# EXPRESIONES REGULARES

Algoritmos y estructuras de datos I

# BÚSQUEDA DE VALORES EN UNA CADENA

En código de ejemplo, buscamos el número "1984" dentro de un texto usando el operador *in*, que es una forma simple y directa de verificar si una palabra o número específico está presente. Si "1984" está en el texto, el código muestra un mensaje confirmando su existencia.

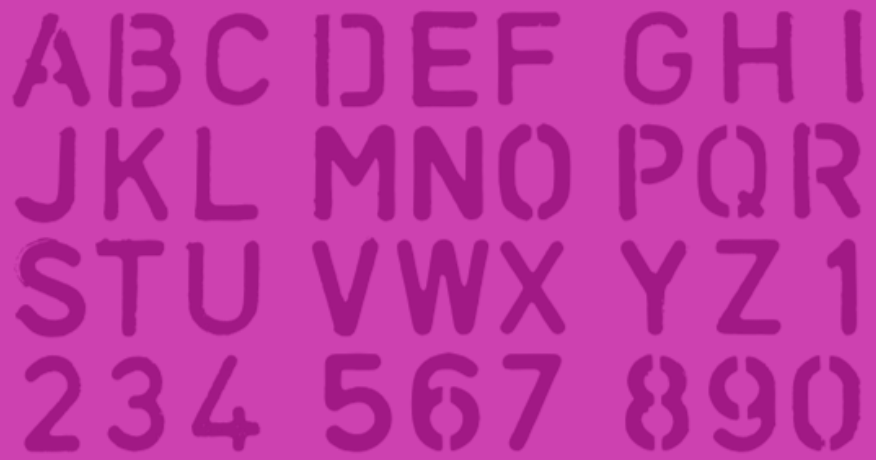
Esta técnica es útil para búsquedas exactas cuando sabemos exactamente qué estamos buscando. Pero ¿qué sucede si necesitamos buscar todos los números de 4 cifras en el texto, no solo "1984"? En este caso, el operador *in* no nos sirve, ya que solo encuentra coincidencias exactas.

Aquí es donde las expresiones regulares nos permiten buscar patrones más complejos, como cualquier número de 4 dígitos, en lugar de solo uno específico.

```
texto = "He leído 1984 de George Orwell y 1Q84 de Haruki Murakami este año."

if "1984" in texto:
    print("Se encontró el número 1984.")
else:
    print("No se encontró el número 1984.")
```

Lic. Yaps David P.




# EXPRESIONES REGULARES

Es una secuencia de caracteres que define un patrón de búsqueda. Este patrón se utiliza para encontrar coincidencias dentro de otras cadenas de texto, basado en delimitadores específicos y reglas de sintaxis.

Por ejemplo, consideremos el patrón *pa* y veamos algunas cadenas con las que podría coincidir:


Si el patrón se encuentra, decimos que hemos encontrado una coincidencia (*match*). Estos ejemplos son básicos, ya que utilizan patrones literales, lo que significa que solo encontramos coincidencias cuando hay una correspondencia exacta.

Lic. Yaps David P.

# EXPRESIONES REGULARES

Las expresiones regulares son patrones que usaremos para encontrar una o varias combinaciones de caracteres en un texto.

Las expresiones regulares, a menudo llamada también *regex*, son unas secuencias de caracteres que forma un patrón de búsqueda, las cuales son formalizadas por medio de una sintaxis específica.

Los patrones se interpretan como un conjunto de instrucciones, que luego se ejecutan sobre un texto de entrada para producir un subconjunto o una versión modificada del texto original.

Lic. Yaps David P.


# CARACTERES Y METACARACTERES

Nuestro patrón puede estar formado por un conjunto de caracteres (letras, números o signos) acompañado de metacaracteres que representan a otros caracteres o permiten una búsqueda contextual.

Los metacaracteres reciben este nombre porque no se representan a ellos mismos, sino que son interpretados de una manera especial. Por ejemplo, a través de metacaracteres podemos definir diferentes condiciones tales como agrupaciones, alternativas, comodines, multiplicadores, entre otros.

Los metacaracteres más usados son:

. * ? + [ ] ( ) { } ^ $ | \

# METACARACTERES DE POSICIONAMIENTO (ANCLAS)

Los signos ^ y $ sirven para indicar donde debe estar situado nuestro patrón dentro de la cadena para considerar que existe una coincidencia.

Cuando usamos el signo ^ queremos decir que el patrón debe aparecer al principio de la cadena de caracteres comparada. Cuando usamos el signo $ estamos indicando que el patrón debe aparecer al final del conjunto de caracteres. O más precisamente, antes de un caracter de nueva línea.

De la misma forma, la expresión ^$ se puede utilizar para encontrar líneas vacías, donde el inicio de una línea es inmediatamente seguido por el final de ésta.

Los metacaracteres ^ y $ también conocidos como anclas (*anchors*) ya que no representan otros caracteres, sino posiciones en una cadena.

| Expresión | Coincidencia |
|---|---|
| ^El | El mejor código es el que no necesitas escribir |
| co$ | El código siempre debe ser lógico |

# ESCAPE DE CARACTERES

Puede suceder que necesitemos incluir en nuestro patrón algún metacaracter como signo literal, es decir, que se interprete por sí mismo y no por su función especial. Para lograr esto, utilizamos el carácter de escape, que es la barra invertida (\).

La barra invertida (\) antepone a los metacaracteres para que sean tratados como caracteres normales. Por ejemplo, si queremos buscar el símbolo de peso ($) en lugar de su significado especial de «final de caracteres», usaremos \$.

Así, la barra invertida convierte a los caracteres especiales en literales dentro del patrón.

| Expresión | Coincidencia |
|---|---|
| \$100 | El monto total es de: $100 |
| \$21 | El cálculo del IVA es de $21.<br>Total de compra: $121. |

Lic. Yaps David P.

# EL COMODÍN . (PUNTO)

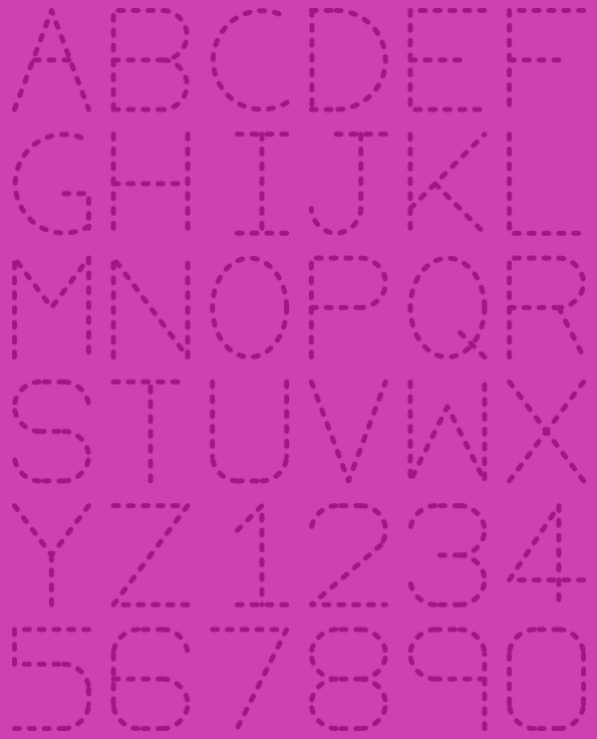
El metacaracter *. (punto)* es el comodín por excelencia en las expresiones regulares. Un punto en el patrón representa cualquier carácter excepto una nueva línea. Esto significa que puede coincidir con letras, números, espacios en blanco y otros símbolos.

Observamos en estos ejemplos cómo el metacaracter punto (.) puede utilizarse para coincidir con diversos caracteres en diferentes contextos dentro de las expresiones regulares.

Cuando usamos la expresión regular *'.l'*, el motor de búsqueda trata de encontrar la primera coincidencia que satisface el patrón, luego avanza el cursor a la siguiente posición, y así sucesivamente. Esto implica que, si una coincidencia comienza en una posición, el motor no vuelve a utilizar esa misma posición para otra coincidencia, incluso si es posible encontrarla. En otras palabras, no se permite que las coincidencias se solapen.

| Expresión | Coincidencia |
|---|---|
| s.l | la sal al sol |
| a..a | aroma, arena, alaba, ariza |
| a...a | aroma, arena, alaba, ariza |
| .l | El sol brilla |
| ^E. | El gato se llama Tomy |

Lic. Yaps David P.




# CLASES DE CARACTERES

Los corchetes [ ] definen una clase de caracteres y permiten encontrar cualquiera de los caracteres dentro de un grupo.

Imaginemos que queremos encontrar la palabra niño, pero también queremos encontrar en caso de que la hayan escrito con n el lugar de ñ. Podríamos lograr esto con una clase de caracteres, de forma que la expresión regular ni[ñn]o se interpretaría como: n, seguida de i, seguida ya sea de ñ o n, seguida de o.

niño
niña
ninio
nino

# CLASES DE CARACTERES

La mayoría de los metacaracteres pierden su significado al ser utilizados dentro de clases de caracteres. Es así como la expresión [a.] se refiere literalmente a la letra a y al caracter punto.

Un caso especial es el carácter ^, que al ser utilizado al comienzo de una clase de caracteres significa negación.

Es decir que la expresión [^a] se refiere a cualquier cadena que NO contenga la letra a.

Oliver
Olivia
Gabriel
Gabriela

Este significado se pierde si el ^ no está al inicio de la clase de caracteres, así que la expresión [2^] se refiere al número 2 y al caracter ^ literalmente:

El circunflejo (^) se utiliza en matemáticas para denotar exponentes, como en 2^3 que significa 2 elevado a la potencia de 3.

Si lo que se quiere es buscar específicamente el símbolo ^ y no usarlo para negar el conjunto, es necesario escaparlo [\^]:

Este es el símbolo de circunflejo: ^

O todo aquello que no sea circunflejo, la letra í o espacio [^^í ]:

Este es el símbolo de circunflejo: ^

# CLASES DE CARACTERES: RANGOS

Así como la mayoría de los metacaracteres pierden su significado especial al ser utilizados dentro de corchetes, existe un caracter que solamente al ser utilizado dentro de corchetes adquiere un significado especial.

Este es el carácter - (guión), el cual se utiliza dentro de una clase de caracteres para indicar un rango.

Por ejemplo, si queremos referirnos a un caracter hexadecimal, en lugar de definir la clase *[01234567890abcdefABCDEF]* utilizaríamos *[0-9a-fA-F]*, que es mucho más conveniente.

| Expresión | Coincidencia |
|---|---|
| [0-9][0-9] | Su fecha de nacimiento fue el 07/08/17. |
| 201[2-5] | 2010 2011 2012 2013 2014 2015 2016 2017 |
| [a-zA-Z] | El monto total es de: $100 |

# ALTERNATIVAS

El metacaracter | (barra vertical) en expresiones regulares funciona como un operador de alternancia (o lógico). Permite especificar múltiples opciones dentro de una misma expresión, de modo que la expresión coincida con cualquiera de las alternativas proporcionadas.

Supongamos que queremos buscar palabras que puedan ser *"septiembre" o "setiembre"*. En lugar de definir dos expresiones separadas, podemos utilizar el metacaracter | para combinarlas en una sola expresión regular.

Este uso del metacaracter | simplifica la búsqueda de múltiples alternativas dentro de una expresión regular, lo que resulta en una expresión más concisa y legible.

| Expresión | Coincidencia |
|---|---|
| [septiembre\|setiembre] | 21 de septiembre<br>21 de setiembre |

# CUANTIFICADORES O MULTIPLICADORES

Los multiplicadores permiten especificar cuántas veces puede aparecer un caracter o grupo de caracteres en una cadena que estamos buscando o comparando. Estos metacaracteres se aplican al elemento precedente y definen la cantidad de ocurrencias permitidas para que haya coincidencia:

| Expresión | Coincidencia |
|---|---|
| sub?scripción | Coincide con suscripción y con subscripción |
| camp*a | Productos: cama, campana y campera |
| car+ | Productos: carrito, cartón, collar y cartel |

**?**

indica que el elemento puede aparecer 0 o 1 vez.

*****

indica que el elemento puede aparecer 0 o más veces.

**+**

indica que el elemento debe aparecer al menos 1 vez, pero puede aparecer más veces.

# CUANTIFICADORES O MULTIPLICADORES

En expresiones regulares, las llaves {} se utilizan para especificar la multiplicidad exacta que es aceptable para un cuantificador.

La sintaxis consiste en indicar primero el valor mínimo de la multiplicidad, seguido de una coma (opcional si no hay límite inferior), y luego el valor máximo de la multiplicidad.

| Expresión | Coincidencia |
|---|---|
| [0-9]{2,3} | Los números son 123-4567, 456-7890 y 111-0000 |
| 1{2,} | Los números son 123-4567, 456-7890 y 111-0000 |
| 1{0,3} | Los números son 123-4567, 456-7890 y 111-0000 |
| 0{4} | Los números son 123-4567, 456-7890 y 111-0000 |

| Expresión | Coincidencia |
| --- | --- |
| ([0-9]{4}-[0-9]{4}) | 4123-4567, 4456-7890 y 111-0000 |
| ([A-Z]{3}[0-9]{3}) \| ([A-Z]{2}[0-9]{3}[A-Z]{2}) | LWO805, B354006, ZZ000UI |

# DELIMITACIÓN

Los paréntesis en expresiones regulares se utilizan para agrupar subconjuntos de caracteres dentro de una expresión más grande.

Esto permite aplicar operadores o cuantificadores a ese subconjunto específico.

Supongamos que queremos encontrar una secuencia similar a la composición de un número telefónico o una patente de automóvil.

Lic. Yaps David P.

# UTILIZACIÓN DE EXPRESIONES REGULARES EN PYTHON

Ahora que hemos establecido las bases, es momento de adentrarnos en el mundo de las expresiones regulares junto con Python.

Python ofrece un módulo específico para trabajar con expresiones regulares, conocido como 're', el cual nos provee todas las herramientas necesarias para realizar operaciones avanzadas.

Para comenzar, nuestro primer paso será importar este módulo:

```
import re
```

Con esto, estaremos listos para explorar y utilizar eficazmente las expresiones regulares en nuestros proyectos

# MÉTODO MATCH

Se utiliza para determinar si una cadena coincide con un patrón de expresión regular desde el inicio de la cadena.

La función retorna un objeto *Match* si encuentra una coincidencia exitosa (se considera *True* cuando se evalúa en un contexto booleano); de lo contrario, devuelve *None*.

Para utilizar *match*, se debe proporcionar un patrón de expresión regular y la cadena en la que se desea buscar. Este método es útil para validaciones iniciales o para extraer información específica que se encuentra al principio de las cadenas, siguiendo el formato definido por el patrón de expresión regular.

```
import re

cadenas = ['A123', 'B456', 'C789', '123A', 'D1234']
patron = '[A-Za-z][1-3]{3}'

#Iteramos sobre las cadenas
for cadena in cadenas:
    if re.match(patron, cadena):
        print(f'La cadena "{cadena}" coincide con el patrón.')
    else:
        print(f'La cadena "{cadena}" no coincide con el patrón.')
```

# MÉTODO SEARCH

Permite buscar un patrón de expresión regular dentro de una cadena en cualquier posición. A diferencia de *match*, que verifica solo el inicio de la cadena, *search* examina toda la cadena en busca de la primera ocurrencia del patrón especificado.

Si encuentra una coincidencia, devuelve un objeto *Match* que contiene detalles sobre la posición y el contenido coincidente. Si no encuentra ninguna coincidencia, devuelve *None*.

Para utilizar *search*, se debe proporcionar un patrón de expresión regular y la cadena en la que se desea buscar.

```
import re

cadena = "El precio del producto es $11.33."
patron = '[0-9]+'

match = re.search(patron, cadena)

if match:
    numeroEncontrado = match.group()
    posicionInicio = match.start()
    posicionFin = match.end()
    posicionSpan = match.span()  #Tupla con (inicio, fin)

    print(f"Primer número encontrado: {numeroEncontrado}")
    print(f"Posición donde comienza: {posicionInicio}")
    print(f"Posición donde termina: {posicionFin}")
    print(f"Posición donde comienza y termina (tupla span):
{posicionSpan}")
else:
    print("No se encontró ningún número en la cadena.")
```

# MÉTODO SEARCH: OMITIR MAYÚSCULAS Y MINÚSCULAS

Para utilizar el método *search* y omitir las diferencias entre mayúsculas y minúsculas, podemos hacer uso de un tercer parámetro con el valor *re.IGNORECASE*.

Este permite que la búsqueda del patrón de expresión regular sea insensible a las diferencias de mayúsculas y minúsculas en la cadena de texto.

Esto es especialmente útil cuando se desea buscar patrones sin importar la capitalización de las letras en el texto.

```
import re

cadena = "Morty, a veces la Ciencia es más arte que Ciencia..."
patron = 'ciencia'

match = re.search(patron, cadena, re.IGNORECASE)

if match:
    print(f"Se encontró '{match.group()}' en la posición {match.start()}.")
else:
    print("No se encontró ninguna coincidencia.")
```

# MÉTODO FINDALL

Se utiliza para encontrar todas las ocurrencias de un patrón de expresión regular en una cadena de texto y devolverlas como una lista de cadenas.

A diferencia de *search*, que encuentra solo la primera ocurrencia, *findall* recorre toda la cadena y retorna todas las coincidencias encontradas.

El método *findall* toma dos argumentos principales: el patrón de expresión regular que se busca y la cadena en la que se realiza la búsqueda.

Si se utilizan grupos de captura en el patrón, *findall* devolverá una lista de tuplas donde cada tupla contiene los grupos capturados para cada coincidencia.

```
import re

texto = "Juan tiene el teléfono 1234-5678, Oliver tiene el
teléfono 11-15-3456-7890 y Gabriela el 11-15-7654-3210."
patron = "([0-9]{2,})-(15-[0-9]{4})-([0-9]{4})"

numeros = re.findall(patron, texto)

#Imprimimos los números de celulares encontrados y sus partes
específicas
if numeros:
    print("Números de celulares encontrados:")
    for numero in numeros:
        print(f"Número completo: {numero[0]}-{numero[1]}-
{numero[2]}")
        print(f"Código de área: {numero[0]}")
        print("----")
else:
    print("No se encontraron números de teléfono.")
```

# MÉTODO FINDITER

Se utiliza para encontrar todas las ocurrencias de un patrón de expresión regular en una cadena de texto, similar a *findall*.

Sin embargo, en lugar de devolver una lista de coincidencias, *finditer* devuelve un iterador que produce objetos *Match* para cada coincidencia encontrada.

Para utilizarlo, se proporciona el patrón de expresión regular y la cadena en la que se realizará la búsqueda.

Cada objeto *Match* generado por el iterador contiene información sobre la posición y el contenido de la coincidencia, permitiendo un procesamiento detallado de cada coincidencia encontrada en el texto.

```python
import re

texto = "Los correos de contacto son oliver@dominio.com y gabriela@dominio.com."

patron = '[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}'

iteradorCorreos = re.finditer(patron, texto)

#Iteramos sobre el iterador para mostrar información detallada de cada correo encontrado
print("Información detallada de correos electrónicos encontrados:")
for match in iteradorCorreos:
    print(f"Correo electrónico encontrado: {match.group()}")
    print(f"Ubicación: inicio={match.start()}, fin={match.end()}")
    print("----")
```

# MÉTODO SUB

Se utiliza para reemplazar todas las ocurrencias de un patrón de expresión regular en una cadena de texto con otra cadena especificada.

Esto es útil cuando se desea modificar o limpiar texto según un patrón definido por expresiones regulares.

Al utilizar *sub*, se proporcionan tres argumentos principales: el patrón de expresión regular que se busca, la cadena que se utilizará para reemplazar las coincidencias encontradas y la cadena de texto en la que se realizará la búsqueda y el reemplazo.

El método devuelve una nueva cadena con todas las sustituciones realizadas según el patrón especificado.

```
import re

texto = "El número de teléfono de Oliver es 123-456-7890, y el de Gabriela es 987-654-3210."

patron = "[0-9]{3}-[0-9]{3}-[0-9]{4}"
cadenaEnmascarada = "XXX-XXX-XXXX"

#Utilizamos re.sub() para reemplazar todos los números de teléfono por la cadena enmascarada
textoOfuscado = re.sub(patron, cadenaEnmascarada, texto)

print("Texto original:")
print(texto)
print("\nTexto después de ofuscar los números de teléfono:")
print(textoOfuscado)
```

# MÉTODO SPLIT

Se utiliza para dividir una cadena en partes utilizando un patrón de expresión regular como delimitador.

A diferencia del método *split* convencional de Python, que solo admite cadenas simples como delimitadores, *re.split* permite utilizar patrones más complejos basados en expresiones regulares para separar la cadena en partes más significativas.

Para usar re.split(), se proporcionan dos argumentos principales: el patrón de expresión regular que se utilizará como separador y la cadena de texto que se dividirá.

El método devuelve una lista de cadenas resultantes después de dividir la cadena original según el patrón especificado.

```
import re

texto = "Hola, ¿cómo estás? Espero que bien. Yo estoy bien también."
#Este patrón divide por cualquier combinación de puntos, comas, signos de interrogación y espacios
patron = '[.,? ]+'

#Utilizamos re.split() para dividir el texto según el patrón definido
partes = re.split(patron, texto)

print("Partes del texto después de dividir:")
for parte in partes:
    print(parte)
```

# MÉTODO COMPILE

La compilación de expresiones regulares es el proceso mediante el cual una expresión regular se convierte en un objeto patrón que puede ser reutilizado eficientemente.

La ventaja de compilar una expresión regular es que permite optimizar las búsquedas, especialmente cuando la misma expresión se utiliza múltiples veces en el código.

Al compilar una expresión regular, no solo se mejora la legibilidad del código, sino también su rendimiento, ya que evita la necesidad de recompilar la misma expresión en cada uso.

Este objeto se puede emplear para invocar métodos antes vistos, de la misma manera que se usarían directamente con el módulo *re*.

```
import re

patron = re.compile('[0-9]{4}')
cadena = "07/08/2017|03/02/1984|17/03/1984"

coincidencias = patron.findall(cadena)
print(f'Coincidencias encontradas con findall(): {coincidencias}')

inicioCoincidencia = patron.match(cadena)
print(f'Coincidencia al inicio con match(): {inicioCoincidencia}')

cadenaReemplazada = patron.sub('XXXX', cadena)
print(f'Cadena después de usar sub(): {cadenaReemplazada}')
```

```python
import re

#Función para validar la fecha en formato DD/MM/AAAA
def validarFecha(fecha):
    patronFecha = "^(0?[1-9]|[12][0-9]|3[01])/(0?[1-9]|1[0-2])/[0-9]{4}$"
    return re.match(patronFecha, fecha) is not None

#Función para validar la dirección IP en formato IPv4
def validarDireccionIp(direccionIp):
    patronIp = "^((25[0-5]|2[0-4][0-9]|[01]?[0-9]?[0-9])\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9]?[0-9])$"
    return re.match(patronIp, direccionIp) is not None

def solicitarDato(mensaje, funcionValidacion, mensajeError):
    dato = input(mensaje)
    while not funcionValidacion(dato):
        print(mensajeError)
        dato = input(mensaje)
    return dato

fechaAcceso = solicitarDato("Ingrese fecha de acceso (DD/MM/AAAA): ", validarFecha, "Fecha inválida. Debe estar en formato DD/MM/AAAA.")
direccionIp = solicitarDato("Ingrese la dirección IP: ", validarDireccionIp, "Dirección IP inválida. Debe ser una dirección IP válida en formato IPv4.")

print(f"Registro exitoso! Fecha de acceso: {fechaAcceso}, Dirección IP: {direccionIp}")
```

# EJEMPLOS DE USO

La carga validada asegura que los datos ingresados, como una dirección IP o una fecha, cumplan con los formatos específicos requeridos antes de ser aceptados.

Al ingresar una fecha, el sistema verifica que esté en el formato correcto (como día/mes/año), y al ingresar una dirección IP, verifica que siga el formato estándar de direcciones IP.

Esto es esencial en aplicaciones que requieren precisión en los datos.

Lic. Yaps David P.

# AUTOEVALUACIÓN

La autoevaluación es crucial para identificar mejoras y fortalezas de forma objetiva, facilitando la fijación de metas y la toma de acciones para alcanzarlas.