Módulo 3 - Valores booleanos, ejecución condicional, bucles, listas y su procesamiento, operaciones lógicas y de bit a bit

Tomando decisiones

- = asignación
- == comparación, es igual
- != no es igual
- >,<,>=,⇐

if

```
if <exp>:
    linea1
    linea2
  elif <exp>:
    linea5
    linea6
  else:
    linea3
    linea4
```

```
if number1 > number2: larger_number = number1
else: larger_number = number2
```

- no mezclar tabs y espacios en blanco en la indentación
- `else` es la última rama de la cascada, opcionalmente

bucles (ciclos)

- algoritmo, pseudocódigo
- `max()`: máximo de X números → `min()`

ejercicio

```
Érase una vez una tierra de leche y miel, habitada por gente feliz y próspera. La
gente pagaba impuestos, por supuesto, su felicidad tenía límites. El impuesto más
importante, denominado Impuesto Personal de Ingresos (IPI, para abreviar) tenía que
pagarse una vez al año y se evaluó utilizando la siguiente regla:

Si el ingreso del ciudadano no era superior a 85,528 pesos, el impuesto era igual
al 18% del ingreso menos 556 pesos y 2 centavos (esta fue la llamada exención
fiscal ).
Si el ingreso era superior a esta cantidad, el impuesto era igual a 14,839 pesos y
2 centavos, más el 32% del excedente sobre 85,528 pesos.
```

```
Tu tarea es escribir una calculadora de impuestos.

Debe aceptar un valor de punto flotante: el ingreso.
A continuación, debe imprimir el impuesto calculado, redondeado a pesos totales.
Hay una función llamada round() que hará el redondeo por ti, la encontrarás en el
código de esqueleto del editor.
Nota: Este país feliz nunca devuelve dinero a sus ciudadanos. Si el impuesto
calculado es menor que cero, solo significa que no hay impuesto (el impuesto es
igual a cero). Ten esto en cuenta durante tus cálculos.

Observa el código en el editor: solo lee un valor de entrada y genera un resultado,
por lo que debes completarlo con algunos cálculos inteligentes.
```

```
income = float(input("Introduce el ingreso anual:"))

if income<=85528:
    tax = (income * 18 / 100) - 556.2
else:
    tax = 14839.2 + (income - 85528) * 32 / 100

if tax < 0: tax = 0.0

tax = round(tax, 0)
print("El impuesto es:", tax, "pesos")
```

ejercicio

```
Como seguramente sabrás, debido a algunas razones astronómicas, el año pueden ser
bisiesto o común. Los primeros tienen una duración de 366 días, mientras que los
últimos tienen una duración de 365 días.

Desde la introducción del calendario Gregoriano (en 1582), se utiliza la siguiente
regla para determinar el tipo de año:

Si el número del año no es divisible entre cuatro, es un año común.
De lo contrario, si el número del año no es divisible entre 100, es un año
bisiesto.
De lo contrario, si el número del año no es divisible entre 400, es un año común.
De lo contrario, es un año bisiesto.
Observa el código en el editor: solo lee un número de año y debe completarse con
las instrucciones que implementan la prueba que acabamos de describir.

El código debe mostrar uno de los dos mensajes posibles, que son Año Bisiesto o Año
Común, según el valor ingresado.

Sería bueno verificar si el año ingresado cae en la era Gregoriana y emitir una
advertencia de lo contrario: No dentro del período del calendario Gregoriano.
Consejo: utiliza los operadores != y %.
```

```
year = int(input("Introduce un año:"))

if year >= 1582:
    if year % 4 != 0: ano="común"
    elif year % 100 != 0: ano="bisiesto"
    elif year % 400 != 0: ano="comun"
```

```
    else: ano="bisiesto"
    print("Año "+ano)
else:
    print("no es gregoriano")
```

while

```
while conditional_expression:
    instruction
```

```
while True:
    print("Estoy atrapado dentro de un bucle.")
```

- Si deseas ejecutar más de una sentencia dentro de un while, debes (como con if) poner sangría a todas las instrucciones de la misma manera.
- Una instrucción o conjunto de instrucciones ejecutadas dentro del while se llama el cuerpo del bucle.
- Si la condición es False (igual a cero) tan pronto como se compruebe por primera vez, el cuerpo no se ejecuta ni una sola vez (ten en cuenta la analogía de no tener que hacer nada si no hay nada que hacer).
- El cuerpo debe poder cambiar el valor de la condición, porque si la condición es True al principio, el cuerpo podría funcionar continuamente hasta el infinito. Observa que hacer una cosa generalmente disminuye la cantidad de cosas por hacer.

ejercicio

```
Un mago junior ha elegido un número secreto. Lo ha escondido en una variable
llamada secret_number. Quiere que todos los que ejecutan su programa jueguen el
juego Adivina el número secreto, y adivina qué número ha elegido para ellos.
¡Quiénes no adivinen el número quedarán atrapados en un bucle sin fin para siempre!
Desafortunadamente, él no sabe cómo completar el código.

Tu tarea es ayudar al mago a completar el código en el editor de tal manera que el
código:

Pedirá al usuario que ingrese un número entero.
Utilizará un bucle while.
Comprobará si el número ingresado por el usuario es el mismo que el número escogido
por el mago. Si el número elegido por el usuario es diferente al número secreto del
mago, el usuario debería ver el mensaje "¡Ja, ja! ¡Estás atrapado en mi bucle!"  y
se le solicitará que ingrese un número nuevamente. Si el número ingresado por el
usuario coincide con el número escogido por el mago, el número debe imprimirse en
la pantalla, y el mago debe decir las siguientes palabras: "¡Bien hecho, muggle!
Eres libre ahora".
¡El mago está contando contigo! No lo decepciones.


INFO EXTRA

Por cierto, observa la función print(). La forma en que lo hemos utilizado aquí se
llama impresión multilínea. Puede utilizar comillas triples para imprimir cadenas
en varias líneas para facilitar la lectura del texto o crear un diseño especial
basado en texto. Experimenta con ello.
```

```
secret_number = 777
```

```
print(
"""
+==================================+
| ¡Bienvenido a mi juego, muggle!  |
| Introduce un número entero       |
| y adivina qué número he          |
| elegido para ti.                 |
| Entonces,                        |
| ¿Cuál es el número secreto?      |
+==================================+
""")
num = 0

while num!=secret_number:
    num = int(input())
    if num!=secret_number: print("¡Ja, ja! ¡Estás atrapado en mi bucle!")
    else: print("¡Bien hecho, muggle! Eres libre ahora")
```

for

```
for i in range(100):
    # do_something()
    pass
```

```
for i in range(2, 8):
    print("El valor de i es actualmente", i)
```

- La palabra reservada for abre el bucle for; nota - No hay condición después de eso; no tienes que pensar en las condiciones, ya que se verifican internamente, sin ninguna intervención.
- Cualquier variable después de la palabra reservada for es la variable de control del bucle; cuenta los giros del bucle y lo hace automáticamente.
- La palabra reservada in introduce un elemento de sintaxis que describe el rango de valores posibles que se asignan a la variable de control.
- La función **range()** (esta es una función muy especial) es responsable de generar todos los valores deseados de la variable de control; en nuestro ejemplo, la función creará (incluso podemos decir que alimentará el bucle con) valores subsiguientes del siguiente conjunto: 0, 1, 2 .. 97, 98, 99; nota: en este caso, la función range() comienza su trabajo desde 0 y lo finaliza un paso (un número entero) antes del valor de su argumento.
- Nota la palabra clave **pass** dentro del cuerpo del bucle - no hace nada en absoluto; es una instrucción vacía : la colocamos aquí porque la sintaxis del bucle for exige al menos una instrucción dentro del cuerpo (por cierto, if, elif, else y while expresan lo mismo).

```
for i in range(2, 8, 3):
    print("El valor de i es actualmente", i)
```

- La función range() también puede aceptar tres argumentos: Echa un vistazo al código del editor.
- El tercer argumento es un incremento: es un valor agregado para controlar la variable en cada giro del bucle (como puedes sospechar, el valor predeterminado del incremento es 1).
- el conjunto generado por range() debe ordenarse en un orden ascendente. No hay forma de forzar el range() para crear un conjunto en una forma diferente. Esto significa que el segundo argumento de range() debe ser mayor que el primero.

ejercicio

¿Sabes lo que es Mississippi? Bueno, es el nombre de uno de los estados y ríos en los Estados Unidos. El río Mississippi tiene aproximadamente 2,340 millas de largo, lo que lo convierte en el segundo río más largo de los Estados Unidos (el más largo es el río Missouri). ¡Es tan largo que una sola gota de agua necesita 90 días para recorrer toda su longitud!

La palabra Mississippi también se usa para un propósito ligeramente diferente: para contar mississippily (mississippimente).

Si no estás familiarizado con la frase, estamos aquí para explicarte lo que significa: se utiliza para contar segundos.

La idea detrás de esto es que agregar la palabra Mississippi a un número al contar los segundos en voz alta hace que suene más cercano al reloj, y por lo tanto "un Mississippi, dos Mississippi, tres Mississippi" tomará aproximadamente unos tres segundos reales de tiempo. A menudo lo usan los niños que juegan al escondite para asegurarse de que el buscador haga un conteo honesto.

Tu tarea es muy simple aquí: escribe un programa que use un bucle for para "contar de forma mississippi" hasta cinco. Habiendo contado hasta cinco, el programa debería imprimir en la pantalla el mensaje final "¡Listos o no, ahí voy!"

Utiliza el esqueleto que hemos proporcionado en el editor.

INFO EXTRA

Ten en cuenta que el código en el editor contiene dos elementos que pueden no ser del todo claros en este momento: la sentencia import time y el método sleep(). Vamos a hablar de ellos pronto.

Por el momento, nos gustaría que supieras que hemos importado el módulo time y hemos utilizado el método sleep() para suspender la ejecución de cada función posterior de print() dentro del bucle for durante un segundo, de modo que el mensaje enviado a la consola se parezca a un conteo real. No te preocupes, pronto aprenderás más sobre módulos y métodos.

```
import time

for count in range(1,6):
    print(count,"Mississippi")
    time.sleep(1)
print("¡Listos o no, ahí voy!")
```

```
import time

for i in range(1,6):
  time.sleep(1)
  print(str(i),"missisipi" + ("s" if i>1 else ""))
```

break y continue

Hasta ahora, hemos tratado el cuerpo del bucle como una secuencia indivisible e inseparable de instrucciones que se realizan completamente en cada giro del bucle. Sin embargo, como desarrollador, podrías enfrentar las siguientes opciones:

Parece que no es necesario continuar el bucle en su totalidad; se debe abstener de seguir ejecutando el cuerpo del bucle e ir más allá. Parece que necesitas comenzar el siguiente giro del bucle sin completar la ejecución del turno actual. Python proporciona dos instrucciones especiales para la implementación de estas dos tareas. Digamos por razones de precisión que su existencia en el lenguaje no es necesaria: un programador experimentado puede codificar cualquier algoritmo sin estas instrucciones. Tales adiciones, que no mejoran el poder expresivo del lenguaje, sino que solo simplifican el trabajo del desarrollador, a veces se denominan dulces sintácticos o azúcar sintáctica.

Estas dos instrucciones son:

- **break**: sale del bucle inmediatamente, e incondicionalmente termina la operación del bucle; el programa comienza a ejecutar la instrucción más cercana después del cuerpo del bucle.
- **continue**: se comporta como si el programa hubiera llegado repentinamente al final del cuerpo; el siguiente turno se inicia y la expresión de condición se prueba de inmediato.

Ambas palabras son palabras clave reservadas.

ejemplos

```
# break - ejemplo

print("La instrucción break:")
for i in range(1, 6):
    if i == 3:
        break
    print("Dentro del bucle.", i)
print("Fuera del bucle.")


# continue - ejemplo

print("\nLa instrucción continue:")
for i in range(1, 6):
    if i == 3:
        continue
    print("Dentro del bucle.", i)
print("Fuera del bucle.")
```

```
largest_number = -99999999
counter = 0

while True:
    number = int(input("Ingresa un número o escribe -1 para finalizar el programa:
"))
    if number == -1:
        break
    counter += 1
    if number > largest_number:
```

```
        largest_number = number

if counter != 0:
    print("El número más grande es", largest_number)
else:
    print("No has ingresado ningún número.")
```

```
largest_number = -99999999
counter = 0

number = int(input("Ingresa un número o escribe -1 para finalizar el programa: "))

while number != -1:
    if number == -1:
        continue
    counter += 1

    if number > largest_number:
        largest_number = number
    number = int(input("Ingresa un número o escribe -1 para finalizar el programa:
"))

if counter:
    print("El número más grande es", largest_number)
else:
    print("No has ingresado ningún número.")
```

ejercicio

```
La instrucción break se implementa para salir/terminar un bucle.

Diseña un programa que use un bucle while y le pida continuamente al usuario que
ingrese una palabra a menos que ingrese "chupacabra" como la palabra de salida
secreta, en cuyo caso el mensaje "¡Has dejado el bucle con éxito". Debe imprimirse
en la pantalla y el bucle debe terminar.

No imprimas ninguna de las palabras ingresadas por el usuario. Utiliza el concepto
de ejecución condicional y la sentencia break.
```

```
while True:
    text = input()
    if text == "chupacabra": break
print("¡Has dejado el bucle con éxito")
```

ejercicio

```
La sentencia continue se usa para omitir el bloque actual y avanzar a la siguiente
iteración, sin ejecutar las sentencias dentro del bucle.

Se puede usar tanto con while y for.

Tu tarea aquí es muy especial: ¡Debes diseñar un devorador de vocales! Escribe un
programa que use:
```

```
Un bucle for.
El concepto de ejecución condicional (if-elif-else).
La sentencia continue.
Tu programa debe:

Pedir al usuario que ingrese una palabra.
Utiliza user_word = user_word.upper() para convertir la palabra ingresada por el
usuario a mayúsculas; hablaremos sobre los llamados métodos de cadena y el upper()
muy pronto, no te preocupes.
Utiliza la ejecución condicional y la instrucción continue para "comer" las
siguientes vocales A , E , I , O , U de la palabra ingresada.
Imprime las letras no consumidas en la pantalla, cada una de ellas en una línea
separada.
```

```
user_word = input()
user_word = user_word.upper()

for letter in user_word:
    if letter=="A" or letter=="E" or letter=="I" or letter=="O" or letter=="U":
continue
    print(letter)
```

ejercicio

```
Tu tarea aquí es aún más especial que antes: ¡Debes rediseñar el devorador de
vocales (feo) del laboratorio anterior (3.1.2.10) y crear un mejor devorador de
vocales (bonito) mejorado! Escribe un programa que use:

Un bucle for.
El concepto de ejecución condicional (if-elif-else).
La instrucción continue.
Tu programa debe:

Pedir al usuario que ingrese una palabra.
Utilizar user_word = user_word.upper() para convertir la palabra ingresada por el
usuario a mayúsculas; hablaremos sobre los llamados métodos de cadena y el upper()
muy pronto, no te preocupes.
Emplea la ejecución condicional y la instrucción continue para "comer" las
siguientes vocales A , E , I , O , U de la palabra ingresada.
Asigne las letras no consumidas a la variable word_without_vowels e imprime la
variable en la pantalla.
Analiza el código en el editor. Hemos creado word_without_vowels y le hemos
asignado una cadena vacía. Utiliza la operación de concatenación para pedirle a
Python que combine las letras seleccionadas en una cadena más larga durante los
siguientes giros de bucle, y asignarlo a la variable word_without_vowels.
```

```
word_without_vowels = ""

user_word = input()
user_word = user_word.upper()

for letter in user_word:
    if letter=="A": continue
    elif letter=="E": continue
    elif letter=="I": continue
```

```
    elif letter=="O": continue
    elif letter=="U": continue
    else: word_without_vowels += letter

print(word_without_vowels)
```

El bucle while y la rama else

```
i = 1
while i < 5:
    print(i)
    i += 1
else:
    print("else:", i)
```

El bucle for y la rama else

```
for i in range(5):
    print(i)
else:
    print("else:", i)
```

ejercicio

```
Escucha esta historia: Un niño y su padre, un programador de computadoras, juegan
con bloques de madera. Están construyendo una pirámide.

Su pirámide es un poco rara, ya que en realidad es una pared en forma de pirámide,
es plana. La pirámide se apila de acuerdo con un principio simple: cada capa
inferior contiene un bloque más que la capa superior.

Tu tarea es escribir un programa que lea la cantidad de bloques que tienen los
constructores, y generar la altura de la pirámide que se puede construir utilizando
estos bloques.

Nota: La altura se mide por el número de capas completas: si los constructores no
tienen la cantidad suficiente de bloques y no pueden completar la siguiente capa,
terminan su trabajo inmediatamente.

Prueba tu código con los datos que hemos proporcionado.
```

```
blocks = int(input("Ingresa el número de bloques: "))

used_blocks = 0
height = 0

while blocks>=used_blocks:
    height += 1
    used_blocks = used_blocks + height
    if (used_blocks+height) >= blocks: break
```

```
print("La altura de la pirámide:", height)
```

ejercicio

```
En 1937, un matemático alemán llamado Lothar Collatz formuló una hipótesis
intrigante (aún no se ha comprobado) que se puede describir de la siguiente manera:

Toma cualquier número entero que no sea negativo y que no sea cero y asígnale el
nombre c0.
Si es par, evalúa un nuevo c0 como c0 Ã· 2.
De lo contrario, si es impar, evalúe un nuevo  c0  como 3 Ã c0 + 1.
Si c0 â  1, salta al punto 2.
La hipótesis dice que, independientemente del valor inicial de c0, el valor siempre
tiende a 1.

Por supuesto, es una tarea extremadamente compleja usar una computadora para probar
la hipótesis de cualquier número natural (incluso puede requerir inteligencia
artificial), pero puede usar Python para verificar algunos números individuales.
Tal vez incluso encuentres el que refutaría la hipótesis.

Escribe un programa que lea un número natural y ejecute los pasos anteriores
siempre que c0 sea diferente de 1. También queremos que cuente los pasos necesarios
para lograr el objetivo. Tu código también debe mostrar todos los valores
intermedios de c0.

Sugerencia: la parte más importante del problema es como transformar la idea de
Collatz en un bucle while- esta es la clave del éxito.
```

Operaciones lógicas y de bits en python: and, or not

- AND
- OR
- NOT
- Expresiones lógicas: Puedes estar familiarizado con las leyes de De Morgan. Dicen que:
  - La negación de una conjunción es la separación de las negaciones. → `not (p and q) == (not p) or (not q)`
  - La negación de una disyunción es la conjunción de las negaciones. → `not (p or q) == (not p) and (not q)`
- Operadores bit a bit
  - **&** (ampersand) - conjunción a nivel de bits. → requieres exactamente dos 1s para proporcionar 1 como resultado.
  - **|** (barra vertical) - disyunción a nivel de bits. → requiere al menos un 1 para proporcionar 1 como resultado.
  - **~** (tilde) - negación a nivel de bits.
  - **^** (signo de intercalación) - o exclusivo a nivel de bits (xor). → requiere exactamente un 1 para proporcionar 1 como resultado.
  - Agreguemos un comentario importante: los argumentos de estos operadores deben ser enteros. No debemos usar flotantes aquí.
- Operaciones lógicas frente a operaciones de bit:

```
i = 15 # 00000000000000000000000000001111
j = 22 # 00000000000000000000000000010110
```

```
print(i and j) # True
print(i & j) # 00000000000000000000000000000110 -> 6
```

- ¿Cómo tratamos los bits individuales?
  - La variable almacena la información sobre varios aspectos de la operación del sistema. Cada bit de la variable almacena un valor de si/no. También se te ha dicho que solo uno de estos bits es tuyo, el tercero (recuerda que los bits se numeran desde cero y el número de bits cero es el más bajo, mientras que el más alto es el número 31). Los bits restantes no pueden cambiar, porque están destinados a almacenar otros datos. Aquí está tu bit marcado con la letra x:

    ```
    flag_register = 0000000000000000000000000000x000
    ```

  - Es posible que tengas que hacer frente a las siguientes tareas:
    - Comprobar el estado de tu bit: deseas averiguar el valor de su bit; comparar la variable completa con cero no hará nada, porque los bits restantes pueden tener valores completamente impredecibles, pero puedes usar la siguiente propiedad de conjunción:

      ```
      x & 1 = x
      x & 0 = 0
      ```

    - Dicha secuencia de ceros y unos, cuya tarea es tomar el valor o cambiar los bits seleccionados, se denomina máscara de bits. Construyamos una máscara de bits para detectar el estado de tus bits. Debería apuntar a el tercer bit. Ese bit tiene el peso de 2^3=8. Se podría crear una máscara adecuada mediante la siguiente sentencia:

      ```
      the_mask = 8
      if flag_register & the_mask:
          # Mi bit se estableció en 1.
      else:
          # Mi bit se restableció a 0.
      ```

    - Reinicia tu bit: asigna un cero al bit, mientras que todos los otros bits deben permanecer sin cambios; usemos la misma propiedad de la conjunción que antes, pero usemos una máscara ligeramente diferente, exactamente como se muestra a continuación:

      ```
      11111111111111111111111111110111
      ```

    - Tenga en cuenta que la máscara se creó como resultado de la negación de todos los bits de la variable the_mask. Restablecer el bit es simple, y se ve así (elige el que más te guste):

      ```
      flag_register = flag_register & ~the_mask
      flag_register &= ~the_mask
      ```

    - Establece tu bit : asigna un 1 a tu bit, mientras que todos los bits restantes deben permanecer sin cambios; usa la siguiente propiedad de disyunción:

      ```
      x | 1 = 1
      x | 0 = x
      ```

      ```
      flag_register = flag_register | the_mask
      flag_register |= the_mask
      ```

    - Niega tu bit: reemplaza un 1 con un 0 y un 0 con un 1. Puedes utilizar una propiedad interesante del operador ~x:

      ```
      x ^ 1 = ~x
      ```

```
x ^ 0 = x
```

```
flag_register = flag_register ^ the_mask
flag_register ^= the_mask
```

- Desplazamiento izquierdo binario y desplazamiento derecho binario:
  - Python ofrece otra operación relacionada con los bits individuales: shifting. Esto se aplica solo a los valores de número entero, y no debe usar flotantes como argumentos para ello.
  - Ya aplicas esta operación muy a menudo y muy inconscientemente. ¿Cómo multiplicas cualquier número por diez? Echa un vistazo:

    ```
    12345 × 10 = 123450
    ```

  - Como puede ver, multiplicar por diez es de hecho un desplazamiento de todos los dígitos a la izquierda y llenar el vacío resultante con cero.¿División entre diez? Echa un vistazo:

    ```
    12340 ÷ 10 = 1234
    ```

  - Dividir entre diez no es más que desplazar los dígitos a la derecha.
  - La computadora realiza el mismo tipo de operación, pero con una diferencia: como dos es la base para los números binarios (no 10), desplazar un valor un bit a la izquierda corresponde a multiplicarlo por dos; respectivamente, desplazar un bit a la derecha es como dividir entre dos (observe que se pierde el bit más a la derecha).

Los operadores de cambio en Python son un par de dígrafos: < < y > >, sugiriendo claramente en qué dirección actuará el cambio.

```
value << bits
value >> bits
```

listas

Hasta ahora, has aprendido como declarar variables que pueden almacenar exactamente un valor dado a la vez. Tales variables a veces se denominan escalares por analogía con las matemáticas. Todas las variables que has usado hasta ahora son realmente escalares.

Piensa en lo conveniente que sería declarar una variable que podría almacenar más de un valor. Por ejemplo, cien, o mil o incluso diez mil. Todavía sería una y la misma variable, pero muy amplia y espaciosa. ¿Suena atractivo? Quizás, pero ¿cómo manejarías un contenedor así lleno de valores diferentes? ¿Cómo elegirías solo el que necesitas?

Digamos lo mismo utilizando una terminología adecuada: numeros es una lista que consta de cinco valores, todos ellos números. También podemos decir que esta sentencia crea una lista de longitud igual a cinco (ya que contiene cinco elementos).

```
numbers = [10, 5, 7, 2, 1]
```

Los elementos dentro de una lista pueden tener diferentes tipos. Algunos de ellos pueden ser enteros, otros son flotantes y otros pueden ser listas.

Python ha adoptado una convención que indica que los elementos de una lista están siempre numerados desde cero. Esto significa que el elemento almacenado al principio de la lista tendrá el número cero. Como hay cinco elementos en nuestra lista, al último de ellos se le asigna el número cuatro. No olvides esto.

Antes de continuar con nuestra discusión, debemos indicar lo siguiente: nuestra lista es una colección de

elementos, pero cada elemento es un escalar.

Indexando Listas

```
numbers = [10, 5, 7, 2, 1]
print("Contenido de la lista original:", numbers)  # Imprimiendo contenido de la
lista original.

numbers[0] = 111
print("Nuevo contenido de la lista: ", numbers)  # Contenido de la lista actual.

numbers = [10, 5, 7, 2, 1]
print("Contenido de la lista original:", numbers)  # Imprimiendo contenido de la
lista original.

numbers[0] = 111
print("\nPrevio contenido de la lista:", numbers)  # Imprimiendo contenido de la
lista anterior.

numbers[1] = numbers[4]  # Copiando el valor del quinto elemento al segundo
elemento.
print("Nuevo contenido de la lista:", numbers)  # Imprimiendo el contenido de la
lista actual.
```

El valor dentro de los corchetes que selecciona un elemento de la lista se llama un índice, mientras que la operación de seleccionar un elemento de la lista se conoce como indexación.

Vamos a utilizar la función print() para imprimir el contenido de la lista cada vez que realicemos los cambios. Esto nos ayudará a seguir cada paso con más cuidado y ver que sucede después de una modificación de la lista en particular.

Nota: todos los índices utilizados hasta ahora son literales. Sus valores se fijan en el tiempo de ejecución, pero cualquier expresión también puede ser un índice. Esto abre muchas posibilidades.

Accediendo al contenido de la lista

Se puede acceder a cada uno de los elementos de la lista por separado. Por ejemplo, se puede imprimir:

```
print(numbers[0]) # Accediendo al primer elemento de la lista.
```

Suponiendo que todas las operaciones anteriores se hayan completado con éxito, el fragmento enviará 111 a la consola. Como puedes ver en el editor, la lista también puede imprimirse como un todo, como aquí:

```
print(numbers)  # Imprimiendo la lista completa.
```

Como probablemente hayas notado antes, Python decora la salida de una manera que sugiere que todos los valores presentados forman una lista. La salida del fragmento de ejemplo anterior se ve así:

```
[111, 1, 7, 2, 1]
```

La función len()

La longitud de una lista puede variar durante la ejecución. Se pueden agregar nuevos elementos a la lista, mientras que otros pueden eliminarse de ella. Esto significa que la lista es una entidad muy dinámica.

Si deseas verificar la longitud actual de la lista, puedes usar una función llamada len() (su nombre proviene de length - longitud).

La función toma el nombre de la lista como un argumento y devuelve el número de elementos almacenados actualmente dentro de la lista (en otras palabras, la longitud de la lista).

Eliminando elementos de una lista

Cualquier elemento de la lista puede ser eliminado en cualquier momento, esto se hace con una instrucción llamada del (eliminar). Nota: es una instrucción, no una función.

Tienes que apuntar al elemento que quieres eliminar, desaparecerá de la lista y la longitud de la lista se reducirá en uno.

Mira el fragmento de abajo. ¿Puedes adivinar qué salida producirá? Ejecuta el programa en el editor y comprueba.

```
del numbers[1]
print(len(numbers))
print(numbers)
```

No puedes acceder a un elemento que no existe , no puedes obtener su valor ni asignarle un valor. Ambas instrucciones causarán ahora errores de tiempo de ejecución.

Los índices negativos son válidos

Puede parecer extraño, pero los índices negativos son válidos y pueden ser muy útiles.

Un elemento con un índice igual a -1 es el último en la lista. Del mismo modo, el elemento con un índice igual a -2 es el anterior al último en la lista.

ejercicio

```
Había una vez un sombrero. El sombrero no contenía conejo, sino una lista de cinco
números: 1, 2, 3, 4 y 5.

Tu tarea es:

Escribir una línea de código que solicite al usuario que reemplace el número
central en la lista con un número entero ingresado por el usuario (Paso 1).
Escribir una línea de código que elimine el último elemento de la lista (Paso 2).
Escribir una línea de código que imprima la longitud de la lista existente (Paso
3).
```

```
hat_list = [1, 2, 3, 4, 5]  # Esta es una lista existente de números ocultos en el
sombrero.
```

```
# Paso 1: escribe una línea de código que solicite al usuario
# reemplazar el número de en medio con un número entero ingresado por el usuario.
number = int(input())
hat_list[2]=number

# Paso 2: escribe aquí una línea de código que elimine el último elemento de la
lista.
del hat_list[-1]

# Paso 3: escribe aquí una línea de código que imprima la longitud de la lista
existente.
print(len(hat_list))

print(hat_list)
```

Funciones frente a métodos

- Un método es un tipo específico de función: se comporta como una función y se parece a una función, pero difiere en la forma en que actúa y en su estilo de invocación.
- Una función no pertenece a ningún dato: obtiene datos, puede crear nuevos datos y (generalmente) produce un resultado.
- Un método hace todas estas cosas, pero también puede cambiar el estado de una entidad seleccionada.
- Un método es propiedad de los datos para los que trabaja, mientras que una función es propiedad de todo el código.
- Esto también significa que invocar un método requiere alguna especificación de los datos a partir de los cuales se invoca el método.
- Puede parecer desconcertante aquí, pero lo trataremos en profundidad cuando profundicemos en la programación orientada a objetos.
- En general, una invocación de función típica puede tener este aspecto:

```
result = function(arg)
```

- La función toma un argumento, hace algo y devuelve un resultado.
- Una invocación de un método típico usualmente se ve así:

```
result = data.method(arg)
```

- Nota: el nombre del método está precedido por el nombre de los datos que posee el método. A continuación, se agrega un punto, seguido del nombre del método y un par de paréntesis que encierran los argumentos.
- El método se comportará como una función, pero puede hacer algo más: puede cambiar el estado interno de los datos a partir de los cuales se ha invocado.

Agregando elementos a una lista: append() e insert()

- Un nuevo elemento puede ser añadido al final de la lista existente:

```
list.append(value)
```

- Dicha operación se realiza mediante un método llamado **append()**. Toma el valor de su argumento y lo coloca al final de la lista que posee el método.
- La longitud de la lista aumenta en uno.
- El método **insert()** es un poco más inteligente: puede agregar un nuevo elemento en cualquier lugar de la lista, no solo al final.

```
list.insert(location, value)
```

    - El primero muestra la ubicación requerida del elemento a insertar. Nota: todos los elementos existentes que ocupan ubicaciones a la derecha del nuevo elemento (incluido el que está en la posición indicada) se desplazan a la derecha, para hacer espacio para el nuevo elemento.
    - El segundo es el elemento a insertar.
- Puedes iniciar la vida de una lista creándola vacía (esto se hace con un par de corchetes vacíos) y luego agregar nuevos elementos según sea necesario.

Haciendo uso de las listas

- El bucle for tiene una variante muy especial que puede procesar las listas de manera muy efectiva. Echemos un vistazo a eso.
- Supongamos que deseas calcular la suma de todos los valores almacenados en la lista my_list.
- Necesitas una variable cuya suma se almacenará y se le asignará inicialmente un valor de 0 - su nombre será total. (Nota: no la vamos a nombrar sum ya que Python utiliza el mismo nombre para una de sus funciones integradas: sum(). Utilizar ese nombre sería considerado una mala práctica. Luego agrega todos los elementos de la lista usando el bucle for. Echa un vistazo al fragmento en el editor.

```
my_list = [10, 1, 8, 3, 5]
total = 0

for i in range(len(my_list)):
    total += my_list[i]

print(total)
```

- Comentemos este ejemplo:
    - A la lista se le asigna una secuencia de cinco valores enteros.
    - La variable i toma los valores 0, 1,2,3, y 4, y luego indexa la lista, seleccionando los elementos siguientes: el primero, segundo, tercero, cuarto y quinto.
    - Cada uno de estos elementos se agrega junto con el operador += a la variable suma, dando el resultado final al final del bucle.
    - Observa la forma en que se ha empleado la función len(), hace que el código sea independiente de cualquier posible cambio en el contenido de la lista.

El segundo aspecto del bucle for

- Pero el bucle for puede hacer mucho más. Puede ocultar todas las acciones conectadas a la indexación de la lista y entregar todos los elementos de la lista de manera práctica.
- Este fragmento modificado muestra como funciona:

```
my_list = [10, 1, 8, 3, 5]
total = 0

for i in my_list:
    total += i

print(total)
```

- ¿Qué sucede aquí?
    - La instrucción for especifica la variable utilizada para navegar por la lista (i) seguida de la palabra clave in y el nombre de la lista siendo procesado (my_list).
    - A la variable i se le asignan los valores de todos los elementos de la lista subsiguiente, y el proceso ocurre tantas veces como hay elementos en la lista.

- Esto significa que usa la variable i como una copia de los valores de los elementos, y no necesita emplear índices.
- La función len() tampoco es necesaria aquí.

Listas en acción

- Dejemos de lado las listas por un breve momento y veamos un tema intrigante.
- Imagina que necesitas reorganizar los elementos de una lista, es decir, revertir el orden de los elementos: el primero y el quinto, así como el segundo y cuarto elementos serán intercambiados. El tercero permanecerá intacto.
- Pregunta: ¿Cómo se pueden intercambiar los valores de dos variables?

```
variable_1 = 1
variable_2 = 2

variable_2 = variable_1
variable_1 = variable_2
```

- Si haces algo como esto, perderás el valor previamente almacenado en variable_2. Cambiar el orden de las tareas no ayudará. Necesitas una tercera variable que sirva como almacenamiento auxiliar.
- Así es como puedes hacerlo:

```
variable_1 = 1
variable_2 = 2

auxiliary = variable_1
variable_1 = variable_2
variable_2 = auxiliary
```

- Python ofrece una forma más conveniente de hacer el intercambio, echa un vistazo:

```
variable_1 = 1
variable_2 = 2

variable_1, variable_2 = variable_2, variable_1
```

- Ahora puedes intercambiar fácilmente los elementos de la lista para revertir su orden:

```
my_list = [10, 1, 8, 3, 5]

my_list[0], my_list[4] = my_list[4], my_list[0]
my_list[1], my_list[3] = my_list[3], my_list[1]

print(my_list
```

```
my_list = [10, 1, 8, 3, 5]
length = len(my_list)

for i in range(length // 2):
    my_list[i], my_list[length - i - 1] = my_list[length - i - 1], my_list[i]

print(my_list)
```

- Nota:
  - Hemos asignado la variable length a la longitud de la lista actual (esto hace que nuestro código

sea un poco más claro y más corto).
- Hemos preparado el bucle for para que se ejecute su cuerpo length // 2 veces (esto funciona bien para listas con longitudes pares e impares, porque cuando la lista contiene un número impar de elementos, el del medio permanece intacto).
- Hemos intercambiado el elemento i (desde el principio de la lista) por el que tiene un índice igual a (length-i-1) (desde el final de la lista); en nuestro ejemplo, for i igual a 0 a la (length-i-1) da 4; for i igual a 3, da 3: esto es exactamente lo que necesitábamos.

ejercicio

```
Los Beatles fueron uno de los grupos de música más populares de la década de 1960 y
la banda más vendida en la historia. Algunas personas los consideran el acto más
influyente de la era del rock. De hecho, se incluyeron en la compilación de la
revista Time de las 100 personas más influyentes del siglo XX.

a banda sufrió muchos cambios de formación, que culminaron en 1962 con la formación
de John Lennon, Paul McCartney, George Harrison y Richard Starkey (mejor conocido
como Ringo Starr).


Escribe un programa que refleje estos cambios y le permita practicar con el
concepto de listas. Tu tarea es:

Paso 1: Crea una lista vacía llamada beatles.
Paso 2: Emplea el método append() para agregar los siguientes miembros de la banda
a la lista: John Lennon, Paul McCartney y George Harrison.
Paso 3: Emplea el bucle for y el append() para pedirle al usuario que agregue los
siguientes miembros de la banda a la lista: Stu Sutcliffe, y Pete Best.
Paso 4: Usa la instrucción del para eliminar a Stu Sutcliffe y Pete Best de la
lista.
Paso 5: Usa el método insert() para agregar a Ringo Starr al principio de la lista.
```

```
Beatles = []
print("Paso 1:", Beatles)

Beatles.append("John Lennon")
Beatles.append("Paul McCartney")
Beatles.append("George Harrison")
print("Paso 2:", Beatles)

for i in range(2):
    valor = input("agrega ")
    Beatles.append(valor)
print("Paso 3:", Beatles)

del Beatles[-1]
del Beatles[-1]
print("Paso 4:", Beatles)

Beatles.insert(0,"Ringo Starr")
print("Paso 5:", Beatles)


# probando la longitud de la lista
```

```
print("Los Fav", len(Beatles))
```

ordenamiento burbuja

¿Cuántos pases necesitamos para ordenar la lista completa?

Resolvamos este problema de la siguiente manera: introducimos otra variable, su tarea es observar si se ha realizado algún intercambio durante el pase o no. Si no hay intercambio, entonces la lista ya está ordenada, y no hay que hacer nada más. Creamos una variable llamada swapped, y le asignamos un valor de False para indicar que no hay intercambios. De lo contrario, se le asignará True.

```
my_list = [8, 10, 6, 2, 4]  # lista a ordenar

for i in range(len(my_list) - 1):  # necesitamos (5 - 1) comparaciones
    if my_list[i] > my_list[i + 1]:  # compara elementos adyacentes
        my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]  # Si terminamos
aquí, tenemos que intercambiar elementos.
```

Deberías poder leer y comprender este programa sin ningún problema:

```
my_list = [8, 10, 6, 2, 4]  # lista a ordenar
swapped = True  # Lo necesitamos verdadero (True) para ingresar al bucle while.

while swapped:
    swapped = False  # no hay intercambios hasta ahora
    for i in range(len(my_list) - 1):
        if my_list[i] > my_list[i + 1]:
            swapped = True  # ¡ocurrió el intercambio!
            my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]

print(my_list)
```

El ordenamiento burbuja - versión interactiva

En el editor, puedes ver un programa completo, enriquecido por una conversación con el usuario, y que permite ingresar e imprimir elementos de la lista: El ordenamiento burbuja: versión final interactiva.

```
my_list = []
swapped = True
num = int(input("¿Cuántos elementos deseas ordenar?: "))

for i in range(num):
    val = float(input("Ingresa un elemento de la lista: "))
    my_list.append(val)

while swapped:
    swapped = False
    for i in range(len(my_list) - 1):
        if my_list[i] > my_list[i + 1]:
            swapped = True
            my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]
```

```
print("\nOrdenada:")
print(my_list)
```

Python, sin embargo, tiene sus propios mecanismos de clasificación. Nadie necesita escribir sus propias clases, ya que hay un número suficiente de herramientas listas para usar.

Te explicamos este sistema de clasificación porque es importante aprender como procesar los contenidos de una lista y mostrarte como puede funcionar la clasificación real.

Si quieres que Python ordene tu lista, puedes hacerlo de la siguiente manera:

```
my_list = [8, 10, 6, 2, 4]
my_list.sort()
print(my_list)
```

Como puedes ver, todas las listas tienen un método denominado sort(), que las ordena lo más rápido posible.

La vida al interior de las listas

Ahora queremos mostrarte una característica importante y muy sorprendente de las listas, que las distingue de las variables ordinarias.

Queremos que lo memorices, ya que puede afectar tus programas futuros y causar graves problemas si se olvida o se pasa por alto.

Echa un vistazo al fragmento en el editor.

El programa:

Crea una lista de un elemento llamada list_1. La asigna a una nueva lista llamada list_2. Cambia el único elemento de list_1. Imprime la list_2. La parte sorprendente es el hecho de que el programa mostrará como resultado: [2], no [1], que parece ser la solución obvia.

Las listas (y muchas otras entidades complejas de Python) se almacenan de diferentes maneras que las variables ordinarias (escalares).

Se podría decir que:

El nombre de una variable ordinaria es el nombre de su contenido. El nombre de una lista es el nombre de una ubicación de memoria donde se almacena la lista. Lee estas dos líneas una vez más, la diferencia es esencial para comprender de que vamos a hablar a continuación.

La asignación: list_2 = list_1 copia el nombre del arreglo no su contenido. En efecto, los dos nombres (list_1 y list_2) identifican la misma ubicación en la memoria de la computadora. Modificar uno de ellos afecta al otro, y viceversa.

Rebanadas Poderosas

Afortunadamente, la solución está al alcance de tu mano: su nombre es **rebanada**.

Una rebanada es un elemento de la sintaxis de Python que permite **hacer una copia nueva de una lista, o partes de una lista.**

En realidad, copia el contenido de la lista, no el nombre de la lista.

Esto es exactamente lo que necesitas. Echa un vistazo al fragmento de código a continuación:

```
list_1 = [1]
list_2 = list_1[:]
list_1[0] = 2
print(list_2)
```

Su salida es **[1]**.

Esta parte no visible del código descrito como [:] puede producir una lista completamente nueva.

Una de las formas más generales de la rebanada es la siguiente:

```
my_list[start:end]
```

Como puedes ver, se asemeja a la indexación, pero los dos puntos en el interior hacen una gran diferencia.

Una rebanada de este tipo **crea una nueva lista (de destino), tomando elementos de la lista de origen: los elementos de los índices desde el principio hasta el fin - 1.**

Nota: no hasta el fin, sino hasta fin-1. Un elemento con un índice igual a fin es el primer elemento el cual no participa en la segmentación.

Es posible utilizar valores negativos tanto para el inicio como para el fin(al igual que en la indexación).

Echa un vistazo al fragmento:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[1:3]
print(new_list)
```

La lista new_list contendrá fin - inicio (3 - 1 = 2) elementos â□□ los que tienen índices iguales a 1 y 2 (pero no 3).

La salida del fragmento es: **[8, 6]**

Rebanadas - índices negativos

Observa el fragmento de código a continuación:

```
my_list[start:end]
```

Para confirmar:

start es el índice del primer elemento **incluido en la rebanada. end** es el índice del primer elemento **no incluido en la rebanada.**

Así es como los índices negativos funcionan en las rebanadas:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[1:-1]
print(new_list)
```

El resultado del fragmento es:

```
[8, 6, 4]
salida
```

Si start especifica un elemento que se encuentra más allá del descrito por end (desde el punto de vista inicial de la lista), la rebanada estará vacía:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[-1:1]
print(new_list)
```

La salida del fragmento es:

```
[]
```

Rebanadas: continuación

Si omites el start en tu rebanada, se supone que deseas obtener un segmento que comienza en el elemento con índice 0.

```
my_list[:end]
```

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[:3]
print(new_list)
```

```
[10, 8, 6]
```

Del mismo modo, si omites el end en tu rebanada, se supone que deseas que el segmento termine en el elemento con el índice len(my_list).

```
my_list[start:]
my_list[start:len(my_list)]
```

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[3:]
print(new_list)
```

```
[4, 2]
```

Como hemos dicho anteriormente, el omitir el inicio y fin crea una copia de toda la lista

La instrucción **del** descrita anteriormente puede **eliminar más de un elemento de la lista a la vez, también puede eliminar rebanadas**:

```
my_list = [10, 8, 6, 4, 2]
del my_list[1:3]
print(my_list)
```

Nota: En este caso, la rebanada **¡no produce ninguna lista nueva!**

Al eliminar la rebanada del código, su significado cambia dramáticamente.

```
my_list = [10, 8, 6, 4, 2]
del my_list
```

```
print(my_list)
```

La instrucción del **eliminará la lista, no su contenido**.

Los operadores in y not in

Python ofrece dos operadores muy poderosos, capaces de revisar la lista para verificar si un valor específico está almacenado dentro de la lista o no.

Estos operadores son:

```
elem in my_list
elem not in my_list
```

El primero de ellos (in) verifica si un elemento dado (el argumento izquierdo) está actualmente almacenado en algún lugar dentro de la lista (el argumento derecho) - el operador devuelve True en este caso.

El segundo (not in) comprueba si un elemento dado (el argumento izquierdo) está ausente en una lista - el operador devuelve True en este caso.

Listas dentro de listas

Las listas pueden constar de escalares (es decir, números) y elementos de una estructura mucho más compleja (ya has visto ejemplos como cadenas, booleanos o incluso otras listas en las lecciones del Resumen de la Sección anterior). Veamos más de cerca el caso en el que los elementos de una lista son listas.

A menudo encontramos estos arreglos en nuestras vidas. Probablemente el mejor ejemplo de esto sea un tablero de ajedrez.

Un tablero de ajedrez está compuesto de filas y columnas. Hay ocho filas y ocho columnas. Cada columna está marcada con las letras de la A a la H. Cada línea está marcada con un número del uno al ocho.

La ubicación de cada campo se identifica por pares de letras y dígitos. Por lo tanto, sabemos que la esquina inferior derecha del tablero (la que tiene la torre blanca) es A1, mientras que la esquina opuesta es H8.

Supongamos que podemos usar los números seleccionados para representar cualquier pieza de ajedrez. También podemos asumir que cada fila en el tablero de ajedrez es una lista.

Observa el siguiente código:

```
row = []

for i in range(8):
    row.append(WHITE_PAWN)
```

Crea una lista que contiene ocho elementos que representan la segunda fila del tablero de ajedrez: la que está llena de peones (supon que WHITE_PAWN es un símbolo predefinido que representa un peón blanco).

El mismo efecto se puede lograr mediante una comprensión de lista, la sintaxis especial utilizada por Python para completar o llenar listas masivas.

Una comprensión de lista es en realidad una lista, pero **se creó sobre la marcha durante la ejecución del programa, y no se describe de forma estática**.

Echa un vistazo al fragmento:

row = [WHITE_PAWN for i in range(8)]

La parte del código colocada dentro de los paréntesis especifica:

- Los datos que se utilizarán para completar la lista (WHITE_PAWN)
- La cláusula que especifica cuántas veces se producen los datos dentro de la lista (for i in range(8))

Permítenos mostrarte otros ejemplos de comprensión de lista:

Ejemplo #1:

```
squares = [x ** 2 for x in range(10)]
```

El fragmento de código genera una lista de diez elementos y la rellena con cuadrados de diez números enteros que comienzan desde cero (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

Ejemplo #2:

```
twos = [2 ** i for i in range(8)]
```

El fragmento crea un arreglo de ocho elementos que contiene las primeras ocho potencias del numero dos (1, 2, 4, 8, 16, 32, 64, 128)

Ejemplo #3:

```
odds = [x for x in squares if x % 2 != 0 ]
```

El fragmento crea una lista con solo los elementos impares de la lista squares.

Listas dentro de listas: arreglos bidimensionales

Supongamos también que un símbolo predefinido denominado EMPTY designa un campo vacío en el tablero de ajedrez.

Entonces, si queremos crear una lista de listas que representan todo el tablero de ajedrez, se puede hacer de la siguiente manera:

```
board = []

for i in range(8):
    row = [EMPTY for i in range(8)]
    board.append(row)
```

Nota:

- La parte interior del bucle crea una fila que consta de ocho elementos (cada uno de ellos es igual a EMPTY) y lo agrega a la lista del board.
- La parte exterior se repite ocho veces.
- En total, la lista board consta de 64 elementos (todos iguales a EMPTY).

Este modelo imita perfectamente el tablero de ajedrez real, que en realidad es una lista de elementos de ocho elementos, todos ellos en filas individuales. Resumamos nuestras observaciones:

- Los elementos de las filas son campos, ocho de ellos por fila.

- Los elementos del tablero de ajedrez son filas, ocho de ellos por tablero de ajedrez.

La variable board ahora es un **arreglo bidimensional**. También se le llama, por analogía a los términos algebraicos, una **matriz**.

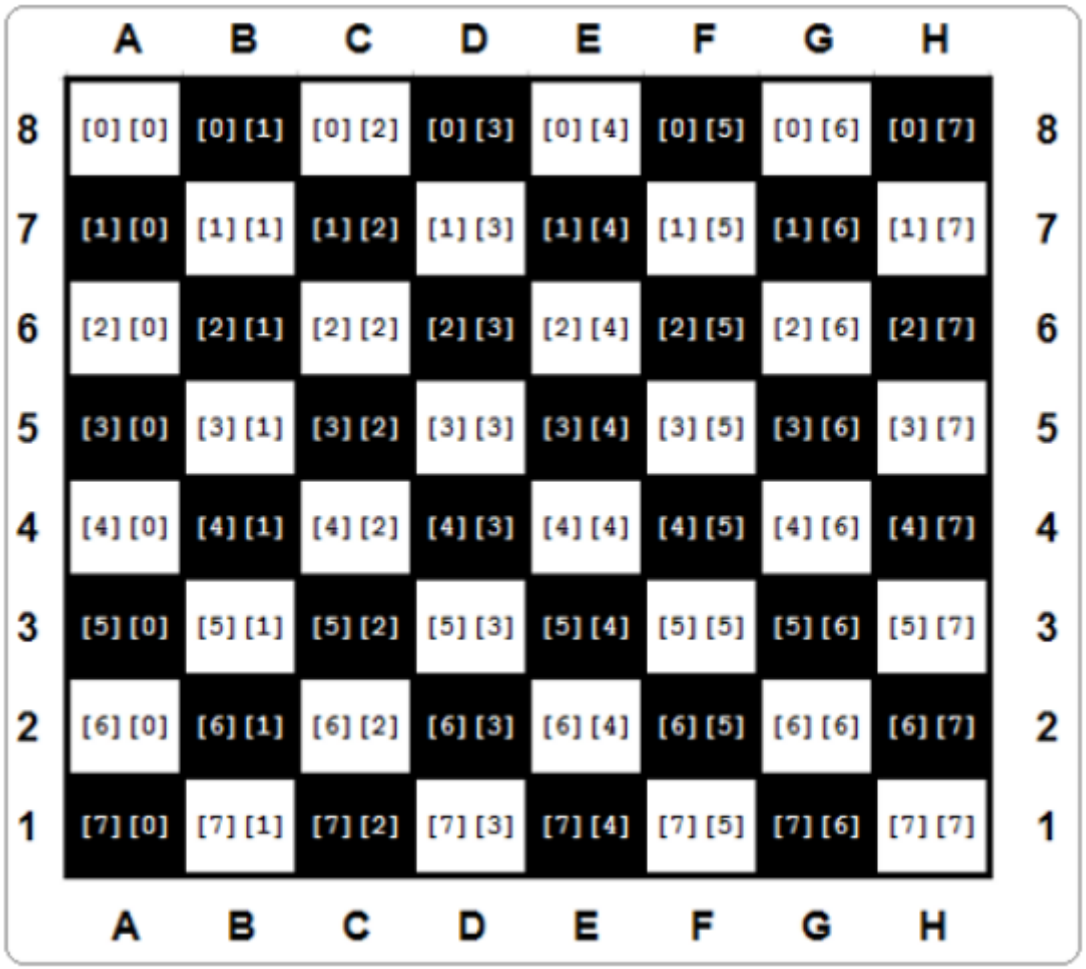
Como las listas de comprensión puede ser anidadas, podemos acortar la creación del tablero de la siguiente manera:

```
board = [[EMPTY for i in range(8)] for j in range(8)]
```

La parte interna (bucle i) crea una fila, y la parte externa (bucle j) crea una lista de filas.

El acceso al campo seleccionado del tablero requiere dos índices: el primero selecciona la fila; el segundo: el número del campo dentro de la fila, el cual es un número de columna.

Echa un vistazo al tablero de ajedrez. Cada campo contiene un par de índices que se deben dar para acceder al contenido del campo:



Echando un vistazo a la figura que se muestra arriba, coloquemos algunas piezas de ajedrez en el tablero. Primero, agreguemos todas las torres:

```
board[0][0] = ROOK
board[0][7] = ROOK
board[7][0] = ROOK
board[7][7] = ROOK
```

Si deseas agregar un caballo a C4, hazlo de la siguiente manera:

```
board[4][2] = KNIGHT
```

Naturaleza multidimensional de las listas: aplicaciones avanzadas

Profundicemos en la naturaleza multidimensional de las listas. Para encontrar cualquier elemento de una lista bidimensional, debes usar dos coordenadas:

Una vertical (número de fila). Una horizontal (número de columna). Imagina que desarrollas una pieza de software para una estación meteorológica automática. El dispositivo registra la temperatura del aire cada hora y lo hace durante todo el mes. Esto te da un total de 24 × 31 = 744 valores. Intentemos diseñar una lista capaz de almacenar todos estos resultados.

Primero, debes decidir qué tipo de datos sería adecuado para esta aplicación. En este caso, sería mejor un float, ya que este termómetro puede medir la temperatura con una precisión de 0.1 ℃.

Luego tomarás la decisión arbitraria de que las filas registrarán las lecturas cada hora exactamente (por lo que la fila tendrá 24 elementos) y cada una de las filas se asignará a un día del mes (supongamos que cada mes tiene 31 días, por lo que necesita 31 filas). Aquí está el par apropiado de comprensiones(h es para las horas, dpara el día):

```
temps = [[0.0 for h in range(24)] for d in range(31)]
```

Toda la matriz está llena de ceros ahora. Puede suponer que se actualiza automáticamente utilizando agentes de hardware especiales. Lo que tienes que hacer es esperar a que la matriz se llene con las mediciones.

Ahora es el momento de determinar la temperatura promedio mensual del mediodía. Suma las 31 lecturas registradas al mediodía y divida la suma por 31. Puedes suponer que la temperatura de medianoche se almacena primero. Aquí está el código:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# La matriz se actualiza aquí.
#

total = 0.0

for day in temps:
    total += day[11]

average = total / 31

print("Temperatura promedio al mediodía:", average)
```

Nota: La variable day utilizada por el bucle for no es un escalar: cada paso a través de la matriz temps lo asigna a la siguiente fila de la matriz; Por lo tanto, es una lista. Se debe indexar con 11 para acceder al valor de temperatura medida al mediodía.

Ahora encuentra la temperatura más alta durante todo el mes, analiza el código:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# La matriz se actualiza aquí.
#

highest = -100.0
```

```
for day in temps:
    for temp in day:
        if temp > highest:
            highest = temp

print("La temperatura más alta fue:", highest)
```

Nota:

- La variable day itera en todas las filas de la matriz temps.
- La variable temp itera a través de todas las mediciones tomadas en un día.

Arreglos tridimensionales

Python no limita la profundidad de la inclusión lista en lista. Aquí puedes ver un ejemplo de un arreglo tridimensional:

Imagina un hotel. Es un hotel enorme que consta de tres edificios, de 15 pisos cada uno. Hay 20 habitaciones en cada piso. Para esto, necesitas un arreglo que pueda recopilar y procesar información sobre las habitaciones ocupadas/libres.

Primer paso: El tipo de elementos del arreglo. En este caso, sería un valor Booleano (True/False).

Paso dos: análisis de la situación. Resume la información disponible: tres edificios, 15 pisos, 20 habitaciones.

Ahora puedes crear el arreglo:

```
rooms = [[[False for r in range(20)] for f in range(15)] for t in range(3)]
```

El primer índice (0 a 2) selecciona uno de los edificios; el segundo (0 a 14) selecciona el piso, el tercero (0 a 19) selecciona el número de habitación. Todas las habitaciones están inicialmente desocupadas.

Ahora ya puedes reservar una habitación para dos recién casados: en el segundo edificio, en el décimo piso, habitación 14:

```
rooms[1][9][13] = True
```

y desocupar el segundo cuarto en el quinto piso ubicado en el primer edificio:

```
rooms[0][4][1] = False
```

Verifica si hay disponibilidad en el piso 15 del tercer edificio:

```
vacancy = 0

for room_number in range(20):
    if not rooms[2][14][room_number]:
        vacancy += 1
```

La variable vacancy contiene 0 si todas las habitaciones están ocupadas, o en dado caso el número de habitaciones disponibles.

From:
https://matebcn.eu/ - **miguel angel torres egea**

Permanent link:
https://matebcn.eu/info:cursos:netacad:python:pe1m3

Last update: **14/06/2022 20:07**