PYTHON 🐍 - Lección 13: EXPRESIONES REGULARES

Presentación basada en el curso de Alex Narváez Programming

Introducción a las Expresiones Regulares

Las expresiones regulares (RegEx) son un lenguaje de patrones utilizado para encontrar cadenas de caracteres específicas dentro de un texto. Son extremadamente útiles para validación de datos, búsqueda y manipulación de texto, y extracción de información.

Algunos usos comunes incluyen:

- Validación de formularios (email, contraseñas, números de teléfono)
- Extraer información estructurada de texto plano
- Verificar formatos de datos (fechas, números, códigos)
- Reemplazar patrones específicos en textos

¿Por qué aprender expresiones regulares?
Las expresiones regulares son una herramienta poderosa que permiten realizar operaciones complejas de búsqueda y manipulación de texto que serían muy difíciles de programar manualmente.

Meta Caracteres

Los meta caracteres son símbolos especiales que tienen un significado específico en las expresiones regulares:

Caracteres Básicos

Carácter	Descripción
\d	Cualquier dígito (0-9)
\D	Cualquier caracter que no sea un dígito
\w	Cualquier caracter alfanumérico (a-z, A-Z, 0-9, _)
\W	Cualquier caracter que no sea alfanumérico
\s	Cualquier espacio en blanco (espacio, tab, nueva línea)
\S	Cualquier caracter que no sea espacio en blanco
\b	Límite de palabra
\.	Punto literal (escapado)

Cuantificadores

Carácter	Descripción
*	0 o más veces (0 o más repeticiones)
+	1 o más veces (al menos una repetición)
?	0 o 1 vez (opcional)
{n}	Exactamente n veces
{n,}	Al menos n veces
{n,m}	Entre n y m veces

Clases y Rangos

Carácter	Descripción
[abc]	Cualquier caracter dentro del conjunto (a, b o c)
[^abc]	Cualquier caracter que no esté en el conjunto
[a-z]	Cualquier caracter en el rango (a hasta z)
[A-Z]	Cualquier caracter en el rango (A hasta Z)
[0-9]	Cualquier dígito entre 0 y 9

Anclajes y Posiciones

Carácter	Descripción
^	Inicio de la cadena
$	Final de la cadena
.	Cualquier caracter (excepto nueva línea)
|	Alternancia (OR)
()	Agrupar expresiones y capturar
(?:...)	Grupo sin captura

Implementación en Python: Módulo re

Python proporciona el módulo `re` para trabajar con expresiones regulares. Este módulo contiene funciones para buscar patrones, extraer información y manipular texto.

Funciones principales:

- `re.search(pattern, string)`: Busca la primera ocurrencia del patrón en la cadena
- `re.match(pattern, string)`: Verifica si el patrón coincide al inicio de la cadena
- `re.findall(pattern, string)`: Encuentra todas las ocurrencias del patrón
- `re.finditer(pattern, string)`: Devuelve un iterador sobre todas las coincidencias
- `re.sub(pattern, repl, string)`: Sustituye las coincidencias por el texto especificado
- `re.split(pattern, string)`: Divide la cadena por las coincidencias del patrón
- `re.compile(pattern)`: Pre-compila un patrón para su reutilización

```
# Importar el módulo re
import re

# Ejemplo básico: verificar si hay una vocal en una cadena
cadena = "lol"
resultado = re.search(r"[aeiou]+", cadena)

if resultado:
    print(f"Se encontró una vocal en la posición {resultado.span()}")
    print(f"Coincidencia: {resultado.group()}")
else:
    print("No se encontraron vocales")
```

Nota: El prefijo `r` antes de la cadena indica que es una "raw string", lo que evita que Python interprete las secuencias de escape como \n o \t.

Ejemplos Prácticos

Ejemplo 1: Grupos de captura

Los grupos de captura permiten extraer partes específicas que coinciden con un patrón.

```
import re

cadena = "Alexander Narváez"
patron = re.compile(r"(?P<nombre>\w+)\s+(?P<apellido>\w+)")
resultado = patron.search(cadena)

if resultado:
    print(f"Nombre: {resultado.group('nombre')}")
    print(f"Apellido: {resultado.group('apellido')}")
```

Este ejemplo extrae el nombre y apellido de una cadena, utilizando grupos con nombre.

Ejemplo 2: Validación de fechas

Validar que una fecha esté en formato correcto DD/MM/YYYY y que sea una fecha válida.

```
import re

def validar_fecha(fecha):
    # Expresión simple que verifica el formato
    patron_formato = r"^\d{2}/\d{2}/\d{4}$"

    if not re.match(patron_formato, fecha):
        return False

    # Expresión avanzada que valida el rango de días y meses
    patron_avanzado = r"^(0[1-9]|[12]\d|3[01])/(0[1-9]|1[0-2])/((19|20)\d{2})$"

    if not re.match(patron_avanzado, fecha):
        return False

    # Verifica si es una fecha válida (ejemplo: 31/02 no existe)
    day, month, year = map(int, fecha.split('/'))

    if month in [4, 6, 9, 11] and day > 30:
        return False
    if month == 2:
        # Verificación de año bisiesto
        is_leap = (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
        if (is_leap and day > 29) or (not is_leap and day > 28):
            return False

    return True

# Ejemplos de uso
fechas = ["29/02/2020", "31/02/2020", "31/04/2020", "31/12/2020"]
for fecha in fechas:
    if validar_fecha(fecha):
        print(f"{fecha} es una fecha válida")
    else:
        print(f"{fecha} no es una fecha válida")
```

Ejemplo 3: Validación de contraseñas

Un ejemplo complejo para validar contraseñas según reglas específicas.

```
import re

def validar_contrasena(password):
    """
    Valida una contraseña según las siguientes reglas:
    1. Debe contener al menos una vocal
    2. No puede contener tres vocales consecutivas
    3. No puede contener tres consonantes consecutivas
    4. No puede repetir la misma letra dos veces consecutivas, excepto 'e' y 'o'
    """
    # Verificar que contiene al menos una vocal
    if not re.search(r"[aeiou]+", password):
        return False

    # Verificar que no contiene tres vocales consecutivas
    if re.search(r"[aeiou]{3}", password):
        return False

    # Verificar que no contiene tres consonantes consecutivas
    if re.search(r"[bcdfghjklmnpqrstvwxyz]{3}", password):
        return False

    # Verificar que no se repite la misma letra dos veces, excepto 'e' y 'o'
    pattern = r"([abcdfghijklmnpqrstuvwxyz])\1|([e])\2{2,}|([o])\3{2,}"
    if re.search(pattern, password, re.IGNORECASE):
        return False

    return True

# Ejemplos de uso
passwords = ["tv", "ptuui", "bontres", "o", "abc", "soonxx", "hoootowel"]
for pwd in passwords:
    if validar_contrasena(pwd):
        print(f"<{pwd}> es aceptable")
    else:
        print(f"<{pwd}> no es aceptable")
```

Este ejemplo está basado en el ejercicio mostrado en el video, donde se validan contraseñas según reglas específicas.

Ejemplo 4: Reconocimiento de tipos de números

Identificar si un número está en formato decimal, binario, octal o hexadecimal.

```
import re

def identificar_tipo_numero(numero):
    # Verificar si es hexadecimal (comienza con 0x o 0X seguido de dígitos o letras a-f)
    if re.match(r"^0[xX][0-9a-fA-F]+$", numero):
        return "hexadecimal"

    # Verificar si es binario (comienza con 0b o 0B seguido de 0s y 1s)
    elif re.match(r"^0[bB][01]+$", numero):
        return "binario"

    # Verificar si es octal (comienza con 0 seguido de dígitos 0-7)
    elif re.match(r"^0[0-7]+$", numero):
        return "octal"

    # Verificar si es decimal (solo dígitos, puede comenzar o no con 0)
    elif re.match(r"^[0-9]+$", numero):
        return "decimal"

    else:
        return "formato desconocido"

# Ejemplos de uso
numeros = ["123", "0123", "0x1A", "0b101", "0o777", "ABC", "0b102"]
for num in numeros:
    print(f"{num} es un número {identificar_tipo_numero(num)}")
```

Herramientas para Expresiones Regulares

Herramientas Online

Estas son algunas herramientas útiles para probar y visualizar expresiones regulares:

Regex101
La más completa. Prueba expresiones, explica cada parte y muestra coincidencias en tiempo real.

Regexper
Genera diagramas visuales de expresiones regulares para entender su funcionamiento.

Debuggex
Depurador visual que permite entender cómo funciona cada parte de la expresión.

Entornos de Desarrollo

Algunas opciones para practicar y trabajar con Python:

OnlineGDB
IDE online para Python y otros lenguajes de programación.

CodeAbbey
Sitio con problemas de programación, incluyendo retos sobre expresiones regulares.

Recomendación: El libro "Regular Expression Cookbook" es un excelente recurso para aprender patrones de expresiones regulares para casos de uso específicos.

Casos de Uso Avanzados

Extracción de datos estructurados

Las expresiones regulares son excelentes para extraer información específica de archivos de log, registros, textos largos, etc.

```
import re

# Extraer información de un log de acceso
log_line = '192.168.1.1 - - [23/Apr/2021:20:30:45 +0300] "GET /index.html HTTP/1.1" 200 4523'

patron = r'^(\S+) - - \[(.*?)\] "(.*?)" (\d+) (\d+)$'
resultado = re.search(patron, log_line)

if resultado:
    ip = resultado.group(1)
    fecha = resultado.group(2)
    peticion = resultado.group(3)
    codigo = resultado.group(4)
    bytes_enviados = resultado.group(5)

    print(f"IP: {ip}")
    print(f"Fecha: {fecha}")
    print(f"Petición: {peticion}")
    print(f"Código: {codigo}")
    print(f"Bytes: {bytes_enviados}")
```

Validación de formatos comunes

```
Validación de Email
import re

def validar_email(email):
    patron = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA
    return re.match(patron, email) is not None

emails = ["usuario@ejemplo.com", "invalido@", "otro@dom
for email in emails:
    print(f"{email}: {'Válido' if validar_email(email)
```

```
Validación de Tarjetas de Crédito
import re

def validar_tarjeta(numero):
    # Elimina espacios y guiones
    numero = re.sub(r'[\s-]', '', numero)

    # Visa: comienza con 4, 13-16 dígitos
    visa = r'^4\d{12,15}$'

    # MasterCard: comienza con 51-55, 16 dígitos
    mastercard = r'^5[1-5]\d{14}$'

    # Amex: comienza con 34 o 37, 15 dígitos
    amex = r'^3[47]\d{13}$'

    if re.match(visa, numero):
        return "Visa"
    elif re.match(mastercard, numero):
        return "MasterCard"
    elif re.match(amex, numero):
        return "American Express"
    else:
        return "Desconocida"

print(validar_tarjeta("4111 1111 1111 1111"))
```

Aplicaciones en el mundo real

Algunos ejemplos de cómo se utilizan las expresiones regulares en entornos profesionales:

- Procesamiento de datos y ETL
- Validación de formularios web
- Parsing de archivos de configuración
- Migración entre bases de datos
- Limpieza y normalización de datos
- Análisis de logs y monitoreo
- Web scraping y extracción de datos
- Seguridad y detección de patrones maliciosos

Como mencionó el profesor Alex Narváez en su video: las expresiones regulares son tan poderosas que muchos profesionales han construido herramientas completas y soluciones basadas principalmente en esta tecnología.

Buenas Prácticas

Recomendaciones

- Comenta tus expresiones regulares complejas
- Divide expresiones complejas en partes más pequeñas
- Usa grupos con nombre para mejorar la legibilidad
- Usa el flag 'x' (VERBOSE) para expresiones complejas
- Pre-compila patrones que vas a usar repetidamente
- Ten cuidado con la sobreutilización de expresiones regulares
- Prueba con casos extremos y bordes

Optimización

Técnicas para mejorar el rendimiento:

```
import re

# Pre-compilar patrones (más eficiente)
patron = re.compile(r'\d+')

# Usar el modo VERBOSE para expresiones complejas
patron_email = re.compile(r'''
    ^                       # Inicio de la cadena
    [a-zA-Z0-9._%+-]+       # Nombre de usuario
    @                       # Símbolo @
    [a-zA-Z0-9.-]+          # Dominio
    \.                      # Punto
    [a-zA-Z]{2,}            # TLD (com, org, etc)
    $                       # Fin de la cadena
''', re.VERBOSE)

# Limitar la recursividad usando cuantificadores no c
texto = "
Contenido 1
Contenido 2
"
# Codicioso (podría causar problemas):
re.findall(r'
(.*)
', texto)  # ['Contenido 1
```

Contenido 2'] # No codicioso (mejor): re.findall(r'
(.*?)
', texto) # ['Contenido 1', 'Contenido 2']

Conclusión

Las expresiones regulares son una herramienta extremadamente poderosa para el procesamiento de texto y la validación de datos. Aunque pueden parecer intimidantes al principio, dominando los conceptos básicos y practicando con ejemplos concretos, pueden convertirse en una de las herramientas más útiles en tu arsenal de programación.

Como vimos en los ejemplos del profesor Alex Narváez, las expresiones regulares pueden resolver problemas complejos de manera elegante y eficiente, desde la validación de contraseñas hasta el procesamiento de grandes volúmenes de datos.

Recursos adicionales

- Canal de YouTube: Alex Narváez Programming
- Video completo: PYTHON 🐍- Lección 13: EXPRESIONES REGULARES
- Documentación oficial de Python: Módulo re