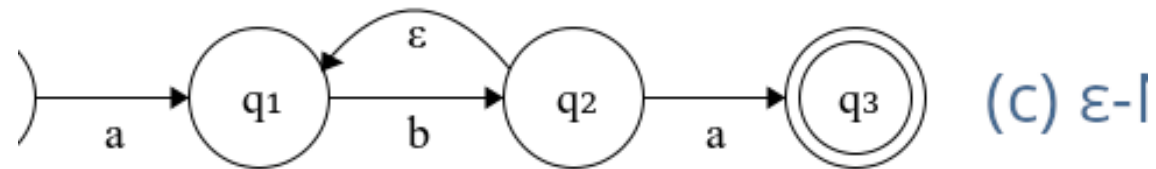
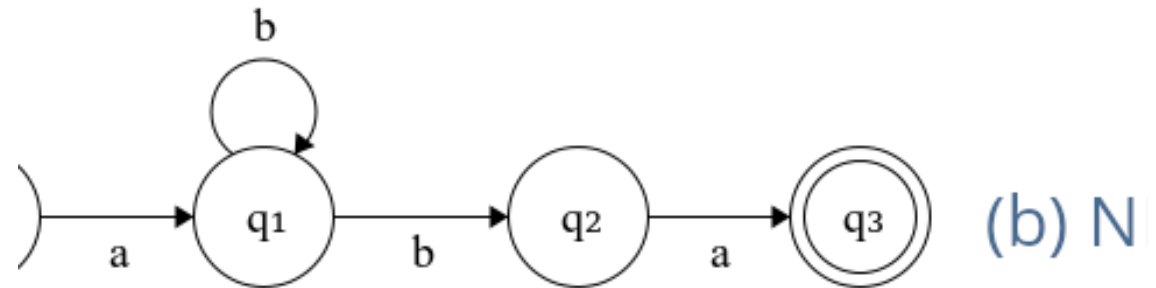
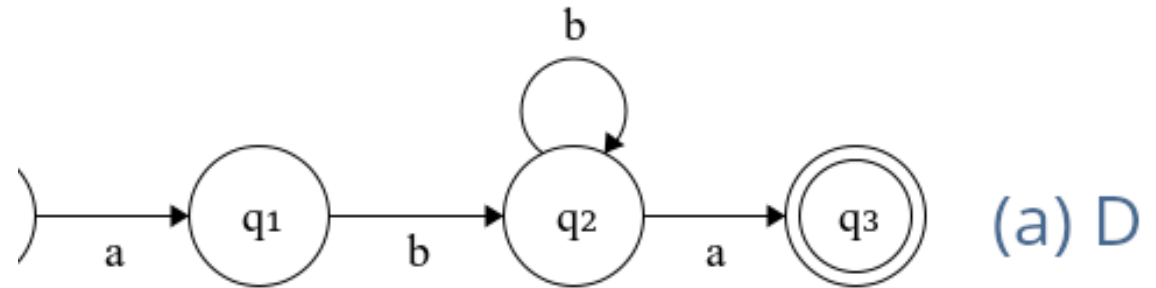


# Expresiones regulares en Python



Tema 12

# Expresiones regulares

Si queremos validar un número, por ejemplo:

- (81)8345-9824

# Validación

(81)8345-9824

Longitud 13 caracteres

Que sean decimales

Que sean paréntesis

Que sean decimales

Que sea guion

Que sean decimales

---

# Función para validar teléfono

```
def verificarNumero(telefono):  
    if len (telefono) != 13:  
        return False  
    for i in range (1,2):  
        if not telefono[i].isdecimal():  
            return False  
    if telefono[0] != "(":  
        return False  
    if telefono[3] != ")":  
        return False  
    for i in range (4,7):  
        if not telefono[i].isdecimal():  
            return False  
    if telefono[8] != "-":  
        return False  
    for i in range (9,13):  
        if not telefono[i].isdecimal():  
            return False  
    return True  
  
telefono = input ("Ingresa un telefono")  
print (verificarNumero(telefono))
```

---

# Limitación

---

Solo funciona con esa cadena en específico.  
Si hay espacios (u otro caracter), deja de funcionar.

---

¿Qué sucede si alguien escribe el número en otro formato?

---

Por ejemplo: 81-8345-9824

---

Por ejemplo: 81 83 45 98 24

---

# Introducción a las expresiones regulares

## ¿Qué son las expresiones regulares?

Las expresiones regulares son secuencias de caracteres que definen patrones de búsqueda para buscar, reemplazar o hacer coincidir el texto de una cadena.

## Sintaxis básica de las expresiones regulares

Las expresiones regulares se escriben utilizando una sintaxis específica que incluye caracteres especiales y símbolos para definir patrones. Algunos ejemplos de sintaxis básica son el uso de paréntesis, el símbolo de barra vertical y los corchetes.

## Clases de caracteres en expresiones regulares

Las clases de caracteres son códigos abreviados especiales que representan grupos de caracteres en expresiones regulares. Algunos ejemplos de clases de caracteres son los dígitos, los caracteres de palabra y los espacios en blanco.

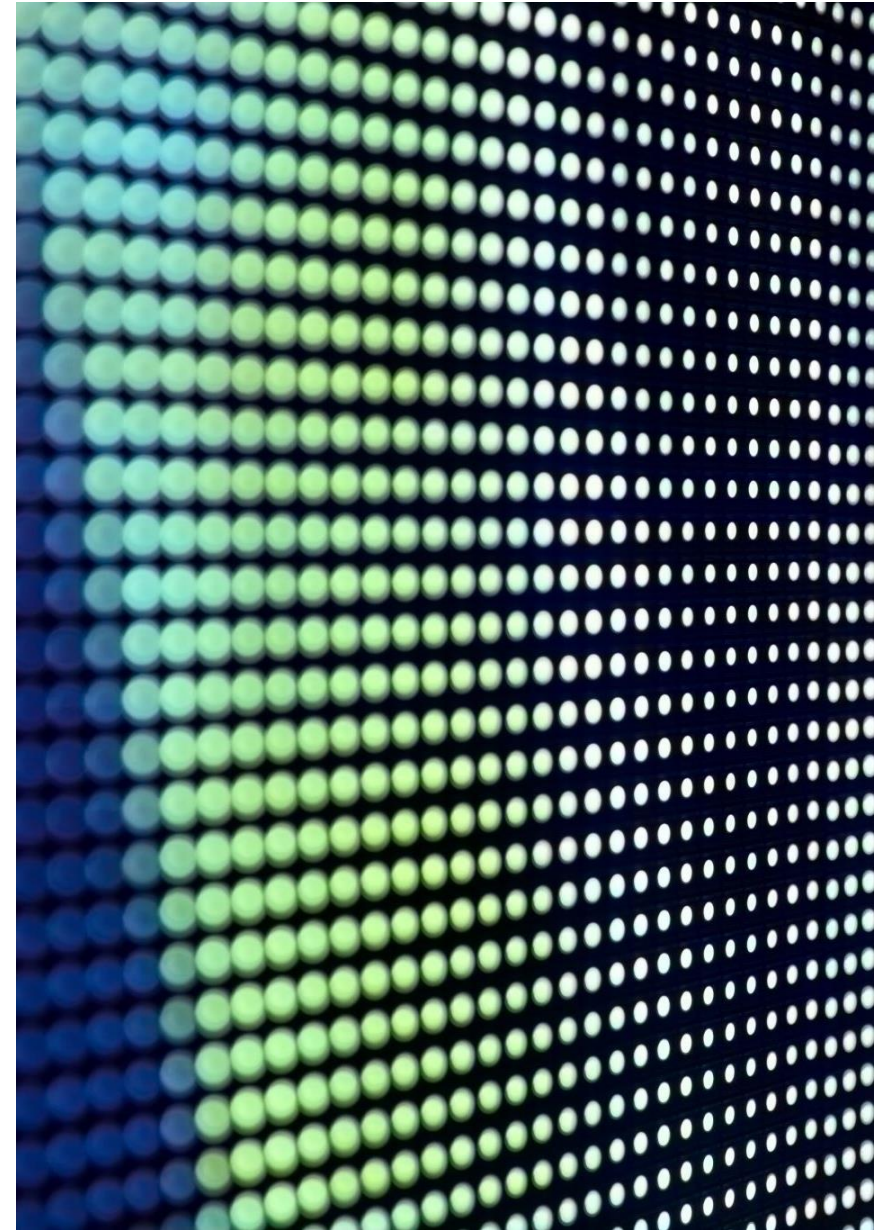




---

# ¿Qué son las expresiones regulares?

Las expresiones regulares, o regex, son una secuencia de caracteres que definen un patrón de búsqueda. Se pueden usar para buscar, reemplazar y validar texto, lo que los convierte en una poderosa herramienta para programadores y desarrolladores.



# Sintaxis básica

Las expresiones regulares utilizan una combinación de caracteres normales y especiales para definir un patrón de coincidencia de texto. La sintaxis básica incluye caracteres literales, clases de caracteres, repetición, alternancia y agrupación.

Caracter	Valor que representa
\d	Cualquier dígito entre 0 y 9
\D	Cualquier caracter que no sea número entre 0 y 9
\w	Cualquier letra, dígito numérico, o guion bajo.
\W	Cualquier caracter que no sea letra, dígito o guion bajo.
\s	Cualquier espacio, tab, salto de línea.
\S	Cualquier caracter que no sea espacio, tab o salto de línea.



# Patrones

Las expresiones regulares (regex) son descripciones de un patrón de texto.

Por ejemplo, `\d` es una expresión regular para un dígito.

- El patrón para teléfono sería `(\d\d)\d\d\d\d-\d\d\d\d`

De manera más sofisticada

- El patrón teléfono sería `(\d{2})\d{4}-\d{4}`

---

# Significado especial en regex

Los siguientes caracteres tienen un significado especial en regex

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

Si quieres crear una regex con estos caracteres **debes de colocar \** antes de ellos

- \ ( \d\d \)\d\d\d\d-\d\d\d\d

---

# Clases de caracteres

---

Las clases de caracteres se utilizan para hacer coincidir un conjunto específico de caracteres en la cadena de entrada. Los diferentes tipos de clases de caracteres incluyen [a-z], [A-Z], [0-9], [^] y [.]. Cada una de estas clases de caracteres tiene un uso único en las expresiones regulares.

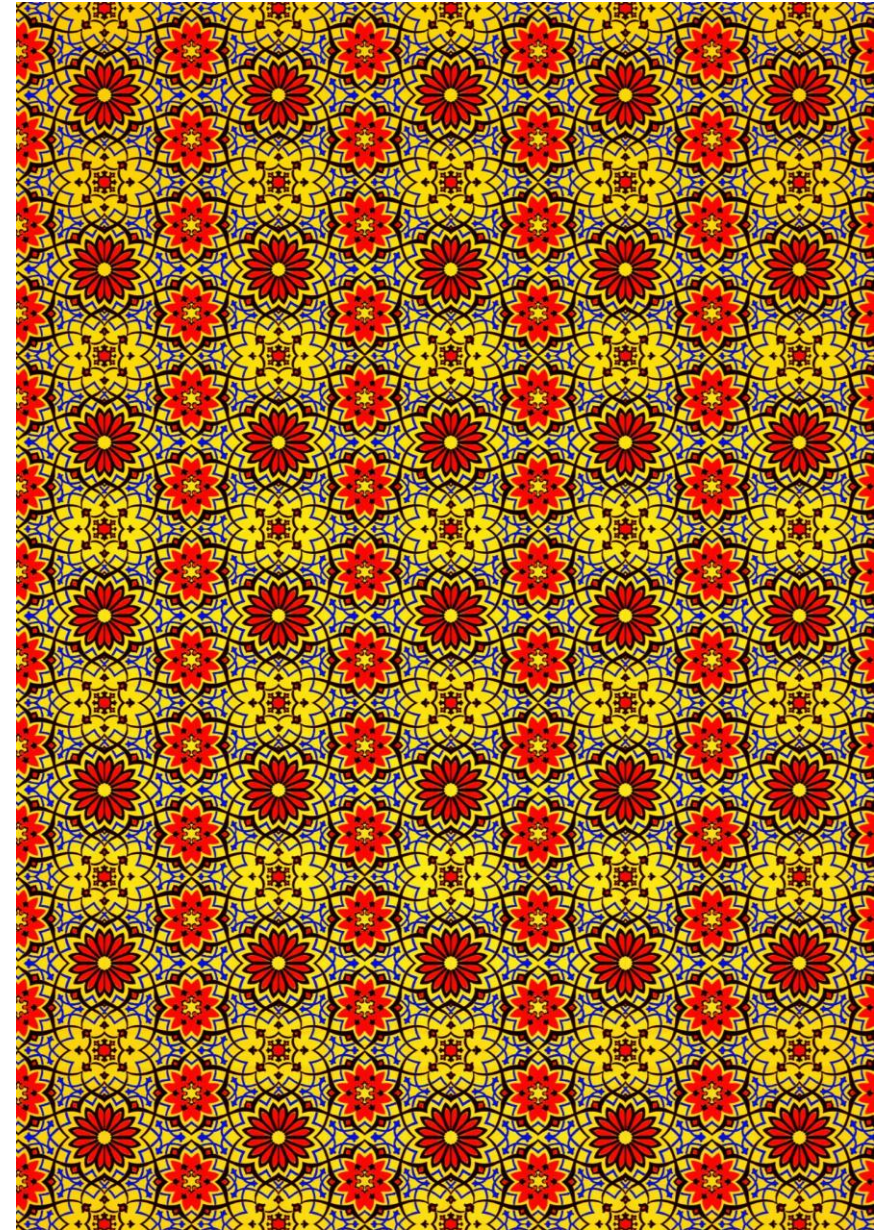
# Cuantificadores y agrupaciones

## Cuantificadores

Los cuantificadores especifican cuántas veces se debe repetir un patrón. Existen diferentes tipos de cuantificadores como  $*$ ,  $+$ ,  $?$  y  $\{n,m\}$ .

## Agrupación

La agrupación nos permite tratar varios caracteres como una sola unidad. La agrupación se denota con paréntesis  $()$  y se puede utilizar con cuantificadores y alternancia.





# Uso de expresiones regulares en Python

## Expresiones regulares en Python

Python proporciona un módulo incorporado llamado `re` para trabajar con expresiones regulares. Las expresiones regulares son una herramienta eficaz para la coincidencia de patrones y la manipulación de cadenas, y se pueden utilizar en una amplia variedad de aplicaciones.

## Emparejamiento y sustitución

Las expresiones regulares en Python se pueden usar para hacer coincidir y reemplazar texto. Esto se hace utilizando los métodos `search()` y `sub()` del módulo `re`, respectivamente. Las expresiones regulares se pueden usar para hacer coincidir patrones en el texto y, a continuación, sustituir esos patrones por texto nuevo.



---

# Módulo re en Python

Python tiene una librería llamada re (regular expressions) con funciones dedicadas a expresiones regulares.

Si quieres usar la librería debes de importarla con la linea

- `import re`

Al inicio de tu código

---

---

# Función **compile**

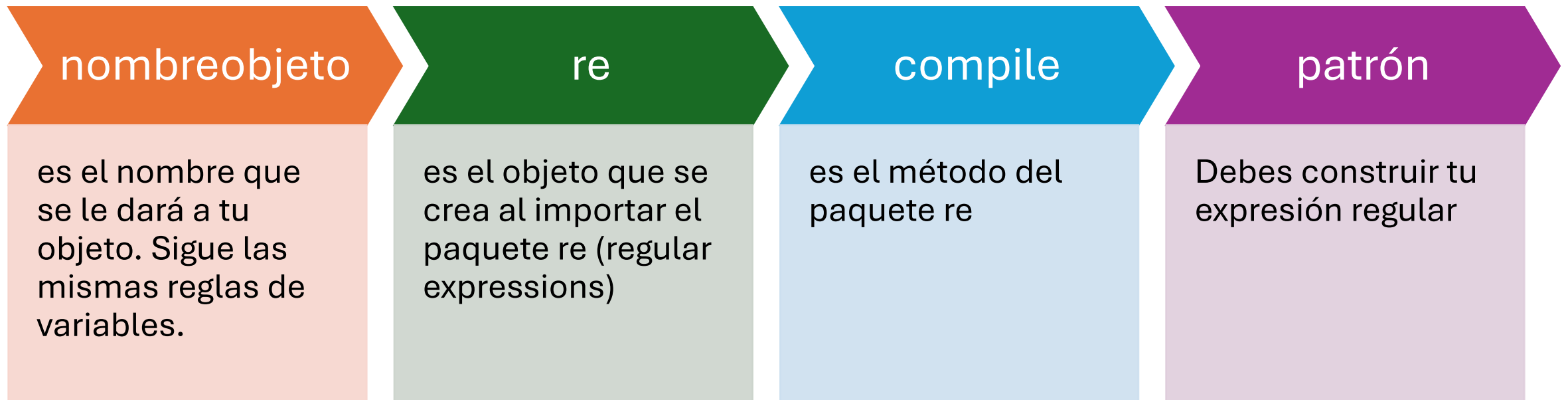
La función **compile** de `re` permite compilar patrones de expresiones regulares en objetos (POO)

Los objetos pueden ser igualados con los métodos **match()** y **search()**

---




# nombreobjeto = re.compile (patrón)



---

# Ejemplo

```
expresion_telefono = re.compile(r'\(\d\d\) \d\d\d\d-\d\d\d\d')
```



```
##Buscamos objetos que empaten con la búsqueda matching objects  
mo = expresion_telefono.search('Mi numero es (81)8345-9824.')
```

**La función search() necesita un String en donde buscar**

---

**group() - Su valor de  
retorno es la cadena**

```
print (mo.group())
```

```
>> (81) 8345-9824
```

---

# group()

re permite agrupar contenidos para una búsqueda eficiente.

1

En el ejemplo anterior no había grupos.

2

Si quieres agrupar, necesitas usar “(“ sin la diagonal invertida.

---

# Los paréntesis marcados agrupan el contenido

```
expresion_telefono_grupo =  
re.compile(r'\((\d{2})\)(\d{4})-(\d{4})')  
mo = expresion_telefono_grupo.search('Mi numero es  
(81)8345-9824.')
```

```
print ("Número encontrado: " + mo.group())  
print ("Lada", mo.group(1))  
print ("Numero",mo.group(2),mo.group(3))
```

```
>>Número encontrado: (81)8345-9824  
>>Lada 81  
>>Numero 8345 9824
```

---

## El caracter |

---

Puede usarlo en cualquier lugar para coincidir con una de las muchas expresiones.

---

Por ejemplo, la expresión regular `r“Batman|Robin “` coincidirá con `“Batman”` o `“Robin”`.

---

Cuando ambas ocurren en `search()`, la primera ocurrencia será retornada.

---



---

# Ejemplo de

|

```
br_regex = re.compile (r"Batman|Robin")  
mo = br_regex.search ("Batman y Robin")  
print (mo.group())
```

```
mo = br_regex.search ("Robin y Batman")  
print (mo.group())  
input()
```

```
>>Batman
```

```
>>Robin
```

---

---

# findall()

Este método retorna las cadenas para cada texto encontrado en el String.



Retorna una lista

---

```
mo = re.findall ("Batman, Robin, Joker, batman, robin, B a t m a n")  
print (mo)  
input()
```

## Ejemplo de findall()

---

# Prefijo

---

Si queremos encontrar todas las coincidencias, por ejemplo: **Bat**man **Bat**imovil **Bat**arang o todos los números que inicien con (81) podemos combinar grupos con |

```
r"Bat (man|imovil|arang) "
```

Buscará si existen palabras con “Bat” como prefijo.

```
bat_regex = re.compile (r"Bat(man|imovil|arang)")
mo = bat_regex.search("Batman subio a su Batimovil olvidó su Batarang.")
print (mo.group())
print (mo.group(1))
input()
mo = bat_regex.findall("Batman subio a su Batimovil olvidó su Batarang.
Batman, Batman")
print (mo)
input()
```

# Ejemplo de |

# Coincidencia opcional (?)

A veces hay un patrón que busca coincidir opcionalmente.

Es decir, la expresión regular debería encontrar una coincidencia independientemente de si ese fragmento de texto está allí.

“?” marca el grupo que lo precede como una parte opcional del patrón.

```
sup_regex = re.compile (r"Interroga(ción)?")
```

Busca “Interroga” opcionalmente busca  
“Interrogación”

Se puede decir que ? Busca 0 o más coincidencias.

---

# Ejemplo de ?

```
sup_regex = re.compile (r"Interroga(ción)?")  
mo = sup_regex.search ("Interrogacion")  
print (mo.group())
```

```
mo = sup_regex.search ("Interroga")  
print (mo.group())
```

```
input()
```

```
>>Interroga
```

```
>>Interroga
```

---



# Buscando 0 o más coincidencias (\*)

---

El grupo que precede \* puede aparecer cualquier número de veces en el texto.

---

Puede ser 0

---

Pueden ser más

```
ast_regex = re.compile (r"(Ha)*")
```

---

Busca Ha desde 0 a cualquier número de veces

---

---

Regresa "" si no encuentra resultados. Esto es importante ya que regresa una cadena vacía y no "none".

---

```
ast_regex = re.compile (r"(Ha) *")
mo = ast_regex.search ("HaHaHaHaHaHaHaHaHa")
print (mo.group())

mo = ast_regex.search ("") #Permite el valor nulo
print (mo.group(), "cadena vacia")
input()
```

>>HaHaHaHaHaHaHaHaHa

>> cadena vacia

## Ejemplo de \*

# Buscando 1 o más coincidencias (+)



Significa "coincide con uno o más".



A diferencia \*, el grupo que precede a un signo más debe aparecer al menos una vez. No es opcional.



```
men_regex =  
re.compile  
(r"Bat(ha)+man")
```

ha de aparecer al menos 1 vez, no es opcional

Si no lo encuentra, regresa "none"

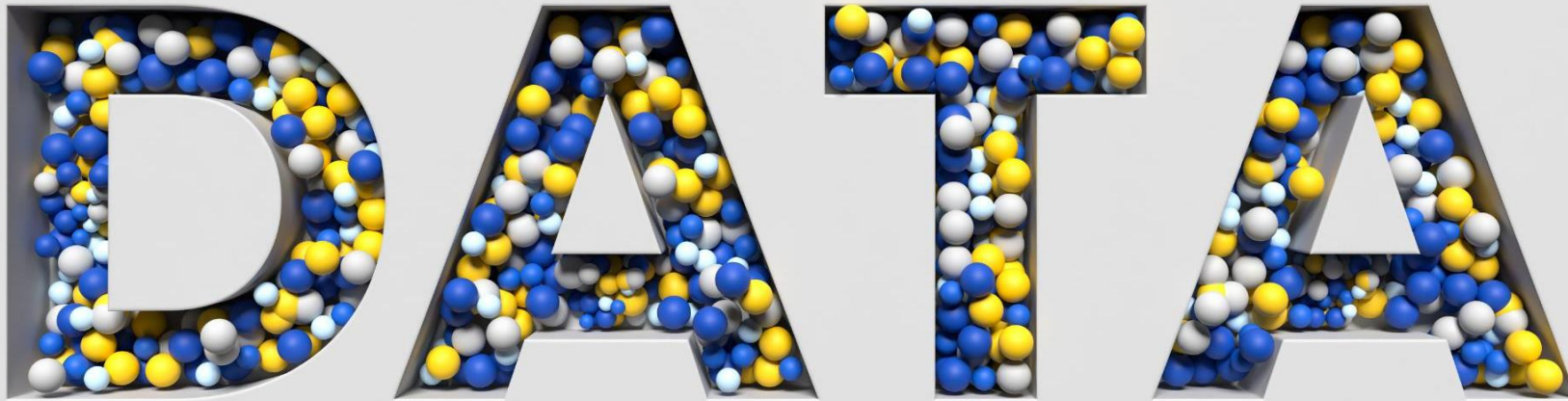
```
men_regex = re.compile (r"Bat(ha)+man")
mo = men_regex.search("Batman")    #No permite valor nulo hay
error
print (mo)
mo = men_regex.search ("Bathahahaman")
print (mo.group())
input()

>>None
>>Bathahahaman
>>HaHaHa
```

## Ejemplo de +

---

# Conclusión



Las expresiones regulares son una herramienta poderosa para trabajar con datos de texto en Python. Pueden ayudarlo a manipular datos de texto con facilidad y realizar búsquedas simples para coincidencias de patrones complejos.