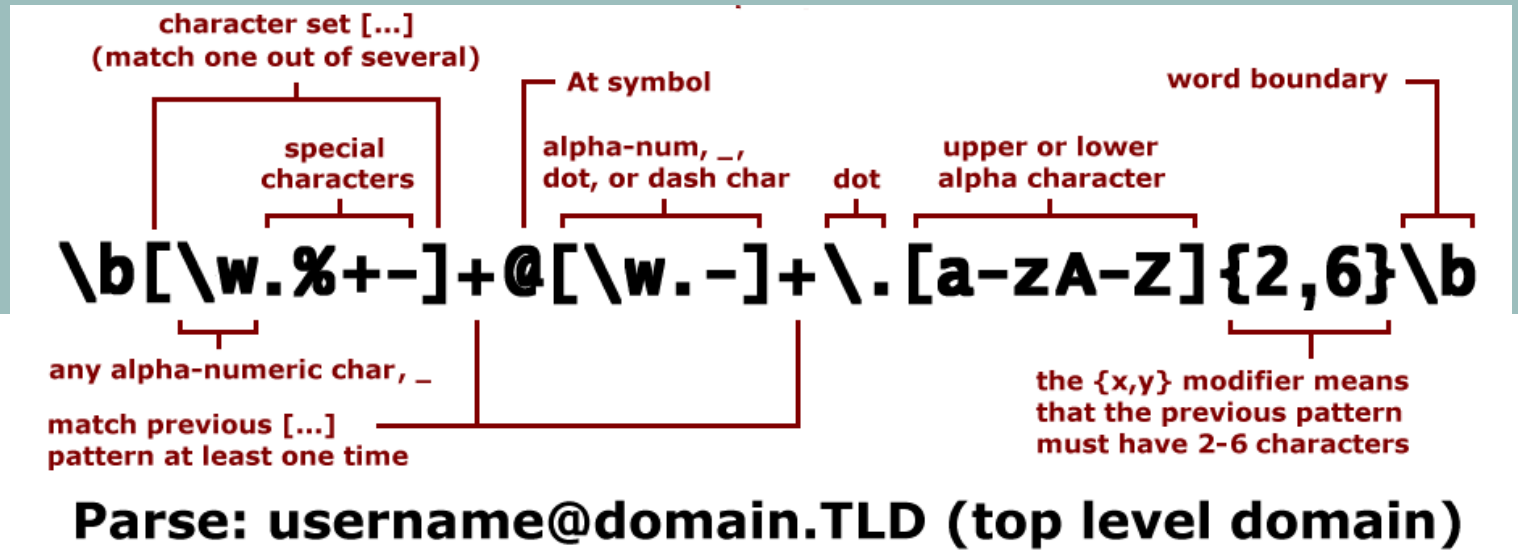
# IMPLEMENTACIÓN CON PYTHON PARA VALIDACIÓN DE DATOS MEDIANTE EXPRESIONES REGULARES



# TEMAS

Introducción.

¿Qué son las Expresiones Regulares?

Componentes de las Expresiones Regulares.

Metacaracteres.

Expresiones Regulares con Python.

Expresiones Regulares con Python – Buscando coincidencias.

- import.
- compile.
- Métodos: match, search, findall, finditer.
- Objeto de coincidencia – Métodos: group, start, end, span

Expresiones Regulares con Python - Modificando el texto de entrada.

Funciones no compiladas.

Banderas de compilación.

Nombrando grupos.

Validaciones.

# INTRODUCCIÓN

Cuando se desarrolla un **programa informático**, generalmente, se requiere

el **procesamiento de texto**.

Para las personas, es muy sencillo, detectar que es un número y que una letra, o cuales son palabras que cumplen con un determinado patrón y cuales no, pero estas mismas tareas no son tan fáciles para una computadora.

Se desarrolló un lenguaje que cualquier computadora puede utilizar para **reconocer patrones de texto, las expresiones regulares**.

Las expresiones regulares permiten realizar las operaciones de validación, búsqueda, extracción y sustitución de texto, de forma más sencilla, a través de las computadoras.

# ¿QUÉ SON LAS EXPRESIONES REGULARES?

Se las llama también regex.

Son **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.

Pueden incluir **patrones de coincidencia** literal, de repetición, de composición, de ramificación, y otras reglas de reconocimiento de texto .

# COMPONENTES DE LAS EXPRESIONES REGULARES

## Literales

- Cualquier **caracter** se encuentra a sí mismo, a menos que se trate de un metacaracter con significado especial. Ejemplo: “a”
- Una **serie de caracteres** encuentra esa misma serie en el texto de entrada. Ejemplo: “expresión”, encontrará todas las apariciones de " expresión" en el texto que procesamos.

## Secuencias de escape

- Pueden ser finales de línea, tabs, barras diagonales, etc.

# PRINCIPALES SECUENCIAS DE ESCAPE

| Secuencia de escape | Significado |
|---|---|
| \n | Nueva línea (new line). El cursor pasa a la primera posición de la línea siguiente. |
| \t | Tabulador. El cursor pasa a la siguiente posición de tabulación. |
| \\ | Barra diagonal inversa |
| \v | Tabulación vertical. |
| \ooo | Carácter ASCII en notación octal. |
| \xhh | Carácter ASCII en notación hexadecimal. |
| \xhhhh | Carácter Unicode en notación hexadecimal. |

# COMPONENTES DE LAS EXPRESIONES REGULARES

## Clases de caracteres

- Se pueden especificar, encerrando una lista de caracteres entre **corchetes** [], la que que encontrará uno cualquiera de los caracteres de la lista. Si el primer símbolo después del "[" es "^", la clase encuentra cualquier carácter que no está en la lista.

## Metacaracteres:

- Son **caracteres especiales.**

# METACARACTERES - DELIMITADORES

Permite **delimitar dónde queremos buscar** los patrones de búsqueda.

| **Metacaracter** | **Descripción** |
|---|---|
| ^ | inicio de línea. |
| $ | fin de línea. |
| \A | inicio de texto. |
| \Z | fin de texto. |
| . | cualquier caracter en la línea. |
| \b | encuentra límite de palabra. |
| \B | encuentra distinto a límite de palabra. |

# METACARACTERES - CLASES PREDEFINIDAS

Son clases predefinidas que **facilitan la utilización** de las expresiones regulares.

| **Metacaracter** | **Descripción** |
|---|---|
| \w | un caracter alfanumérico (incluye "_"). |
| \W | un caracter no alfanumérico. |
| \d | un caracter numérico. |
| \D | un caracter no numérico. |
| \s | cualquier espacio (lo mismo que [ \t\n\r\f]). |
| \S | un no espacio. |

# METACARACTERES - ITERADORES

Pueden **especificar** el **número de ocurrencias** del caracter previo, de un metacaracter o de una subexpresión. los dígitos entre llaves de la forma {n,m}, especifican el mínimo número de ocurrencias en n y el máximo en m.

| Metacaracter | Descripción |
|---|---|
| * | cero o más, similar a {0,}. |
| + | una o más, similar a {1,}. |
| ? | cero o una, similar a {0,1}. |
| {n} | exactamente n veces. |
| {n,} | por lo menos n veces. |
| {n,m} | por lo menos n pero no más de m veces. |
| *? | cero o más, similar a {0,}?. |
| +? | una o más, similar a {1,}?. |
| ?? | cero o una, similar a {0,1}?. |
| {n}? | exactamente n veces. |
| {n,}? | por lo menos n veces. |
| {n,m}? | por lo menos n pero no más de m veces. |

# METACARACTERES - ALTERNATIVAS

Se puede especificar una serie de alternativas para una plantilla usando "|" para separarlas.

Ejemplo: do|re|mi encontrará cualquier "do", "re", o "mi" en el texto de entrada.

Las alternativas son evaluadas de izquierda a derecha, por lo tanto la primera alternativa que coincide plenamente con la expresión analizada es la que se selecciona.

Ejemplo: si se buscan foo|foot en "barefoot", sólo la parte "foo" da resultado positivo, porque es la primera alternativa probada, y porque tiene éxito en la búsqueda de la cadena analizada.

Ejemplo: foo(bar|foo), encuentra las cadenas 'foobar' o 'foofoo'.

# METACARACTERES - SUBEXPRESIONES

La construcción ( ... ) también puede ser empleada para definir subexpresiones de expresiones regulares.

Ejemplos

- (foobar){10}, encuentra cadenas que contienen 8, 9 o 10 instancias de 'foobar‘
- foob([0-9]|a+)r , encuentra 'foob0r', 'foob1r' , 'foobar', 'foobaar', 'foobaar' etc.

# METACARACTERES - MEMORIAS (BACKREFERENCES)

Los metacaracteres \1 a \9 son interpretados como memorias. \ encuentra la subexpresión previamente encontrada #.

Ejemplos

- (.)\1+, encuentra 'aaaa' y 'cc'.
- (.+)\1+, también encuentra 'abab' y '123123'
- (["']?)(\d+)\1, encuentra "13" (entre comillas dobles), o '4' (entre comillas simples) o 77 (sin comillas) etc.

# EXPRESIONES REGULARES CON PYTHON

Python cuenta con el módulo re, que proporciona todas las operaciones necesarias para trabajar con las expresiones regulares.

# EXPRESIONES REGULARES CON PYTHON - BUSCANDO COINCIDENCIAS - IMPORT

## import

Para usar el módulo re, primero se debe importar:

## En Python

```
# importando el modulo de regex de python

import re
```

# EXPRESIONES REGULARES CON PYTHON - BUSCANDO COINCIDENCIAS - COMPILE

## compile

Para buscar coincidencias con un determinado patrón de búsqueda, **primero se debe compilar la expresión regular** en un objeto de patrones de Python, el cual posee métodos para diversas operaciones, tales como la búsqueda de coincidencias de patrones o realizar sustituciones de texto.

## En Python

```
# compilando la regex

patron = re.compile(r'\bfoo\b')

# busca la palabra foo
```

# EXPRESIONES REGULARES CON PYTHON - BUSCANDO COINCIDENCIAS - MÉTODOS

Métodos para buscar coincidencias

- match(): determina si la expresión regular tiene coincidencias en el comienzo del texto.
- search(): escanea todo el texto buscando cualquier ubicación donde haya una coincidencia.
- findall(): encuentra todos los subtextos donde haya una coincidencia y nos devuelve estas coincidencias como una lista.
- finditer(): es similar al anterior pero en lugar de devolvernos una lista nos devuelve un iterador (puntero que permite recorrer los elementos).

# EXPRESIONES REGULARES CON PYTHON - BUSCANDO COINCIDENCIAS - MÉTODOS

## Match

- Determina si la expresión regular tiene coincidencias en el comienzo del texto.
- En Python:

```
import re
# compilando la regex
patron = re.compile(r'\bfoo\b')
# busca la palabra foo
# texto de entrada
texto = """ bar foo bar
foo barbarfoo
foofoo foo bar
"""
# match devuelve None porque no hubo coincidencia al comienzo del texto
print(patron.match(texto))
# salida None
# match encuentra una coindencia en el comienzo del texto
m = patron.match('foo bar')
print(m)
# salida <_sre.SRE_Match object; span=(0, 3), match='foo'>
```

# EXPRESIONES REGULARES CON PYTHON - BUSCANDO COINCIDENCIAS - MÉTODOS

## Search

- Escanea todo el texto buscando cualquier ubicación donde haya una coincidencia.
- En Python:

```
import re
# compilando la regex
patron = re.compile(r'\bfoo\b')
# busca la palabra foo
# texto de entrada
texto = """ bar foo bar
foo barbarfoo
foofoo foo bar
"""
# search devuelve la coincidencia en cualquier ubicación.
s = patron.search(texto)
print(s)
# salida <_sre.SRE_Match object; span=(5, 8), match='foo'>
```

# EXPRESIONES REGULARES CON PYTHON - BUSCANDO COINCIDENCIAS - MÉTODOS

## Findall

- Encuentra todos los subtextos donde haya una coincidencia y nos devuelve estas coincidencias como una lista.
- En Python:

```
import re
# compilando la regex
patron = re.compile(r'\bfoo\b')
# busca la palabra foo
# texto de entrada
texto = """ bar foo bar
# findall devuelve una lista con todas las coincidencias
fa = patron.findall(texto)
print(fa)
# salida ['foo', 'foo', 'foo']
```

# EXPRESIONES REGULARES CON PYTHON - BUSCANDO COINCIDENCIAS - MÉTODOS

## Finditer

- Es similar al anterior pero en lugar de devolvernos una lista nos devuelve un iterador (puntero que permite recorrer los elementos).
- En Python:

```
import re
# compilando la regex
patron = re.compile(r'\bfoo\b')
# busca la palabra foo
# texto de entrada
texto = """ bar foo bar
# finditer devuelve un iterador
fi = patron.finditer(texto)
print(fi)
# Salida <callable_iterator at 0x7f413db74240>
# iterando por las distintas coincidencias
print(next(fi))
# Salida <_sre.SRE_Match object; span=(5, 8), match='foo'>
print(next(fi))
# Salida <_sre.SRE_Match object; span=(13, 16), match='foo'>
```

# EXPRESIONES REGULARES CON PYTHON - BUSCANDO COINCIDENCIAS - OBJETO DE COINCIDENCIA - MÉTODOS

Cuando hay coincidencias, Python devuelve un Objeto de coincidencia (salvo por el método findall() que devuelve una lista).

El Objeto de coincidencia también tiene sus propios métodos que proporcionan información adicional sobre la coincidencia:

- group(): Devuelve el texto que coincide con la expresion regular.
- start(): Devuelve la posición inicial de la coincidencia.
- end(): Devuelve la posición final de la coincidencia.
- span(): Devuelve una tupla con la posición inicial y final de la coincidencia.

# EXPRESIONES REGULARES CON PYTHON - BUSCANDO COINCIDENCIAS - OBJETO DE COINCIDENCIA - MÉTODOS

En Python:

```
import re
patron = re.compile(r'\bfoo\b')
texto = """ bar foo bar
foo barbarfoo
foofoo foo bar
"""
m = patron.match('foo bar')
# Métodos del objeto de coincidencia
print(m.group(), m.start(), m.end(), m.span())
# salida ('foo', 0, 3, (0, 3))
s = patron.search(texto)
# Métodos del objeto de coincidencia
print(s.group(), s.start(), s.end(), s.span())
# salida ('foo', 5, 8, (5, 8))
```

# EXPRESIONES REGULARES CON PYTHON - MODIFICANDO EL TEXTO DE ENTRADA - MÉTODOS

Métodos para modificar el texto de entrada

- Se puede utilizar un patrón, que se utiliza para las búsquedas, para realizar modificaciones al texto de entrada.
- split(): Divide el texto en una lista, realizando las divisiones del texto en cada lugar donde se cumple con la expresión regular.
- sub(): Encuentra todos los subtextos donde existe una coincidencia con la expresion regular y luego los reemplaza con un nuevo texto.
- subn(): Es similar al anterior pero además de devolver el nuevo texto, también devuelve el numero de reemplazos que realizó.

# EXPRESIONES REGULARES CON PYTHON - MODIFICANDO EL TEXTO DE ENTRADA - MÉTODOS

## Split - Python

```
import re
# texto de entrada
becquer = """Podrá nublarse el sol eternamente;
Podrá secarse en un instante el mar;
Podrá romperse el eje de la tierra
como un débil cristal.
¡todo sucederá! Podrá la muerte
cubrirme con su fúnebre crespón;
Pero ¡jamás en mí podrá apagarse
la llama de tu amor."""
# patron para dividir donde no encuentre un caracter alfanumerico
patron = re.compile(r'\W+')
palabras = patron.split(becquer)
palabras[:10]  # 10 primeras palabras
print(palabras[:10])
```

## Salida

```
['Podrá',
 'nublarse',
 'el',
 'sol',
 'eternamente',
 'Podrá',
 'secarse',
 'en',
 'un',
 'instante']
```

# EXPRESIONES REGULARES CON PYTHON - MODIFICANDO EL TEXTO DE ENTRADA - MÉTODOS

## Split - Python

```
import re
# texto de entrada
becquer = """Podrá nublarse el sol eternamente;
Podrá secarse en un instante el mar;
Podrá romperse el eje de la tierra
como un débil cristal.
¡todo sucederá! Podrá la muerte
cubrirme con su fúnebre crespón;
Pero jamás en mí podrá apagarse
la llama de tu amor."""
# patron para dividir donde no encuentre un caracter alfanumerico
patron = re.compile(r'\W+')
# Utilizando la version no compilada de split.
print(re.split(r'\n', becquer))  # Dividiendo por linea.
```

## Salida

```
['Podrá nublarse el sol eternamente; ',
'Podrá secarse en un instante el mar; ',
'Podrá romperse el eje de la tierra ',
'como un débil cristal. ',
'¡todo sucederá! Podrá la muerte ',
'cubrirme con su fúnebre crespón; ',
'Pero jamás en mí podrá apagarse ',
'la llama de tu amor.']
```

# EXPRESIONES REGULARES CON PYTHON - MODIFICANDO EL TEXTO DE ENTRADA - MÉTODOS

## Split - Python

```
import re
# texto de entrada
becquer = """Podrá nublarse el sol eternamente;
Podrá secarse en un instante el mar;
Podrá romperse el eje de la tierra
como un débil cristal.
¡todo sucederá! Podrá la muerte
cubrirme con su fúnebre crespón;
Pero jamás en mí podrá apagarse
la llama de tu amor."""
# patron para dividir donde no encuentre un caracter alfanumerico
patron = re.compile(r'\W+')
# Utilizando el tope de divisiones
print(patron.split(becquer, 5))
```

## Salida

```
['Podrá',
 'nublarse',
 'el',
 'sol',
 'eternamente',
 'Podrá secarse en un instante el mar; \nPodrá romperse el eje de la tierra \ncomo un débil cristal. \n¡todo sucederá! Podrá la muerte \ncubrirme con su fúnebre crespón; \nPero jamás en mí podrá apagarse \nla llama de tu amor.']
```

# EXPRESIONES REGULARES CON PYTHON - MODIFICANDO EL TEXTO DE ENTRADA - MÉTODOS

## Sub - Python

```
import re
# texto de entrada
becquer = """Podrá nublarse el sol eternamente;
Podrá secarse en un instante el mar;
Podrá romperse el eje de la tierra
como un débil cristal.
¡todo sucederá! Podrá la muerte
cubrirme con su fúnebre crespón;
Pero jamás en mí podrá apagarse
la llama de tu amor."""
# Cambiando "Podrá" o "podra" por "Puede"
podra = re.compile(r'\b(P|p)odrá\b')
puede = podra.sub("Puede", becquer)
print(puede)
```

## Salida

Puede nublarse el sol eternamente;

Puede secarse en un instante el mar;

Puede romperse el eje de la tierra

como un débil cristal.

¡todo sucederá! Puede la muerte

cubrirme con su fúnebre crespón;

Pero jamás en mí Puede apagarse

la llama de tu amor.

# EXPRESIONES REGULARES CON PYTHON - MODIFICANDO EL TEXTO DE ENTRADA - MÉTODOS

## Sub - Python

```
import re
# texto de entrada
becquer = """Podrá nublarse el sol eternamente;
Podrá secarse en un instante el mar;
Podrá romperse el eje de la tierra
como un débil cristal.
¡todo sucederá! Podrá la muerte
cubrirme con su fúnebre crespón;
Pero jamás en mí podrá apagarse
la llama de tu amor."""
# Limitando el número de reemplazos
puede = podra.sub("Puede", becquer, 2)
print(puede)
```

## Salida

Puede nublarse el sol eternamente;

Puede secarse en un instante el mar;

Podrá romperse el eje de la tierra

como un débil cristal.

¡todo sucederá! Podrá la muerte

cubrirme con su fúnebre crespón;

Pero jamás en mí podrá apagarse

la llama de tu amor.

# EXPRESIONES REGULARES CON PYTHON - MODIFICANDO EL TEXTO DE ENTRADA - MÉTODOS

## Subn - Python

```
import re
# texto de entrada
becquer = """Podrá nublarse el sol eternamente;
Podrá secarse en un instante el mar;
Podrá romperse el eje de la tierra
como un débil cristal.
¡todo sucederá! Podrá la muerte
cubrirme con su fúnebre crespón;
Pero jamás en mí podrá apagarse
la llama de tu amor."""
# Utilizando la version no compilada de subn
print(re.subn(r'\b(P|p)odrá\b', "Puede", becquer))
# se realizaron 5 reemplazos
```

## Salida

('Puede nublarse el sol eternamente; \nPuede secarse en un instante el mar; \nPuede romperse el eje de la tierra \ncomo un débil cristal. \n¡todo sucederá! Puede la muerte \ncubrirme con su fúnebre crespón; \nPero ¡jamás en mí Puede apagarse \nla llama de tu amor.',

5)

# FUNCIONES NO COMPILADAS

Existen casos donde se utilizan las **funciones al nivel del módulo**: split() y subn().

Para los siguientes métodos: match, search, findall, finditer, split, sub, existe una versión al nivel del módulo que **se puede utilizar sin necesidad de compilar primero el patrón de búsqueda**.

En los casos en los que no se requiere compilar primero, la expresión regular y el resultado será el mismo.

La **ventaja** que tiene la versión que se **compila** sobre las funciones no compiladas es que si se utiliza la expresión regular dentro de un bucle se evitan varias llamadas de funciones y se **mejora la performance** del programa.

# FUNCIONES NO COMPILADAS

Ejemplo de findall con la funcion a nivel del modulo

```
import re
# findall nos devuelve una lista con todas
las coincidencias
texto = """ bar foo bar
foo barbarfoo
foofoo foo bar
"""
print(re.findall(r'\bfoo\b', texto))
```

Salida

```
['foo', 'foo', 'foo']
```

# BANDERAS DE COMPILACIÓN

Permiten modificar algunos aspectos de cómo funcionan las expresiones regulares.

Están disponibles en el módulo re bajo dos nombres:

- Nombre largo como IGNORECASE
- Forma abreviada de una sola letra como I.
- Múltiples banderas pueden ser especificadas utilizando el operador "|" OR
- Ejemplo

  re.I | re.M establece las banderas de E y M.

IGNORECASE, I: Para realizar búsquedas sin tener en cuenta las minúsculas o mayúsculas.

VERBOSE, X: Para ignorar los comentarios y espacios en la expresión.

ASCII, A: Hace que las secuencias de escape \w, \b, \s and \d funciones para coincidencias con los caracteres ASCII.

DOTALL, S: Hace que el metacaracter . funcione para cualquier caracter, incluyendo el las líneas nuevas.

LOCALE, L: Hace que \w, \W, \b, \B, \s, y \S dependientes de la localización actual.

MULTILINE, M: Habilita la coincidencia en múltiples líneas, afectando el funcionamiento de los metacaracteres ^ and $.

# BANDERAS DE COMPILACIÓN

## IGNORECASE

```
import re
# texto de entrada
becquer = """Podrá nublarse el sol eternamente;
Podrá secarse en un instante el mar;
Podrá romperse el eje de la tierra
como un débil cristal.
¡todo sucederá! Podrá la muerte
cubrirme con su fúnebre crespón;
Pero jamás en mí podrá apagarse
la llama de tu amor."""
# Cambiando "Podrá" o "podra" por "Puede"
podra = re.compile(r'podrá\b', re.I)
# el patrón se vuelve más sencillo
puede = podra.sub("puede", becquer)
print(puede)
```

## Salida

puede nublarse el sol eternamente;

puede secarse en un instante el mar;

puede romperse el eje de la tierra

como un débil cristal.

¡todo sucederá! puede la muerte

cubrirme con su fúnebre crespón;

Pero jamás en mí puede apagarse

la llama de tu amor.

# BANDERAS DE COMPILACIÓN

## VERBOSE

```
import re
mail = re.compile(r"""
\b            # comienzo de delimitador de palabra
[\w.%+-]      # usuario: Cualquier caracter alfanumerico mas los signos
(.%+-)
+@            # seguido de @
[\w.-]        # dominio: Cualquier caracter alfanumerico mas los signos (.-)
+\.           # seguido de .
[a-zA-Z]{2,6}  # dominio de alto nivel: 2 a 6 letras en minúsculas o
mayúsculas.
\b            # fin de delimitador de palabra
""", re.X)
mails = """raul.lopez@relopezbriega.com, Raul Lopez Briega,
foo bar, relopezbriega@relopezbriega.com.ar, raul@github.io,
https://relopezbriega.com.ar, https://relopezbriega.github.io,
python@python,   river@riverplate.com.ar,   pythonAR@python.pythonAR
"""
# filtrando los mails con estructura válida
print(mail.findall(mails))
```

## Salida

['raul.lopez@relopezbriega.com',

'relopezbriega@relopezbriega.com.ar',
'raul@github.io',
'river@riverplate.com.ar']

# NOMBRANDO LOS GRUPOS

Se puede colocar nombres a los grupos de las expresiones regulares, para no tener que acceder a los grupos por sus índices.

Se  puede utilizar la siguiente sintaxis especial para nombrar grupos y que sea más fácil identificarlos:

(?P<nombre>patron)

# NOMBRANDO LOS GRUPOS

## Ejemplo: Acceso a los grupos por sus índices

```
import re

# Accediendo a los grupos por sus indices

patron = re.compile(r"(\w+) (\w+)")

s = patron.search("Raul Lopez")

# grupo 1

print(s.group(1))

# grupo 2

print(s.group(2))
```

## Ejemplo: Acceso a los grupos por los nombre

```
import re

# Accediendo a los grupos por nombres

patron = re.compile(r"(?P<nombre>\w+) (?P<apellido>\w+)")

s = patron.search("Raul Lopez")

# grupo nombre

print(s.group("nombre"))

# grupo apellido

print(s.group("apellido"))
```

# VALIDACIONES

## Extensión imagen (compile)

```
import re

regex = re.compile(r"jpg|png|gif|bmp|svg")

img_ext = input("Ingrese una extensión de una imagen: ")

if regex.match(img_ext):

        print('La extensión ', img_ext, 'se corresponde con la extensón de una imagen')

else:

        print('La extensión ', img_ext, 'no se corresponde con la extensón de una imagen')
```

## Extensión imagen a nivel modulo

```
import re

regex = "jpg|png|gif|bmp|svg"

img_ext = input("Ingrese una extensión de una imagen: ")

if re.match(regex,img_ext):

        print('La extensión ', img_ext, 'se corresponde con la extensón de una imagen')

else:

        print('La extensión ', img_ext, 'no se corresponde con la extensón de una imagen')
```

# VALIDACIONES

## IP (compile)

```
import re
regex = re.compile(r"^(?:(?: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]?)$")
ip = input("Ingrese una IP: ")
if regex.match(ip):
        print ('El IP es correcto: ', ip)
else:
        print('No es una IP correcta: ', ip)
```

## IP a nivel modulo

```
import re
print ("validador de ip")
ip = input("\ningrese una ip\n")
if re.match("^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$", ip):
        print ('El IP es correcto: ', ip)
else:
        print('No es una IP correcta: ', ip)
```

# BIBLIOGRAFÍA

https://docs.python.org/es/3/library/re.html

https://www.w3schools.com/python/python_regex.asp

https://regex101.com/

Friedl, Jeffrey. Mastering Regular Expressions. 3a ed., O'Reilly Media, 2006.