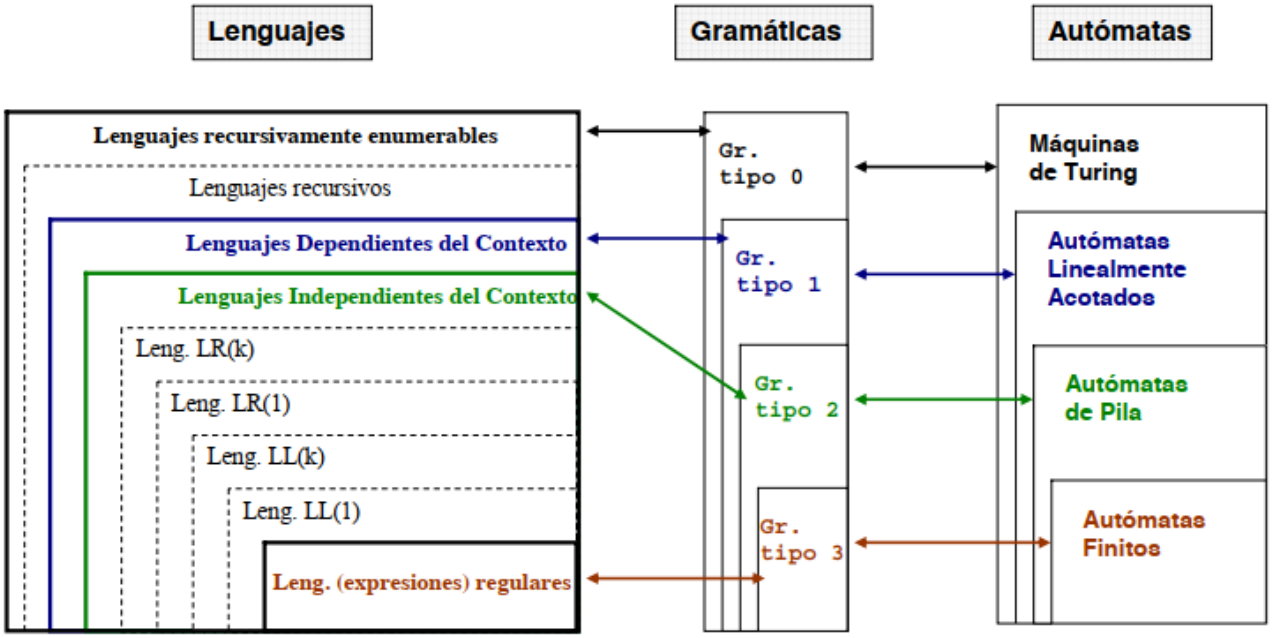
😎 Expresiones Regulares en Python con Re

Introducción

Las expresiones regulares, también llamadas regex, son un conjunto de caracteres que forman un patrón de búsqueda, y que están normalizadas por medio de una sintaxis específica. Los patrones se interpretan como un conjunto de instrucciones, que se ejecutan sobre un texto de entrada para producir un subconjunto o una versión modificada del texto original.

Las expresiones regulares pueden incluir patrones de coincidencia literal, de repetición, de composición, de ramificación, y otras reglas de reconocimiento de texto.

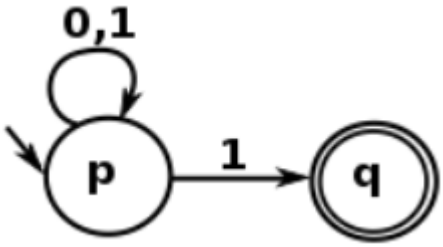
Chomsky clasificó las gramáticas formales (y los lenguajes que éstas generan) de acuerdo a una jerarquía de 4 niveles. A cada nivel de gramática se le puede asociar de forma natural un conjunto de lenguajes que serán los que esas gramáticas generan, pero además, se le puede asociar una clase de autómatas formada por aquellos que podrán reconocer a dichos lenguajes.



Cada nivel de lenguaje se corresponde con un tipo de autómata. Por ejemplo, dado un lenguaje de tipo 3 siempre será posible encontrar un autómata finito que reconozca dicho lenguaje, es decir, que permita determinar si una determinada palabra pertenece o no al lenguaje. Si el lenguaje es de tipo 2 será necesario utilizar un autómata más complejo, concretamente un autómata de pila.

Y cada nivel contiene al anterior. Por ejemplo, cualquier lenguaje de tipo 3 es a su vez un lenguaje de tipo 2, es decir ($L3 \subsetneq L2 \subsetneq L1 \subsetneq L0$). De la misma forma, un autómata finito puede considerarse como un caso particular de autómata de pila y éste como un caso particular de máquina de Turing.

Las expresiones regulares expresan un lenguaje definido por una gramática regular que se puede resolver con un autómata finito no determinista (AFND), donde la coincidencia está representada por los estados. Ejemplo simple:



El lenguaje de Expresión Regular es una representación textual de tal autómata. Ese último ejemplo es expresado por la siguiente regex: **^[01]*1$** Que coincide con cualquier cadena que comience con 0 ó 1, repitiendo 0 o más veces, y que termine con 1. En otras palabras, es una expresión regular para hacer coincidir los números impares de su representación binaria.

Gramáticas

Las gramáticas formales definen un lenguaje describiendo cómo se pueden generar las cadenas del lenguaje. Una gramática formal es una cuádrupla G = (N, T, P, S) donde:

- N es un conjunto finito de símbolos no terminales
- T es un conjunto finito de símbolos terminales $N \cap T = \emptyset$
- P es un conjunto finito de producciones
- S es el símbolo distinguido o axioma $S \notin (N \cup T)$

Gramáticas regulares (Tipo 3)

Generan los lenguajes regulares (aquellos reconocidos por un autómata finito). Son las gramáticas más restrictivas. El lado derecho de una producción debe contener un símbolo terminal y, como máximo, un símbolo no terminal. Estas gramáticas pueden ser:

- Lineales a derecha
- Lineales a izquierda

Expresiones regulares

Se denominan expresiones regulares sobre un alfabeto A, a las expresiones que se pueden construir a partir de las siguientes reglas:

- $\emptyset$ es una expresión regular que describe el lenguaje vacío.
- ε es una expresión regular que describe el lenguaje {ε}, esto es el lenguaje que contiene únicamente la cadena vacía.
- Para cada símbolo $a \in A$, a es una expresión regular que describe el lenguaje {a}, esto es el lenguaje que contiene únicamente la cadena a.
- Si r y s son expresiones regulares que describen los lenguajes L(r) y L(s) respectivamente:
  1. unión: r + s es una expresión regular que describe el lenguaje $L(r) \cup L(s)$
  2. concatenación: r . s es una expresión regular que describe el lenguaje L(r) . L(s)
  3. clausura de Kleene: r* es una expresión regular que describe el lenguaje L(r)*.

El operador de clausura es el que tiene mayor precedencia, seguido por el operador de concatenación y por último el operador de unión. Las expresiones regulares describen los lenguajes regulares (aquellos reconocidos por autómatas finitos).

Componentes de la sintaxis en Python:

- **Literales: Cualquier caracter se encuentra a sí mismo, a menos que se trate de un metacaracter con significado especial. Una serie de caracteres encuentra esa misma serie en el texto de entrada.**

- **Secuencias de escape: La sintaxis permite utilizar las secuencias de escape que conocemos de otros lenguajes de programación.**

Secuencia de escape	Significado
\n	Nueva línea . 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.

Clases de caracteres:	Metacaracteres:
Se pueden especificar encerrando una lista entre corchetes [ ] y encontrará uno cualquiera. Si el primer símbolo después del "[" es "^", encuentra cualquier caracter que no está en la lista.	Son caracteres especiales, sumamente importantes para entender la sintaxis de las expresiones regulares y existen diferentes tipos

Ejemplo:

```
In [1]: print("C:\\Users\\Desktop\\Python\n")
```

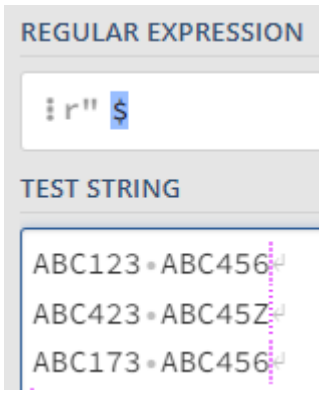
```
C:\Users\Desktop\Python
```

Metacaracteres – delimitadores

Permiten delimitar dónde queremos buscar los patrones de búsqueda. Algunos de ellos son:

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.

Ejemplo:

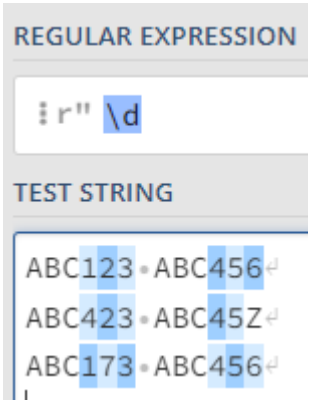


Metacaracteres - clases

Son clases predefinidas que facilitan la utilización de las expresiones regulares. Algunos de ellos son:

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.

Ejemplo:

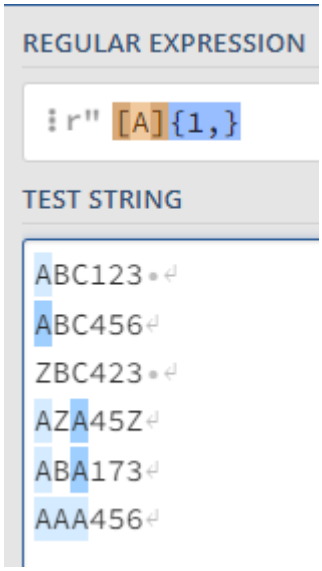


Metacaracteres – iteradores

Cualquier elemento de una expresion regular puede ser seguido por otro tipo de metacaracteres, los iteradores. Usando estos metacaracteres se puede especificar el número de ocurrencias del caracter previo, de un metacaracter o de una subexpresión. Algunos de ellos son:

Metacaracter	Descripción
*	cero o más, similar a {0,}. indica “cero o más coincidencias” del carácter que viene immediatamente antes.
+	una o más, similar a {1,}. indica “por lo menos una coincidencia”.
?	cero o una, similar a {0,1} indica “como máximo una coincidencia” del carácter que viene inmediatamente antes.
{n}	indica “exactamente n coincidencias” del carácter anterior.
{n,}	por lo menos n veces.
{n,m}	por lo menos n pero no más de m veces. indica “entre n y m coincidencias”.
*?	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.
()	sirven para agrupar los términos y especificar el orden de las operaciones.

Ejemplo:



In [2]:
```
import re
```

Metacaracteres

In [3]:
```
texto = 'hi hla hola hoola hooola hooooola huuuuulaaaa'

def buscar(patrones, texto):
    for patron in patrones:
        print(re.findall(patron, texto))
```

In [4]:
```
'''
Metacaracter * : ninguna o más veces ese carácter a su izquierda
'''
patrones = ['ho*la']
buscar (patrones, texto)
```

```
['hla', 'hola', 'hoola', 'hooola', 'hooooola']
```

In [5]:
```
'''
Metacaracter +: una o más repeticiones de la letra a su izquierda
'''
patrones = ['ho+']
buscar(patrones, texto)
```

```
['ho', 'hoo', 'hooo', 'hoooooo']
```

In [6]:
```
'''
Metacaracter ?: una o ninguna repetición de la letra a su izquierda
'''
patrones = ['ho?','ho?la']
buscar(patrones, texto)
```

```
['h', 'h', 'ho', 'ho', 'ho', 'ho', 'h']
['hla', 'hola']
```

Repeticiones y rangos

In [7]:
```
texto = 'hi hla hola hoola hooola hooooola huuuuulaaaa'

def buscar(patrones, texto):
    for patron in patrones:
        print(re.findall(patron, texto))
```

In [8]:
```
'''
Número de repeticiones explicito de la letra a su izquierda: {n}
'''
patrones = ['ho{0}','ho{1}la','ho{3}la']
buscar(patrones, texto)
```

```
['h', 'h', 'h', 'h', 'h', 'h', 'h']
['hola']
['hooola']
```

In [9]:
```
'''
Sintaxis con rango {n, m}
'''
patrones = ['ho{0,1}la', 'ho{1,2}la', 'ho{2,10}la']
buscar(patrones, texto)
```

```
['hla', 'hola']
['hola', 'hoola']
['hoola', 'hooola', 'hoooooola']
```

Conjuntos de caracteres, repeticiones y exclusión

In [10]:
```
def buscar(patrones, texto):
    for patron in patrones:
        print(re.findall(patron, texto))
```

In [11]:
```
'''
Conjuntos de caracteres [ ] - sets
'''
texto = 'hele hala hela hila heli hola hula'
patrones = ['h[ou]la', 'h[aio]la', 'h[aeiou]la']
buscar(patrones, texto)
```

```
['hola', 'hula']
['hala', 'hila', 'hola']
['hala', 'hela', 'hila', 'hola', 'hula']
```

In [12]:
```
'''
Utilizar repeticiones
'''
texto = 'heeele haala heeela hiiiila hoooooola hiiilooo'
patrones = ['h[ae]la', 'h[ae]*la', 'h[io]{3,9}la']
buscar(patrones, texto)
```

```
[]
['haala', 'heeela']
['hiiiila', 'hoooooola']
```

In [13]:
```
'''
Operador de exclusión [^ ] para indicar una búsqueda contraria
'''
texto = 'hala hela hila hilo hola hula'
patrones = ['h[o]la', 'h[^o]la']
buscar(patrones, texto)
```

```
['hola']
['hala', 'hela', 'hila', 'hula']
```

Rangos y códigos escapados

In [14]:
```
def buscar(patrones, texto):
    for patron in patrones:
        print(re.findall(patron, texto))
'''
[A-Z]: Cualquier carácter alfabético en mayúscula (no especial ni número)
[a-z]: Cualquier carácter alfabético en minúscula (no especial ni número)
[A-Za- z]: Cualquier carácter alfabético en minúscula o mayúscula (no especial ni nú
[A-z]: Cualquier carácter alfabético en minúscula o mayúscula (no especial ni numero
[0-9]: Cualquier carácter numérico (no especial ni alfabético)
[a-zA-Z0-9]: Cualquier carácter alfanumérico (no especial)
'''

texto = 'hola h0la Hola mola m0la M0la'
patrones = ['h[a-z]la', 'h[0-9]la', '[A-z]{4}', '[A-Z][A-z0-9]{3}']
buscar(patrones, texto)
```

```
['hola']
['h0la']
['hola', 'Hola', 'mola']
['Hola', 'M0la']
```

In [15]:
```
'''
Códigos escapados:
\d numérico, \D no numérico
\s espacio en blanco, \S no espacio en blanco
\w alfanumérico, \W no alfanumérico
'''
texto = 'Matemática III - 1er. cuatrimestre - 2022'
patrones = [r'\d+', r'\D+', r'\s', r'\S+', r'\w+', r'\W+']
buscar(patrones, texto)
```

```
['1', '2022']
['Matemática III - ', 'er. cuatrimestre - ']
[' ', ' ', ' ', ' ', ' ', ' ']
['Matemática', 'III', '-', '1er.', 'cuatrimestre', '-', '2022']
['Matemática', 'III', '1er', 'cuatrimestre', '2022']
[' ', ' - ', '. ', ' - ']
```

search() escanea todo el texto buscando cualquier ubicación donde haya una coincidencia.

In [16]:
```
texto = "Las expresiones regulares casi son un lenguaje de \
programación en miniatura para buscar y analizar cadenas. De hecho, \
se han escrito libros enteros sobre las expresiones regulares"

a_buscar = "casi"

x = re.search(a_buscar, texto)

'''
Escribe la posicion inicial y final de la ocurrencia.
'''
print(x.span())
```

```
(26, 30)
```

In [17]:
```
'''
Devuelve el texto que coincide con la expresion regular.
'''
print(x.group())
```

```
casi
```

In [18]:
```
'''
devuelve la posición inicial de la coincidencia.
'''
print(x.start())
```

```
26
```

In [19]:
```
'''
Devuelve la posición final de la coincidencia.
'''
print(x.end())
```

```
30
```

Para buscar un patrón en un string podemos utilizar el método search() que busca el patrón dentro del texto y escribe la posición inicial y final de la ocurrencia

match() determina si la regex tiene coincidencias en el comienzo del texto.

In [20]:
```
texto = "Las expresiones regulares casi son un lenguaje de \
programación en miniatura para buscar y analizar cadenas. De hecho, \
se han escrito libros enteros sobre las expresiones regulares"

a_buscar1= "Las"
'''
Busca el patron dentro del texto
'''
x = re.match(a_buscar1, texto)
print(x.span())
```

```
(0, 3)
```

In [21]:
```
a_buscar2 = "expresiones"
y = re.match(a_buscar2, texto)
'''
Error!
'''
print(y.span())
print(y)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [21], in <cell line: 6>()
      2 y = re.match(a_buscar2, texto)
      3 '''
      4 Error!
      5 '''
----> 6 print(y.span())
      7 print(y)

AttributeError: 'NoneType' object has no attribute 'span'
```

El segundo match()dará un error al intentar hacer el print() pues el patrón "expresiones" no se encuentra al principio del texto y por tanto el método match() devuelve Error

findall() encuentra todos los subtextos donde haya una coincidencia y devuelve estas coincidencias como una lista.

In [22]:
```
texto = """El poder de las expresiones regulares se manifiesta cuando agregamos cara
especiales a la cadena de búsqueda que nos permite controlar de manera más precisa
qué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestra
expresión regular nos permitirá buscar coincidencias y extraer datos usando unas
pocas líneas de código."""

a_buscar = "est"
x = re.findall(a_buscar, texto)
print(x)
```

```
['est', 'est', 'est']
```

Busca todas las coincidencias en una cadena, si agregamos len(re.findall(a_buscar, texto)), podríamos saber cuantas veces se repite un patrón dentro de una cadena

finditer() similar a findall pero en lugar de devolver una lista devuelve un iterador.

In [23]:
```
texto = """El poder de las expresiones regulares se manifiesta cuando agregamos cara
especiales a la cadena de búsqueda que nos permite controlar de manera más precisa
qué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestra
expresión regular nos permitirá buscar coincidencias y extraer datos usando unas
pocas líneas de código."""

a_buscar = "est"

x = re.finditer(a_buscar, texto)

for i in x:
    print(i.span())
```

```
(47, 50)
(207, 210)
(239, 242)
```

Tanto match() como search() sólo se quedan con la primera ocurrencia encontrada. Se puede utilizar la función finditer() para buscarlas todas, y se devuelve las posiciones de las ocurrencias. También se puede recorrer una a una con next()

Modificando el texto de entrada

Además de buscar coincidencias podemos utilizar ese mismo patrón para realizar modificaciones al texto de entrada.

In [24]:
```
texto = """Las expresiones regulares casi son un lenguaje de
programación para buscar y analizar cadenas."""

'''
patron para dividir donde no encuentre un caracter alfanumerico
'''
patron = re.compile(r'\W+')
palabras = patron.split(texto)
```

In [25]:
```
'''
8 primeras palabras
'''
palabras[:8]
```

Out[25]:
```
['Las', 'expresiones', 'regulares', 'casi', 'son', 'un', 'lenguaje', 'de']
```

In [26]:
```
'''
Utilizando la version no compilada de split. Dividiendo por linea.
'''
re.split(r'\n', texto)
```

Out[26]:
```
['Las expresiones regulares casi son un lenguaje de',
 'programación para buscar y analizar cadenas.']
```

In [27]:
```
'''
Utilizando el tope de divisiones
'''
patron.split(texto, 3)
```

Out[27]:
```
['Las',
 'expresiones',
 'regulares',
 'casi son un lenguaje de\nprogramación para buscar y analizar cadenas.']
```

In [28]:
```
texto = """El poder de las expresiones regulares se manifiesta cuando agregamos cara
especiales a la cadena de búsqueda que nos permite controlar de manera más precisa
qué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestras
expresiones regulares nos permitirá buscar coincidencias y extraer datos usando unas
pocas líneas de código."""

reg = re.compile(r'\b(R|r)egulares\b')

regex = reg.sub("REGEX", texto)
print(regex)
```

```
El poder de las expresiones REGEX se manifiesta cuando agregamos caracteres
especiales a la cadena de búsqueda que nos permite controlar de manera más precisa
qué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestras
expresiones REGEX nos permitirá buscar coincidencias y extraer datos usando unas
pocas líneas de código.
```

In [29]:
```
regex = reg.sub("REGEX", texto, 1)
print(regex)

re.subn(r'\b(R|r)egulares\b', "REGEX", texto)
```

```
El poder de las expresiones REGEX se manifiesta cuando agregamos caracteres
especiales a la cadena de búsqueda que nos permite controlar de manera más precisa
qué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestras
expresiones regulares nos permitirá buscar coincidencias y extraer datos usando un
as
pocas líneas de código.
```

Out[29]:
```
('El poder de las expresiones REGEX se manifiesta cuando agregamos caracteres\nesp
eciales a la cadena de búsqueda que nos permite controlar de manera más precisa\nq
ué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestras
\nexpresiones REGEX nos permitirá buscar coincidencias y extraer datos usando unas
\npocas líneas de código.',
 2)
```

Banderas de compilación

Las banderas de compilación permiten modificar algunos aspectos de cómo funcionan las expresiones regulares. Todas ellas están disponibles en el módulo re con un nombre largo y una letra que lo identifica.

IGNORECASE, I, búsquedas sin tener en cuenta las minúsculas o mayúsculas. Múltiples banderas pueden ser especificadas utilizando el operador "|" OR.

VERBOSE, X, los comentarios y espacios son ignorados (en la expresión). Es importante indicar que los espacios SON ignorados por lo que si necesitamos indicar un espacio hay que escaparlo.

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

DOTALL, S, permite que el punto (.) coincida con la nueva línea.

LOCALE, L, esta opción hace que \w, \W, \b, \B, \s, y \S dependientes de la localización actual.

MULTILINE, M, dentro de una cadena compuesta de muchas líneas, permite que ^ y $ coincidan con el inicio y el final de cada línea. Normalmente ^ / $ solo coincidiría con el inicio y el final de toda la cadena.

In [30]:
```
'''
IGNORECASE
'''

texto = """El poder de las expresiones regulares se manifiesta cuando agregamos cara
especiales a la cadena de búsqueda que nos permite controlar de manera más precisa
qué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestras
expresiones Regulares nos permitirá buscar coincidencias y extraer datos usando unas
pocas líneas de código."""

reg = re.compile(r'regulares\b', re.I)
regex = reg.sub("REGEX", texto)
print(regex)
```

```
El poder de las expresiones REGEX se manifiesta cuando agregamos caracteres
especiales a la cadena de búsqueda que nos permite controlar de manera más precisa
qué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestras
expresiones REGEX nos permitirá buscar coincidencias y extraer datos usando unas
pocas líneas de código.
```

Validar mails

Una regex aproximada puede ser: \b[\w.%+-]+@[\w.-]+.[a-zA-Z]{2,6}\b

In [31]:
```
'''
VERBOSE
'''

mails = """aaa.bbbbbb@gmail.com, Pepe Pepitito,
ccc.dddddd@yahoo.com.ar, qué lindo día , eeeee@github.io,
https://pypi.org/project/regex/, https://ffffff.github.io,
python@python, river@riverplate.com.ar, pythonAR@python.pythonAR
"""

mail = re.compile(r"""
\b # comienzo de delimitador de palabra
[\w.%+-] # 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)

mail.findall(mails)
```

Out[31]:
```
['aaa.bbbbbb@gmail.com',
 'ccc.dddddd@yahoo.com.ar',
 'eeeee@github.io',
 'river@riverplate.com.ar']
```

Validar una URL

Una regex aproximada puede ser: ^(https?://)?([\da-z.-]+).([a-z.]{2,6})([/\w .-])/?$

In [32]:
```
url = re.compile(r"^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?$")
print(url.search("https://www.python.org/"))
```

```
<re.Match object; span=(0, 23), match='https://www.python.org/'>
```

In [33]:
```
print(url.search("https://www.google.com/!.html"))
```

```
None
```

Validar una dirección IP

Una regex aproximada puede ser: ^(?:(?: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]?)$

In [34]:
```
patron = (r'^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]

ip = re.compile(patron)
ip.search("98.61.125.138")
```

Out[34]:
```
<re.Match object; span=(0, 13), match='98.61.125.138'>
```

In [35]:
```
print(ip.search("256.60.124.136"))
```

```
None
```

Validar una fecha

Una regex aproximada puede ser: ^(0?[1-9]|[12][0-9]|3[01])/(0?[1-9]|1[012])/((19|20)\d\d)$

In [36]:
```
fecha = re.compile(r'^(0?[1-9]|[12][0-9]|3[01])/(0?[1-9]|1[012])/((19|20)\d\d)$')
print(fecha.search("2/10/1990"))
```

```
<re.Match object; span=(0, 9), match='2/10/1990'>
```

In [37]:
```
print(fecha.search("2-10-1990"))
```

```
None
```

In [38]:
```
print(fecha.search("32/12/2021"))
```

```
None
```

In [39]:
```
print(fecha.search("30/13/2020"))
```

```
None
```

Análisis the HTML mediante expresiones regulares

La regex del ejemplo busca cadenas que comiencen con “href="http://” o “href=https://, seguido de uno o más caracteres (.+?), seguidos por otra comilla doble . El signo de interrogación después de [s]? , indica que la coincidencia debe ser hecha en modo "no codicioso", en vez de en modo "codicioso".

- **Una búsqueda no-codiciosa intenta encontrar la cadena coincidente más pequeña posible, mientras que**
- **Una búsqueda codiciosa intentaría localizar la cadena coincidente más grande.**

Agregamos paréntesis a la regex para indicar qué parte de la cadena localizada queremos extraer.

In [40]:
```
import urllib.request as ur

url= 'https://docs.python.org'
```

In [41]:

```
'''
probamos con https://docs.python.org ó http://python.org
'''
with ur.urlopen(url) as f:
    html = f.read().decode('utf-8')

regex = re.compile('href="(http*s+?://.*?)"', re.IGNORECASE)
links = re.findall(regex, html)

for link in links:
    print(link)
```

```
https://docs.python.org/3/index.html (https://docs.python.org/3/index.html)
https://www.python.org/ (https://www.python.org/)
https://docs.python.org/3.12/ (https://docs.python.org/3.12/)
https://docs.python.org/3.11/ (https://docs.python.org/3.11/)
https://docs.python.org/3.10/ (https://docs.python.org/3.10/)
https://docs.python.org/3.9/ (https://docs.python.org/3.9/)
https://docs.python.org/3.8/ (https://docs.python.org/3.8/)
https://docs.python.org/3.7/ (https://docs.python.org/3.7/)
https://docs.python.org/3.6/ (https://docs.python.org/3.6/)
https://docs.python.org/3.5/ (https://docs.python.org/3.5/)
https://docs.python.org/2.7/ (https://docs.python.org/2.7/)
https://www.python.org/doc/versions/ (https://www.python.org/doc/versions/)
https://www.python.org/dev/peps/ (https://www.python.org/dev/peps/)
https://wiki.python.org/moin/BeginnersGuide (https://wiki.python.org/moin/Beginner
sGuide)
https://wiki.python.org/moin/PythonBooks (https://wiki.python.org/moin/PythonBook
s)
https://www.python.org/doc/av/ (https://www.python.org/doc/av/)
https://devguide.python.org/ (https://devguide.python.org/)
https://www.python.org/ (https://www.python.org/)
https://devguide.python.org/docquality/#helping-with-documentation (https://devgui
de.python.org/docquality/#helping-with-documentation)
https://docs.python.org/3.12/ (https://docs.python.org/3.12/)
https://docs.python.org/3.11/ (https://docs.python.org/3.11/)
https://docs.python.org/3.10/ (https://docs.python.org/3.10/)
https://docs.python.org/3.9/ (https://docs.python.org/3.9/)
https://docs.python.org/3.8/ (https://docs.python.org/3.8/)
https://docs.python.org/3.7/ (https://docs.python.org/3.7/)
https://docs.python.org/3.6/ (https://docs.python.org/3.6/)
https://docs.python.org/3.5/ (https://docs.python.org/3.5/)
https://docs.python.org/2.7/ (https://docs.python.org/2.7/)
https://www.python.org/doc/versions/ (https://www.python.org/doc/versions/)
https://www.python.org/dev/peps/ (https://www.python.org/dev/peps/)
https://wiki.python.org/moin/BeginnersGuide (https://wiki.python.org/moin/Beginner
sGuide)
https://wiki.python.org/moin/PythonBooks (https://wiki.python.org/moin/PythonBook
s)
https://www.python.org/doc/av/ (https://www.python.org/doc/av/)
https://devguide.python.org/ (https://devguide.python.org/)
https://www.python.org/ (https://www.python.org/)
https://www.python.org/psf/donations/ (https://www.python.org/psf/donations/)
https://www.sphinx-doc.org/ (https://www.sphinx-doc.org/)
```

Leer (https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/web-scraping-con-python/)

Ejemplo (https://www.octoparse.es/blog/web-scraping-con-python)

Expresiones Regulares, archivos y grupos

In [42]:
```
with open("archs/codigos_postales.txt", encoding="utf-8") as f_codigos_postales:
    codigos = {}
    for linea in f_codigos_postales:
        res1 = re.search(r"[\d ]+([^\d]+[a-z])\s(\d+)", linea)
        if res1:
            ciudad, cp = res1.groups()
            if ciudad in codigos:
                codigos[ciudad].add(cp)
            else:
                codigos[ciudad] = {cp}

with open("archs/ciudades.txt", encoding="utf-8") as f_ciudades:
    for linea in f_ciudades:
        res2 = re.search(r"^[0-9]{1,2}\.\s+([\w\s-]+\w)\s+[0-9]", linea)
        ciudad = res2.group(1)
        print(ciudad, codigos[ciudad])
        print('\n')
```

```
Berlin {'13053', '10178', '10555', '10789', '12489', '14195', '13355', '13405',
'12683', '13439', '14167', '12526', '10435', '13435', '12357', '12045', '14199',
'13465', '12109', '14165', '10827', '10551', '12487', '14129', '13051', '10407',
'10963', '12055', '13353', '10315', '13629', '12053', '13627', '10709', '12279',
'10117', '10629', '12157', '13127', '12349', '12587', '12555', '10707', '14052',
'13129', '12351', '12161', '14109', '12359', '12167', '12247', '12623', '13593',
'13187', '10318', '10715', '13437', '12527', '14053', '13407', '10589', '13583',
'10405', '12203', '10439', '10119', '12435', '13089', '13159', '12621', '10781',
'12355', '12439', '10369', '12689', '13189', '10319', '10779', '12307', '12685',
'10711', '13349', '13589', '13507', '10623', '13595', '10179', '10969', '13469',
'12209', '12459', '12305', '14163', '13587', '10777', '13088', '14055', '12353',
'13585', '10967', '10585', '13351', '13403', '10587', '13059', '14059', '13158',
'12057', '12103', '13505', '12681', '13467', '10783', '12059', '13599', '12627',
'10243', '10965', '12205', '10627', '12249', '14197', '10823', '13581', '12107',
'12679', '10999', '13509', '13357', '10409', '10625', '10829', '12687', '12049',
'12159', '12559', '12437', '14193', '10713', '12051', '13591', '10559', '10247',
'10367', '10557', '12619', '13503', '10317', '13347', '10249', '12099', '10115',
'12207', '12347', '13597', '13409', '14050', '12169', '12043', '10437', '14057',
'12277', '12524', '12589', '10785', '10825', '12309', '10553', '13156', '13086',
```

dir() permite localizar los métodos del módulo.

In [43]:
```
print(dir(re))
```

```
['A', 'ASCII', 'DEBUG', 'DOTALL', 'I', 'IGNORECASE', 'L', 'LOCALE', 'M', 'MULTILIN
E', 'Match', 'Pattern', 'RegexFlag', 'S', 'Scanner', 'T', 'TEMPLATE', 'U', 'UNICOD
E', 'VERBOSE', 'X', '_MAXCACHE', '__all__', '__builtins__', '__cached__', '__doc_
_', '__file__', '__loader__', '__name__', '__package__', '__spec__', '__version_
_', '_cache', '_compile', '_compile_repl', '_expand', '_locale', '_pickle', '_spec
ial_chars_map', '_subx', 'compile', 'copyreg', 'enum', 'error', 'escape', 'findal
l', 'finditer', 'fullmatch', 'functools', 'match', 'purge', 'search', 'split', 'sr
e_compile', 'sre_parse', 'sub', 'subn', 'template']
```

help(), brinda ayuda sobre los objetos, clases y métodos.

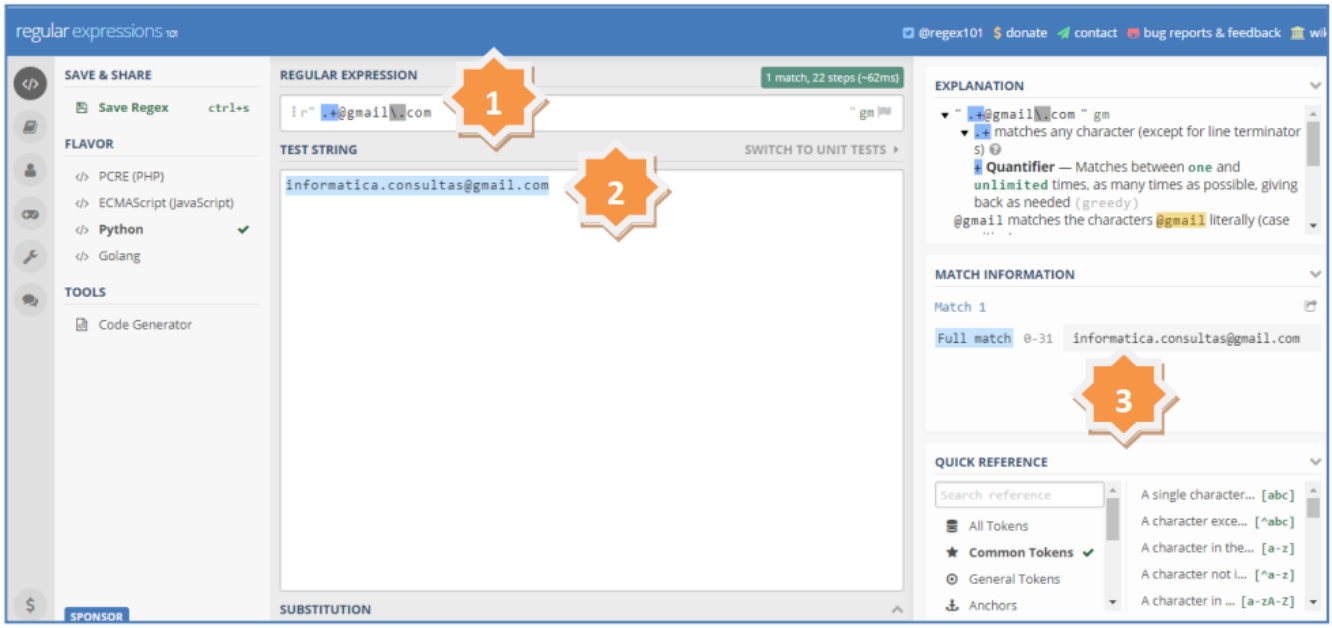
In [44]:
```
help(re.search)
```

```
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a Match object, or None if no match was found.
```

Cómo usar Regex101.com para construir expresiones regulares

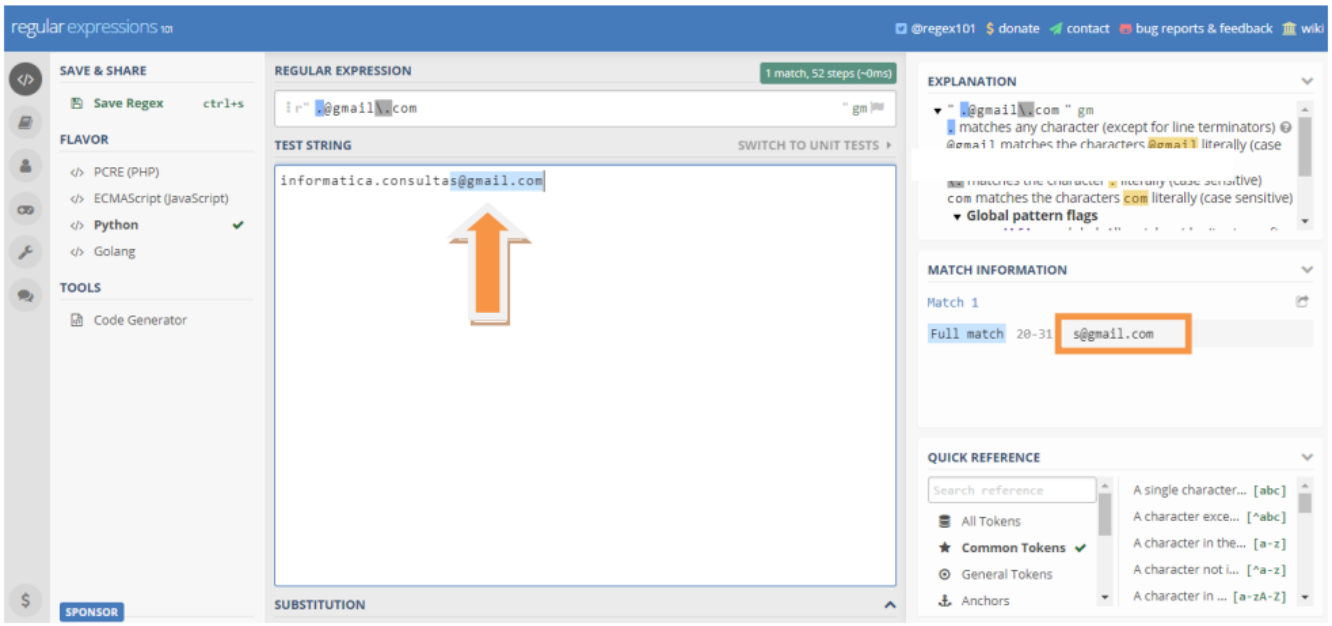
Para empezar con la validación de un correo la expresión regular puede ser: [.+@gmail.com (mailto:.+@gmail.com)](mailto:.+@gmail.com)



1. Se escribe la expresión regular,
2. Se escribe el texto en donde vamos a realizar la búsqueda,
3. Se emite el resultado, como vemos en el ejemplo, encontró que en el texto de búsqueda sí existe un correo electrónico con el dominio.

Cómo funciona esta expresión regular?

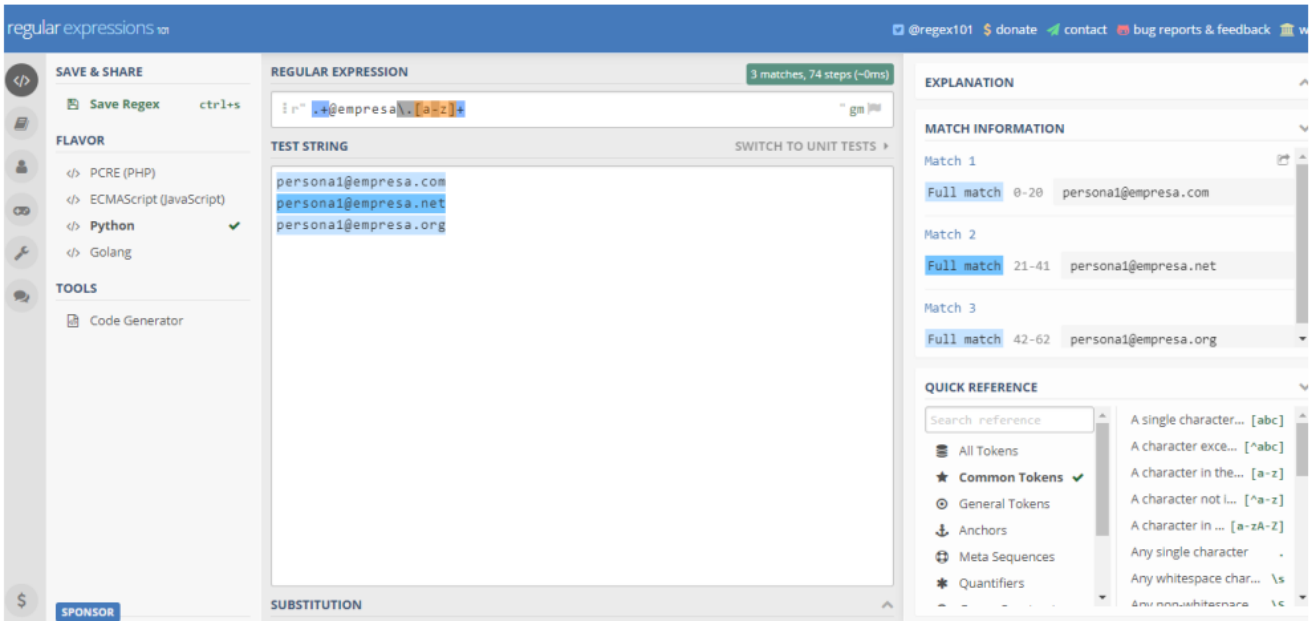
Primero utilice un punto **(.)** significa que espera cualquier carácter (letras, números o símbolos) pero un solo **(.)** Cómo se vería la expresión regular si solo usamos el **(.)** sin el signo **(+)** del paso anterior?:



Como se muestra, solo encontró un carácter (la letra s que es la que está junto a la **@**), entonces el signo **+** le dice a la expresión regular que el patrón que esta antes que él (a la izquierda) puede repetirse más de una vez.

Luego está un texto que es el dominio que buscamos gmail.com, pero aquí hay un problema, el dominio lleva un **(.)** y como vimos antes el **(.)** le dice que representa cualquier carácter, entonces tomaría como válidos otros dominios, para evitar esto se usa un carácter de escape **** antes del **.** esto hace que el **.** o cualquier otra comando de expresión regular sea ignorado y tratado como texto.

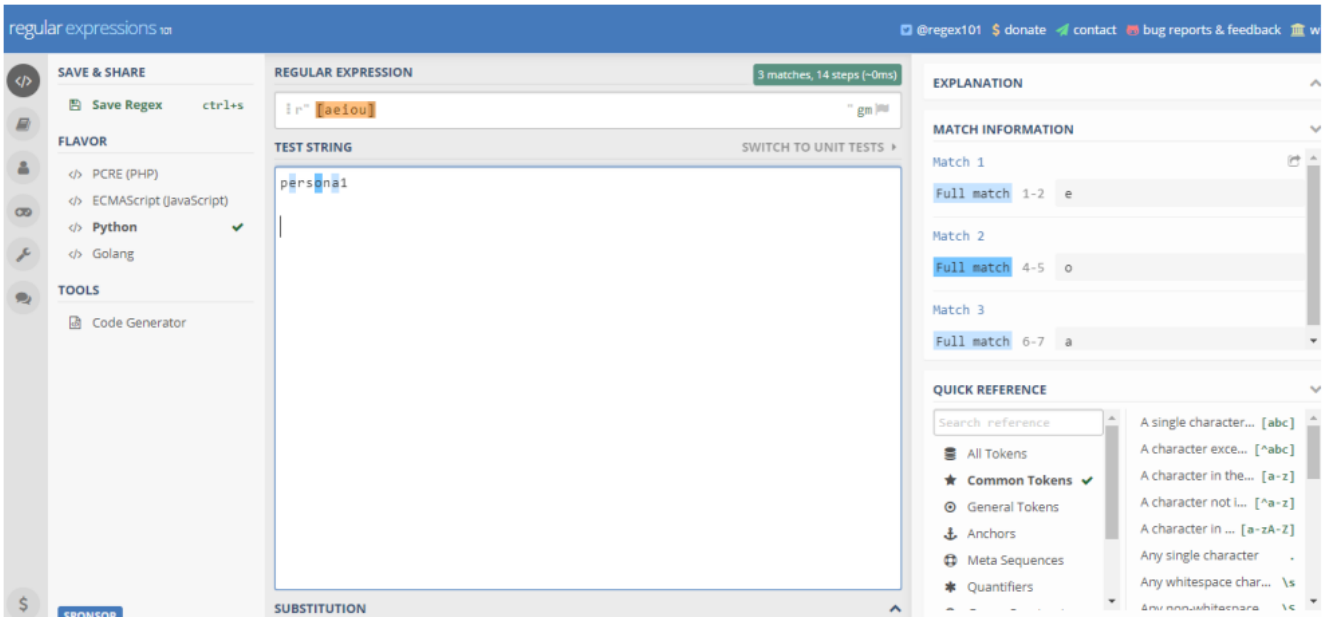
Vamos a ajustar un poco este código para que pueda encontrar un email con el dominio empresa pero en versiones .com .net .org etc. Cambiemos la regex por esta **.+@empresa.[a-z]+**



Con una sola expresión regular encontramos el dominio empresa en las versiones .com .net y .org. El cambio fue sustituir com por [a-z] los signos **[** y **]** se usan para definir un rango, en este caso le dice que puede ir cualquier carácter de la a a la z y el signo **+** al final le dice que ese patrón se puede repetir más de una vez.

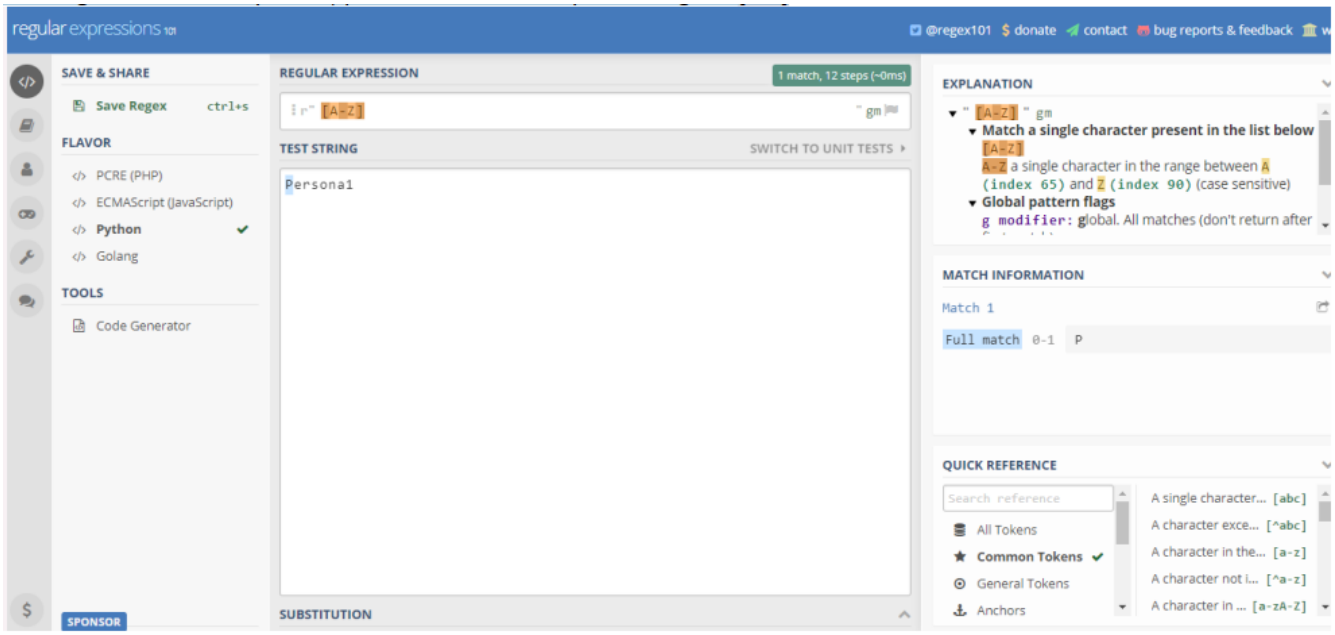
Caracteres, grupos de caracteres y rangos

Las expresiones regulares pueden contener palabras, grupos o rangos, por ejemplo una regex pueden usarse para buscar una palabra específica pero también podemos definir que contenga un rango de caracteres o grupo usando los signos **[** y **]**, por ejemplo la expresión regular **[aeiou]** puede usarse para saber si el texto contiene una vocal.

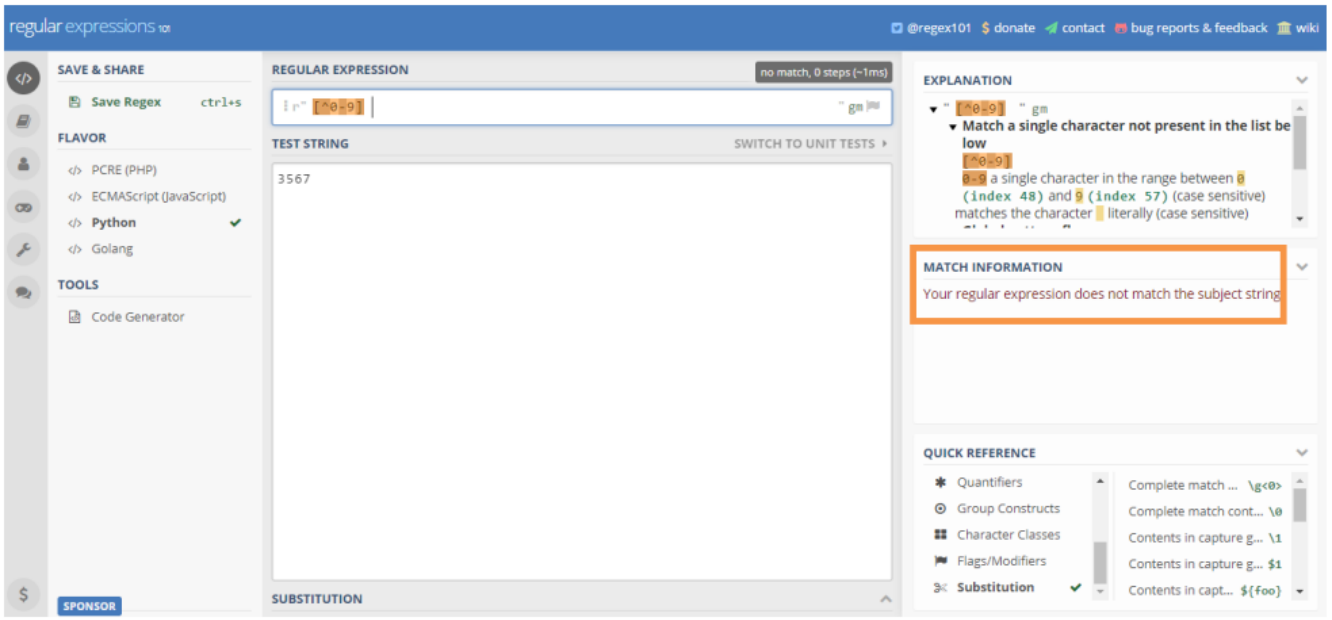


Como vemos la expresión regular encontró las tres vocales en la palabra Persona1, y detectó que la palabra contiene vocales. También hay que notar que cada letra [aeiou] se considera de forma independiente y no como una palabra.

Si lo que necesitamos es buscar un rango debemos separar el rango por un **(-)** por ejemplo para buscar palabras que contengan una letra mayúscula, podemos usar esta expresión regular [A-Z]



También podemos «negar» el rango buscando el resultado opuesto con el caracter **^** al inicio del rango, por ejemplo la expresión **[^0-9]** intenta buscar caracteres que no sean números. Al aplicar la expresión a un texto con números, entonces no encuentra ninguna coincidencia.



Cuantificadores de expresiones regulares

Estos se colocan después de un rango, texto o meta-caracter para modificar o definir la cantidad de veces que debe repetirse o encontrarse. Aquí está la lista de ellos:

- ? Coincide 0 ó 1 vez.
- *? cero o más, similar a {0,}?. Coincide 0 ó más veces
- +? una o más, similar a {1,}?. Coincide 1 ó más veces
- {n} Coincide exactamente n veces, donde n es un número entero.
- {n,} Coincide al menos n veces, donde n es un número entero.
- {,m} Coincide un máximo de m veces, donde m es un número entero.
- {n,m} Coincide exactamente de n a m veces, donde n y m son números entero.

Meta-caracteres

En las expresiones regulares existen caracteres especiales con un significado y tratamiento especial, a estos se les llama meta-caracteres o caracteres especiales, aquí hay una lista de ellos:

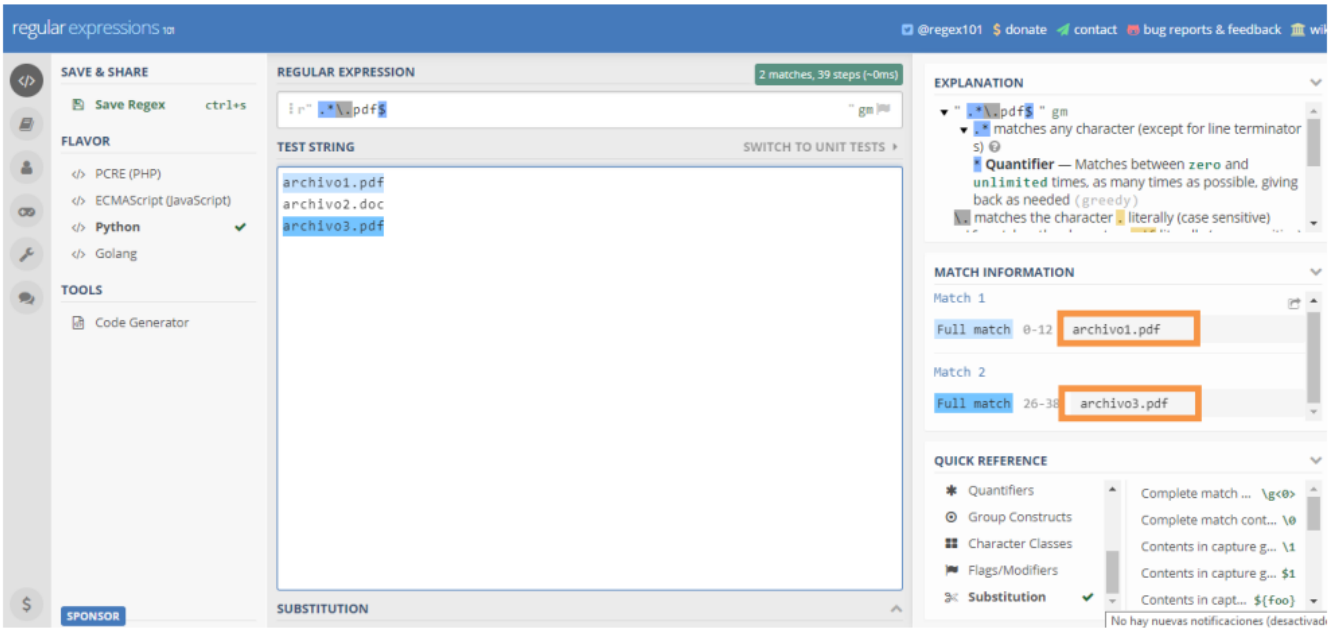
- . Representa cualquier caracter excepto el salto de línea
- \w Representa cualquier letra o número

- \W Representa cualquier caracter que no sea una letra o un número.
- \d Representa cualquier número del 0 al 9
- \D Representa cualquier caracter que no sea un número del 0 al 9
- \s Representa un espacio en blanco
- \S Representa cualquier caracter que no sea un espacio en blanco.
- $ Representa que ahí finaliza el texto, por ejemplo la expresión com$ busca que com sea lo último en el texto. Este caracter solo puede usarse al final de la expresión regular.
- ^ Representa el inicio del texto, por ejemplo la expresión ^hola busca que el texto inicie con hola, este caracter solo puede usarse al inicio de la expresión regular.
- \b Representa que ahí inicia o finaliza una palabra, por ejemplo la expresión \b[A-Z][a-z]* busca palabras que inicien con una letra mayúscula y luego lleven cualquier cantidad de letras minúsculas. En cambio la expresión \w*os\b busca palabras que finalicen en os.

Ejemplos prácticos.

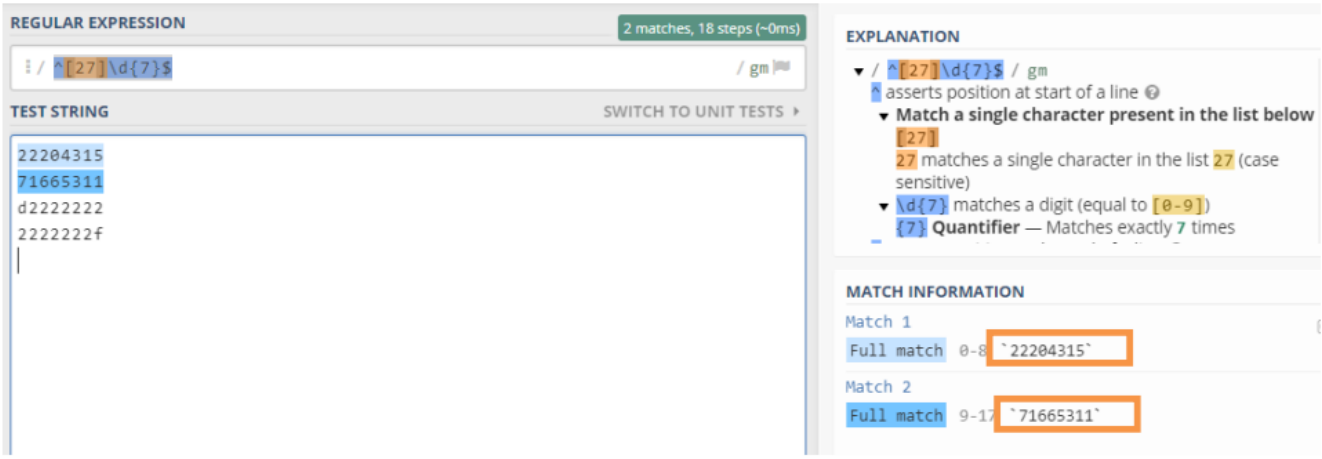
1. Detectar si un archivo es de tipo pdf, en base a su nombre y extensión, para ello podemos usar una expresión como esta: .*.pdf$

Lo que hace la expresión es buscar cualquier caracter, sea numero, letra o símbolo usando el . luego se coloca un * para decir que puede ir cualquier cantidad de caracteres, después sabemos que la extensión de los archivos inicia con un . pero como él . es un caracter especial usamos un signo \ para que lo ignore y trate como un simple . luego, debemos buscar el texto pdf y este debe ser lo último en el texto, no debe haber nada después, esto lo definimos usando el signo $



2. Validar un número telefónico que solo puede iniciar con un 2 o un 7 y debe de tener exactamente 8 números, puede usarse esta expresión regular: ^[27]\d{7}$

Primero usamos el signo ^ para decir que estamos por definir el inicio de la cadena, luego le decimos que debe llevar un 2 o un 7 con [27] ahora faltan 7 números más para completar los 8 números, entonces usamos \d para decir que continua un número del 0 al 9 y usamos {7} para definir que este último patrón se repite 7 veces, luego usamos $ para definir que aquí debe terminar el texto (después de los últimos 7 números.)



Documentación:

https://docs.python.org/3.10/library/re.html?highlight=re#module-re (https://docs.python.org/3.10/library/re.html?highlight=re#module-re)

https://pypi.org/project/regex/ (https://pypi.org/project/regex/)

https://rico-schmidt.name/pymotw-3/re/index.html (https://rico-schmidt.name/pymotw-3/re/index.html)

Bibliografía:

https://www.frlp.utn.edu.ar/materias/sintaxis/automatas-lenguajes-computacion.pdf (https://www.frlp.utn.edu.ar/materias/sintaxis/automatas-lenguajes-computacion.pdf)

https://www.regular-expressions.info/ (https://www.regular-expressions.info/)

Jugar:

https://regexcrossword.com/ (https://regexcrossword.com/)

Regex dictionary

https://www.visca.com/regexdict/ (https://www.visca.com/regexdict/)

App:

https://regex101.com/ (https://regex101.com/)

https://pythex.org/ (https://pythex.org/)

https://www.debuggex.com/ (https://www.debuggex.com/)

Consulta:

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