `join()` colando uno a lado de otro todas las piezas y al final a toda esta fila de piezas la cambiamos de color con `negative()`. Finalmente obreponemos la fila de piezas negras sobre `squares` usando el método `under()` y creando asi las Piezas negras.
  - **Pawns:** Para crear las fila de peones, simplemente usamos el método `horizontalRepeat()` y lo superponemos a `squares`. Veremos aqui que tanto la fila de peones como la fila de `squares` no coinciden con su color respectivo, por lo tanto a este resultado le cambiamos de color con `negative()`. A

  Para culminar esta parte necesitamos colocar encima por encima de los peones negros, las piezas negras y esta acción la hacemos con `up()`. A este resultado lo referenciamos con el identificador `partBlack`
- **Tablero vacío:** Para esta parte simplemente usamos `squares`, lo ponemos arriba de `squares.negative()` y repetimos verticamente 2 veces. Esta figura la referenciamos como `tableEmpty`.

- **Piezas blancas:** Para esta parte simplemente reutilizamos los resultados de **Piezas negras**. En vez de poner a los peones por debajo de las piezas como lo hicimos en esa sección, ahora lo hacemos al revez y al resultado le cambiamos de color con el método `negative()`. Este resultado lo referenciamos con el identificador `partWhite`

Finalmente tenemos que unir estas tres partes, colocando **Piezas negras** encima de **Tablero vacío** y este encima de **Piezas blancas**.

```
from interpreter import draw
from chessPictures import *

squares = square.join(square.negative()).horizontalRepeat(4)
piecesPart = rock.join(knight).join(bishop)

# Part black
pieces = squares.under(piecesPart.join(queen).join(king).join(piecesPart).negative())
pawns = squares.under(pawn.horizontalRepeat(8)).negative()
partBlack = pieces.up(pawns)

# Part empty
tableEmpty = squares.up(squares.negative()).verticalRepeat(2)

# Part white
partWhite = pawns.up(pieces).negative()

# table
table = partBlack.up(tableEmpty).up(partWhite)

draw(table)
```

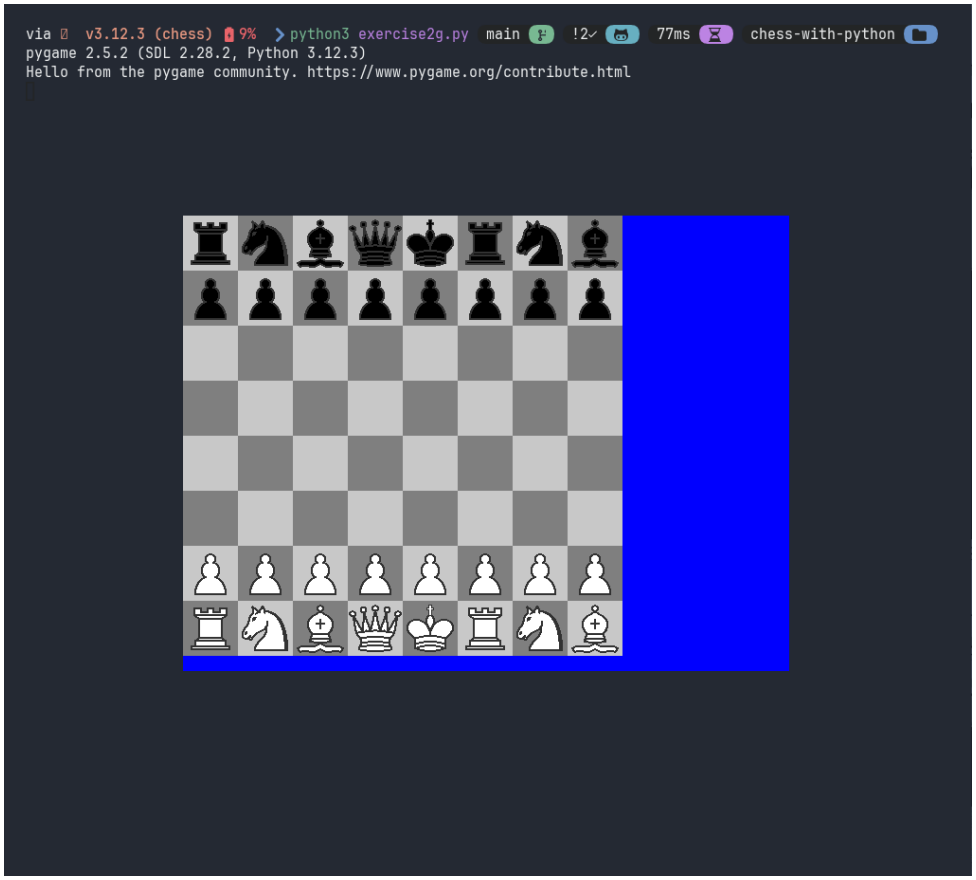


Figure 7: Ejercicio 2G