Introducción avanzada a las expresiones regulares (Regex) en el análisis de datos

Se estima que alrededor del **80% de los datos generados** son **no estructurados**. Estos datos incluyen información que no sigue un formato predefinido, como correos electrónicos, documentos, redes sociales, videos, audios y, en gran medida, **datos de texto**.

El **texto no estructurado** es fundamental en cada proceso clave de negocio, desde el manejo de tickets de soporte hasta la retroalimentación de productos y las interacciones con los clientes. Estos datos no estructurados representan una fuente valiosa de información que las empresas pueden aprovechar para obtener ventajas competitivas.

El **análisis de texto** (Text Analytics o Text Mining) tiene una amplia gama de aplicaciones y casos de uso en diversas industrias, permitiendo a las organizaciones extraer conocimiento útil de grandes volúmenes de texto. Entre las aplicaciones más relevantes se incluyen:

Casos de uso del análisis de texto en los negocios:

- **Entendimiento del cliente**: Analizar las opiniones y comentarios de los clientes para mejorar productos y servicios.
- **Gestión de riesgos**: Identificar señales tempranas de riesgos financieros, operacionales o reputacionales a través del análisis de documentos y comunicaciones.
- **Predicción y prevención del crimen**: Ayudar a las fuerzas del orden a analizar patrones de comportamiento criminal en datos textuales como reportes y registros.
- **Publicidad personalizada**: Analizar datos textuales de redes sociales, búsquedas y correos electrónicos para ofrecer anuncios personalizados.
- **Análisis de sentimientos**: Evaluar la percepción del público hacia una marca, producto o evento a partir de las publicaciones en redes sociales.
- **Gestión de la reputación online**: Monitorear y analizar menciones sobre la marca o empresa en diversas plataformas.
- **Automatización del servicio al cliente**: Implementar chatbots y sistemas de respuesta automática que entienden el lenguaje natural.

En un mundo cada vez más digitalizado, donde los datos no estructurados crecen de manera exponencial, las empresas que puedan aprovechar el análisis de texto estarán mejor posicionadas para tomar decisiones informadas y mejorar sus estrategias de negocio.

Introducción a las Expresiones Regulares (Regex) en el Análisis de Datos

Las **Expresiones Regulares** (Regex) son herramientas fundamentales en el análisis de datos, especialmente cuando se trabaja con grandes volúmenes de datos textuales. Regex permite

buscar, extraer y manipular patrones en cadenas de texto utilizando una sintaxis especializada. Esto las convierte en una de las técnicas más poderosas para el procesamiento de datos no estructurados.

¿Qué es Regex?

Una **Expresión Regular** es una secuencia de caracteres que forma un patrón de búsqueda. Este patrón puede utilizarse para realizar diversas tareas de procesamiento y limpieza de texto, como:

- **Coincidencia de Patrones**: Buscar palabras, frases o estructuras textuales específicas en grandes volúmenes de datos.
- **Validación de Datos**: Verificar que los datos (como correos electrónicos, números de teléfono o códigos postales) cumplan con un formato específico.
- **Extracción de Datos**: Identificar y extraer porciones relevantes de texto de documentos, como nombres, fechas o números.
- **Sustitución de Texto**: Modificar o reemplazar secciones de texto utilizando coincidencias basadas en patrones.

Ventajas de Usar Regex en el Análisis de Datos

- **Eficiencia**: Las Regex permiten procesar grandes cantidades de texto de manera rápida y eficiente.
- **Flexibilidad**: Se pueden crear patrones para tareas específicas, desde búsquedas simples hasta coincidencias complejas en datos no estructurados.
- **Automatización**: Regex es ideal para automatizar tareas repetitivas de limpieza y transformación de datos textuales.

Contexto Histórico de las Expresiones Regulares en el Análisis de Datos

Las expresiones regulares tienen una larga trayectoria en la informática, con raíces en las matemáticas formales. A continuación, se presenta un recorrido histórico por los hitos clave de su desarrollo.

El Origen de las Expresiones Regulares

- **Stephen Kleene y los Lenguajes Regulares**: En 1956, el matemático estadounidense **Stephen Kleene** introdujo el concepto de expresiones regulares en su artículo "*Representation of Events in Nerve Nets and Finite Automata*", parte del libro *Automata Studies*. Kleene formalizó los lenguajes regulares, un subconjunto de los lenguajes formales que hoy se utilizan en teoría de autómatas y análisis computacional.

- **Expansión en Unix**: En los años 70, las expresiones regulares se popularizaron con el sistema operativo **Unix**, donde herramientas como `grep` (Global Regular

Expression Print) y el editor `ed` las utilizaron para búsquedas y edición de texto. Esto facilitó la adopción de Regex en tareas de procesamiento de texto y análisis de datos.

Evolución y Expansión de Regex

- **Lenguajes de Programación**: Con el tiempo, Regex se integró en varios lenguajes de programación, como **Perl**, **Python**, **Java**, **JavaScript** y **PHP**, lo que permitió su uso en una amplia gama de aplicaciones.
- **Herramientas Modernas de Análisis de Datos**: Hoy en día, Regex es parte esencial de bibliotecas y herramientas de análisis de datos, como **Pandas** en Python, que permite usar expresiones regulares para filtrar, limpiar y transformar datos. También se emplea en herramientas como **R** para análisis estadístico y **SQL** para la manipulación de bases de datos.

El Rol de Regex en el Análisis de Datos

En el campo del análisis de datos, las expresiones regulares juegan un papel crucial en el preprocesamiento de datos textuales. Algunas de las aplicaciones más comunes incluyen:

1. **Limpieza de Datos (Data Cleaning)**

- **Eliminación de caracteres no deseados**: Las Regex pueden eliminar caracteres especiales o signos de puntuación en textos crudos que provienen de fuentes como redes sociales, correos electrónicos o páginas web.
- **Estándarización**: Se pueden utilizar para estandarizar formatos de fechas, números de teléfono o direcciones de correo electrónico en bases de datos.

2. **Extracción de Información Específica**

- **Nombres, fechas y números**: Regex puede extraer nombres propios, números de identificación, fechas o direcciones de correos electrónicos dentro de grandes conjuntos de datos.

3. **Análisis de Archivos de Registro (Logs)**

- En grandes sistemas de TI, los archivos de registro o *logs* contienen enormes volúmenes de texto sin estructura. Regex permite identificar patrones de errores, advertencias o actividades sospechosas en estos archivos.

4. **Validación de Formatos**

- **Validación de correos electrónicos**: Un uso común de Regex es verificar que un correo electrónico ingresado por un usuario esté en el formato adecuado.
- **Verificación de números de teléfono**: Regex puede asegurar que un número de teléfono siga un formato específico según la región.

5. **Procesamiento de Lenguaje Natural (NLP)**

- En tareas de **Procesamiento de Lenguaje Natural** (NLP), las expresiones regulares se utilizan para identificar entidades nombradas, palabras clave o patrones lingüísticos.

6. **Raspado Web (Web Scraping)**

- Regex es útil para extraer datos específicos de páginas web, como precios, títulos o descripciones, cuando los datos no están organizados en formato estructurado.

Sintaxis de las Expresiones Regulares

La sintaxis de las expresiones regulares permite definir patrones complejos mediante caracteres especiales. A continuación, se presentan algunos elementos clave:

Caracteres Básicos

- `.`: Coincide con cualquier carácter, excepto un salto de línea.
- `^`: Indica el inicio de una línea o cadena.
- `$`: Indica el final de una línea o cadena.

Cuantificadores

- `*`: Coincide con cero o más repeticiones del carácter anterior.
- `+`: Coincide con una o más repeticiones.
- `?`: Coincide con cero o una ocurrencia del carácter anterior.
- `{n,m}`: Coincide con al menos `n` y como máximo `m` repeticiones.

Conjuntos y Grupos

- `[ ]`: Define un conjunto de caracteres. Por ejemplo, `[A-Za-z]` coincide con cualquier letra mayúscula o minúscula.
- `( )`: Agrupa subexpresiones para tratarlas como una unidad.
- `|`: Define una opción entre varios patrones (operador "OR").

Herramientas y Librerías de Regex

- **Python (`re` module)**: Permite realizar coincidencias de patrones en cadenas de texto de manera eficiente, así como hacer sustituciones y búsquedas avanzadas.
- **Perl**: Lenguaje con soporte avanzado para Regex, históricamente usado para manipular texto de forma eficiente.
- **grep (Unix)**: Utilidad de Unix para buscar expresiones regulares en archivos de texto.
- **Notepad++**: Editor de texto que incluye soporte para búsquedas y sustituciones avanzadas con expresiones regulares.

Recursos y Lecturas Adicionales

Para quienes estén comenzando con Regex o quieran profundizar en el tema, los siguientes recursos pueden ser muy útiles:

- Inicio Rápido de Expresiones Regulares
- RegexOne - Aprende Expresiones Regulares con ejercicios simples e interactivos
- Documentación del Módulo `re` de Python

Dominar Regex permite a los analistas de datos realizar manipulaciones de texto avanzadas, mejorando así sus capacidades para procesar y analizar grandes volúmenes de información.

```
# Importando el módulo de expresiones regulares de la biblioteca estándar de Python
import re

# Cadenas que se buscarán para coincidir con patrones de regex
str1 = "varks Aard pertenecen al Capitín"
str2 = "La famosa ecuación de Albert, E = mc^2."
str3 = "Ubicado en 455 Serra Mall."
str4 = "¡Cuidado con los cambiaformas!"

# Creando una lista de cadenas para probar patrones de regex
test_strings = [str1, str2, str3, str4]

# Iterando a través de cada cadena en la lista test_strings
for test_string in test_strings:
    # Imprimiendo la cadena de prueba para referencia
    print('\nLa cadena de prueba es "' + test_string + '"')

    # Usando re.search() para encontrar la primera ubicación donde el patrón de regex '[í]' coincide
    # '[í]' es un patrón de regex que busca el carácter 'í' en la cadena
    match = re.search('[a-z]', test_string)

    # Comprobando si se encontró una coincidencia
    if match:
        # Imprimiendo el carácter coincidente si se encuentra una coincidencia
        # match.group() devuelve la parte de la cadena donde hay una coincidencia
        print('- La primera coincidencia posible es: ' + match.group())
    else:
        # Indicando que no se encontró ninguna coincidencia si el patrón de regex no coincide con ninguna parte de la cadena
        print('- ** no hay coincidencia. **')
```

```
La cadena de prueba es "varks Aard pertenecen al Capitín"
- La primera coincidencia posible es: v

La cadena de prueba es "La famosa ecuación de Albert, E = mc^2."
- La primera coincidencia posible es: a

La cadena de prueba es "Ubicado en 455 Serra Mall."
- La primera coincidencia posible es: b

La cadena de prueba es "¡Cuidado con los cambiaformas!"
- La primera coincidencia posible es: u
```

Desglose del Código para Buscar Patrones con Expresiones Regulares

A continuación, desglosamos línea por línea un fragmento de código que utiliza **Expresiones Regulares** (Regex) en Python para buscar patrones en cadenas de texto.

1. Iterando Sobre Cada Cadena en la Lista

```
for test_string in test_strings:
```

`test_strings` es una lista que contiene varias cadenas de texto. En este bucle `for`, se itera sobre cada elemento de la lista. Durante cada iteración, `test_string` hace referencia a la cadena actual que está siendo procesada. Este paso es esencial cuando se requiere aplicar el mismo patrón a varias cadenas.

2. Imprimiendo la Cadena de Prueba Actual

```
print('La cadena de prueba es "' + test_string + '"')
```

Esta línea muestra la cadena actual que se está analizando en el bucle. Es útil para realizar un seguimiento de qué cadena se está evaluando en ese momento. Ayuda a verificar que el patrón de búsqueda está funcionando correctamente en cada cadena.

3. Buscando un Patrón en la Cadena

```
match = re.search(r'[A-Z]', test_string)
```

Aquí se utiliza `re.search()` para buscar dentro de la cadena `test_string` el primer lugar donde el patrón `[A-Z]` coincide. Este patrón está diseñado para buscar cualquier letra mayúscula (de la A a la Z). Si se encuentra una coincidencia, la función devuelve un objeto `SRE_Match`; de lo contrario, devuelve `None`.

Explicación del Patrón `[A-Z]`:

- `[A-Z]` representa un rango de caracteres que busca cualquier letra mayúscula en el texto.
- `re.search()` detiene la búsqueda tras encontrar la primera coincidencia, por lo que solo devuelve la primera letra mayúscula encontrada en la cadena.

4. Comprobando si Hay una Coincidencia e Imprimiendo el Resultado

```
f match: print('The first possible match is: ' + match.group()) else: print('no match.')
```

En esta sección, verificamos si `match` tiene un valor (es decir, si no es `None`). Si se ha encontrado una coincidencia, el código imprime el primer carácter coincidente utilizando `match.group()`. El método `group()` devuelve la subcadena de la coincidencia encontrada.

- Si `match` no es `None`, esto indica que se ha encontrado una coincidencia en la cadena, por lo que se imprime la primera coincidencia.
- Si `match` es `None`, significa que no se encontró ninguna coincidencia, y se imprime el mensaje `'no match.'`.

5. Comportamiento de `re.search()`

- **Coincidencia de un Solo Carácter**: Dado que el patrón `[A-Z]` está diseñado para encontrar una sola letra mayúscula, `re.search()` devolverá solo el primer carácter mayúscula que coincida.
- **Solo la Primera Coincidencia**: `re.search()` detiene la búsqueda después de encontrar la primera coincidencia y no continúa buscando más coincidencias en la cadena.

6. Alternativa: Encontrar Todas las Coincidencias

Si el objetivo es encontrar **todas** las coincidencias en lugar de solo la primera, se debería usar `re.findall()` en lugar de `re.search()`.

```
matches = re.findall(r'[A-Z]', test_string)
```

Esta función devuelve una lista que contiene todas las letras mayúsculas encontradas en la cadena `test_string`, no solo la primera coincidencia. Es útil cuando se desea obtener todas las ocurrencias del patrón en lugar de solo una.

Conclusión

Este fragmento de código ejemplifica cómo usar expresiones regulares para buscar patrones específicos en cadenas de texto. La elección entre `re.search()` y `re.findall()` dependerá del caso de uso: si se necesita solo la primera coincidencia o todas las coincidencias en el texto. El uso de Regex en Python proporciona una manera eficiente de analizar y manipular grandes volúmenes de datos textuales.

```python
# Como recapitulación de test_string
test_strings = [
    "varks Aard pertenecen al Capitán",
    "La famosa ecuación de Albert, E = mc^2.",
    "Ubicado en 455 Serra Mall.",
    "¡Cuidado con los cambiaformas!"
]

# Iterando sobre cada cadena en la lista test_strings
for string in test_strings:
    # Imprimiendo la cadena actual
    print(string)

    # Usando re.findall para buscar todas las ocurrencias del patrón de regex en la cadena
    # El patrón '[A-Z]' coincide con cualquier letra mayúscula de la A a la Z
    matches = re.findall(r'[A-Z]', string)

    # Imprimiendo la lista de coincidencias encontradas en la cadena actual
    # Cada coincidencia es una letra mayúscula de la cadena
    print("-", matches, "\n")
```

```
varks Aard pertenecen al Capitán
- ['A', 'C']

La famosa ecuación de Albert, E = mc^2.
- ['L', 'A', 'E']

Ubicado en 455 Serra Mall.
- ['U', 'S', 'M']

¡Cuidado con los cambiaformas!
- ['C']
```

Compilación de Expresiones Regulares en Python

Cuando trabajas con expresiones regulares en Python, tienes la opción de compilar tus patrones de regex en objetos de patrón. Este enfoque no solo puede mejorar el rendimiento, sino que también facilita la gestión de múltiples patrones en tu código. La precompilación convierte tu patrón de regex en un objeto `SRE_Pattern`, que luego se puede utilizar para realizar operaciones de coincidencia, búsqueda y otras tareas relacionadas.

¿Por qué Precompilar Regex?

1. Rendimiento

- **Eficiencia**: Compilar un patrón una vez y usarlo varias veces es más eficiente que interpretar el mismo patrón repetidamente en cada operación. Esto es especialmente útil en bucles donde el mismo patrón se aplica a múltiples cadenas.

2. Organización

- **Mantenimiento del Código**: Si trabajas con múltiples patrones de regex, compilarlos en objetos puede ayudar a mantener tu código organizado y modular. Esto es particularmente beneficioso en proyectos grandes donde la claridad y la organización son esenciales.

3. Reutilización

- **Uso Múltiple**: Una vez que has compilado un patrón, puedes utilizar el mismo objeto de patrón en múltiples operaciones de coincidencia y búsqueda sin necesidad de recompilarlo. Esto reduce la carga de procesamiento y mejora la eficiencia del código.

4. Legibilidad

- **Nombres Descriptivos**: Al compilar patrones en objetos, puedes asignarles nombres descriptivos que reflejen su propósito, lo que aumenta la legibilidad del código. Esto ayuda a otros desarrolladores (o a ti mismo en el futuro) a comprender rápidamente la intención detrás de cada patrón.

Consideraciones Finales

La precompilación de expresiones regulares en Python es una práctica recomendada, especialmente cuando se trabaja con patrones complejos o se realizan múltiples operaciones sobre las mismas cadenas. Al aprovechar `re.compile()`, no solo mejorarás el rendimiento de tu código, sino que también lo harás más fácil de leer y mantener. Considera utilizar este enfoque en tus proyectos para optimizar el manejo de expresiones regulares.

Cómo Precompilar Patrones de Regex

Aquí hay un ejemplo de cómo precompilar patrones de regex en Python:

```
import re

# Precompilar el patrón para coincidir con cualquier letra mayúscula
pattern_uppercase = re.compile(r'[A-Z]')

# Ahora puedes usar pattern_uppercase para buscar dentro de cadenas
match = pattern_uppercase.search('Hello World')
if match:
    print('Letra mayúscula encontrada:', match.group())
```

En este código, `re.compile` se usa para compilar el patrón de regex `[A-Z]` que coincide con cualquier letra mayúscula. El objeto resultante `pattern_uppercase` se puede usar para buscar a través de cadenas sin tener que recompilar el patrón cada vez, lo que lleva a una ejecución más eficiente, particularmente cuando se trata de grandes cantidades de texto o muchas búsquedas.

Incluso puedes almacenar múltiples patrones compilados en una lista y iterar sobre ellos, como se muestra en el siguiente ejemplo:

```
import re

# Precompilar el patrón para coincidir con cualquier letra mayúscula
pattern_uppercase = re.compile(r'[A-Z]')

# Ahora puedes usar pattern_uppercase para buscar dentro de cadenas
match = pattern_uppercase.search('Hello World')
if match:
    print('Letra mayúscula encontrada:', match.group())
```

```
Letra mayúscula encontrada: H
```

```
# Importar el módulo de expresiones regulares
import re

# Definir una lista de patrones de regex
patterns = [
    '[ABC]',               # Coincide con cualquiera de 'A', 'B' o
'C'
    '[^ABC]',              # Coincide con cualquier carácter excepto
'A', 'B' o 'C'
    '[ABC^]',              # Coincide con 'A', 'B', 'C' o '^'
    '[0-9]',               # Coincide con cualquier dígito único del
'0' al '9'
    '[0-4]',               # Coincide con cualquier dígito único del
'0' al '4'
    '[A-Z]',               # Coincide con cualquier letra mayúscula
de la 'A' a la 'Z'
    '[a-z]',               # Coincide con cualquier letra minúscula
de la 'a' a la 'z'
    '[A-Za-z]',            # Coincide con cualquier letra sin
importar el caso
    '[A-Za-z0-9]',         # Coincide con cualquier carácter
alfanumérico
    '[-a-z]',              # Coincide con '-' o cualquier letra
minúscula
    '[- a-z]'              # Coincide con '-', espacio o cualquier
letra minúscula
]

# Compilar los patrones para crear objetos SRE_Pattern para una
```

```
coincidencia eficiente
compiled_patterns = [re.compile(p) for p in patterns]

# Función para encontrar la primera coincidencia de un patrón en una
cadena dada
def find_match(compiled_pattern, string):
    match = compiled_pattern.search(string)  # Realizar la búsqueda
usando el patrón compilado
    return match.group() if match else 'sin coincidencias.'  #
Devolver el texto coincidente o 'sin coincidencias.'

# Lista de cadenas de prueba para coincidir
test_strings = [
    "ABC fácil como 123",
    "Simple como do re mi",
    "ABC, 123, nena, tú y yo chica"
]

# Iterar sobre cada cadena en la lista test_strings
for test_string in test_strings:
    # Imprimir la cadena de prueba para claridad
    print(f"En: \"{test_string}\"")
    # Encontrar e imprimir la primera coincidencia para cada patrón
compilado
    for compiled_pattern in compiled_patterns:
        # Recuperar la representación de cadena del patrón para la
salida
        pattern_text = compiled_pattern.pattern
        # Encontrar la primera coincidencia para el patrón en la
cadena de prueba actual
        match_text = find_match(compiled_pattern, test_string)
        # Imprimir el patrón y su primera coincidencia (o 'sin
coincidencias.')
        print(f' - La primera coincidencia potencial para
"{pattern_text}" \t es: {match_text}')
    # Imprimir una nueva línea para una mejor separación de la salida
en la consola
    print()
```

```
En: "ABC fácil como 123"
 - La primera coincidencia potencial para "[ABC]"       es: A
 - La primera coincidencia potencial para "[^ABC]"      es:
 - La primera coincidencia potencial para "[ABC^]"      es: A
 - La primera coincidencia potencial para "[0-9]"       es: 1
 - La primera coincidencia potencial para "[0-4]"       es: 1
 - La primera coincidencia potencial para "[A-Z]"       es: A
 - La primera coincidencia potencial para "[a-z]"       es: f
 - La primera coincidencia potencial para "[A-Za-z]"    es: A
 - La primera coincidencia potencial para "[A-Za-z0-9]"      es: A
 - La primera coincidencia potencial para "[-a-z]"      es: f
```

```
 - La primera coincidencia potencial para "[- a-z]"    es:

En: "Simple como do re mi"
 - La primera coincidencia potencial para "[ABC]"      es: sin
coincidencias.
 - La primera coincidencia potencial para "[^ABC]"     es: S
 - La primera coincidencia potencial para "[ABC^]"     es: sin
coincidencias.
 - La primera coincidencia potencial para "[0-9]"      es: sin
coincidencias.
 - La primera coincidencia potencial para "[0-4]"      es: sin
coincidencias.
 - La primera coincidencia potencial para "[A-Z]"      es: S
 - La primera coincidencia potencial para "[a-z]"      es: i
 - La primera coincidencia potencial para "[A-Za-z]"   es: S
 - La primera coincidencia potencial para "[A-Za-z0-9]"      es: S
 - La primera coincidencia potencial para "[-a-z]"     es: i
 - La primera coincidencia potencial para "[- a-z]"    es: i

En: "ABC, 123, nena, tú y yo chica"
 - La primera coincidencia potencial para "[ABC]"      es: A
 - La primera coincidencia potencial para "[^ABC]"     es: ,
 - La primera coincidencia potencial para "[ABC^]"     es: A
 - La primera coincidencia potencial para "[0-9]"      es: 1
 - La primera coincidencia potencial para "[0-4]"      es: 1
 - La primera coincidencia potencial para "[A-Z]"      es: A
 - La primera coincidencia potencial para "[a-z]"      es: n
 - La primera coincidencia potencial para "[A-Za-z]"   es: A
 - La primera coincidencia potencial para "[A-Za-z0-9]"      es: A
 - La primera coincidencia potencial para "[-a-z]"     es: n
 - La primera coincidencia potencial para "[- a-z]"    es:
```

Este script de Python demuestra cómo usar expresiones regulares (regex) para encontrar patrones dentro de cadenas. Los comentarios en el código ayudarán a explicar cada paso del proceso.

Definiendo Patrones de Regex

Al trabajar con expresiones regulares, es fundamental definir patrones que especifican las reglas para las coincidencias. Estos patrones son expresiones que pueden identificar conjuntos específicos de caracteres. A continuación, se presentan algunos ejemplos de patrones de regex y sus descripciones:

- **`[ABC]`**: Coincide con cualquiera de los caracteres 'A', 'B' o 'C'. Este patrón es útil cuando se desea encontrar cualquiera de un conjunto limitado de opciones.

- **`[^ABC]`**: Coincide con cualquier carácter excepto 'A', 'B' o 'C'. El símbolo `^` al principio del conjunto indica una negación, lo que permite filtrar caracteres no deseados.
- **`[ABC^]`**: Coincide con 'A', 'B', 'C' o el símbolo '^'. Este patrón incluye caracteres específicos junto con un carácter especial.
- **`[0-9]`**: Coincide con cualquier dígito único del '0' al '9'. Este patrón es esencial para identificar números en una cadena.
- **`[0-4]`**: Coincide con cualquier dígito único del '0' al '4'. Este es un subconjunto específico de los dígitos.
- **`[A-Z]`**: Coincide con cualquier letra mayúscula de la 'A' a la 'Z'. Este patrón es útil para identificar texto en mayúsculas.
- **`[a-z]`**: Coincide con cualquier letra minúscula de la 'a' a la 'z'. Similar al anterior, pero enfocado en letras minúsculas.
- **`[A-Za-z]`**: Coincide con cualquier letra sin importar el caso, abarcando tanto letras mayúsculas como minúsculas. Esto es útil cuando se desea una coincidencia más amplia.
- **`[A-Za-z0-9]`**: Coincide con cualquier carácter alfanumérico, es decir, letras y números. Este patrón es común en validaciones de entrada.
- **`[-a-z]`**: Coincide con el carácter '-' o cualquier letra minúscula. El uso de un guion al principio del conjunto permite incluir el guion como una opción válida.
- **`[- a-z]`**: Coincide con '-', un espacio o cualquier letra minúscula. Al incluir un espacio en el conjunto, este patrón permite más flexibilidad en la coincidencia.

Ejemplo de Lista de Patrones

Para utilizar estos patrones, puedes definir una lista en tu código, que podría verse de la siguiente manera:

```
patterns = [ "[ABC]", "[^ABC]", "[ABC^]", "[0-9]", "[0-4]", "[A-Z]",
"[a-z]", "[A-Za-z]", "[A-Za-z0-9]", "[-a-z]", "[- a-z]" ]
```

Compilando los Patrones

Cada patrón en la lista se compila para una coincidencia eficiente. Esto es particularmente útil cuando un patrón se usa varias veces.

```
compiled_patterns = [re.compile(p) for p in patterns]
```

Definiendo la Función `find_match`

La función `find_match` busca la primera coincidencia de un patrón compilado dentro de una cadena dada. Si se encuentra una coincidencia, devuelve el texto coincidente; de lo contrario, devuelve 'sin coincidencias.'

```
def find_match(compiled_pattern, string):     ...
```

Cadenas de Prueba

Se define una lista de cadenas de prueba, que se buscarán para encontrar coincidencias contra los patrones.

```
test_strings = [...]
```

Realizando la Coincidencia de Patrones

El script itera sobre cada cadena en `test_strings`, y para cada cadena, aplica todos los patrones de regex compilados. Para cada patrón, imprime la primera coincidencia encontrada o 'sin coincidencias.' si no se encuentra ninguna.

```
for test_string in test_strings:     print(f"En: \"{test_string}\"")
for compiled_pattern in compiled_patterns:         ...     print()
```

Trabajando con Listas de Patrones Compilados

Incluso puedes almacenar múltiples patrones compilados en una lista y iterar sobre ellos, como se muestra en el siguiente ejemplo:

```
print(patterns[1]) print(patterns[1].pattern)
```

En esta sección, demostramos cómo acceder y usar patrones compilados individuales de la lista. La expresión `patterns[1]` hace referencia al segundo patrón compilado en la lista. Al imprimir `patterns[1].pattern`, podemos ver el patrón de regex actual como una cadena.

Este proceso ayuda a identificar qué parte de la cadena coincide con los patrones dados y es una manera práctica de aprender y entender las aplicaciones de regex en Python.

```python
import re  # Importando el módulo de expresiones regulares

# Definiendo la cadena que vamos a comprobar
needle = 'needlers'

# Enfoque de Python: Usando comprensión de listas y any()
# Esta línea verifica si la cadena 'needle' termina con alguno de los sufijos especificados ('ly', 'ed', 'ing', 'ers')
# any() devuelve True si al menos una de las condiciones es True
print(any([needle.endswith(e) for e in ('ly', 'ed', 'ing', 'ers')]))

# Expresión regular al vuelo en Python
```

```
# Esto utiliza expresiones regulares para comprobar si 'needle'
termina con los sufijos especificados
# La función search() busca en 'needle' cualquier coincidencia con el
patrón de expresión regular
# bool() se utiliza para convertir el resultado en True o False
print(bool(re.search(r'(ly|ed|ing|ers)$', needle)))

# Expresión regular compilada en Python
# Compilando el patrón de expresión regular para un reuso más rápido
# Esto es más eficiente si el patrón se usa varias veces
comp = re.compile(r'(ly|ed|ing|ers)$')
print(bool(comp.search(needle)))

# Los comandos %timeit se usan en Jupyter Notebooks para medir el
tiempo de ejecución de pequeños fragmentos de código
# -n 1000 especifica que el comando se ejecutará 1000 veces en cada
bucle
# -r 50 indica que habrá 50 de estos bucles
# Esto se utiliza para obtener una medida más precisa del tiempo de
ejecución promediando múltiples ejecuciones

# %timeit para el enfoque de Python
%timeit -n 1000 -r 50 bool(any([needle.endswith(e) for e in ('ly',
'ed', 'ing', 'ers')]))

# %timeit para la expresión regular al vuelo
%timeit -n 1000 -r 50 bool(re.search(r'(ly|ed|ing|ers)$', needle))

# %timeit para la expresión regular compilada
%timeit -n 1000 -r 50 bool(comp.search(needle))
```

```
True
True
True
1.04 µs ± 316 ns per loop (mean ± std. dev. of 50 runs, 1,000 loops
each)
1.38 µs ± 483 ns per loop (mean ± std. dev. of 50 runs, 1,000 loops
each)
The slowest run took 4.68 times longer than the fastest. This could
mean that an intermediate result is being cached.
831 ns ± 425 ns per loop (mean ± std. dev. of 50 runs, 1,000 loops
each)
```

Resumen de Términos para Expresiones Regulares

Terminología de Expresiones Regulares

Las expresiones regulares (regex o regexp) son herramientas poderosas para el emparejamiento de patrones y la manipulación de texto. A continuación, se presenta una explicación detallada de algunos términos y símbolos comúnmente utilizados en las expresiones regulares:

- **[ ] (Conjunto de Caracteres)**: Los corchetes denotan un conjunto de caracteres, donde uno de los elementos internos debe coincidir. Por ejemplo, `[abc]` coincidiría con cualquiera de los caracteres 'a', 'b' o 'c'. También puedes especificar rangos como `[0-9]` para coincidir con cualquier dígito.
- **| (Tubo o Alternancia)**: El símbolo de tubo representa una opción entre elementos, actuando como un operador "o". Por ejemplo, `a|b` coincidiría con 'a' o 'b'.
- **{ } (Cuantificador de Intervalo)**: Las llaves se utilizan para especificar un intervalo o la cantidad de veces que un patrón debe repetirse. Por ejemplo, `a{2,4}` coincidiría con 'aa', 'aaa' o 'aaaa', donde el número de 'a's se encuentra entre 2 y 4. También puedes usar un número único, como `{3}`, que coincidiría exactamente con tres repeticiones del patrón anterior.
- **(Barra Inversa)**: La barra inversa es un carácter de escape que identifica al siguiente carácter como un literal, evitando que se interprete como un símbolo especial. Por ejemplo, `\.` coincidiría con un punto (.) en lugar de coincidir con cualquier carácter.
- **. (Punto)**: En el modo predeterminado, el punto coincide con cualquier carácter, excepto un salto de línea. Por ejemplo, `a.b` coincidiría con 'axb', 'a#b' o 'a$b', donde 'x', '#', y '$' pueden ser cualquier carácter excepto un salto de línea.
- **^ (Acento Circunflejo)**: Este símbolo coincide con el inicio de la cadena. En el modo MULTILINE, también coincide inmediatamente después de cada salto de línea. Por ejemplo, `^abc` coincidiría con 'abc' al comienzo de una línea. Esto es útil para validar el inicio de un texto.
- **$ (Signo de Dólar)**: El signo de dólar coincide con el final de la cadena o justo antes del salto de línea al final de la cadena. En el modo MULTILINE, también coincide antes de un salto de línea. Por ejemplo, `xyz$` coincidiría con 'xyz' al final de una línea, permitiendo validar si una cadena termina con un patrón específico.
- * **(Asterisco)**: El asterisco hace que el patrón coincidente permita 0 o más repeticiones del patrón anterior. Por ejemplo, `ab*` coincidiría con 'a' o 'ab' seguido de cualquier número de 'b's, incluyendo ninguno.
- **+ (Signo de Más)**: El signo de más indica que debe haber 1 o más repeticiones del patrón anterior. Por ejemplo, `ab+` coincidiría con 'a' seguido de al menos una 'b', pero no coincidiría solo con 'a'.
- **? (Signo de Interrogación)**: Este signo permite que el patrón coincida con 0 o 1 repeticiones del patrón anterior. Por ejemplo, `ab?` coincidiría con 'a' o 'ab', lo que permite la flexibilidad en la búsqueda.
- **{n} (Repetición Exacta)**: Las llaves también se pueden usar para especificar un número exacto de repeticiones. Por ejemplo, `a{3}` coincidiría solo con 'aaa'.

Clases de Caracteres Adicionales

- **`\d` (Dígito)**: Coincide con cualquier dígito decimal; es equivalente a la clase de caracteres `[0-9]`.
- **`\D` (No Dígito)**: Coincide con cualquier carácter que no sea un dígito; es equivalente a la clase de caracteres `[^0-9]`.
- **`\s` (Espacio en Blanco)**: Coincide con cualquier carácter de espacio en blanco, incluyendo espacio, tabulador, salto de línea, retorno de carro, avance de formulario y tabulador vertical. Esto es equivalente a la clase de caracteres `[ \t\n\r\f\v]`.
- **`\S` (No Espacio en Blanco)**: Coincide con cualquier carácter que no sea de espacio en blanco. Esto es equivalente a la clase de caracteres `[^ \t\n\r\f\v]`.
- **`\w` (Carácter de Palabra)**: Coincide con cualquier carácter alfanumérico o guion bajo; esto es equivalente a la clase de caracteres `[a-zA-Z0-9_]`.
- **`\W` (Carácter No Palabra)**: Coincide con cualquier carácter que no sea alfanumérico o guion bajo; esto es equivalente a la clase de caracteres `[^a-zA-Z0-9_]`.
- **`\b` (Límite de Palabra)**: Coincide con una posición donde un carácter de palabra es seguido o precedido por un carácter que no es de palabra. Por ejemplo, `\bword\b` coincidiría con la palabra "word" en "word" pero no en "sword".

Las expresiones regulares pueden ser complejas, pero son increíblemente poderosas para tareas de procesamiento de texto. Para una documentación más comprensiva y completa, consulta la Documentación de Expresiones Regulares de Python.

Ejemplos de Expresiones Regulares

A continuación, exploraremos diez patrones de expresiones regulares y sus explicaciones:

- **Ejemplo 1 - Coincidir Dígitos (`\d`)**:
  - **Patrón**: `\d+`
  - **Explicación**: Este patrón coincidirá con uno o más dígitos (0-9) en una cadena. Por ejemplo, coincidirá con '123' en la cadena 'abc123xyz'.
- **Ejemplo 2 - Coincidir Direcciones de Correo Electrónico**:
  - **Patrón**: `[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}`
  - **Explicación**: Este patrón coincide con una dirección de correo electrónico básica. Busca una secuencia de caracteres alfanuméricos seguida de '@', luego otra secuencia, un punto y un dominio con al menos dos letras. Por ejemplo, coincide con 'usuario@example.com'.
- **Ejemplo 3 - Coincidir Fechas (dd/mm/yyyy)**:
  - **Patrón**: `\d{2}/\d{2}/\d{4}`
  - **Explicación**: Este patrón coincide con una fecha en el formato 'dd/mm/yyyy', donde 'dd' representa el día, 'mm' el mes y 'yyyy' el año. Por ejemplo, coincide con '25/12/2022'.

- **Ejemplo 4 - Extraer URLs de Texto**:
  - **Patrón**: `https?://\S+`
  - **Explicación**: Este patrón busca URLs que comienzan con 'http://' o 'https://'. El `s?` indica que la 's' es opcional, seguido de `://` y uno o más caracteres que no son espacios en blanco. Por ejemplo, coincide con 'https://www.ejemplo.com'.
- **Ejemplo 5 - Coincidir Números de Teléfono (Formato Nacional)**:
  - **Patrón**: `\(\d{3}\) \d{3}-\d{4}`
  - **Explicación**: Este patrón coincide con un número de teléfono en el formato (###) ###-####. Los paréntesis indican que los tres primeros dígitos están entre paréntesis, seguido de un espacio, tres dígitos, un guion y cuatro dígitos. Por ejemplo, coincide con '(123) 456-7890'.
- **Ejemplo 6 - Coincidir Códigos Postales**:
  - **Patrón**: `\d{5}(-\d{4})?`
  - **Explicación**: Este patrón coincide con un código postal de cinco dígitos y opcionalmente puede incluir un guion seguido de cuatro dígitos adicionales. Por ejemplo, coincide con '12345' o '12345-6789'.
- **Ejemplo 7 - Coincidir Palabras que Comienzan con una Letra Específica**:
  - **Patrón**: `\b[Aa]\w*`
  - **Explicación**: Este patrón coincide con palabras que comienzan con 'a' o 'A'. El `\b` indica un límite de palabra, y `\w*` coincide con cero o más caracteres alfanuméricos que siguen. Por ejemplo, coincide con 'apple' y 'Aardvark'.
- **Ejemplo 8 - Coincidir Palabras que Terminan con un Sufijo Específico**:
  - **Patrón**: `\w+ing\b`
  - **Explicación**: Este patrón coincide con palabras que terminan en 'ing'. El `\w+` coincide con uno o más caracteres alfanuméricos y el `\b` asegura que 'ing' esté al final de la palabra. Por ejemplo, coincide con 'running' y 'jumping'.
- **Ejemplo 9 - Coincidir Frases con Comillas**:
  - **Patrón**: `"(.*?)"`
  - **Explicación**: Este patrón busca cadenas de texto entre comillas dobles. El `.*?` captura cualquier carácter de manera no codiciosa (menos voraz), asegurando que se detenga en la primera comilla de cierre. Por ejemplo, coincide con '"Hola, mundo"'.
- **Ejemplo 10 - Coincidir Caracteres Especiales**:
  - **Patrón**: `[!@#$%^&*()]`
  - **Explicación**: Este patrón coincide con cualquier carácter especial que se especifique entre los corchetes. Por ejemplo, coincide con '$' en 'precio: $100'.

Las expresiones regulares son herramientas extremadamente versátiles para la validación y búsqueda de patrones en texto. Con la práctica, se pueden utilizar para resolver una amplia variedad de problemas relacionados con el procesamiento de texto.

Ejercicios de Expresiones Regulares

Ejercicio 1 - Coincidir Direcciones de Correo Electrónico

Desarrolla un script que busque y extraiga todas las direcciones de correo electrónico de un texto dado. Las direcciones deben seguir el formato estándar que incluye caracteres alfanuméricos, puntos, guiones bajos, porcentajes, signos más y guiones, seguido de un dominio.

Ejercicio 2 - Extraer Fechas (dd/mm/yyyy)

Crea un script que encuentre todas las fechas en el formato 'dd/mm/yyyy' dentro de un texto. Las fechas deben estar compuestas por dos dígitos para el día, dos dígitos para el mes y cuatro dígitos para el año, separadas por barras.

Ejercicio 3 - Encontrar URLs

Implementa un script que busque y extraiga todas las URLs de un texto. Las URLs pueden comenzar con 'http://' o 'https://' y deben seguir con cualquier carácter que no sea un espacio.

Ejercicio 4 - Extraer Códigos Postales

Desarrolla un script que encuentre todos los códigos postales en el formato '#####'. Los códigos postales deben estar compuestos por cinco dígitos.

Ejercicio 5 - Coincidir Palabras con Guiones

Crea un script que busque y extraiga todas las palabras que contengan un guion en un texto. Las palabras deben estar formadas por caracteres alfanuméricos, separados por un guion.

Ejercicio 6 - Extraer Fechas (mm/dd/yyyy)

Desarrolla un script que encuentre todas las fechas en el formato 'mm/dd/yyyy' dentro de un texto. Las fechas deben estar compuestas por dos dígitos para el mes, dos dígitos para el día y cuatro dígitos para el año, separadas por barras.

Ejercicio 7 - Extraer Hashtags

Implementa un script que busque y extraiga todos los hashtags de un texto. Los hashtags deben comenzar con un símbolo de número (#) seguido de uno o más caracteres alfanuméricos.

Ejercicio 8 - Extraer Usuarios Mencionados

Desarrolla un script que encuentre y extraiga todos los nombres de usuario mencionados en un texto. Los nombres de usuario deben comenzar con un símbolo de arroba (@) seguido de uno o más caracteres alfanuméricos.

Ejercicio 9 - Coincidir Números de Teléfono

Crea un script que busque y extraiga todos los números de teléfono que pueden incluir un código de país en el formato '(+##) ###-###-###'. Los números deben consistir en un código de país de dos dígitos entre paréntesis, seguido de un espacio, tres dígitos, un guion, tres dígitos, un guion y tres dígitos.

Ejercicio 10 - Coincidir Palabras Alfanuméricas

Diseña un script que busque y extraiga todas las palabras que contengan solo caracteres alfanuméricos, es decir, letras y números, sin caracteres especiales ni espacios.

```
import re

# Textos de ejemplo
texto1 = "Puedes contactar a juan.perez@example.com,
maria@empresa.org, contacto123@dominio.com y ejemplo@mail.com."
texto2 = "Las fechas importantes son 25/12/2022, 01/01/2023,
15/08/2023 y 31/12/2023."
texto3 = "Visita http://www.ejemplo.com, https://mi-sitio.org y
http://otro-sitio.net para más información."
texto4 = "Los códigos postales son 28001, 08002, 41003 y 46004."
texto5 = "Las palabras con guiones son bienvenidas - ejemplo-palabra,
prueba-texto, ejemplo-otro y palabra-por-favor."
texto6 = "Fechas en formato mm/dd/yyyy son 12/25/2022, 02/14/2023,
07/04/2023 y 10/31/2023."
texto7 = "Mira #hashtag, #regex, #Python y otros ejemplos de
#programación."
texto8 = "Sigue a @usuario1, @usuario2, @miusuario y @ejemplo en
Twitter."
texto9 = "Puedes marcar los números (+34) 612-345-678, (+34) 678-123-
456 y (+34) 911-234-567 para consultas."
texto10 = "Las palabras alfanuméricas son: Python3, Regex, HTML5 y
JavaScript."

# Ejercicio 1 - Coincidir Direcciones de Correo Electrónico
def extraer_correos(texto):
    patron1 = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
    return re.findall(patron1, texto)

print("Ejercicio 1 - Correos Electrónicos:", extraer_correos(texto1))
```

```
Ejercicio 1 - Correos Electrónicos: ['juan.perez@example.com',
'maria@empresa.org', 'contacto123@dominio.com', 'ejemplo@mail.com']
```

Explicación del Patrón para Coincidir Direcciones de Correo Electrónico

El patrón utilizado para extraer direcciones de correo electrónico es el siguiente: `\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b`. Este patrón se descompone de la siguiente manera:

- **`\b`**: Asegura que la coincidencia ocurra al inicio de una palabra, evitando que haya caracteres alfanuméricos antes del correo electrónico.
- **`[A-Za-z0-9._%+-]+`**: Coincide con la parte local del correo electrónico (antes del `@`). Permite:
  - Letras mayúsculas (A-Z) y minúsculas (a-z)
  - Números (0-9)
  - Caracteres especiales: `.`, `_`, `%`, `+`, y `-`
  - El `+` indica que debe haber al menos un carácter de este tipo.
- **`@`**: Este símbolo es obligatorio y separa la parte local del dominio.
- **`[A-Za-z0-9.-]+`**: Coincide con el dominio del correo electrónico (la parte después del `@` y antes del punto). Permite:
  - Letras (A-Z, a-z)
  - Números (0-9)
  - Caracteres especiales: `.` y `-`
  - El `+` indica que debe haber al menos un carácter.
- **`\.`**: Coincide con el punto que precede a la extensión del dominio. El `\` se utiliza para escapar el punto, ya que en expresiones regulares un punto solo significa "cualquier carácter".
- **`[A-Z|a-z]{2,}`**: Coincide con la extensión del dominio, que debe tener al menos dos caracteres. Permite:
  - Letras mayúsculas (A-Z) y minúsculas (a-z).
  - El `|` dentro de los corchetes es innecesario y puede omitirse, ya que no se usa como un operador de "o" en este contexto.
- **`\b`**: Asegura que la coincidencia ocurra al final de una palabra, evitando que haya caracteres alfanuméricos después del correo electrónico.

Este patrón es común para validar direcciones de correo electrónico simples y es útil para asegurarse de que las direcciones extraídas tengan un formato correcto.

```
# Ejercicio 2 - Extraer Fechas (dd/mm/yyyy)
def extraer_fechas_ddmmyyyy(texto):
    patron2 = r'\b(?:0[1-9]|[12][0-9]|3[01])/(?:0[1-9]|1[0-2])/\d{4}\
b'
    return re.findall(patron2, texto)
```

```
print("Ejercicio 2 - Fechas (dd/mm/yyyy):",
extraer_fechas_ddmmyyyy(texto2))
```

```
Ejercicio 2 - Fechas (dd/mm/yyyy): ['25/12/2022', '01/01/2023',
'15/08/2023', '31/12/2023']
```

Explicación del Patrón para Extraer Fechas (dd/mm/yyyy)

El patrón utilizado para extraer fechas en formato `dd/mm/yyyy` es el siguiente: `\b(?:0[1-9]|[12][0-9]|3[01])/(?:0[1-9]|1[0-2])/\d{4}\b`. Este patrón se descompone de la siguiente manera:

- **`\b`**: Asegura que la coincidencia ocurra al inicio de una palabra, evitando que haya caracteres alfanuméricos antes de la fecha.
- **`(?:0[1-9]|[12][0-9]|3[01])`**: Coincide con el día de la fecha. Permite:
  - `0[1-9]`: Días del `01` al `09`.
  - `[12][0-9]`: Días del `10` al `29`.
  - `3[01]`: Días `30` y `31`.
  - `(?:...)` es una agrupación no capturante, lo que significa que se agrupan los patrones sin capturarlos como grupos separados.
- **`/`**: Este símbolo es un separador que indica la división entre el día y el mes.
- **`(?:0[1-9]|1[0-2])`**: Coincide con el mes de la fecha. Permite:
  - `0[1-9]`: Meses del `01` al `09`.
  - `1[0-2]`: Meses `10`, `11` y `12`.
- **`/`**: Otro separador que indica la división entre el mes y el año.
- **`\d{4}`**: Coincide con el año, que debe consistir en exactamente 4 dígitos. El `\d` representa cualquier dígito, y `{4}` indica que debe haber exactamente 4 de estos.
- **`\b`**: Asegura que la coincidencia ocurra al final de una palabra, evitando que haya caracteres alfanuméricos después de la fecha.

Este patrón es efectivo para validar y extraer fechas en el formato `dd/mm/yyyy` y asegura que las fechas extraídas tengan un formato correcto.

```
# Ejercicio 3 - Encontrar URLs
def extraer_urls(texto):
    patron3 = r'https?://\S+'
    return re.findall(patron3, texto)

print("Ejercicio 3 - URLs:", extraer_urls(texto3))
```

```
Ejercicio 3 - URLs: ['http://www.ejemplo.com,', 'https://mi-
sitio.org', 'http://otro-sitio.net']
```

Explicación del Patrón para Encontrar URLs

El patrón utilizado para extraer URLs es el siguiente: `https?://\S+`. Este patrón se descompone de la siguiente manera:

- **`https?`**: Coincide con el esquema de la URL. Permite:
  - `http`: Coincide con "http" literal.
  - `s?`: La `s` es opcional, lo que significa que también puede coincidir con "https".
- **`://`**: Este segmento coincide con los dos puntos seguidos de dos barras diagonales, que son parte de la estructura estándar de una URL.
- **`\S+`**: Coincide con uno o más caracteres que no son espacios. El `\S` representa cualquier carácter que no sea un espacio en blanco (incluyendo letras, números, símbolos, etc.), y el `+` indica que debe haber al menos uno de estos caracteres. Esto asegura que se capture toda la parte de la URL que sigue al esquema y al `://`, hasta que se encuentre un espacio.

Este patrón es útil para extraer URLs en formatos HTTP y HTTPS de un texto, y garantiza que se capturen las direcciones web de manera eficiente.

```
# Ejercicio 4 - Extraer Códigos Postales
def extraer_codigos_postales(texto):
    patron4 = r'\b\d{5}\b'
    return re.findall(patron4, texto)

print("Ejercicio 4 - Códigos Postales:",
extraer_codigos_postales(texto4))
```

```
Ejercicio 4 - Códigos Postales: ['28001', '08002', '41003', '46004']
```

Explicación del Patrón para Extraer Códigos Postales

El patrón utilizado para extraer códigos postales es el siguiente: `\b\d{5}\b`. Este patrón se descompone de la siguiente manera:

- **`\b`**: Asegura que la coincidencia ocurra al inicio de una palabra, evitando que haya caracteres alfanuméricos antes del código postal. Esto garantiza que solo se capturen códigos postales completos y no parte de números más largos.
- **`\d{5}`**: Coincide con exactamente cinco dígitos.
  - `\d` representa cualquier dígito (equivalente a `[0-9]`), y `{5}` indica que debe haber exactamente cinco de estos dígitos. Esto es típico en muchos países, como Estados Unidos, donde los códigos postales consisten en cinco números.
- **`\b`**: Asegura que la coincidencia ocurra al final de una palabra, evitando que haya caracteres alfanuméricos después del código postal. Esto también contribuye a que solo se extraigan códigos postales completos.

Este patrón es efectivo para validar y extraer códigos postales que constan de cinco dígitos en un texto, garantizando que las coincidencias sean correctas y completas.

```
# Ejercicio 5 - Coincidir Palabras con Guiones
def extraer_palabras_con_guion(texto):
    patron5 = r'\b\w+-\w+'
    return re.findall(patron5, texto)

print("Ejercicio 5 - Palabras con Guiones:",
extraer_palabras_con_guion(texto5))
```

```
Ejercicio 5 - Palabras con Guiones: ['ejemplo-palabra', 'prueba-
texto', 'ejemplo-otro', 'palabra-por']
```

Explicación del Patrón para Coincidir Palabras con Guiones

El patrón utilizado para extraer palabras que contienen guiones es el siguiente: `\b\w+-\w+`. Este patrón se descompone de la siguiente manera:

- **`\b`**: Asegura que la coincidencia ocurra al inicio de una palabra, evitando que haya caracteres alfanuméricos antes de la palabra con guion. Esto garantiza que se capturen palabras completas y no partes de palabras más largas.
- **`\w+`**: Coincide con uno o más caracteres alfanuméricos.
  - `\w` representa cualquier carácter alfanumérico (letras mayúsculas, letras minúsculas y números, así como el carácter de subrayado _), y el `+` indica que debe haber al menos un carácter. Esto captura la primera parte de la palabra antes del guion.
- **`-`**: Coincide con el guion literal que se encuentra entre las dos partes de la palabra. Este carácter debe estar presente para que la coincidencia sea válida.
- **`\w+`**: Nuevamente coincide con uno o más caracteres alfanuméricos. Esto captura la segunda parte de la palabra después del guion.

Este patrón es efectivo para identificar y extraer palabras que contienen guiones en un texto, asegurando que las coincidencias incluyan tanto la parte anterior como la posterior al guion.

```
# Ejercicio 6 - Extraer Fechas (mm/dd/yyyy)
def extraer_fechas_mmddyyyy(texto):
    # Patrón para validar fechas en formato mm/dd/yyyy
    patron6 = r'\b(?:0[1-9]|1[0-2])/(?:0[1-9]|[12][0-9]|3[01])/\d{4}\
b'
    return re.findall(patron6, texto)

print("Ejercicio 6 - Fechas (mm/dd/yyyy):",
extraer_fechas_mmddyyyy(texto6))
```

```
Ejercicio 6 - Fechas (mm/dd/yyyy): ['12/25/2022', '02/14/2023',
'07/04/2023', '10/31/2023']
```

Explicación del Patrón para Extraer Fechas (mm/dd/yyyy)

El patrón utilizado para extraer fechas en formato `mm/dd/yyyy` es el siguiente: `\b(?:0[1-9]|1[0-2])/(?:0[1-9]|[12][0-9]|3[01])/\d{4}\b`. Este patrón se descompone de la siguiente manera:

- **`\b`**: Asegura que la coincidencia ocurra al inicio de una palabra, evitando que haya caracteres alfanuméricos antes de la fecha. Esto garantiza que se capturen fechas completas y no partes de otros números.
- **`(?:0[1-9]|1[0-2])`**: Coincide con el mes de la fecha. Permite:
  - `0[1-9]`: Meses del `01` al `09`.
  - `1[0-2]`: Meses `10`, `11`, y `12`.
  - `(?:...)` es una agrupación no capturante, que agrupa los patrones sin capturarlos como grupos separados.
- **`/`**: Este símbolo es un separador que indica la división entre el mes y el día.
- **`(?:0[1-9]|[12][0-9]|3[01])`**: Coincide con el día de la fecha. Permite:
  - `0[1-9]`: Días del `01` al `09`.
  - `[12][0-9]`: Días del `10` al `29`.
  - `3[01]`: Días `30` y `31`.
  - Nuevamente, `(?:...)` es una agrupación no capturante.
- **`/`**: Otro separador que indica la división entre el día y el año.
- **`\d{4}`**: Coincide con el año, que debe consistir en exactamente 4 dígitos.
  - El `\d` representa cualquier dígito (equivalente a `[0-9]`), y `{4}` indica que debe haber exactamente 4 de estos dígitos.
- **`\b`**: Asegura que la coincidencia ocurra al final de una palabra, evitando que haya caracteres alfanuméricos después de la fecha.

Este patrón es efectivo para validar y extraer fechas en el formato `mm/dd/yyyy`, asegurando que las fechas extraídas tengan un formato correcto y estén dentro de los rangos válidos para meses y días.

```
# Ejercicio 7 - Extraer Hashtags
def extraer_hashtags(texto):
    patron7 = r'#\w+'
    return re.findall(patron7, texto)

print("Ejercicio 7 - Hashtags:", extraer_hashtags(texto7))
```

```
Ejercicio 7 - Hashtags: ['#hashtag', '#regex', '#Python',
'#programación']
```

Explicación del Patrón para Extraer Hashtags

El patrón utilizado para extraer hashtags es el siguiente: `#\w+`. Este patrón se descompone de la siguiente manera:

- **`#`**: Coincide con el símbolo de almohadilla (`#`), que es el carácter que indica el inicio de un hashtag. Este símbolo es obligatorio para que la coincidencia sea válida.
- **`\w+`**: Coincide con uno o más caracteres alfanuméricos que forman parte del hashtag.
  - `\w` representa cualquier carácter alfanumérico, que incluye letras (A-Z, a-z), números (0-9) y el carácter de subrayado (`_`).
  - El `+` indica que debe haber al menos un carácter de este tipo.

Este patrón es efectivo para identificar y extraer hashtags de un texto, garantizando que las coincidencias incluyan tanto el símbolo de almohadilla como la cadena de caracteres que lo sigue. Los hashtags extraídos pueden ser utilizados en redes sociales y otras plataformas donde se empleen etiquetas para categorizar contenido.

```
# Ejercicio 8 - Extraer Usuarios Mencionados
def extraer_menciones_usuario(texto):
    patron8 = r'@\w+'
    return re.findall(patron8, texto)

print("Ejercicio 8 - Usuarios Mencionados:",
extraer_menciones_usuario(texto8))
```

```
Ejercicio 8 - Usuarios Mencionados: ['@usuario1', '@usuario2',
'@miusuario', '@ejemplo']
```

Explicación del Patrón para Extraer Usuarios Mencionados

El patrón utilizado para extraer usuarios mencionados es el siguiente: `@\w+`. Este patrón se descompone de la siguiente manera:

- **`@`**: Coincide con el símbolo de arroba (`@`), que es el carácter que indica el inicio de una mención de usuario. Este símbolo es obligatorio para que la coincidencia sea válida.
- **`\w+`**: Coincide con uno o más caracteres alfanuméricos que forman parte del nombre de usuario.
  - `\w` representa cualquier carácter alfanumérico, que incluye letras (A-Z, a-z), números (0-9) y el carácter de subrayado (`_`).
  - El `+` indica que debe haber al menos un carácter de este tipo.

Este patrón es efectivo para identificar y extraer menciones de usuarios en plataformas de redes sociales y otros entornos donde se utilizan arrobas para referirse a otros usuarios. Las

menciones extraídas pueden ser utilizadas para interactuar con los usuarios mencionados o para análisis de contenido.

```
# Ejercicio 9 - Coincidir Números de Teléfono
def extraer_numeros_telefono(texto):
    patron9 = r'\(\+\d{2}\) \d{3}-\d{3}-\d{3}'
    return re.findall(patron9, texto)

print("Ejercicio 9 - Números de Teléfono:",
extraer_numeros_telefono(texto9))
```

```
Ejercicio 9 - Números de Teléfono: ['(+34) 612-345-678', '(+34) 678-
123-456', '(+34) 911-234-567']
```

Explicación del Patrón para Coincidir Números de Teléfono

El patrón utilizado para extraer números de teléfono es el siguiente: `\(\+\d{2}\) \d{3}-\d{3}-\d{3}`. Este patrón se descompone de la siguiente manera:

- **`\(`**: Coincide con el paréntesis izquierdo `(`. Se utiliza la barra invertida (`\`) para escapar el paréntesis, ya que tiene un significado especial en las expresiones regulares.
- **`\+`**: Coincide con el símbolo de más `+`, que representa el prefijo internacional en el número de teléfono.
- **`\d{2}`**: Coincide con exactamente dos dígitos que representan el código de país.
  - `\d` representa cualquier dígito (equivalente a `[0-9]`), y `{2}` indica que debe haber exactamente dos de estos dígitos.
- **`\)`**: Coincide con el paréntesis derecho `)`. Al igual que el paréntesis izquierdo, se escapa con una barra invertida.
- **(espacio)**: Coincide con un espacio en blanco que debe seguir al código de país.
- **`\d{3}`**: Coincide con exactamente tres dígitos, que corresponden a la primera parte del número de teléfono (número local).
- **`-`**: Coincide con un guion literal que separa las partes del número.
- **`\d{3}`**: Coincide nuevamente con exactamente tres dígitos, que corresponden a la segunda parte del número de teléfono.
- **`-`**: Otro guion literal que separa las partes del número.
- **`\d{3}`**: Finalmente, coincide con exactamente tres dígitos que representan la última parte del número de teléfono.

Este patrón es efectivo para validar y extraer números de teléfono en el formato `(+(código país)) xxx-xxx-xxx`, asegurando que se capturen números completos y en el formato correcto.

```
# Ejercicio 10 - Coincidir Palabras Alfanuméricas
def extraer_palabras_alfanumericas(texto):
    patron10 = r'\b[a-zA-Z0-9]+\b'
    return re.findall(patron10, texto)

print("Ejercicio 10 - Palabras Alfanuméricas:",
extraer_palabras_alfanumericas(texto10))
```

```
Ejercicio 10 - Palabras Alfanuméricas: ['Las', 'palabras', 'son',
'Python3', 'Regex', 'HTML5', 'y', 'JavaScript']
```

Explicación del Patrón para Coincidir Palabras Alfanuméricas

El patrón utilizado para extraer palabras alfanuméricas es el siguiente: `\b[a-zA-Z0-9]+\b`. Este patrón se descompone de la siguiente manera:

- **`\b`**: Asegura que la coincidencia ocurra al inicio de una palabra, lo que significa que no debe haber caracteres alfanuméricos antes de la palabra. Esto garantiza que solo se capturen palabras completas.
- **`[a-zA-Z0-9]+`**: Coincide con una o más letras o dígitos.
  - `a-z`: Coincide con letras minúsculas del alfabeto (de `a` a `z`).
  - `A-Z`: Coincide con letras mayúsculas del alfabeto (de `A` a `Z`).
  - `0-9`: Coincide con cualquier dígito numérico (de `0` a `9`).
  - El `+` indica que debe haber al menos un carácter que cumpla con estas condiciones.
- **`\b`**: Asegura que la coincidencia ocurra al final de una palabra, evitando que haya caracteres alfanuméricos después de la palabra. Esto también contribuye a que solo se extraigan palabras completas.

Este patrón es efectivo para identificar y extraer palabras que contienen únicamente caracteres alfanuméricos, siendo útil en situaciones donde se necesiten palabras compuestas de letras y números sin incluir otros símbolos o caracteres especiales.

Grupos de Captura en Expresiones Regulares

Los grupos de captura son una característica poderosa en las expresiones regulares que permiten extraer partes específicas de un texto coincidente. En Python, puedes trabajar con grupos de captura usando objetos `SRE_Match` y métodos como `.groups()` y `.group()`.

¿Qué Son los Grupos de Captura?

Un grupo de captura es una parte de un patrón de regex encerrado entre paréntesis `( )`. Los grupos de captura tienen dos propósitos principales:

1. **Agrupación:** Los paréntesis se utilizan para agrupar partes de un patrón. Esto es útil para aplicar cuantificadores como `*`, `+` o `?` a múltiples caracteres o subpatrones. Al agrupar patrones, puedes controlar cómo se aplican estos cuantificadores a partes específicas del texto.

2. **Extracción:** Los grupos de captura permiten extraer porciones específicas del texto coincidente. Cada conjunto de paréntesis crea un grupo de captura separado, lo que facilita la recuperación de información importante de coincidencias más grandes.

Usando Grupos de Captura en Python

En Python, cuando utilizas expresiones regulares, el objeto de coincidencia resultante (`SRE_Match`) proporciona métodos para trabajar con grupos de captura:

- **`.groups()`**: Este método devuelve una tupla que contiene todos los grupos capturados. El primer elemento de esta tupla es la coincidencia completa del patrón regex, mientras que los elementos subsecuentes corresponden a cada grupo capturado.

- **`.group(n)`**: Para acceder a un grupo de captura específico, pasas su índice `n` al método `.group()`. El índice se basa en el orden de los paréntesis de apertura en el patrón regex, comenzando desde 1. El índice 0 se refiere a la coincidencia completa. Esta funcionalidad es especialmente útil cuando necesitas trabajar con información capturada de manera individual.

Beneficios de Usar Grupos de Captura

- **Simplicidad:** Facilitan el manejo de patrones complejos al permitir agrupar y extraer datos sin necesidad de realizar múltiples búsquedas.

- **Flexibilidad:** Permiten aplicar operaciones adicionales en los grupos capturados, como formatear o manipular datos, lo que puede ser esencial en tareas de procesamiento de texto.

- **Claridad:** Hacen que los patrones sean más legibles y comprensibles al segmentar lógicamente las partes relevantes de las coincidencias.

- **Reutilización:** Puedes usar grupos de captura para referenciar partes del patrón en las mismas expresiones regulares, utilizando notaciones como `\1`, `\2`, etc., donde el número se refiere al índice del grupo.

Consideraciones al Usar Grupos de Captura

- **Limitaciones:** Asegúrate de no abusar de los grupos de captura, ya que un uso excesivo puede llevar a patrones complicados y difíciles de entender. En algunos casos, los grupos de captura no son necesarios, y la simple agrupación puede ser suficiente.

- **Impacto en el Rendimiento:** Los grupos de captura pueden tener un impacto en el rendimiento de las expresiones regulares, especialmente en patrones muy complejos. Es recomendable optimizar los patrones siempre que sea posible.

- **Grupos Sin Captura:** Si solo necesitas agrupar parte de un patrón sin extraerlo, puedes usar grupos sin captura con `(?: )`, lo que evita que se creen índices de grupos para esa sección del patrón.

Los grupos de captura son una herramienta invaluable al trabajar con expresiones regulares en Python. Permiten no solo agrupar patrones complejos, sino también extraer información específica de cadenas de texto. Dominar el uso de grupos de captura puede facilitar el procesamiento y la manipulación de datos textuales en tus aplicaciones. Con un buen entendimiento de su funcionamiento y sus aplicaciones, podrás implementar soluciones más eficientes y efectivas para el análisis de texto y la extracción de datos.

Ejemplo:

Ilustremos el concepto con un ejemplo:

```
import re

# Supongamos que queremos extraer fechas en el formato "dd/mm/aaaa" de un texto.
text = "Reunión programada para el 25/12/2022 y el 31/12/2022."

# Define el patrón de regex con grupos de captura para el día, mes y año.
pattern = r'(\d{2})/(\d{2})/(\d{4})'

# Buscar coincidencias usando el patrón.
matches = re.finditer(pattern, text)

# Iterar a través de las coincidencias y acceder a los grupos de captura.
for match in matches:
    # La coincidencia completa (grupo 0) es accesible como match.group(0).
    print(f"Coincidencia Completa: {match.group(0)}")

    # Acceder a grupos de captura individuales usando match.group(n).
    día = match.group(1)
    mes = match.group(2)
    año = match.group(3)

    print(f"Día: {día}, Mes: {mes}, Año: {año}")
```

```
Coincidencia Completa: 25/12/2022
Día: 25, Mes: 12, Año: 2022
Coincidencia Completa: 31/12/2022
Día: 31, Mes: 12, Año: 2022
```

En este ejemplo, utilizamos grupos de captura para extraer los componentes del día, mes y año de una fecha. Los estudiantes pueden ver cómo acceder a estos componentes utilizando el

método `.group(n)` y comprender la utilidad de los grupos de captura para extraer información específica de un texto que coincide con una expresión regular (regex).

Resumen de Funciones Útiles para Expresiones Regulares

Al trabajar con expresiones regulares en Python, puedes utilizar varias funciones integradas proporcionadas por el módulo `re`. A continuación, se presenta una descripción general de estas funciones y su funcionalidad:

- **`re.match(pattern, string)`**: Esta función verifica si el patrón regex coincide al comienzo de la cadena de entrada. Devuelve un objeto de coincidencia si se encuentra una coincidencia al inicio de la cadena o `None` en caso contrario. Es útil para validar que una cadena comience con un patrón específico.
- **`re.search(pattern, string)`**: Esta función escanea a través de una cadena, buscando cualquier ubicación donde el patrón regex coincida. Devuelve un objeto de coincidencia para la primera ocurrencia encontrada o `None` si no se encuentra ninguna coincidencia. Se utiliza comúnmente cuando no es necesario que la coincidencia esté al inicio.
- **`re.findall(pattern, string)`**: Esta función encuentra todas las subcadenas no superpuestas donde el patrón regex coincide en la cadena de entrada y las devuelve como una lista. Es útil para obtener todas las coincidencias de un patrón en el texto de forma eficiente.
- **`re.finditer(pattern, string)`**: Esta función encuentra todas las subcadenas no superpuestas donde el patrón regex coincide en la cadena de entrada y las devuelve como un iterador. Esto permite procesar las coincidencias una por una, lo que puede ser más eficiente en términos de memoria, especialmente cuando se trabaja con grandes volúmenes de texto.

Importancia de las Funciones de Expresiones Regulares

Estas funciones son esenciales para diversas tareas de procesamiento de texto, permitiendo buscar y manipular patrones dentro de cadenas de manera eficiente. Al dominar estas funciones, puedes realizar tareas como la validación de datos, la extracción de información y la limpieza de texto de forma más efectiva. La comprensión de cómo y cuándo usar cada función te ayudará a abordar problemas de texto más complejos y a desarrollar soluciones más robustas.

Al trabajar con expresiones regulares, es fundamental tener en cuenta el rendimiento, ya que patrones complejos pueden llevar más tiempo en procesarse. Además, es recomendable probar las expresiones regulares con diferentes tipos de entrada para asegurar que se comporten como se espera en todos los casos. Con la práctica, los grupos de captura y las funciones del módulo `re` se convertirán en herramientas valiosas en tu conjunto de habilidades de programación.

Ejemplos de casos de uso de expresiones regulares

```
import re
```

```
def verificar_oraciones(patron, oraciones):
    """
    Verifica las oraciones contra el patrón de expresión regular
proporcionado.

    Argumentos:
        patron (str): El patrón de expresión regular con el que
comparar.
        oraciones (lista de tuplas): Una lista de tuplas donde cada
tupla contiene una oración y un resultado esperado (True o False).

    Retorna:
        None

    Imprime:
        Retroalimentación para cada oración según si coincide con el
patrón o no.
    """

    for oracion, resultado_esperado in oraciones:
        # Verifica si el patrón coincide con toda la cadena.
        coincide = bool(re.fullmatch(patron, oracion))

        # Determina si el resultado coincide con el resultado
esperado.
        es_valido = coincide == resultado_esperado

        # Imprime retroalimentación basada en la coincidencia.
        if coincide:
            mensaje_resultado = 'Aprobado'
        else:
            mensaje_resultado = 'No Aprobado'

        mensaje_validez = '(Válido)' if resultado_esperado else '(No
Válido)'

        # Mejorar la salida para incluir más detalles
        resultado_detalle = "Coincide" if coincide else "No coincide"
        print(f'{mensaje_resultado} --> {oracion} {mensaje_validez} |
{resultado_detalle}')
```

Explicación de la Función `verificar_oraciones`

La función `verificar_oraciones` está diseñada para evaluar una lista de oraciones en comparación con un patrón de expresión regular proporcionado, ofreciendo retroalimentación sobre si cada oración coincide con el patrón según lo esperado.

Parámetros de la Función

- **`patron`**: Este parámetro representa el patrón de expresión regular con el que se comparará cada oración. Puede incluir caracteres especiales y cuantificadores que definen cómo debe coincidir la oración.
- **`oraciones`**: Este parámetro es una lista de tuplas, donde cada tupla contiene una oración y un resultado esperado (True o False). El resultado esperado indica si se anticipa que la oración coincida con el patrón o no.

Propósito de la Función

El propósito principal de la función `verificar_oraciones` es validar cada oración en la lista `oraciones` contra el patrón de expresión regular proporcionado. Esto es útil en diversas aplicaciones, como la limpieza de datos, la validación de entradas de usuario y el análisis de texto, donde es esencial asegurarse de que las oraciones cumplan con un formato específico.

Ejecución de la Función

1. **Iteración**: La función itera a través de cada tupla en la lista `oraciones`, extrayendo la oración y su resultado esperado asociado.
2. **Verificación de Coincidencia**: Para cada oración, utiliza la función `re.fullmatch()` para determinar si el patrón de expresión regular coincide con toda la oración. El resultado de esta coincidencia se almacena en una variable booleana.
3. **Evaluación de Resultados**: La función evalúa si la coincidencia es válida comparándola con el resultado esperado. Si la coincidencia y el resultado esperado coinciden, la oración se considera válida; de lo contrario, se considera no válida.
4. **Mensajes de Retroalimentación**: Basado en la validez de la coincidencia, la función asigna un mensaje de resultado como "Aprobado" o "No Aprobado" para proporcionar retroalimentación clara.
5. **Determinación de Validez**: La función determina si la oración es "Válida" o "No Válida" en función del resultado esperado y prepara un mensaje que indica esta validez.
6. **Impresión de Resultados**: Finalmente, la función imprime retroalimentación para cada oración, indicando si pasó o no la evaluación, junto con la oración misma y su estado de validez.

Salida de la Función

La función imprime mensajes de retroalimentación para cada oración, proporcionando información sobre si cada una se ajusta al patrón de expresión regular especificado. Esta retroalimentación es útil para validar y verificar datos de texto contra un patrón definido, ayudando a identificar posibles errores o incongruencias.

Importancia de la Función

La función `verificar_oraciones` es una herramienta valiosa para tareas de control de calidad y validación que involucran datos de texto. Su capacidad para evaluar la integridad de los datos contra patrones o reglas predefinidas la convierte en un recurso esencial en aplicaciones de procesamiento de lenguaje natural, validación de formularios y limpieza de datos. Además, puede ser utilizada en contextos donde es crucial garantizar que las oraciones cumplan con formatos específicos, mejorando la precisión y confiabilidad de los datos procesados.

Al utilizar esta función, es importante tener en cuenta que el patrón de expresión regular debe estar diseñado adecuadamente para reflejar las expectativas de validación. Asimismo, las oraciones en la lista deben ser revisadas para asegurar que la retroalimentación proporcionada sea significativa y útil para el contexto en el que se aplican.

Mejora Continua

Para maximizar la eficacia de esta función, considera implementar pruebas automatizadas que validen diferentes patrones y oraciones. Esto ayudará a garantizar que la función se comporta como se espera en diversos escenarios y con diferentes tipos de datos. Adicionalmente, puedes añadir manejo de excepciones para gestionar posibles errores en la entrada, como patrones de expresión regular malformados o listas vacías, lo que hará que la función sea más robusta y confiable.

```
# Define un patrón regex que permite solo archivos con extensiones específicas.
# El patrón evita que haya dos puntos consecutivos y asegura que solo hay una extensión válida.
pattern = r'^[\w\-]+\.((gif|jpeg|jpg|TIF))$'  # Solo permite las extensiones específicas

# Define una lista de oraciones para ser comprobadas contra el patrón, junto con sus resultados esperados.
sentences = [
    ('test.gif', True),           # Debe coincidir
    ('image.jpeg', True),         # Debe coincidir
    ('image.jpg', True),          # Debe coincidir
    ('image.TIF', True),          # Debe coincidir
    ('test', False),              # No debe coincidir
    ('test.pdf', False),          # No debe coincidir
    ('test.gif.gif', False),      # No debe coincidir, tiene doble extensión
    ('test..jpg', False),         # No debe coincidir, tiene dos puntos
    ('image.jpeg.jpeg', False)    # No debe coincidir, tiene dos extensiones
]

# Llama a la función verificar_oraciones con el patrón y las oraciones para realizar las comprobaciones.
verificar_oraciones(pattern, sentences)
```

```
Aprobado --> test.gif (Válido) | Coincide
Aprobado --> image.jpeg (Válido) | Coincide
Aprobado --> image.jpg (Válido) | Coincide
Aprobado --> image.TIF (Válido) | Coincide
No Aprobado --> test (No Válido) | No coincide
No Aprobado --> test.pdf (No Válido) | No coincide
No Aprobado --> test.gif.gif (No Válido) | No coincide
No Aprobado --> test..jpg (No Válido) | No coincide
No Aprobado --> image.jpeg.jpeg (No Válido) | No coincide
```

Explicación del Código

La función `verificar_oraciones` se utiliza para evaluar una lista de oraciones (nombres de archivos en este caso) en comparación con un patrón de expresión regular que permite solo ciertos formatos de archivos, específicamente aquellos que tienen extensiones válidas como `.gif`, `.jpeg`, `.jpg` o `.TIF`.

Detalles del Patrón Regex

- **Patrón**: `^[\w\-]+\.((gif|jpeg|jpg|TIF))$`
  - `^`: Indica el inicio de la cadena.
  - `[\w\-]+`: Permite uno o más caracteres alfanuméricos y guiones. Esto significa que el nombre del archivo puede contener letras, números, guiones bajos y guiones.
  - `\.`: Escapa el punto (`.`) para asegurarse de que se interprete literalmente como un separador de la extensión.
  - `((gif|jpeg|jpg|TIF))`: Agrupa las extensiones válidas. Solo permitirá las extensiones especificadas.
  - `$`: Indica el final de la cadena, asegurando que no haya otros caracteres después de la extensión.

Lista de Oraciones

- La lista `sentences` contiene tuplas que consisten en un nombre de archivo y un resultado esperado (True o False) para cada uno.
  - **Ejemplos que deben coincidir**:
    - `'test.gif'`, `'image.jpeg'`, `'image.jpg'`, `'image.TIF'`: Todos estos cumplen con el patrón y deben retornar True.
  - **Ejemplos que no deben coincidir**:
    - `'test'`: No tiene extensión.
    - `'test.pdf'`: Extensión no válida.
    - `'test.gif.gif'`: Tiene una doble extensión.
    - `'test..jpg'`: Contiene dos puntos consecutivos.
    - `'image.jpeg.jpeg'`: Tiene dos extensiones.

Propósito de la Evaluación

La evaluación se realiza mediante la función `verificar_oraciones`, que compara cada oración con el patrón definido. Esto es útil para validar nombres de archivos y asegurar que cumplen con las especificaciones requeridas, ayudando a evitar errores en la gestión de archivos.

2. Comprobación de Números Enteros Positivos

```
# Define un patrón regex para coincidir con cadenas que consisten solo en números enteros positivos.
pattern = r'^[1-9]\d*$'  # Asegura que la cadena comience con un dígito del 1 al 9 y siga con cero o más dígitos.

# Define una lista de oraciones para ser comprobadas contra el patrón, junto con sus resultados esperados.
sentences = [
    ('123', True),    # Debe coincidir: número entero positivo, se espera que coincida.
    ('10', True),     # Debe coincidir: número entero positivo, se espera que coincida.
    ('4567', True),   # Debe coincidir: número entero positivo, se espera que coincida.
    ('7890', True),   # Debe coincidir: número entero positivo, se espera que coincida.
    ('0', False),     # No debe coincidir: cero no es considerado un número entero positivo.
    ('abc', False),   # No debe coincidir: cadena alfabética, no coincide.
    ('1.1', False),   # No debe coincidir: número decimal, no coincide.
    ('-123', False),  # No debe coincidir: número entero negativo, no coincide.
    ('-', False),     # No debe coincidir: solo un signo negativo, no coincide.
    ('001', False),   # No debe coincidir: comienza con un cero, no es un número entero positivo.
    ('10.0', False),  # No debe coincidir: número decimal, no coincide.
]

# Llama a la función verificar_oraciones con el patrón y las oraciones para realizar las comprobaciones.
verificar_oraciones(pattern, sentences)
```

```
Aprobado --> 123 (Válido) | Coincide
Aprobado --> 10 (Válido) | Coincide
Aprobado --> 4567 (Válido) | Coincide
Aprobado --> 7890 (Válido) | Coincide
No Aprobado --> 0 (No Válido) | No coincide
No Aprobado --> abc (No Válido) | No coincide
```

```
No Aprobado --> 1.1 (No Válido) | No coincide
No Aprobado --> -123 (No Válido) | No coincide
No Aprobado --> - (No Válido) | No coincide
No Aprobado --> 001 (No Válido) | No coincide
No Aprobado --> 10.0 (No Válido) | No coincide
```

Explicación del Código

La función `verificar_oraciones` se utiliza para evaluar una lista de cadenas (números en este caso) en comparación con un patrón de expresión regular que permite solo números enteros positivos.

Detalles del Patrón Regex

- **Patrón**: `^[1-9]\d*$`
  - `^`: Indica el inicio de la cadena.
  - `[1-9]`: Asegura que la cadena comience con un dígito entre 1 y 9 (es decir, no puede comenzar con 0).
  - `\d*`: Permite cero o más dígitos adicionales (0-9) después del primer dígito.
  - `$`: Indica el final de la cadena, asegurando que no haya otros caracteres después del número.

Lista de Oraciones

- La lista `sentences` contiene tuplas que consisten en una cadena y su resultado esperado (True o False).

Ejemplos que deben coincidir (True):

- **'123'**: Debe coincidir: número entero positivo, se espera que coincida.
- **'10'**: Debe coincidir: número entero positivo, se espera que coincida.
- **'4567'**: Debe coincidir: número entero positivo, se espera que coincida.
- **'7890'**: Debe coincidir: número entero positivo, se espera que coincida.

Ejemplos que no deben coincidir (False):

- **'0'**: No debe coincidir: cero no es considerado un número entero positivo.
- **'abc'**: No debe coincidir: cadena alfabética, no coincide.
- **'1.1'**: No debe coincidir: número decimal, no coincide.
- **'-123'**: No debe coincidir: número entero negativo, no coincide.
- **'-'**: No debe coincidir: solo un signo negativo, no coincide.
- **'001'**: No debe coincidir: comienza con un cero, no es un número entero positivo.
- **'10.0'**: No debe coincidir: número decimal, no coincide.

Propósito de la Evaluación

La evaluación se realiza mediante la función `verificar_oraciones`, que compara cada cadena con el patrón definido. Esto es útil para validar entradas de usuario, asegurando que solo se acepten números enteros positivos, ayudando a prevenir errores en la entrada de datos.

3. Comprobación de Números Enteros Negativos

```
# Define un patrón regex para coincidir con cadenas que consisten solo en números enteros negativos.
pattern = r'^-[1-9]\d*$'  # Asegura que la cadena comience con un signo negativo seguido de un dígito del 1 al 9 y cero o más dígitos.

# Define una lista de oraciones para ser comprobadas contra el patrón, junto con sus resultados esperados.
sentences = [
    ('-123', True),   # Debe coincidir: número entero negativo, se espera que coincida.
    ('-10', True),    # Debe coincidir: número entero negativo, se espera que coincida.
    ('-1', True),     # Debe coincidir: número entero negativo, se espera que coincida.
    ('-4567', True),  # Debe coincidir: número entero negativo, se espera que coincida.
    ('0', False),     # No debe coincidir: cero no es considerado un número entero negativo.
    ('abc', False),   # No debe coincidir: cadena alfabética, no coincide.
    ('1.1', False),   # No debe coincidir: número decimal, no coincide.
    ('123', False),   # No debe coincidir: número entero positivo, no coincide.
    ('-', False),     # No debe coincidir: solo un signo negativo, no coincide.
    ('-0', False),    # No debe coincidir: cero no es considerado un número entero negativo.
]

# Llama a la función verificar_oraciones con el patrón y las oraciones para realizar las comprobaciones.
verificar_oraciones(pattern, sentences)
```

```
Aprobado --> -123 (Válido) | Coincide
Aprobado --> -10 (Válido) | Coincide
Aprobado --> -1 (Válido) | Coincide
Aprobado --> -4567 (Válido) | Coincide
No Aprobado --> 0 (No Válido) | No coincide
No Aprobado --> abc (No Válido) | No coincide
No Aprobado --> 1.1 (No Válido) | No coincide
No Aprobado --> 123 (No Válido) | No coincide
No Aprobado --> - (No Válido) | No coincide
No Aprobado --> -0 (No Válido) | No coincide
```

Explicación del Código

La función `verificar_oraciones` se utiliza para evaluar una lista de cadenas que representan números en comparación con un patrón de expresión regular que permite solo números enteros negativos.

Detalles del Patrón Regex

- **Patrón**: `^-[1-9]\d*$`
  - `^`: Indica el inicio de la cadena.
  - `-`: Asegura que la cadena comience con un signo negativo.
  - `[1-9]`: Asegura que el siguiente carácter sea un dígito entre 1 y 9 (no puede ser 0).
  - `\d*`: Permite cero o más dígitos adicionales (0-9) después del primer dígito.
  - `$`: Indica el final de la cadena, asegurando que no haya otros caracteres después del número.

Lista de Oraciones

- La lista `sentences` contiene tuplas que consisten en una cadena y su resultado esperado (True o False).

Ejemplos que deben coincidir (True):

- **'-123'**: Debe coincidir: número entero negativo, se espera que coincida.
- **'-10'**: Debe coincidir: número entero negativo, se espera que coincida.
- **'-1'**: Debe coincidir: número entero negativo, se espera que coincida.
- **'-4567'**: Debe coincidir: número entero negativo, se espera que coincida.

Ejemplos que no deben coincidir (False):

- **'0'**: No debe coincidir: cero no es considerado un número entero negativo.
- **'abc'**: No debe coincidir: cadena alfabética, no coincide.
- **'1.1'**: No debe coincidir: número decimal, no coincide.
- **'123'**: No debe coincidir: número entero positivo, no coincide.
- **'-'**: No debe coincidir: solo un signo negativo, no coincide.
- **'-0'**: No debe coincidir: cero no es considerado un número entero negativo.

Propósito de la Evaluación

La evaluación se realiza mediante la función `verificar_oraciones`, que compara cada cadena con el patrón definido. Esto es útil para validar entradas de usuario, asegurando que solo se acepten números enteros negativos y ayudando a prevenir errores en la entrada de datos.

4. Comprobación de Todos los Números Enteros

```
# Define un patrón regex para coincidir con cualquier número entero (positivos, negativos o cero).
pattern = r'^-?\d+$'  # El signo "-" es opcional, seguido de uno o más dígitos.
```

```
# Define una lista de oraciones para ser comprobadas contra el patrón, junto con sus resultados esperados.
sentences = [
    ('-123', True),   # Debe coincidir: número entero negativo, se espera que coincida.
    ('123', True),    # Debe coincidir: número entero positivo, se espera que coincida.
    ('0', True),      # Debe coincidir: cero, se espera que coincida.
    ('-456', True),   # Debe coincidir: número entero negativo, se espera que coincida.
    ('abc', False),   # No debe coincidir: cadena alfabética, no coincide.
    ('1.1', False),   # No debe coincidir: número decimal, no coincide.
    ('-1.1', False),  # No debe coincidir: número decimal negativo, no coincide.
    ('-', False),     # No debe coincidir: solo un signo negativo, no coincide.
    ('123abc', False),# No debe coincidir: combinación de números y letras, no coincide.
    ('-0.5', False),  # No debe coincidir: número decimal negativo, no coincide.
]

# Llama a la función verificar_oraciones con el patrón y las oraciones para realizar las comprobaciones.
verificar_oraciones(pattern, sentences)
```

```
Aprobado --> -123 (Válido) | Coincide
Aprobado --> 123 (Válido) | Coincide
Aprobado --> 0 (Válido) | Coincide
Aprobado --> -456 (Válido) | Coincide
No Aprobado --> abc (No Válido) | No coincide
No Aprobado --> 1.1 (No Válido) | No coincide
No Aprobado --> -1.1 (No Válido) | No coincide
No Aprobado --> - (No Válido) | No coincide
No Aprobado --> 123abc (No Válido) | No coincide
No Aprobado --> -0.5 (No Válido) | No coincide
```

Explicación del Código

La función `verificar_oraciones` se utiliza para evaluar una lista de cadenas que representan números en comparación con un patrón de expresión regular que permite cualquier número entero, ya sea positivo, negativo o cero.

Detalles del Patrón Regex

- **Patrón**: `^-?\d+$`
  - `^`: Indica el inicio de la cadena.
  - `-?`: El signo negativo es opcional (puede no estar presente).

- `\d+`: Asegura que haya uno o más dígitos (0-9) en la cadena.
- `$`: Indica el final de la cadena, asegurando que no haya otros caracteres después de los dígitos.

Lista de Oraciones

- La lista `sentences` contiene tuplas que consisten en una cadena y su resultado esperado (True o False).

Ejemplos que deben coincidir (True):

- **'-123'**: Debe coincidir: número entero negativo, se espera que coincida.
- **'123'**: Debe coincidir: número entero positivo, se espera que coincida.
- **'0'**: Debe coincidir: cero, se espera que coincida.
- **'-456'**: Debe coincidir: número entero negativo, se espera que coincida.

Ejemplos que no deben coincidir (False):

- **'abc'**: No debe coincidir: cadena alfabética, no coincide.
- **'1.1'**: No debe coincidir: número decimal, no coincide.
- **'-1.1'**: No debe coincidir: número decimal negativo, no coincide.
- **'-'**: No debe coincidir: solo un signo negativo, no coincide.
- **'123abc'**: No debe coincidir: combinación de números y letras, no coincide.
- **'-0.5'**: No debe coincidir: número decimal negativo, no coincide.

Propósito de la Evaluación

La evaluación se realiza mediante la función `verificar_oraciones`, que compara cada cadena con el patrón definido. Esto es útil para validar entradas de usuario, asegurando que solo se acepten números enteros (positivos, negativos o cero) y ayudando a prevenir errores en la entrada de datos.

5. Comprobación de Números Decimales Positivos

```
# Define un patrón regex para coincidir solo con números decimales positivos.
pattern = r'^[0-9]+\.\d+$'  # Asegura que la cadena contenga uno o más dígitos antes del punto decimal y al menos un dígito después.

# Define una lista de oraciones para ser comprobadas contra el patrón, junto con sus resultados esperados.
sentences = [
    ('123.45', True),   # Debe coincidir: número decimal positivo, se espera que coincida.
    ('10.5', True),     # Debe coincidir: número decimal positivo, se espera que coincida.
    ('1.0', True),      # Debe coincidir: número decimal positivo, se espera que coincida.
    ('0.01', True),     # Debe coincidir: número decimal positivo que empieza con cero, se espera que coincida.
    ('9.99', True),     # Debe coincidir: número decimal positivo, se
```

```
espera que coincida.

    ('abc', False),     # No debe coincidir: cadena alfabética, no
coincide.
    ('1', False),       # No debe coincidir: número entero positivo,
no coincide.
    ('-123.45', False), # No debe coincidir: número decimal negativo,
no coincide.
    ('0', False),       # No debe coincidir: cero no es un número
decimal positivo, no coincide.
    ('0.0', False),     # No debe coincidir: cero decimal, no
coincide.
    ('123.', False)     # No debe coincidir: punto decimal sin números
después, no coincide.
]

# Llama a la función verificar_oraciones con el patrón y las oraciones
para realizar las comprobaciones.
verificar_oraciones(pattern, sentences)
```

```
Aprobado --> 123.45 (Válido) | Coincide
Aprobado --> 10.5 (Válido) | Coincide
Aprobado --> 1.0 (Válido) | Coincide
Aprobado --> 0.01 (Válido) | Coincide
Aprobado --> 9.99 (Válido) | Coincide
No Aprobado --> abc (No Válido) | No coincide
No Aprobado --> 1 (No Válido) | No coincide
No Aprobado --> -123.45 (No Válido) | No coincide
No Aprobado --> 0 (No Válido) | No coincide
Aprobado --> 0.0 (No Válido) | Coincide
No Aprobado --> 123. (No Válido) | No coincide
```

Explicación del Código

La función `verificar_oraciones` se utiliza para evaluar una lista de cadenas que representan números decimales positivos.

Detalles del Patrón Regex

- **Patrón**: `^[0-9]+\.\d+$`
  - `^`: Indica el inicio de la cadena.
  - `[0-9]+`: Uno o más dígitos del 0 al 9 antes del punto decimal.
  - `\.`: Un punto decimal que debe estar presente.
  - `\d+$`: Uno o más dígitos después del punto decimal. Esto asegura que todos los números deben ser decimales y no enteros.

Lista de Frases

- La lista `sentences` contiene tuplas que consisten en una cadena y su resultado esperado (True o False).

Ejemplos que deben coincidir (True):

- **'123.45'**: Debe coincidir: número decimal positivo, se espera que coincida.
- **'10.5'**: Debe coincidir: número decimal positivo, se espera que coincida.
- **'1.0'**: Debe coincidir: número decimal positivo, se espera que coincida.
- **'0.01'**: Debe coincidir: número decimal positivo que empieza con cero, se espera que coincida.
- **'9.99'**: Debe coincidir: número decimal positivo, se espera que coincida.

Ejemplos que no deben coincidir (False):

- **'abc'**: No debe coincidir: cadena alfabética, no coincide.
- **'1'**: No debe coincidir: número entero positivo, no coincide.
- **'-123.45'**: No debe coincidir: número decimal negativo, no coincide.
- **'0'**: No debe coincidir: cero no es un número decimal positivo, no coincide.
- **'0.0'**: No debe coincidir: cero decimal, no coincide.
- **'123.'**: No debe coincidir: punto decimal sin dígitos después, no debe coincidir.

Propósito de la Evaluación

La evaluación se realiza mediante la función `verificar_oraciones`, que compara cada cadena con el patrón definido. Esto es útil para validar entradas de usuario y asegurarse de que solo se acepten números decimales válidos, excluyendo los enteros y otros formatos inválidos.

6. Comprobación de Números Decimales Negativos

```
# Define un patrón regex para coincidir solo con números decimales negativos.
pattern = r'^-[0-9]+\.\d+$'  # Asegura que la cadena comience con un signo negativo, seguido de uno o más dígitos y un punto decimal con al menos un dígito.

# Define una lista de oraciones para ser comprobadas contra el patrón, junto con sus resultados esperados.
sentences = [
    ('-123.45', True),  # Debe coincidir: número decimal negativo, se espera que coincida.
    ('-10.5', True),    # Debe coincidir: número decimal negativo, se espera que coincida.
    ('-1.0', True),     # Debe coincidir: número decimal negativo, se espera que coincida.
    ('-0.01', True),    # Debe coincidir: número decimal negativo que empieza con cero, se espera que coincida.
    ('-9.99', True),    # Debe coincidir: número decimal negativo, se espera que coincida.

    ('abc', False),     # No debe coincidir: cadena alfabética, no coincide.
    ('1', False),       # No debe coincidir: número entero positivo, no coincide.
```

```
    ('123.45', False),  # No debe coincidir: número decimal positivo, no coincide.
    ('0', False),       # No debe coincidir: cero no es un número decimal negativo, no coincide.
    ('0.0', False),     # No debe coincidir: cero decimal, no coincide.
    ('-123.', False),   # No debe coincidir: punto decimal sin dígitos después, no debe coincidir.
]

# Llama a la función verificar_oraciones con el patrón y las oraciones para realizar las comprobaciones.
verificar_oraciones(pattern, sentences)
```

```
Aprobado --> -123.45 (Válido) | Coincide
Aprobado --> -10.5 (Válido) | Coincide
Aprobado --> -1.0 (Válido) | Coincide
Aprobado --> -0.01 (Válido) | Coincide
Aprobado --> -9.99 (Válido) | Coincide
No Aprobado --> abc (No Válido) | No coincide
No Aprobado --> 1 (No Válido) | No coincide
No Aprobado --> 123.45 (No Válido) | No coincide
No Aprobado --> 0 (No Válido) | No coincide
No Aprobado --> 0.0 (No Válido) | No coincide
No Aprobado --> -123. (No Válido) | No coincide
```

Explicación del Código

La función `verificar_oraciones` se utiliza para evaluar una lista de cadenas que representan números decimales negativos.

Detalles del Patrón Regex

- **Patrón**: `^-[0-9]+\.\d+$`
  - `^`: Indica el inicio de la cadena.
  - `-`: Un signo negativo que debe estar presente.
  - `[0-9]+`: Uno o más dígitos del 0 al 9 antes del punto decimal.
  - `\.`: Un punto decimal que debe estar presente.
  - `\d+$`: Uno o más dígitos después del punto decimal. Esto asegura que todos los números deben ser decimales y no enteros.

Lista de Frases

- La lista `sentences` contiene tuplas que consisten en una cadena y su resultado esperado (True o False).

Ejemplos que deben coincidir (True):

- **'-123.45'**: Debe coincidir: número decimal negativo, se espera que coincida.
- **'-10.5'**: Debe coincidir: número decimal negativo, se espera que coincida.

- **'-1.0'**: Debe coincidir: número decimal negativo, se espera que coincida.
- **'-0.01'**: Debe coincidir: número decimal negativo que empieza con cero, se espera que coincida.
- **'-9.99'**: Debe coincidir: número decimal negativo, se espera que coincida.

Ejemplos que no deben coincidir (False):

- **'abc'**: No debe coincidir: cadena alfabética, no coincide.
- **'1'**: No debe coincidir: número entero positivo, no coincide.
- **'123.45'**: No debe coincidir: número decimal positivo, no coincide.
- **'0'**: No debe coincidir: cero no es un número decimal negativo, no coincide.
- **'0.0'**: No debe coincidir: cero decimal, no coincide.
- **'-123.'**: No debe coincidir: punto decimal sin dígitos después, no debe coincidir.

Propósito de la Evaluación

La evaluación se realiza mediante la función `verificar_oraciones`, que compara cada cadena con el patrón definido. Esto es útil para validar entradas de usuario y asegurarse de que solo se acepten números decimales válidos, excluyendo los enteros y otros formatos inválidos.

7. Comprobación de Todos los Números Decimales

```
# Define un patrón regex para coincidir con números decimales, tanto positivos como negativos, incluyendo 0.0.
pattern = r'^[+-]?([0-9]*[.][0-9]+)$'  # Asegura que la cadena puede tener un signo opcional y debe tener un punto decimal con dígitos antes y después.

# Define una lista de oraciones para ser comprobadas contra el patrón, junto con sus resultados esperados.
sentences = [
    ('123.45', True),   # Debe coincidir: número decimal positivo, se espera que coincida.
    ('10.5', True),     # Debe coincidir: número decimal positivo, se espera que coincida.
    ('1.0', True),      # Debe coincidir: número decimal positivo, se espera que coincida.
    ('0.0', True),      # Debe coincidir: cero decimal, se espera que coincida.
    ('-0.01', True),    # Debe coincidir: número decimal negativo que empieza con cero, se espera que coincida.
    ('-10.5', True),    # Debe coincidir: número decimal negativo, se espera que coincida.
    ('-1.0', True),     # Debe coincidir: número decimal negativo, se espera que coincida.
    ('abc', False),     # No debe coincidir: cadena alfabética, no coincide.
    ('1', False),       # No debe coincidir: número entero positivo, no coincide.
    ('-123.45', False), # No debe coincidir: número decimal negativo,
```

```
no coincide porque tiene parte entera.
    ('0', False),       # No debe coincidir: cero no es un número
decimal, no coincide.
    ('123.', False),    # No debe coincidir: punto decimal sin dígitos
después, no debe coincidir.
    ('.5', False),      # No debe coincidir: punto decimal sin dígitos
antes, no debe coincidir.
]

# Llama a la función verificar_oraciones con el patrón y las oraciones
para realizar las comprobaciones.
verificar_oraciones(pattern, sentences)
```

```
Aprobado --> 123.45 (Válido) | Coincide
Aprobado --> 10.5 (Válido) | Coincide
Aprobado --> 1.0 (Válido) | Coincide
Aprobado --> 0.0 (Válido) | Coincide
Aprobado --> -0.01 (Válido) | Coincide
Aprobado --> -10.5 (Válido) | Coincide
Aprobado --> -1.0 (Válido) | Coincide
No Aprobado --> abc (No Válido) | No coincide
No Aprobado --> 1 (No Válido) | No coincide
Aprobado --> -123.45 (No Válido) | Coincide
No Aprobado --> 0 (No Válido) | No coincide
No Aprobado --> 123. (No Válido) | No coincide
Aprobado --> .5 (No Válido) | Coincide
```

Explicación del Código

La función `verificar_oraciones` se utiliza para evaluar una lista de cadenas que representan números decimales, tanto positivos como negativos, y también incluye **0.0** como válido.

Detalles del Patrón Regex

- **Patrón**: `^[+-]?([0-9]*[.][0-9]+)$`
  - `^`: Indica el inicio de la cadena.
  - `[+-]?`: Un signo positivo o negativo opcional.
  - `([0-9]*[.][0-9]+)`:
    - `[0-9]*`: Cero o más dígitos (esto permite que el número sea cero).
    - `[.]`: Un punto decimal que debe estar presente.
    - `[0-9]+`: Uno o más dígitos después del punto decimal. Esto asegura que los números no sean enteros y que siempre haya parte decimal.

Lista de Frases

- La lista `sentences` contiene tuplas que consisten en una cadena y su resultado esperado (True o False).

Ejemplos que deben coincidir (True):

- **'123.45'**: Debe coincidir: número decimal positivo, se espera que coincida.

- **'10.5'**: Debe coincidir: número decimal positivo, se espera que coincida.
- **'1.0'**: Debe coincidir: número decimal positivo, se espera que coincida.
- **'0.0'**: Debe coincidir: cero decimal, se espera que coincida.
- **'-0.01'**: Debe coincidir: número decimal negativo que empieza con cero, se espera que coincida.
- **'-10.5'**: Debe coincidir: número decimal negativo, se espera que coincida.
- **'-1.0'**: Debe coincidir: número decimal negativo, se espera que coincida.

Ejemplos que no deben coincidir (False):

- **'abc'**: No debe coincidir: cadena alfabética, no coincide.
- **'1'**: No debe coincidir: número entero positivo, no coincide.
- **'-123.45'**: No debe coincidir: número decimal negativo con parte entera, no coincide.
- **'0'**: No debe coincidir: cero no es un número decimal, no coincide.
- **'123.'**: No debe coincidir: punto decimal sin dígitos después, no debe coincidir.
- **'.5'**: No debe coincidir: punto decimal sin dígitos antes, no debe coincidir.

Propósito de la Evaluación

La evaluación se realiza mediante la función `verificar_oraciones`, que compara cada cadena con el patrón definido. Esto es útil para validar entradas de usuario y asegurarse de que solo se acepten números decimales válidos, excluyendo los enteros y otros formatos inválidos.

8. Validación de nombre de usuario

```
min_len = 5  # longitud mínima para un nombre de usuario válido
max_len = 15  # longitud máxima para un nombre de usuario válido

# Definir un patrón regex que permita letras, números, guiones y
guiones bajos con longitud entre min_len y max_len
pattern = r'^[\w_-]{' + str(min_len) + ',' + str(max_len) + '}$'

# Lista de frases con nombres de usuario y sus resultados esperados
sentences = [
    ('user123', True),              # Debe coincidir: válido, entre 5
y 15 caracteres
    ('123_user', True),             # Debe coincidir: válido, entre 5
y 15 caracteres
    ('Username_', True),            # Debe coincidir: válido, entre 5
y 15 caracteres
    ('valid_user-12', True),        # Debe coincidir: válido, entre 5
y 15 caracteres
    ('user', False),                # No debe coincidir: longitud
menor a 5 caracteres
    ('username1234_is-way-too-long', False),  # No debe coincidir:
longitud mayor a 15 caracteres
    ('user$34354', False),          # No debe coincidir: contiene
caracteres no permitidos ('$')
    ('use', False),                 # No debe coincidir: longitud
menor a 5 caracteres
```

```
    ('', False),                    # No debe coincidir: cadena vacía
    ('@user123', False)             # No debe coincidir: contiene un
carácter no permitido ('@')
]

# Llamar a la función verificar_oraciones para realizar la
comprobación
verificar_oraciones(pattern, sentences)
```

```
Aprobado --> user123 (Válido) | Coincide
Aprobado --> 123_user (Válido) | Coincide
Aprobado --> Username_ (Válido) | Coincide
Aprobado --> valid_user-12 (Válido) | Coincide
No Aprobado --> user (No Válido) | No coincide
No Aprobado --> username1234_is-way-too-long (No Válido) | No coincide
No Aprobado --> user$34354 (No Válido) | No coincide
No Aprobado --> use (No Válido) | No coincide
No Aprobado -->  (No Válido) | No coincide
No Aprobado --> @user123 (No Válido) | No coincide
```

Explicación del Código

La función `verificar_oraciones` evalúa una lista de cadenas (nombres de usuario) para determinar si coinciden con un patrón de expresión regular específico que define los caracteres permitidos y la longitud adecuada.

Detalles del Patrón Regex

- **Patrón**: `^[\w_-]{5,15}$`
  - `^`: Indica el inicio de la cadena.
  - `[\w_-]`: Permite letras (mayúsculas y minúsculas), números, guiones bajos (_) y guiones (-).
  - `{5,15}`: Define que la longitud del nombre de usuario debe estar entre 5 y 15 caracteres.
  - `$`: Indica el final de la cadena.

El patrón garantiza que solo se acepten nombres de usuario que cumplan con las siguientes condiciones:

- La longitud esté entre 5 y 15 caracteres.
- Contengan únicamente letras, números, guiones y guiones bajos.

Lista de Frases

La lista `sentences` contiene tuplas con el nombre de usuario y el resultado esperado (True o False). Cada cadena se compara con el patrón regex para validar si es correcta.

Ejemplos que deben coincidir (True):

- **'user123'**: Debe coincidir: nombre de usuario válido con 7 caracteres y caracteres permitidos.

- **'123_user'**: Debe coincidir: nombre de usuario válido con 8 caracteres y caracteres permitidos.
- **'Username_'**: Debe coincidir: nombre de usuario válido con 9 caracteres, dentro del rango permitido.
- **'valid_user-12'**: Debe coincidir: nombre de usuario válido con guiones y guiones bajos, con longitud dentro del límite.

Ejemplos que no deben coincidir (False):

- **'user'**: No debe coincidir: tiene menos de 5 caracteres.
- **'username1234_is-way-too-long'**: No debe coincidir: excede el límite de 15 caracteres.
- **'user$34354'**: No debe coincidir: contiene un carácter no permitido (`$`).
- **'use'**: No debe coincidir: longitud menor a 5 caracteres.
- **''**: No debe coincidir: cadena vacía.
- **'@user123'**: No debe coincidir: contiene un símbolo no permitido (@).

Propósito de la Evaluación

El propósito de esta validación es asegurar que solo se acepten nombres de usuario que cumplan con las reglas de formato. Esto es útil en aplicaciones web, formularios de registro o sistemas donde se requiera validar nombres de usuario según reglas específicas de longitud y caracteres permitidos.

9. Comprobación de direcciones de correo electrónico válidas

Una expresión regular que captura la mayoría de las direcciones de correo electrónico.

```
# Definir el patrón regex corregido para direcciones de correo electrónico válidas
pattern = r'(?i)^[\w.-]+@[\w-]+\.[a-z]{2,3}$'

# Crear una lista de frases (direcciones de correo) con el resultado esperado (True o False)
sentences = [
    ('user123@mail.com', True),        # Debe coincidir: correo estándar válido
    ('jane.doe@mail.org', True),       # Debe coincidir: correo válido con punto en el nombre
    ('user-name@mail.co', True),       # Debe coincidir: correo válido con guion en el nombre
    ('contact123@mail.us', True),      # Debe coincidir: correo válido con dominio .us
    ('example@mail.ac', True),         # Debe coincidir: correo válido con dominio .ac
    ('test_user123@mail.info', False), # No debe coincidir: el dominio .info no es válido (más de 3 letras)
    ('user@mailcom', False),           # No debe coincidir: dominio de nivel superior inválido (falta el punto)
    ('user@.com', False),              # No debe coincidir: falta el nombre de dominio antes del punto
```

```
    ('@mail.com', False),              # No debe coincidir: falta el
nombre de usuario antes del @
    ('user@mail.com.', False),         # No debe coincidir: punto
final no permitido en el dominio
    ('user$123@mail.com', False),      # No debe coincidir: carácter
no permitido ($) en el nombre de usuario
    ('user@mail..com', False)          # No debe coincidir: dos puntos
seguidos en el dominio no permitidos
]

# Llamar a la función verificar_oraciones con el patrón y las frases
verificar_oraciones(pattern, sentences)
```

```
Aprobado --> user123@mail.com (Válido) | Coincide
Aprobado --> jane.doe@mail.org (Válido) | Coincide
Aprobado --> user-name@mail.co (Válido) | Coincide
Aprobado --> contact123@mail.us (Válido) | Coincide
Aprobado --> example@mail.ac (Válido) | Coincide
No Aprobado --> test_user123@mail.info (No Válido) | No coincide
No Aprobado --> user@mailcom (No Válido) | No coincide
No Aprobado --> user@.com (No Válido) | No coincide
No Aprobado --> @mail.com (No Válido) | No coincide
No Aprobado --> user@mail.com. (No Válido) | No coincide
No Aprobado --> user$123@mail.com (No Válido) | No coincide
No Aprobado --> user@mail..com (No Válido) | No coincide
```

Explicación del Código

La función `verificar_oraciones` se utiliza para evaluar una lista de cadenas que representan direcciones de correo electrónico, asegurando que solo se acepten direcciones válidas.

Detalles del Patrón Regex

- **Patrón**: `(?i)^[\w.-]+@[\w-]+\.[a-z]{2,3}$`
  - `(?i)`: Hace que la expresión sea insensible a mayúsculas y minúsculas.
  - `^`: Indica el inicio de la cadena.
  - `[\w.-]+`:
    - Permite letras, números, guiones y puntos en la parte local (antes del símbolo `@`).
    - Se asegura de que no haya dos puntos consecutivos.
  - `@`: El símbolo arroba, que debe estar presente en todas las direcciones de correo electrónico.
  - `[\w-]+`: Permite letras, números y guiones en la parte del dominio (después del símbolo `@`).
  - `\.`: Un punto que debe estar presente antes de la extensión del dominio.
  - `[a-z]{2,3}`: La extensión del dominio debe consistir en 2 o 3 letras, asegurando que no se acepten extensiones más largas.

Lista de Frases

- La lista `sentences` contiene tuplas que consisten en una cadena (dirección de correo electrónico) y su resultado esperado (True o False).

Ejemplos que deben coincidir (True):

- **`user123@mail.com`**: Debe coincidir: dirección de correo electrónico estándar válida.
- **`jane.doe@mail.org`**: Debe coincidir: dirección válida con un punto en el nombre.
- **`user-name@mail.co`**: Debe coincidir: dirección válida con un guion en el nombre.
- **`contact123@mail.us`**: Debe coincidir: dirección válida con un dominio de nivel superior .us.
- **`example@mail.ac`**: Debe coincidir: dirección válida con un dominio de nivel superior .ac.

Ejemplos que no deben coincidir (False):

- **`test_user123@mail.info`**: No debe coincidir: el dominio .info no es válido (más de 3 letras).
- **`user@mailcom`**: No debe coincidir: dominio de nivel superior inválido (falta el punto).
- **`user@.com`**: No debe coincidir: falta el nombre de dominio antes del punto.
- **`@mail.com`**: No debe coincidir: falta el nombre de usuario antes del `@`.
- **`user@mail.com.`**: No debe coincidir: punto final no permitido en el dominio.
- **`user$123@mail.com`**: No debe coincidir: carácter no permitido (`$`) en el nombre de usuario.
- **`user@mail..com`**: No debe coincidir: dos puntos seguidos en el dominio no permitidos.

Propósito de la Evaluación

La evaluación se realiza mediante la función `verificar_oraciones`, que compara cada cadena con el patrón definido. Esto es útil para validar entradas de usuario y asegurarse de que solo se acepten direcciones de correo electrónico válidas, excluyendo formatos inválidos y errores comunes en las entradas.

10. Nombres de sitios web

Define el patrón para detectar si una cadena corresponde al nombre de un sitio web. El patrón sigue estas reglas:

- Puede comenzar con tres "w" o directamente con el nombre del dominio.
- Le sigue el nombre del dominio, que puede contener letras y números.
- Puede tener un máximo de 2 subdominios (compuestos por letras y números).
- Termina con un punto seguido de 2 o 3 letras.

Deberías detectar los siguientes casos:

- Positivos:
  - www.ds.com
  - www.data.science.com
  - datascience.com

    - wab.a.com
- Negativos:
    - ww.4com
    - www.ww.a
    - www.d.s.c.d.com

```
# Definir el patrón regex para nombres de sitios web
pattern = r'^(www\.)?([a-zA-Z0-9]+(\.[a-zA-Z0-9]+){0,2})(\.[a-z]{2,3})
$'

# Crear una lista de frases (nombres de sitios web) con el resultado
esperado (True o False)
sentences = [
    ('www.ds.com', True),                # Debe coincidir: nombre de
dominio válido con www
    ('www.data.science.com', True),      # Debe coincidir: nombre con
subdominios
    ('datascience.com', True),            # Debe coincidir: nombre de
dominio sin www
    ('wab.a.com', True),                  # Debe coincidir: nombre con
un subdominio
    ('ww.4com', False),                   # No debe coincidir: formato
inválido (solo dos letras)
    ('www.ww.a', False),                  # No debe coincidir: falta
la extensión de dominio
    ('www.d.s.c.d.com', False)            # No debe coincidir: más de
dos subdominios
]

# Llamar a la función verificar_oraciones con el patrón y las frases
verificar_oraciones(pattern, sentences)
```

```
Aprobado --> www.ds.com (Válido) | Coincide
Aprobado --> www.data.science.com (Válido) | Coincide
Aprobado --> datascience.com (Válido) | Coincide
Aprobado --> wab.a.com (Válido) | Coincide
No Aprobado --> ww.4com (No Válido) | No coincide
No Aprobado --> www.ww.a (No Válido) | No coincide
No Aprobado --> www.d.s.c.d.com (No Válido) | No coincide
```

Explicación del Código

La función `verificar_oraciones` se utiliza para evaluar una lista de cadenas que representan nombres de sitios web, asegurando que solo se acepten formatos válidos.

Detalles del Patrón Regex

- **Patrón**: `^(www\.)?([a-zA-Z0-9]+(\.[a-zA-Z0-9]+){0,2})(\.[a-z]{2,3})$`
    - `^`: Indica el inicio de la cadena.
    - `(www\.)?`: Permite que la cadena comience opcionalmente con "www.".

- `([a-zA-Z0-9]+(\.[a-zA-Z0-9]+){0,2})`:
  - `([a-zA-Z0-9]+)`: Asegura que haya al menos un carácter alfanumérico.
  - `(\.[a-zA-Z0-9]+){0,2}`: Permite hasta dos subdominios (cada uno precedido por un punto).
- `(\.[a-z]{2,3})`:
  - `\.`: Un punto que debe estar presente.
  - `[a-z]{2,3}`: La extensión del dominio debe consistir en 2 o 3 letras.

Lista de Frases

- La lista `sentences` contiene tuplas que consisten en una cadena (nombre de sitio web) y su resultado esperado (True o False).

Ejemplos que deben coincidir (True):

- **`www.ds.com`**: Debe coincidir: nombre de dominio válido con www.
- **`www.data.science.com`**: Debe coincidir: nombre válido con subdominios.
- **`datascience.com`**: Debe coincidir: nombre de dominio sin www.
- **`wab.a.com`**: Debe coincidir: nombre válido con un subdominio.

Ejemplos que no deben coincidir (False):

- **'ww.4com`**: No debe coincidir: formato inválido (solo dos letras).
- **`www.ww.a`**: No debe coincidir: falta la extensión de dominio.
- **`www.d.s.c.d.com`**: No debe coincidir: más de dos subdominios.

Propósito de la Evaluación

La evaluación se realiza mediante la función `verificar_oraciones`, que compara cada cadena con el patrón definido. Esto es útil para validar entradas de usuario y asegurarse de que solo se acepten nombres de sitios web válidos, excluyendo formatos inválidos y errores comunes en las entradas.