ÍNDICE

1. Introducción

En esta sección, exploraremos sentencias más avanzadas que pueden aplicarse en nuestros programas. Comenzaremos por analizar el uso de expresiones regulares para buscar patrones dentro de textos. A continuación, abordaremos cómo se manejan los errores en Python y las estrategias que podemos emplear para evitar que la ejecución del programa se detenga abruptamente. Finalmente, nos sumergiremos en el concepto de compresión de listas, una técnica simple pero poderosa para crear listas al aplicar operaciones a los elementos de otra secuencia.

1.1. Objetivos

- Familiarizarnos con las funciones para buscar y modificar elementos en textos.
- Aprender la sintaxis de las expresiones regulares para definir patrones de texto.
- Aplicar estructuras que nos permitan controlar y gestionar errores en nuestros programas de manera efectiva.
- Dominar la creación de diversas estructuras de datos mediante la técnica de compresión.

2. Expresiones Regulares

Las expresiones regulares (también conocidas como regex o regexp) son secuencias de caracteres que forman un patrón de búsqueda. Estos patrones se utilizan para buscar, validar o manipular texto de una manera más flexible y poderosa que las simples búsquedas de cadenas exactas. A continuación, exploraremos algunos aspectos clave sobre las expresiones regulares

- **Sintaxis básica:** Las expresiones regulares se construyen utilizando una combinación de caracteres literales (que coinciden exactamente) y metacaracteres (que tienen un significado especial). Por ejemplo, el metacaracter "." coincide con cualquier carácter, y el metacaracter "\d" coincide con un dígito
- **Coincidencia de patrones:** Las expresiones regulares te permiten buscar patrones específicos en cadenas de texto. Puedes buscar palabras, números, direcciones de correo electrónico, formatos de fecha y mucho más. Por ejemplo, puedes usar la expresión regular "\b\w+\b" para buscar todas las palabras en una cadena.
- **Modificadores:** Puedes utilizar modificadores para hacer que las expresiones regulares sean insensibles a mayúsculas y minúsculas, para buscar múltiples ocurrencias de un patrón y para realizar búsquedas más amplias. Por ejemplo, el modificador "i" hace que la búsqueda sea insensible a mayúsculas y minúsculas.
- **Grupos y captura:** Puedes usar paréntesis para crear grupos de captura, lo que te permite extraer partes específicas del texto que coinciden con esos grupos. Esto es útil cuando necesitas procesar partes específicas del texto que has encontrado.
- **Sustitución:** Las expresiones regulares no solo se utilizan para buscar, sino también para reemplazar. Puedes buscar un patrón y reemplazarlo con otro texto, lo que es útil para realizar cambios en el formato de los datos.

2.1. Librería *re* de Python

En Python, puedes utilizar la librería re para trabajar con expresiones regulares. Proporciona funciones para buscar, coincidir, dividir y reemplazar texto utilizando patrones de expresiones regulares.

- **search:** Este método busca el patrón dado en toda la cadena y devuelve un objeto de coincidencia si se encuentra una coincidencia en cualquier lugar. Si no se encuentra ninguna coincidencia, devuelve None.

```
import re

text = "busca el patrón dado en toda la cadena"
pattern = r"patrón"
result = re.search(pattern, text)
if result:
    print("Se encontró una coincidencia:", result.group())
else:
    print("No se encontró ninguna coincidencia")
```

En este ejemplo, search encuentra la palabra " patrón " en el texto y devuelve un objeto de coincidencia. El método group() se utiliza para obtener la cadena que coincide con el patrón.

- **match**: Comprueba si el patrón coincide al principio de la cadena. Devuelve un objeto de coincidencia si hay una coincidencia, de lo contrario, devuelve None.

```
import re

text = "Comprueba si el patrón coincide al principio de la
cadena"
pattern = r"patrón"
result = re.match(pattern, text)
if result:
    print("Se encontró una coincidencia:", result.group())
else:
    print("No se encontró ninguna coincidencia al principio")
```

En este caso, match no encuentra una coincidencia al principio de la cadena porque " patrón " no está al principio.

- **findall:** Busca todas las ocurrencias del patrón en la cadena y devuelve una lista con todas las coincidencias.

```
import re

text = "gata gato patrón"
pattern = r"gat\w+"
result = re.findall(pattern, text)
print("Coincidencias encontradas:", result)
```

En este ejemplo, findall encuentra todas las palabras que comienzan con "gat" seguidas de cualquier palabra (carácter de palabra).

- **split:** Divide la cadena en una lista utilizando el patrón como separador y devuelve la lista resultante.

```
import re

text = "apple,banana,grape,orange"
pattern = r","
result = re.split(pattern, text)
print("Lista después de dividir:", result)
```

Aquí, split divide la cadena en elementos de la lista cada vez que encuentra una coma.

- **sub:** Reemplaza todas las ocurrencias del patrón en la cadena con el texto proporcionado y devuelve la cadena resultante.

```
import re

text = "Hola, mundo!"
pattern = r"mundo"
replacement = "amigos"
result = re.sub(pattern, replacement, text)
print("Texto después del reemplazo:", result)

# Texto después del reemplazo: Hola, amigos!
```

Estos son los conceptos clave y ejemplos de los métodos principales de la librería re en Python. La manipulación de expresiones regulares puede volverse más compleja a medida que trabajas con patrones más elaborados y tareas más específicas.

2.2. Componentes Principales

A. Literales

- Los literales son caracteres que representan ellos mismos en una cadena de texto. En otras palabras, un literal es un carácter que se debe coincidir exactamente tal como está escrito.
- Por ejemplo, en la expresión regular apple, "apple" es un literal. Esto significa que la expresión buscará exactamente la palabra "apple" en el texto.
- Los caracteres alfabéticos, numéricos y la mayoría de los caracteres especiales son considerados literales.

B. Caracteres de escape

Los caracteres de escape son especialmente útiles cuando trabajas con expresiones regulares y cadenas de texto, ya que permiten incluir caracteres especiales sin que sean interpretados de manera incorrecta. Por ejemplo, si intentas buscar el carácter de paréntesis (literal en una expresión regular, debes usar \(para evitar que se interprete como un grupo de captura.

- Los caracteres de escape son caracteres que tienen un significado especial en expresiones regulares y en muchas otras situaciones en programación y en cadenas de texto.
- Estos caracteres se utilizan para representar elementos que no se pueden incluir literalmente en una cadena, como saltos de línea, tabulaciones, comillas, etc.
- En las expresiones regulares, el carácter de escape más común es la barra invertida \.
- Algunos caracteres de escape comunes son:
    - \n: Representa un salto de línea.
    - \t: Representa una tabulación.
    - \": Representa una comilla doble.

- \': Representa una comilla simple.
- \\: Representa una barra invertida literal.

Por ejemplo, si quieres buscar una cita dentro de un texto, podrías usar la expresión regular \".*?\" para encontrar cualquier contenido entre comillas dobles.

```
import re

text = '"Juan Pérez", "María López", "Carlos Gómez"'
pattern = r'"(.*?)"'  # El patrón busca cualquier contenido entre
comillas

result = re.findall(pattern, text)
print("Nombres completos encontrados:", result)
```

Supongamos que tienes el siguiente texto que contiene una lista de direcciones de correo electrónico:

```
import re

text = """alice@example.com
bob123@gmail.com
carol_45@yahoo.com"""

pattern = r'(\w+)@'  # El patrón busca cualquier palabra antes del símbolo
"@"

result = re.findall(pattern, text)
print("Nombres de usuario encontrados:", result)
```

C. Grupos de caracteres:

- Un grupo de caracteres es una forma de especificar un conjunto de caracteres que deseas buscar en una determinada posición en el texto.
- Se definen utilizando corchetes [] y puedes enumerar los caracteres que deseas que coincidan en esa posición.
- También puedes definir rangos de caracteres utilizando el guion -. Por ejemplo, [a-z] representa cualquier letra minúscula del alfabeto.

- Los grupos de caracteres permiten que una posición en el texto coincida con cualquiera de los caracteres enumerados en el grupo.

Por ejemplo, [aeiou] coincidiría con cualquier vocal en una cadena.

```
import re

text = "The quick brown fox jumps over the lazy dog."
pattern = r"[aeiou]"
result = re.findall(pattern, text)
print("Vocales encontradas:", result)
```

D. Metacaracteres:

- Los metacaracteres son caracteres con un significado especial en las expresiones regulares y se utilizan para definir patrones más complejos.
- Algunos metacaracteres comunes incluyen:
- .: Coincide con cualquier carácter excepto el salto de línea.
- *: Coincide con cero o más repeticiones del elemento anterior.
- +: Coincide con una o más repeticiones del elemento anterior.
- ?: Coincide con cero o una repetición del elemento anterior.
- ^: Coincide con el principio de una línea.
- $: Coincide con el final de una línea.
- |: Utilizado para alternancia, es decir, para buscar cualquiera de las expresiones separadas por el operador.
- (): Utilizados para agrupar elementos y aplicar operadores a grupos enteros.

Por ejemplo, a.*b coincidiría con cualquier cadena que comience con "a" y termine con "b", sin importar los caracteres intermedios.

```
import re

text = "apple banana cherry date"
pattern = r"\b\w{5}\b"
result = re.findall(pattern, text)
print("Palabras de 5 letras:", result)
```

3. Errores y Excepciones

En la programación, los errores son situaciones en las que el código no puede ejecutarse correctamente debido a diversos motivos, como sintaxis incorrecta, problemas lógicos o condiciones imprevistas. Para manejar estas situaciones, Python ofrece un mecanismo llamado "excepciones", que son eventos excepcionales que pueden ocurrir durante la ejecución del programa y que pueden ser capturados y manejados para evitar que el programa se detenga abruptamente.

3.1. Tipos de Errores:

- **Errores de Sintaxis**: Estos ocurren cuando el código no sigue las reglas de sintaxis del lenguaje.

```
if x > 5
    print("Mayor que 5")
```

- **Errores de Nombre (NameError):** Estos ocurren cuando se intenta acceder a una variable o función que no está definida

```
print(variable_no_definida)
```

- **Errores de Tipo (TypeError):** Estos ocurren cuando se realiza una operación en tipos de datos incompatibles.

```
resultado = "2" + 3
```

- **Errores de índice:** Estos ocurren cuando se intenta acceder a una posición que no existe.

```
lista = []
lista[2]
```

3.2. Tipos de Excepciones:

Las excepciones son eventos que ocurren durante la ejecución del programa y que interrumpen el flujo normal de ejecución. Python proporciona una forma de manejar estas excepciones utilizando bloques *try–except–else-finally*.

- **ZeroDivisionError:** Ocurre cuando se intenta dividir por cero. Puedes manejarlo con un bloque try y except:

```
try:
    resultado = 10 / 0
except ZeroDivisionError:
    print("División por cero no permitida")
```

- **ValueError:** Ocurre cuando se intenta convertir un valor a un tipo incorrecto. Puedes manejarlo de esta manera:

```
try:
    numero = int("abc")
except ValueError:
    print("No se puede convertir a entero")
```

- **IndexError:** Ocurre cuando se intenta acceder a un índice que está fuera del rango válido en una lista o secuencia.

```
lista = [1, 2, 3]
try:
    elemento = lista[10]
except IndexError:
    print("Índice fuera de rango")
```

- **KeyError:** Ocurre cuando se intenta acceder a una clave que no está presente en un diccionario.

```
diccionario = {"nombre": "Juan", "edad": 25}
try:
    valor = diccionario["apellido"]
except KeyError:
    print("Clave no encontrada")
```

- **FileNotFoundError:** Ocurre cuando se intenta acceder a un archivo que no existe.

```
try:
    archivo = open("archivo_inexistente.txt", "r")
except FileNotFoundError:
    print("El archivo no existe")
```

- **TypeError:** Ocurre cuando se realiza una operación en tipos de datos incompatibles.

```
try:
    resultado = "2" + 3
except TypeError:
    print("Operación entre tipos incompatibles")
```

- **Manejo de Excepciones en un Bloque else y finally:** Los bloques else y finally proporcionan aún más control sobre el flujo de manejo de excepciones:
    - El bloque else se ejecuta si no se produce ninguna excepción en el bloque try.
    - El bloque finally siempre se ejecuta, independientemente de si se lanzó una excepción o no.

```
try:
    resultado = 10 / 2
except ZeroDivisionError:
    print("División por cero")
else:
    print("Resultado:", resultado)
finally:
    print("Terminando proceso")
```

En este ejemplo, el bloque else imprimirá el resultado si no se lanza una excepción, y el bloque finally siempre imprimirá "Terminando proceso".

```
try:
    archivo = None
    try:
```

```
        archivo = open("archivo.txt", "r")
        contenido = archivo.read()
    except FileNotFoundError:
        print("El archivo no existe")
    else:
        print("Contenido:", contenido)
finally:
    if archivo:
        archivo.close()
    print("Liberando recursos")
```

La variable archivo se inicializa con None fuera del bloque try. Luego, dentro del bloque try, se asigna el objeto de archivo real si la operación tiene éxito. Finalmente, en el bloque finally, verificamos si archivo tiene un valor (es decir, si se abrió el archivo) y luego lo cerramos. De esta manera, evitamos el NameError incluso si se produce una excepción antes de que el archivo se abra.

- **Errores Controlados:** Puedes lanzar excepciones manualmente utilizando la palabra clave raise para indicar que algo inesperado ocurrió. Esto te permite definir tus propias excepciones para situaciones específicas.

```
def dividir(a, b):
    if b == 0:
        raise ZeroDivisionError("División por cero no permitida")
    return a / b

try:
    resultado = dividir(10, 0)
except ZeroDivisionError as e:
    print("Excepción:", e)
```

En este ejemplo, la función dividir lanza una excepción ZeroDivisionError cuando se intenta dividir por cero. Luego, el bloque try y except captura esa excepción y muestra un mensaje personalizado.

Además de usar raise para lanzar excepciones, también puedes definir tus propias clases de excepción personalizadas. Esto te permite crear excepciones más específicas para tu aplicación:

```python
class MiErrorPersonalizado(Exception):
    def __init__(self, mensaje):
        self.mensaje = mensaje

def funcion_personalizada():
    raise MiErrorPersonalizado("Ocurrió un error personalizado")

try:
    funcion_personalizada()
except MiErrorPersonalizado as e:
    print("Excepción personalizada:", e.mensaje)
```

4. Compresión de Listas

La compresión de listas es una forma elegante de crear y transformar listas en una sola línea de código. En lugar de utilizar bucles tradicionales, puedes aprovechar esta técnica para simplificar tu código y hacerlo más legible.

4.1. Sintaxis:

```
nueva_lista = [expresion for elemento in secuencia if condicion]
```

- **expresion:** La operación que deseas realizar en cada elemento.
- **elemento:** Variable que toma los valores de la secuencia.
- **secuencia:** La secuencia de elementos sobre la cual iterar.
- **condicion:** (Opcional) Una condición para filtrar elementos.

```
numeros = [1, 2, 3, 4, 5]
cuadrados = [x ** 2 for x in numeros]
cuadrados
```

4.2. Tipos de Compresión

A. Compresión de Listas con Condición:

La compresión de listas también puede incluir una condición para filtrar los elementos que se agregarán a la nueva lista. En este ejemplo, se creó una lista de números pares del 1 al 10 utilizando la condición x % 2 == 0 para seleccionar solo los números pares. Esto resulta en una lista que contiene [2, 4, 6, 8, 10].

```
pares = [x for x in range(1, 11) if x % 2 == 0]
pares
```

B. Compresión de Listas con Transformación:

La compresión de listas no se limita a operaciones numéricas. En este ejemplo, se toma una palabra y se crea una lista de las letras en mayúsculas. Cada letra de la palabra original se transforma a mayúscula usando el método .upper().

```
palabra = "compresion"
mayusculas = [letra.upper() for letra in palabra]
mayúsculas
```

C. Compresión de Listas con Diccionarios:

La compresión de listas también puede aplicarse a la creación de diccionarios. En este ejemplo, se utiliza la función enumerate() para obtener tanto el índice como el valor de cada letra en la palabra. Luego, se crea un diccionario donde las letras son las claves y los índices son los valores.

```
palabra = "compresion"
diccionario = {letra: indice for indice, letra in enumerate(palabra)}
diccionario
```

D. Compresión de Listas con Conjuntos:

Además de crear listas, la compresión de listas también se puede usar para generar conjuntos (sets). Esto es útil para eliminar duplicados y mantener elementos únicos.

```
numeros = [1, 2, 2, 3, 4, 4, 5, 6]
conjunto_unicos = {numero for numero in numeros}
conjunto_unicos
```

E. Compresión de Listas Anidadas:

La compresión de listas no se limita a crear listas simples. También puedes utilizarla para crear listas anidadas, es decir, listas que contienen otras listas como elementos. Esto es útil cuando deseas aplicar operaciones en múltiples niveles de datos.

```
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
transpuesta = [[fila[i] for fila in matriz] for i in range(3)]
transpuesta
```