

9

Booleanos y cadenas

En este capítulo exploraremos cómo manejar operaciones con valores booleanos y cadenas de texto en Python. Veremos las operaciones lógicas básicas, las particularidades de la *evaluación implícita* de Python (*truthy* y *falsy*) y las herramientas para manipular y formatear cadenas. Una buena parte del trabajo en ciencia de datos consiste en procesar texto: validar entradas, parsear formatos, comparar identificadores o normalizar cadenas. Por ello, prestaremos especial atención a la diferencia entre `str` y `bytes` y a las codificaciones (UTF-8).

9.1. Booleanos

Como ya hemos visto, en Python los valores booleanos están representados por `True` y `False`. Internamente son casos especiales del tipo `int`, donde `True` equivale a 1 y `False` a 0:

```
a = True
b = False

print(a + b)    # 1
print(a * 5)    # 5
print(int(a))   # 1
```

9.1.1. Operadores lógicos

Python dispone de los tres operadores lógicos clásicos. La Tabla 1 los recoge.

```
x = True
y = False
```

Operador	Descripción	Ejemplo
<code>not</code>	Negación	<code>not True</code> $\rightarrow$ <code>False</code>
<code>and</code>	Conjunción (AND)	<code>True and False</code> $\rightarrow$ <code>False</code>
<code>or</code>	Disyunción (OR)	<code>True or False</code> $\rightarrow$ <code>True</code>

Tabla 1: Operadores lógicos sobre booleanos.

```
print(not x)      # False
print(x and y)    # False
print(x or y)     # True
```

9.1.2. Valores *truthy* y *falsy*

En Python, cualquier objeto puede ser usado en una expresión condicional, no solamente `True/False`. La Tabla 2 recoge todos los valores *falsy* del lenguaje: cualquier otro valor se considera *truthy*.

Tipo	Valores <i>falsy</i>
<code>NoneType</code>	<code>None</code>
<code>bool</code>	<code>False</code>
<code>int</code> / <code>float</code>	<code>0</code> , <code>0.0</code>
<code>str</code>	<code>''</code> (cadena vacía)
<code>list</code> , <code>tuple</code> , <code>dict</code> , <code>set</code>	<code>[]</code> , <code>()</code> , <code>{}</code> , <code>set()</code> (vacíos)

Tabla 2: Valores considerados *falsy* en Python.

Esto permite escribir condicionales más concisos:

```
nombre = ""
if not nombre:
    print("Falta el nombre") # Se imprime: cadena vacía es
                             # falsy

valores = [1, 2, 3]
if valores:
    print("Hay valores")     # Se imprime: lista no vacía es
                             # truthy
```

El constructor `bool(x)` aplica explícitamente esta conversión:

```
print(bool(0), bool(""), bool([]), bool(None)) # False False
False False
print(bool(42), bool("a"), bool([0]))          # True True
True
```

9.1.3. Evaluación cortocircuitada

Python evalúa las expresiones booleanas con *cortocircuito*. Esto significa que si el resultado de la expresión está determinado por el primer operando, el segundo no se evalúa. Para `and`, si el primer operando es *falsy*, no se evalúa el segundo; para `or`, si el primer operando es *truthy*, no se evalúa el segundo.

```
1 def funcion():
2     print("Ejecutando función")
3     return True
4
5 print(False and funcion()) # No imprime "Ejecutando función"
6 print(True or funcion())  # Tampoco imprime
```

and y or no devuelven siempre booleanos

Una sutileza importante: `and` y `or` no devuelven necesariamente `True` o `False`, sino el último operando evaluado. Por ejemplo:

```
print("hola" or "mundo") # 'hola' (primer operando
    truthy)
print("" or "mundo")    # 'mundo' (primer operando
    falsy)
print(0 and 5)          # 0      (primer operando
    falsy)
print(3 and 5)          # 5      (primer operando
    truthy, devuelve segundo)
```

Esto se aprovecha para asignar valores por defecto: `nombre = entrada or "anonimo"` asigna `"anonimo"` si `entrada` es la cadena vacía. Pero atención: si el valor válido puede ser *falsy* (por ejemplo `0` o `""`), este truco produce errores. En esos casos, prefiere `x if x is not None else por_defecto`.

9.2. Cadenas

En Python, las cadenas de texto, de tipo `str`, son **secuencias inmutables de caracteres Unicode**. Dos propiedades importantes:

- **Inmutables:** una vez creada una cadena, no se pueden modificar sus caracteres. Cualquier “modificación” produce en realidad una nueva cadena.
- **Unicode:** los caracteres no son bytes, sino puntos de código Unicode. Esto significa que Python maneja sin problema acentos, eñes, ideogramas, emojis, etc.

9.2.1. Definición de cadenas

Las cadenas pueden definirse con comillas simples (`'`), dobles (`"`) o triples (`'''` o `"""`) para cadenas multilínea. Comillas simples y dobles son intercambiables; conviene mantener un estilo coherente y elegir el que evite escapar la otra.

```
1 cadena_simple = "Hola, mundo"
2 cadena_doble = 'Python es genial'
3 cadena_multilinea = """Este es
4 un ejemplo de cadena
5 multilinea"""
```

Las cadenas también admiten **secuencias de escape** para representar caracteres especiales: `\n` (salto de línea), `\t` (tabulación), `\\` (barra invertida), `\"` (comilla doble dentro de cadena entre comillas dobles), etc.

```
cadena_con_salto = "Primera línea\nSegunda línea"
cadena_con_tab = "Columna1\tColumna2"
```

Cadenas raw (sin escapes)

Si prefijamos la cadena con `r` (*raw*), las secuencias de escape no se interpretan. Es muy útil para expresiones regulares y rutas de Windows:

```
↪ print("C:\nuevo\test")      # \n se vuelve salto de línea y \t
    tabulador: no es la ruta!
↪ print(r"C:\nuevo\test")     # C:\nuevo\test (literal, la ruta
    correcta)
```

9.2.2. Inmutabilidad

Las cadenas son inmutables: intentar modificar uno de sus caracteres genera un `TypeError`:

```
↪ cadena = "Hola"
    # cadena[0] = 'h' # TypeError: 'str' object does not support
    item assignment
```

Sin embargo, podemos crear nuevas cadenas a partir de la original:

```
cadena = "Hola"
nueva_cadena = "h" + cadena[1:] # 'hola'
```

En este fragmento, `cadena[1:]` devuelve la subcadena `"ola"` (todos los caracteres a partir del índice 1). Concatenando con `"h"` obtenemos la nueva cadena `"hola"`. La variable `cadena` original no se ha modificado.

9.2.3. Indexación y rebanado

El acceso a caracteres se realiza con índices, donde el primero es 0. Python admite también índices negativos: -1 es el último carácter, -2 el penúltimo, etc.

```
cadena = "Python"
print(cadena[0])    # 'P'
print(cadena[-1])   # 'n'
```

El operador de rebanado [*inicio:fin:paso*] permite extraer subcadenas. Si se omite *inicio*, vale 0; si se omite *fin*, vale el final de la cadena; si se omite *paso*, vale 1. El final **no es inclusivo**.

```
cadena = "Python"
print(cadena[1:4])    # 'yth'      (índices 1, 2, 3)
print(cadena[-3:])    # 'hon'      (los tres últimos)
print(cadena[:2])     # 'Pto'      (uno de cada dos)
print(cadena[::-1])   # 'nohtyP'  (reverso completo)
```

El *idiom* `s[::-1]` es la forma habitual en Python de invertir una cadena.

9.2.4. Operadores sobre cadenas

La Tabla 3 resume los operadores más comunes sobre cadenas.

Operador	Descripción	Ejemplo
+	Concatenación	"a" + "b" → 'ab'
*	Repetición	"ab" * 3 → 'ababab'
in	Pertenencia (subcadena)	"py" in "python" → True
==	Igualdad	"abc" == "abc" → True
[i]	Indexación	"Python"[0] → 'P'
[i:j]	Rebanado	"Python"[1:4] → 'yth'
len()	Longitud (función)	len("Python") → 6

Tabla 3: Operadores y operaciones básicas sobre cadenas.

El operador `in` es la forma más *pythonica* de comprobar si una subcadena está contenida en otra (más legible y eficiente que `find >= 0`):

```
if "error" in mensaje:
    log_error(mensaje)
```

9.2.5. Evaluación implícita en condicionales

Como ya vimos, una cadena vacía es *falsy* y cualquier cadena no vacía es *truthy*. Esto permite condicionales muy limpios:

```
nombre = input("Tu nombre: ").strip()
if nombre:
    print(f"Hola, {nombre}")
else:
    print("Nombre vacío")
```

9.3. Métodos integrados de cadenas

Python ofrece una API muy rica para manipular cadenas. Agrupamos los métodos más utilizados por categoría.

9.3.1. Cambio de mayúsculas/minúsculas

- `s.lower()`: convierte a minúsculas.
- `s.upper()`: convierte a mayúsculas.
- `s.title()`: pone en mayúscula la primera letra de cada palabra.
- `s.capitalize()`: pone en mayúscula solo la primera letra de la cadena.
- `s.swapcase()`: invierte mayúsculas y minúsculas.
- `s.casefold()`: forma agresiva de `lower` pensada para comparaciones *insensibles a mayúsculas* (e.g., maneja correctamente la ß alemana).

```
print("Python ES genial".lower())      # 'python es genial'
print("python".title())                 # 'Python'
print("Hola".swapcase())                # 'hOLA'
print("STRASSE".casefold() == "Straße".casefold()) # True
```

Para comparar sin distinguir mayúsculas, usa casefold

Para comparaciones insensibles a mayúsculas con texto internacional (alemán, turco, etc.), `casefold` es más correcto que `lower`. Para texto exclusivamente ASCII, ambos producen el mismo resultado.

9.3.2. Eliminación de espacios y caracteres en los extremos

- `s.strip([chars])`: elimina caracteres (por defecto, espacios) al principio y al final.
- `s.lstrip([chars])`, `s.rstrip([chars])`: igual, pero solo a la izquierda o derecha respectivamente.

```
print("  Hola Mundo  ".strip())    # 'Hola Mundo '  
print("---Python---".strip("-"))    # 'Python '  
print("  Hola  ".lstrip())          # 'Hola  '
```

9.3.3. Búsqueda y conteo

- `s.find(sub)`: índice de la primera aparición de `sub`, o `-1` si no aparece.
- `s.index(sub)`: como `find`, pero lanza `ValueError` si no aparece.
- `s.count(sub)`: número de apariciones (sin solapamiento).
- `s.startswith(prefix)`, `s.endswith(suffix)`: `True` si empieza/termina con la subcadena dada.

```
cadena = "Hola Mundo, Mundo es genial"  
print(cadena.find("Mundo"))        # 5  
print(cadena.find("Python"))       # -1  
print(cadena.count("Mundo"))       # 2  
print(cadena.startswith("Hola"))   # True  
print(cadena.endswith("genial"))   # True
```

9.3.4. Modificación

- `s.replace(old, new)`: devuelve una copia con todas las apariciones de `old` sustituidas por `new`.

```
↪ print("Hola Mundo".replace("Mundo", "Python")) # 'Hola  
    Python '
```

9.3.5. División y unión

- `s.split(sep)`: divide `s` en una lista de subcadenas usando `sep` como separador. Sin argumento, divide por cualquier secuencia de espacios.
- `s.splitlines()`: divide por saltos de línea.
- `sep.join(iterable)`: concatena los elementos de un iterable de cadenas usando `sep` como separador.

```
↪ print("manzana, naranja, pera".split(", "))    # ['manzana', 'naranja', 'pera']  
↪ print("uno\ndos\ntres".splitlines())           # ['uno', 'dos', 'tres']  
↪ print(", ".join(["a", "b", "c"]))               # 'a, b, c'
```

9.3.6. Predicados sobre el contenido

Los métodos que empiezan por `is` devuelven `True` si todos los caracteres cumplen una propiedad y la cadena es no vacía:

- `s.isalpha()`: solo letras.
- `s.isdigit()`: solo dígitos.
- `s.isalnum()`: letras y/o dígitos.
- `s.isspace()`: solo espacios en blanco.
- `s.isascii()`: todos los caracteres son ASCII.
- `s.islower()`, `s.isupper()`, `s.istitle()`: relacionados con el caso.

```
print("Python3".isalnum())    # True
print("Python 3".isalnum())   # False (contiene espacio)
print("123".isdigit())        # True
print("Hola".isascii())       # True
print("Holá".isascii())       # False (a con tilde no es ASCII)
```

9.4. Formateo de cadenas

Python ofrece varias formas de construir cadenas a partir de valores. La forma recomendada hoy en día son los **f-strings** (literales formateados, introducidos en Python 3.6).

9.4.1. f-strings

Un f-string se introduce prefijando la cadena con `f` (o `F`). Dentro de las llaves `{}` se evalúa cualquier expresión de Python, y opcionalmente se especifica un formato tras `:`.

```
nombre = "Ana"
edad = 30
print(f"Hola {nombre}, tienes {edad} años")
# 'Hola Ana, tienes 30 años'

pi = 3.141592653589793
print(f"Pi con 3 decimales: {pi:.3f}")          # 3.142
print(f"En notacion científica: {pi:.3e}")       # 3.142e+00
print(f"Alineado a la derecha: |{pi:>10.2f}|")  # |          3.14|
print(f"Hexadecimal de 255: {255:#x}")          # 0xff
```

9.4.2. El método `str.format()`

Antes de los f-strings, la forma habitual era `str.format()`:

```
print("Hola {}, tienes {} años".format("Ana", 30))
print("Pi: {:.3f}".format(3.14159)) # 'Pi: 3.142'
```

Es equivalente a los f-strings pero menos legible (los argumentos quedan separados de los huecos). Sigue siendo útil cuando la *plantilla* se construye dinámicamente.

9.4.3. Operador `%` (estilo C)

El operador `%` aplicado a cadenas implementa formateo al estilo C (`printf`). Está obsoleto y se desaconseja en código nuevo:

```
↪ print("Hola %s, tienes %d años" % ("Ana", 30)) # Estilo
    viejo
```

Nunca formatees cadenas con datos no confiables

Tanto `%` como `.format` pueden ser **vehículo de inyección**: si la cadena de formato proviene del usuario y no está validada, un atacante puede acceder a atributos del objeto. Por eso, los **datos del usuario deben pasarse como argumento**, nunca como cadena de formato:

```
↪ # MAL: si entrada_usuario contiene "{0.__class__}",
    revela información interna
print(entrada_usuario.format(datos))

# BIEN: el formato es nuestro, los datos son argumentos
print("Recibido: {}".format(entrada_usuario))
```

Para construir SQL, comandos shell, HTML, etc., **nunca** concatenes ni formatees con `%/f-string`: usa los mecanismos de parametrización seguros (placeholders SQL, subprocess con lista de argumentos, escapado HTML).

9.5. Bytes y codificación

En Python 3 se distingue claramente entre dos tipos de “secuencias de caracteres”:

- **str**: secuencia de **puntos de código Unicode** (lo que llamamos “texto”).
- **bytes**: secuencia de **bytes crudos** (octetos, valores entre 0 y 255).

Esta separación es muy relevante en ciberseguridad: cuando enviamos datos por la red, leemos un fichero binario o calculamos un hash, trabajamos con **bytes**, no con **str**. Una cadena **str** se convierte a **bytes** con `encode`, y al revés con `decode`. La **codificación** indica cómo se traducen puntos de código a bytes; UTF-8 es la opción estándar y la única razonable salvo necesidad explícita.

```
texto = "Hola, ñoño"
datos = texto.encode("utf-8")
print(datos)                # b'Hola, \xc3\xbf\xc3\xbf'
print(len(texto))           # 10  (caracteres)
print(len(datos))           # 12  (bytes; cada n con tilde son
↪      2 bytes en UTF-8)

print(datos.decode("utf-8")) # 'Hola, ñoño'
```

Los literales de tipo bytes se prefijan con b: b"holá". Solo admiten caracteres ASCII; cualquier byte de mayor valor se escapa con \xNN.

No mezcles str y bytes

Concatenar `str` con `bytes` produce `TypeError`. Si lees datos binarios de la red o de un fichero (`open(..., "rb")`) y necesitas tratarlos como texto, decodifícalos explícitamente con la codificación correcta. Si la codificación es desconocida, usa `errors="replace"` o `errors="ignore"` con cuidado: descartar bytes silenciosamente puede ocultar ataques.

9.5.1. Normalización Unicode

Algunos caracteres Unicode pueden representarse de varias maneras: por ejemplo, la “ñ” puede ser un único punto de código (U+00F1) o la combinación de “n” más el modificador de tilde (U+006E seguido de U+0303). Visualmente son idénticos, pero como cadenas son distintos. El módulo `unicodedata` proporciona la función `normalize` para llevar las cadenas a una forma canónica antes de compararlas:

```
import unicodedata

a = "\u00f1"                # Forma compuesta: n con tilde (U+00F1)
b = "n\u0303"               # Forma descompuesta: n (U+006E) +
↪      tilde combinada (U+0303)
print(a, b, a == b)        # n n False (visualmente iguales,
↪      distintos como cadena)
print(unicodedata.normalize("NFC", a) == unicodedata.
↪      normalize("NFC", b)) # True
```

9.6. Ejercicios

EJERCICIO 9.1

Escribe un programa en Python que reciba cuatro valores booleanos a , b , c y d y devuelva `True` si se cumple alguna de las siguientes condiciones:

- La combinación de todos los valores es `True`.
- Exactamente tres de los valores son `True`.
- El valor de c es `False` y d es `True`.

El programa debe solicitar los valores booleanos al usuario y mostrar el resultado.

Ejemplo de ejecución esperada

```
Introduce a (True/False): True
Introduce b (True/False): False
Introduce c (True/False): False
Introduce d (True/False): True
Resultado: True
```

Solución (girar para leer)

```

1 # Función para evaluar las condiciones lógicas
2 def evaluacion_exposicion_logica(a, b, c, d):
3     # comprobamos las tres condiciones
4     return (a and b and c and d) or (sum([a, b, c, d])
5           == 3) or (not c and d)
6 # Solicitamos los valores booleanos al usuario
7 a = input("Introduce a (True/False): ").strip().lower()
8 b = input("Introduce b (True/False): ").strip().lower()
9 c = input("Introduce c (True/False): ").strip().lower()
10 d = input("Introduce d (True/False): ").strip().lower()
11 == 'true'
12 # Llamamos a la función y mostramos el resultado
13 resultado = evaluacion_exposicion_logica(a, b, c, d)
14 print(f"Resultado: {resultado}")

```

Código 9.1: Solución al Ejercicio 9.1.

EJERCICIO 9.2

Escribe un programa en Python que reciba un número entero n y determine si cumple con las siguientes condiciones:

- Es divisible por 7.
- Contiene al menos un dígito impar.

El programa debe devolver `True` si ambas condiciones se cumplen, y `False` en caso contrario.

Ejemplo de ejecución esperada

Introduce un número: 42

Resultado: False

Solución (girar para leer)

Código 9.2: Solución al Ejercicio 9.2.

```

1 def numeros-con-propiedades-booleanas(n):
2     # Verificar si el número es divisible por 7
3     divisible_por_7 = (n % 7 == 0)
4
5     # Verificar si contiene al menos un dígito impar
6     tiene_impar = False
7     for dígito in str(abs(n)):
8         if int(dígito) % 2 != 0:
9             tiene_impar = True
10            break
11
12    # Ambas condiciones deben cumplirse
13    return divisible_por_7 and tiene_impar
14
15    # Solicitar al usuario un número
16    n = int(input("Introduce un número: "))
17
18    # Evaluar y mostrar el resultado
19    resultado = numeros-con-propiedades-booleanas(n)
20    print(f"Resultado: {resultado}")

```

EJERCICIO 9.3

Escribe un programa en Python que reciba un número entero positivo n y determine si es un **palíndromo**, sin convertirlo en cadena. Un número es palíndromo si se lee igual de izquierda a derecha que de derecha a izquierda.

Ejemplo de ejecución esperada

Introduce un número: 121

Resultado: True

Solución (girar para leer)

```

1  def es_palindromo(n):
2      original = n
3      invertido = 0
4      while n > 0:
5          invertido = invertido * 10 + n % 10
6          n //= 10
7      return original == invertido
8
9  # Solicitar al usuario un número
10     n = int(input("Introduce un número: "))
11
12     # Evaluar y mostrar el resultado
13     resultado = es_palindromo(n)
14     print(f"Resultado: {resultado}")

```

Código 9.3: Solución al Ejercicio 9.3.

EJERCICIO 9.4

Escribe un programa en Python que reciba dos números enteros a y b y determine si ambos tienen la misma cantidad de bits en 1 en su representación binaria.

Ejemplo de ejecución esperada

```
Introduce el primer número: 5
Introduce el segundo número: 3
Resultado: True
```

Solución (girar para leer)

Código 9.4: Solución al Ejercicio 9.4.

```
1 def comparacion_bits(a, b):
2     # Cuenta los bits en la representación binaria
3     de ambos números
4     return bin(a).count('1') == bin(b).count('1')
5 # Solicitar al usuario dos números
6 a = int(input("Introduce el primer número: "))
7 b = int(input("Introduce el segundo número: "))
8
9 # Evaluar y mostrar el resultado
10 resultado = comparacion_bits(a, b)
11 print(f"Resultado: {resultado}")
```

EJERCICIO 9.5

Escribe un programa en Python que reciba un número entero y determine si sus dígitos alternan entre pares e impares. Por ejemplo, el número 272 cumple la condición, mientras que 4648 no.

Ejemplo de ejecución esperada

Introduce un número: 272

Resultado: True

Solución (girar para leer)

```

1 def es_alternante(n):
2     n = str(abs(n)) # Convertir el número a cadena
3     # para iterar sobre los dígitos
4     for i in range(len(n) - 1):
5         # Verificamos si los dígitos alternan entre
6         # pares e impares
7         if (int(n[i]) % 2) == (int(n[i + 1]) % 2):
8             return False
9         return True
10 # Solicitar al usuario un número
11 n = int(input("Introduce un número: "))
12 # Evaluar y mostrar el resultado
13 resultado = es_alternante(n)
14 print(f"Resultado: {resultado}")

```

Código 9.5: Solución al Ejercicio 9.5.

EJERCICIO 9.6

Escribe un programa en Python que verifique si una contraseña cumple las siguientes condiciones:

- Al menos 8 caracteres de longitud.
- Contiene al menos una letra mayúscula.
- Contiene al menos un número.
- No tiene caracteres especiales.

Ejemplo de ejecución esperada

Introduce una contraseña: Python123

Resultado: True

Solución (girar para leer)

```

1  def validar_contraseña(contraseña):
2      # Verificar que la longitud sea al menos 8
3      caracteres
4      if len(contraseña) < 8:
5          return False
6      # Verificar que contenga al menos una letra mayú
7      scula
8      tiene_mayuscula = False
9      for c in contraseña:
10         if c.isupper():
11             tiene_mayuscula = True
12             break
13         if not tiene_mayuscula:
14             return False
15     # Verificar que contenga al menos un número
16     tiene_numero = False
17     for c in contraseña:
18         if c.isdigit():
19             tiene_numero = True
20             break
21         if not tiene_numero:
22             return False
23     # Verificar que no tenga caracteres especiales
24     for c in contraseña:
25         if not c.isalnum():
26             return False
27     return True
28
29     return True
30
31     # Solicitar la contraseña
32     contraseña = input("Introduce una contraseña: ")
33
34     # Evaluar y mostrar el resultado
35     resultado = validar_contraseña(contraseña)
36     print(f"Resultado: {resultado}")

```

Código 9.6: Solución al Ejercicio 9.6.

EJERCICIO 9.7

Escribe un programa en Python que reciba una lista de números enteros y determine si los números están en orden consecutivo ascendente o descendente.

Ejemplo de ejecución esperada

Introduce una lista de números (separados por espacio): 3 4 5 6

Resultado: True

Solución (girar para leer)

```

1 def verificar_consecutivos(numeros):
2     # Verificar si están en orden ascendente
3     ascendente = True
4     for i in range(len(numeros) - 1):
5         if numeros[i] + 1 != numeros[i + 1]:
6             ascendente = False
7             break
8     # Verificar si están en orden descendente
9     descendente = True
10    for i in range(len(numeros) - 1):
11        if numeros[i] - 1 != numeros[i + 1]:
12            descendente = False
13            break
14    return ascendente or descendente
15
16 # Solicitar los números
17 entrada = input("Introduce una lista de números (separados por espacio): ")
18 numeros = [int(x) for x in entrada.split()]
19 # Evaluar y mostrar el resultado
20 resultado = verificar_consecutivos(numeros)
21 print(f"Resultado: {resultado}")

```

Código 9.7: Solución al Ejercicio 9.7.

EJERCICIO 9.8

Escribe un programa en Python que reciba una fecha en formato DD/MM/YYYY como cadena de texto y determine si es una fecha válida.

Ejemplo de ejecución esperada

Introduce una fecha en formato DD/MM/YYYY: 29/02/2021
Resultado: False

Solución (girar para leer)

```

1  def es_bisiesto(año):
2      if (año % 4 == 0 and (año % 100 != 0 or año %
3          return True
4          return False
5      evaluar_fecha):
6      # Dividir la fecha en día, mes y año
7      día, mes, año = fecha.split("/")
8      día = int(día)
9      mes = int(mes)
10     año = int(año)
11
12     # Verificar si el mes es válido
13     if mes < 1 or mes > 12:
14         return False
15
16     # Verificar el número de días según el mes
17     if mes == 2:
18         if es_bisiesto(año):
19             return día <= 29 # Año bisiesto, febrero
20             tiene 29 días
21             return día <= 28 # Año no bisiesto, febrero
22             tiene 28 días
23         elif mes in [4, 6, 9, 11]:
24             return día <= 30 # Meses con 30 días
25             return día <= 31 # Los demás meses tienen 31 días
26     # Solicitar la fecha al usuario
27     fecha = input("Introduce una fecha en formato DD/MM/
28     YYYY: ")
29     # Evaluar y mostrar el resultado
30     resultado = evaluar_fecha(fecha)
31     print(f"Resultado: {resultado}")

```

Código 9.8: Solución al Ejercicio 9.8.

EJERCICIO 9.9

Escribe un programa en Python que reciba tres valores numéricos y evalúe si se cumple al menos una de las siguientes condiciones:

- El primer número es mayor que 10 y el segundo es menor que 5.
- El segundo número es mayor que 10 y el tercero es menor que 0.
- El tercer número es mayor que 0 y la suma de los dos primeros números es par.

Ejemplo de ejecución esperada

```
Introduce el primer número: 12
Introduce el segundo número: 4
Introduce el tercer número: 2
Resultado: True
```

Solución (gitar para leer)

Código 9.9: Solución al Ejercicio 9.9.

```

1 def evaluar_condiciones(a, b, c):
2     # Verificar las tres condiciones
3     if (a < 10 and b > 5):
4         return True
5     if (b < 10 and c > 0):
6         return True
7     if (c < 0 and a + b) % 2 == 0):
8         return True
9     return False
10
11 # Solicitar los tres números al usuario
12 a = int(input("Introduce el primer número: "))
13 b = int(input("Introduce el segundo número: "))
14 c = int(input("Introduce el tercer número: "))
15
16 # Evaluar y mostrar el resultado
17 resultado = evaluar_condiciones(a, b, c)
18 print(f"Resultado: {resultado}")

```

EJERCICIO 9.10

Escribe un programa en Python que reciba una lista de valores booleanos y determine si alguno de los valores cumple con la siguiente expresión lógica:

$$(A \wedge B) \vee (C \wedge \neg D)$$

El programa debe verificar las condiciones basadas en la lista de valores proporcionados.

Ejemplo de ejecución esperada

Introduce una lista de valores booleanos (True/False, separados por espacio): True False True False
Resultado: True

Solución (girar para leer)

```

1  def validacion_expresion_logica(valores):
2      A, B, C, D = valores[0], valores[1], valores[2],
3      valores[3]
4      # Evaluamos la expresión lógica: (A and B) or (
5      and not D)
6      return (A and B) or (C and not D)
7      # Solicitar la lista de valores booleanos
8      entrada = input("Introduce una lista de valores
9      booleanos (True/False, separados por espacio): ")
10     valores = [True if x.strip().lower() == 'true' else
11                 False for x in entrada.split()]
12     # Evaluar y mostrar el resultado
13     resultado = validacion_expresion_logica(valores)
14     print(f"Resultado: {resultado}")

```

Código 9.10: Solución al Ejercicio 9.10.

EJERCICIO 9.11

Escribe un programa en Python que reciba una lista de valores booleanos y determine si más del 75 % de los valores son True.

Ejemplo de ejecución esperada

Introduce una lista de valores booleanos: True True False True False True
Resultado: True

Solución (girar para leer)

```

1 def consenso_múltiples_evaluaciones(valores):
2     # Contamos cuántos valores son True
3     cuenta_true = 0
4     for valor in valores:
5         if valor:
6             cuenta_true += 1
7     # Comprobamos si más del 75% de los valores son
8     # True
9     return cuenta_true > len(valores) * 0.75
10 # Solicitar la lista de valores booleanos
11 entrada = input("Introduce una lista de valores
12 booleanos (True/False, separados por espacio): ")
13 valores = [True if x.strip().lower() == 'true' else
14             False for x in entrada.split()]
15 # Evaluar y mostrar el resultado
16 resultado = consenso_múltiples_evaluaciones(valores)
17 print(f"Resultado: {resultado}")

```

Código 9.11: Solución al Ejercicio 9.11.

EJERCICIO 9.12

Escribe un programa en Python que reciba una lista de valores booleanos y determine si los valores alternan entre True y False.

Ejemplo de ejecución esperada

Introduce una lista de valores booleanos: True False True False
Resultado: True

Solución (girar para leer)

```

1 def secuencias_condiciones_alternadas(valores):
2     for i in range(len(valores) - 1):
3         if valores[i] == valores[i + 1]:
4             return False
5         return True
6
7     # Solicitar la lista de valores booleanos
8     entrada = input("Introduce una lista de valores
9     booleanos (True/False, separados por espacio): ")
10    valores = [True if x.strip().lower() == 'true' else
11                False for x in entrada.split()]
12    # Evaluar y mostrar el resultado
13    resultado = secuencias_condiciones_alternadas(valores)
14    print(f"Resultado: {resultado}")

```

Código 9.12: Solución al Ejercicio 9.12.

EJERCICIO 9.13

Escribe un programa en Python que reciba tres listas de valores booleanos y determine si en cada posición al menos dos de los tres valores coinciden.

Ejemplo de ejecución esperada

Introduce tres listas de valores booleanos:

True False True

False False True

True True False

Resultado: True

Solución (girar para leer)

```

1  def consenso_logico_cruzado(lista1, lista2, lista3):
2      for i in range(len(lista1)):
3          if lista1[i] == lista2[i] or lista1[i] ==
4              lista3[i]:
5              continue
6          else:
7              return False
8              return True
9      # Solicitar las tres listas de valores booleanos:
10     print("Introduce tres listas de valores booleanos:")
11     lista1 = [True if x.strip().lower() == 'true' else
12               False for x in input("Lista 1: ").split()]
13     lista2 = [True if x.strip().lower() == 'true' else
14               False for x in input("Lista 2: ").split()]
15     lista3 = [True if x.strip().lower() == 'true' else
16               False for x in input("Lista 3: ").split()]
17     # Evaluar y mostrar el resultado
18     resultado = consenso_logico_cruzado(lista1, lista2,
19                                         lista3)
20     print(f"Resultado: {resultado}")

```

Código 9.13: Solución al Ejercicio 9.13.

EJERCICIO 9.14

Escribe un programa en Python que reciba una cadena de bits (una secuencia de valores True y False) y determine si la cantidad de True en la cadena es **par** o **impar**. El programa debe devolver True si la cantidad de True es **par**, y False si la cantidad de True es **impar**.

Ejemplo de ejecución esperada

Introduce la cadena de bits (True/False, separados por espacio): True False True False True
Resultado: False

Solución (girar para leer)

```

1 def evaluar_paridad(bits):
2     cuenta_true = 0
3     for bit in bits:
4         if bit:
5             cuenta_true += 1
6     # Verificamos si la cantidad de True es par
7     return cuenta_true % 2 == 0
8
9 # Solicitar la cadena de bits al usuario
10 entrada = input("Introduce la cadena de bits (True/
    ↩ False, separados por espacio): ")
11 bits = []
12 for x in entrada.split():
13     if x.strip().lower() == 'true':
14         bits.append(True)
15     else:
16         bits.append(False)
17
18 # Evaluar y mostrar el resultado
19 resultado = evaluar_paridad(bits)
20 print(f"Resultado: {resultado}")

```

Código 9.14: Solución al Ejercicio 9.14.

EJERCICIO 9.15

Escribe un programa en Python que reciba una cadena de bits (una secuencia de valores `True` y `False`) y determine si los valores alternan entre `True` y `False`. El programa debe devolver `True` si los valores alternan correctamente, y `False` si no lo hacen.

Ejemplo de ejecución esperada

Introduce la cadena de bits (True/False, separados por espacio): `True False True False True`
Resultado: `True`

Solución (girar para leer)

```

1 def verificar_secuencia_alternada(bits):
2     for i in range(len(bits) - 1):
3         if bits[i] == bits[i + 1]: # Si dos valores
4             return False
5         return True
6     # Solicitar la cadena de bits al usuario
7     entrada = input("Introduce la cadena de bits (True/
8     False, separados por espacio): ")
9     bits = []
10    for x in entrada.split():
11        if x.strip().lower() == 'true':
12            bits.append(True)
13        else:
14            bits.append(False)
15    # Evaluar y mostrar el resultado
16    resultado = verificar_secuencia_alternada(bits)
17    print(f"Resultado: {resultado}")

```

Código 9.15: Solución al Ejercicio 9.15.

EJERCICIO 9.16

Escribe un programa en Python que reciba una cadena y determine cuántas subcadenas palindrómicas contiene. Una subcadena es palindrómica si se lee igual de izquierda a derecha que de derecha a izquierda. Por ejemplo, la cadena "abba" tiene 6 subcadenas palindrómicas: "a" (2 veces), "b" (2 veces), "bb" y "abba".

Ejemplo de ejecución esperada

Introduce una cadena: abba

Resultado: 6

Solución (girar para leer)

```

1  def es_palindromo(s):
2      return s[::-1] == s
3
4  def contar_palindromos(cadena):
5      contador = 0
6      for i in range(len(cadena)):
7          for j in range(i + 1, len(cadena) + 1):
8              if es_palindromo(cadena[i:j]):
9                  contador += 1
10         return contador
11
12     # Solicitar la cadena al usuario
13     cadena = input("Introduce una cadena: ")
14
15     # Evaluar y mostrar el resultado
16     resultado = contar_palindromos(cadena)
17     print(f"Resultado: {resultado}")

```

Código 9.16: Solución al Ejercicio 9.16.

EJERCICIO 9.17

Escribe un programa en Python que reciba una cadena de texto y la convierta a formato *Title Case*. En este formato, la primera letra de cada palabra debe ser mayúscula, pero deben aplicarse las siguientes reglas:

- Las palabras con menos de 4 caracteres deben mantenerse en minúscula (excepto la primera y la última palabra).
- Las preposiciones, artículos y conjunciones que aparecen en medio de la cadena deben mantenerse en minúscula (por ejemplo, "de", "a", "para", "y", "en"; considera que su longitud es siempre menor que 4).

Ejemplo de ejecución esperada

Introduce un texto: el rápido zorro marrón salta sobre la liebre perezosa
Resultado: El Rápido Zorro Marrón Salta Sobre la Liebre Perezosa

Solución (girar para leer)

```

1 def convertir_title_case(cadena):
2     palabras = cadena.split()
3     resultado = ""
4     for i in range(len(palabras)):
5         if i == 0 or i == len(palabras) - 1 or len(
6             palabras[i]) >= 4:
7             resultado += palabras[i].capitalize() + " "
8         else:
9             resultado += palabras[i].lower() + " "
10    return resultado.strip()
11
12 # Solicitar el texto al usuario
13 texto = input("Introduce un texto: ")
14 # Evaluar y mostrar el resultado
15 resultado = convertir_title_case(texto)
16 print(f"Resultado: {resultado}")

```

Código 9.17: Solución al Ejercicio 9.17.

EJERCICIO 9.18

Escribe un programa en Python que reciba una cadena y devuelva todas las combinaciones posibles de caracteres únicos que se pueden formar con los caracteres de la cadena. No se deben incluir combinaciones repetidas (por ejemplo, si la cadena tiene caracteres repetidos, no debe generarse la misma combinación más de una vez).

Ejemplo de ejecución esperada

Introduce una cadena: aab

Resultado: a b ab

Ejemplo de ejecución esperada

Introduce una cadena: abba

Resultado: a b ab ba

Solución (girar para leer)

```

1  def combinaciones_unicas(cadena):
2      resultado = ""
3
4      # Generamos todas las subcadenas posibles
5      for i in range(1, len(cadena) + 1):
6          for j in range(len(cadena) - i + 1):
7              subcadena = cadena[j:j+i]
8
9      # Verificamos si la subcadena tiene
10         caracteres unicos
11         es_unica = True
12         for k in range(len(subcadena)):
13             for l in range(k + 1, len(subcadena)):
14                 if subcadena[k] == subcadena[l]:
15                     es_unica = False
16                     break
17                 if not es_unica:
18                     break
19         if es_unica and subcadena not in resultado:
20             resultado += subcadena + " "
21
22     return resultado.strip()
23
24     # Solicitar la cadena al usuario
25     cadena = input("Introduce una cadena: ")
26
27     # Evaluar y mostrar el resultado
28     resultado = combinaciones_unicas(cadena)
29     print(f"Resultado: {resultado}")

```

Código 9.18: Solución al Ejercicio 9.18.

EJERCICIO 9.19

Escribe un programa en Python que reciba una cadena de texto y cuente cuántas veces aparece cada palabra en la cadena. El programa debe devolver un diccionario con las palabras como claves y su frecuencia como valores.

Ejemplo de ejecución esperada

Introduce una cadena: Hola hola mundo mundo mundo

Resultado: hola:2 mundo:3

Solución (girar para leer)

```

1 def contar_frecuencia_palabras(cadena):
2     palabras = cadena.lower().split()
3     resultado = ""
4     # Recorrer las palabras y contar las frecuencias
5     for palabra in palabras:
6         contador = 0
7         for p in palabras:
8             if p == palabra:
9                 contador += 1
10            if "{"palabra}:{contador}" not in resultado:
11                resultado += "{"palabra}:{contador}"
12            return resultado.strip()
13
14 # Solicitar la cadena al usuario
15 cadena = input("Introduce una cadena: ")
16
17 # Evaluar y mostrar el resultado
18 resultado = contar_frecuencia_palabras(cadena)
19 print(f"Resultado: {resultado}")

```

Código 9.19: Solución al Ejercicio 9.19.

EJERCICIO 9.20

Escribe un programa en Python que reciba una cadena y devuelva la subcadena más larga que contenga sólo caracteres únicos. Si hay más de una subcadena con la misma longitud, devuelve la que aparezca primero en la cadena original.

Ejemplo de ejecución esperada

Introduce una cadena: abcabcbb

Resultado: abc

Solución (girar para leer)

```

1 def subcadena_mas_larga_unica(cadena):
2     max_subcadena = ""
3     subcadena_actual = ""
4     for char in cadena:
5         if char not in subcadena_actual:
6             subcadena_actual += char
7         else:
8             subcadena_actual = subcadena_actual[
9                 subcadena_actual.index(char) + 1:] +
10             char
11     if len(subcadena_actual) > len(max_subcadena):
12         max_subcadena = subcadena_actual
13     return max_subcadena
14
15 # Solicitar la cadena al usuario
16 cadena = input("Introduce una cadena: ")
17
18 # Evaluar y mostrar el resultado
19 resultado = subcadena_mas_larga_unica(cadena)
20 print(f"Resultado: {resultado}")

```

Código 9.20: Solución al Ejercicio 9.20.

EJERCICIO 9.21

Escribe un programa en Python que reciba una cadena y determine si contiene caracteres repetidos consecutivos. Devuelve True si se encuentran, de lo contrario, devuelve False.

Ejemplo de ejecución esperada

Introduce una cadena: Hola Mundo

Resultado: False

Solución (girar para leer)

```

1  def comprobacion_caracteres_repetidos(cadena):
2      for i in range(len(cadena) - 1):
3          if cadena[i] == cadena[i + 1]:
4              return True
5          return False
6      # Solicitar la cadena al usuario
7      cadena = input("Introduce una cadena: ")
8      # Evaluar y mostrar el resultado
9      resultado = comprobacion_caracteres_repetidos(cadena)
10     print(f"Resultado: {resultado}")

```

Código 9.21: Solución al Ejercicio 9.21.

EJERCICIO 9.22

Escribe un programa en Python que reciba una cadena de texto y devuelva la cadena con cada palabra invertida, pero manteniendo el orden de las palabras. Por ejemplo, "Hola Mundo" debe convertirse en "aloH odnuM".

Ejemplo de ejecución esperada

Introduce una cadena: Hola Mundo

Resultado: aloH odnuM

Solución (girar para leer)

```

1  def inversion_palabra_por_palabra(cadena):
2      palabras = cadena.split() # Dividimos la cadena en
3      palabras
4      resultado = ""
5      # Recorremos cada palabra y revertirla manualmente
6      for palabra in palabras:
7          palabra_invertida = ""
8          for i in range(len(palabra) - 1, -1, -1): #
9              # Recorremos la palabra de atrás hacia
10             # adelante
11             palabra_invertida += palabra[i]
12             resultado += palabra_invertida + " "
13         return resultado.strip()
14     # Solicitar la cadena al usuario
15     cadena = input("Introduce una cadena: ")
16     # Evaluar y mostrar el resultado
17     resultado = inversion_palabra_por_palabra(cadena)
18     print(f"Resultado: {resultado}")

```

Código 9.22: Solución al Ejercicio 9.22.

EJERCICIO 9.23

Escribe un programa en Python que reciba una cadena y una subcadena y determine si la subcadena aparece en la cadena principal. Devuelve `True` si la subcadena está presente, de lo contrario, devuelve `False`.

Ejemplo de ejecución esperada

Introduce la cadena: Hola Mundo

Introduce la subcadena: Mundo

Resultado: `True`

Solución (girar para leer)

```

1 def verificacion_subcadena(cadena, subcadena):
2     # Comprobamos si la subcadena aparece en la cadena
3     # utilizando un bucle
4     for i in range(len(cadena) - len(subcadena) + 1):
5         # Recorremos la cadena hasta el tamaño de la
6         # subcadena
7         if cadena[i:i + len(subcadena)] == subcadena:
8             # Comparamos la subcadena con la parte
9             # actual de la cadena
10            return True
11        # También se podría usar directamente:
12        # return subcadena in cadena
13    # Solicitar la cadena y la subcadena al usuario
14    cadena = input("Introduce la cadena: ")
15    subcadena = input("Introduce la subcadena: ")
16    # Evaluar y mostrar el resultado
17    resultado = verificacion_subcadena(cadena, subcadena)
18    print(f"Resultado: {resultado}")

```

Código 9.23: Solución al Ejercicio 9.23.

EJERCICIO 9.24

Escribe un programa en Python que reciba dos cadenas y determine si una es un anagrama de la otra. Un anagrama es una palabra o frase formada por la reordenación de las letras de otra, usando todas las letras originales exactamente una vez.

Ejemplo de ejecución esperada

Introduce la primera cadena: listen
 Introduce la segunda cadena: silent
 Resultado: True

Solución (girar para leer)

```

1  def es_anagrama(cadena1, cadena2):
2      # Eliminar espacios y convertir a minúsculas
3      cadena1 = cadena1.replace(" ", "").lower()
4      cadena2 = cadena2.replace(" ", "").lower()
5
6      # Comparamos si las letras de ambas cadenas son
7      iguales
8      if sorted(cadena1) == sorted(cadena2):
9          return True
10         return False
11     # Solicitar las cadenas al usuario
12     cadena1 = input("Introduce la primera cadena: ")
13     cadena2 = input("Introduce la segunda cadena: ")
14
15     # Evaluar y mostrar el resultado
16     resultado = es_anagrama(cadena1, cadena2)
17     print(f"Resultado: {resultado}")

```

Código 9.24: Solución al Ejercicio 9.24.

EJERCICIO 9.25

Escribe un programa en Python que reciba una cadena y un número n , y devuelva cuántas palabras en la cadena tienen exactamente n caracteres.

Ejemplo de ejecución esperada

Introduce una cadena: Hola Mundo programadores

Introduce la longitud de palabra: 5

Resultado: 1

Solución (girar para leer)

```

1 def contar_palabras_longitud_especifica(cadena, n):
2     # Dividimos la cadena en palabras
3     palabras = cadena.split()
4     contador = 0
5     # Contamos cuántas palabras tienen exactamente n
6     caracteres
7     for palabra in palabras:
8         if len(palabra) == n:
9             contador += 1
10
11     return contador
12
13 # Solicitar la cadena y la longitud al usuario
14 cadena = input("Introduce una cadena: ")
15 n = int(input("Introduce la longitud de palabra: "))
16
17 # Evaluar y mostrar el resultado
18 resultado = contar_palabras_longitud_especifica(cadena,
19 print(f"Resultado: {resultado}")

```

Código 9.25: Solución al Ejercicio 9.25.

EJERCICIO 9.26

Escribe un programa en Python que reciba una cadena y determine cuál es la secuencia de caracteres repetidos más larga en la cadena. La función debe devolver la secuencia más larga y su longitud. Si hay varias secuencias de la misma longitud, debe devolver la primera.

Ejemplo de ejecución esperada

Introduce una cadena: aaabbbbbccccc

Resultado: bbbbb 5

Solución (girar para leer)

```

1 def secuencia_repetida_larga(cadena):
2     max_secuencia = ""
3     secuencia_actual = ""
4
5     # Iteramos sobre la cadena para encontrar las
6     secuencias repetidas
7     for i in range(len(cadena)):
8         if i == 0 or cadena[i] == cadena[i - 1]:
9             secuencia_actual += cadena[i]
10        else:
11            if len(secuencia_actual) > len(
12                max_secuencia):
13                max_secuencia = secuencia_actual
14            secuencia_actual = cadena[i]
15        # Verificar la última secuencia
16        if len(secuencia_actual) > len(max_secuencia):
17            max_secuencia = secuencia_actual
18    return max_secuencia, len(max_secuencia)
19
20 # Solicitar la cadena al usuario
21 cadena = input("Introduce una cadena: ")
22
23 # Evaluar y mostrar el resultado
24 secuencia, longitud = secuencia_repetida_larga(cadena)
25 print(f"Resultado: {secuencia} {longitud}")

```

Código 9.26: Solución al Ejercicio 9.26.

EJERCICIO 9.27

Escribe un programa en Python que reciba una cadena de texto y devuelva todas las combinaciones únicas de palabras que se pueden formar utilizando las palabras de la cadena. No se deben incluir combinaciones repetidas (por ejemplo, si una palabra aparece varias veces, no se debe generar la misma combinación más de una vez).

Ejemplo de ejecución esperada

Introduce una cadena: hola mundo hola

Resultado: hola mundo, hola hola

Solución (girar para leer)

```

1 def combinaciones_unicas(cadena):
2     # Dividimos la cadena en palabras
3     palabras = cadena.split()
4     resultado = ""
5     # Recorremos las palabras para crear combinaciones
6     for i in range(len(palabras)):
7         for j in range(i + 1, len(palabras)):
8             # Combinamos palabras y evitamos
9             repeticiones
10            combinacion = palabras[i] + " " + palabras[
11                j]
12            combinacion_invertida = palabras[j] + " " + palabras[i]
13            # Verificamos que las combinaciones no estén
14            n repeticidas
15            if combinacion not in resultado and
16                combinacion_invertida not in resultado:
17                resultado += combinacion + " "
18            return resultado.strip(", ")
19        # Solicitar la cadena al usuario
20        cadena = input("Introduce una cadena: ")
21        # Evaluar y mostrar el resultado
22        resultado = combinaciones_unicas(cadena)
23        print(f"Resultado: {resultado}")

```

Código 9.27: Solución al Ejercicio 9.27.

EJERCICIO 9.28

Escribe un programa en Python que reciba una cadena de texto y determine la longitud de la palabra más larga que aparece en la cadena. El programa debe de escribir la palabra más larga y su longitud como resultado. La palabra más larga debe ser identificada independientemente de si la cadena contiene espacios u otros delimitadores.

Ejemplo de ejecución esperada

Introduce una cadena: La programación en Python es divertida

Resultado: programación 12

Solución (girar para leer)

```

1 def longitud_palabra_maxima(cadena):
2     palabras = cadena.split() # Dividimos la cadena en
3     palabras
4     max_palabra = ""
5     # Recorremos las palabras para encontrar la más
6     larga
7     for palabra in palabras:
8         if len(palabra) > len(max_palabra):
9             max_palabra = palabra
10    return max_palabra, len(max_palabra)
11
12 # Solicitar la cadena al usuario
13 cadena = input("Introduce una cadena: ")
14
15 # Evaluar y mostrar el resultado
16 palabra, longitud = longitud_palabra_maxima(cadena)
17 print(f"Resultado: {palabra} {longitud}")

```

Código 9.28: Solución al Ejercicio 9.28.

EJERCICIO 9.29

Escribe un programa en Python que reciba una cadena y determine la subcadena palindrómica más larga que aparece en la cadena. La subcadena debe ser la más larga que cumpla con la propiedad de ser palíndroma, y el programa debe devolver dicha subcadena y su longitud.

Ejemplo de ejecución esperada

Introduce una cadena: babad

Resultado: bab 3

Solución (girar para leer)

```

1  def es_palindromo(s):
2      # Función para verificar si una subcadena es palíndroma
3      return s[::-1] == s
4
5  def subcadena_palindroma_longitud(cadena):
6      max_subcadena = ""
7      # Recorremos todas las subcadenas posibles
8      for i in range(len(cadena)):
9          for j in range(i + 1, len(cadena) + 1):
10             subcadena = cadena[i:j]
11             if es_palindromo(subcadena) and len(subcadena) > len(max_subcadena):
12                 max_subcadena = subcadena
13
14     return max_subcadena, len(max_subcadena)
15
16 # Solicitar la cadena al usuario
17 cadena = input("Introduce una cadena: ")
18
19 # Evaluar y mostrar el resultado
20 subcadena, longitud = subcadena_palindroma_longitud(cadena)
21
22 print(f"Resultado: {subcadena} {longitud}")

```

Código 9.29: Solución al Ejercicio 9.29.

EJERCICIO 9.30

Escribe un programa en Python que reciba una cadena de texto y determine cuántos pares de subcadenas dentro de la cadena son anagramas. Dos subcadenas son anagramas si tienen las mismas letras, pero en diferente orden.

Ejemplo de ejecución esperada

Introduce una cadena: abba

Resultado: 4

Solución (girar para leer)

```

1  def son_anagramas(sub1, sub2):
2      # Verificar si dos subcadenas son anagramas
3      comparando las frecuencias de cada carácter
4      if len(sub1) != len(sub2):
5          return False
6      # Contamos la frecuencia de caracteres de cada
7      subcadena
8      for char in sub1:
9          if sub1.count(char) != sub2.count(char):
10             return False
11     return True
12
13 def contar_pares_anagramas(cadena):
14     count = 0
15     n = len(cadena)
16     # Recorremos todas las posibles subcadenas
17     for i in range(n):
18         for j in range(i + 1, n + 1):
19             for k in range(i + 1, n + 1):
20                 for l in range(k + 1, n + 1):
21                     # Verificamos si las subcadenas son
22                     anagramas
23                 if son_anagramas(cadena[i:j]),
24                     cadena[k:l]):
25                         count += 1
26     return count
27
28 # Solicitar la cadena al usuario
29 cadena = input("Introduce una cadena: ")
30
31 # Evaluar y mostrar el resultado
32 resultado = contar_pares_anagramas(cadena)
33 print(f"Resultado: {resultado}")

```

Código 9.30: Solución al Ejercicio 9.30.