UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

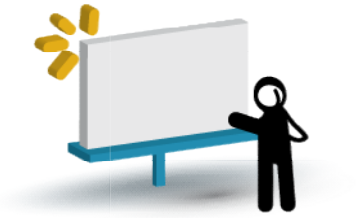
UNIDAD DIDÁCTICA VII

DIPLOMATURA EN PYTHON

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo II – Nivel Inicial II

Unidad III – Expresiones regulares I.



Presentación:

Las expresiones regulares (regex) son patrones de caracteres que conforman un patrón de búsqueda, son extremadamente útiles para extraer información de cualquier texto al buscar una o más coincidencias de un patrón de búsqueda específico (es decir, una secuencia específica de caracteres ASCII o Unicode).

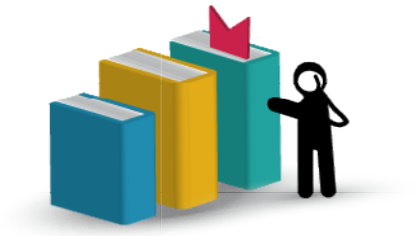
Los campos de aplicación van desde la validación hasta el análisis, reemplazo de cadenas y la traducción de datos a otros formatos. Esta herramienta no es exclusiva de python, la podemos utilizar en otros lenguajes como JavaScript, Java, VB, C #, C / C ++, Python, Perl, Ruby y muchos otros, aunque existen pequeñas diferencias en las sintaxis al pasar de un lenguaje a otro que deberemos tener en cuenta.

Python ha realizado modificaciones en la sintaxis en las últimas versiones de la rama 3.x, en el siguiente material consideraremos que estamos trabajando con la versión 3.7.

Objetivos:

Que los participantes:

Se introduzcan en el mundo de las expresiones regulares de Python y en su terminología básica aplicada.



Bloques temáticos:

1.- Introducción.

2.- Caracteres, clases y rangos.

3.- Meta-caracteres.

4.- Repetición.

5.- Sub-patrones – ()

6.- Herramientas de trabajo.

7.- Descripción del módulo.

Consignas para el aprendizaje colaborativo

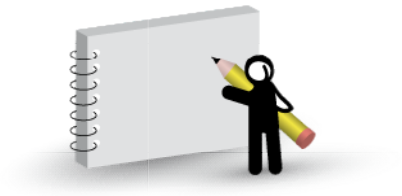
En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.



Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1.- Introducción.

Una expresión regular es un conjunto de caracteres que conforman un patrón de búsqueda. Son muy útiles cuando tenemos que realizar búsquedas en bases de datos o archivos, cuando tenemos que modificar una url para tornarla amigable, etc. Pueden ser utilizadas con cualquier lenguaje de programación moderno.

Para poder utilizarlas en python, podemos importar el módulo “re” y utilizar los métodos:

compile(): Para tomar la expresión regular, la cual en este caso es ‘p+’

match(): Para comparar la expresión regular con el string sobre el cual estamos buscando.

regex1.py

```
import re

patron = re.compile('p+')
lista = ['pera', 'vfg']

print(patron.match(lista[0]))
print(patron.match(lista [1]))
```

El código anterior retorna:

```
<re.Match object; span=(0, 1), match='p'>
None
```

Podemos ver que el patrón “**p+**” está encontrando una coincidencia en el string “**pera**” el cual es el primer elemento de la lista. Esto se da ya que el carácter “+” es un cuantificador de una o más veces, que lo que está diciendo, es que la letra “**p**” se puede repetir una o más veces. En la palabra “**p**era” la letra “**p**” se encuentra al inicio por lo que encuentra en este caso una coincidencia.

En python, se mezclan tres formas de expresiones regulares, que no están identificadas de esta forma y en el manual se consideran una sola, lo cual hace que al inicio trabajar con esta herramienta sea un dolor de cabeza:

- **Primera**: Posix, es la que utiliza las barras inclinadas como: **\w, \W, \s, \S.** Aquí por ejemplo **\s** significa un espacio en blanco, y **\S** que no es un espacio en blanco. La mayúscula siempre es lo contrario a la minúscula y niega.
- **Segunda** - PCRE, es la que usa clases **[a-zA-Z0-9]**
- **Tercera** - Banderas del módulo re de python (es una forma particular solo de python), como en **flags=re.IGNORECASE** o **flags=re.I**

Esta tercera opción se usa para abreviar, por ejemplo poner el **flag I** es equivalente a en PCRE poner **[a-zA-Z]**, ya que se está considerando mayúsculas y minúsculas.

La confusión muchas veces se presenta pues en python se puede mezclar todo y nos podemos encontrar por ejemplo con:

- [a-zA-Z0-9 áéíóú]
- [a-zA-Z\d áéíóú]
- [a-zA-Z\d\sáéíóú]
- [a-z\d\sáéíóú] con una bandera I

En todas estas variantes estamos buscando un carácter alfanumérico con espacios y vocales con acentos. Python contempla a diferentes programadores, tantos los viejos, que han aprendido con Posix como los nuevos que lo han hecho con PCRE, como los que vienen de otros lenguajes y manejan ambos o incluso agrega opciones propias como los flags.

Recomendación: Las REGEX son conveniente tomarlas como una ayuda y no intentar aprenderlas todas sin un objetivo concreto, es decir, es conveniente por ejemplo ponerse el objetivo de lograr validar un campo de nombre y a partir de ahí tratar de ver cómo realizarlo.

2. Caracteres, clases y rangos

Las expresiones regulares utilizan conjuntos de caracteres, clases, rangos y conjuntos de rangos.

Caracteres

Los caracteres son símbolos que actúan como comodines. En nuestro primer ejemplo al considerar la expresión regular 'p+', la letra: p puede ser usada para encontrar la primer ocurrencia de la letra a dentro del string 'pera' o 'vfg' y el signo + indica que la p se puede repetir una o más veces:

Clases

La parte de un patrón que está entre corchetes se llama una "clase carácter". Las clases de caracteres van *entre corchetes*

[aeiou]uto	Específica conjuntos de caracteres aeiou delante de **uto**
a[up]to	Coincide con **auto** o con **apto**

Rangos

Para indicar un rango de caracteres dentro de una clase se utiliza un *guión*.

[0-9]	Dígitos entre 0 y 9
[a-z]	Indica de la 'a' a la 'z' minúsculas.
[a-zA-Z]	Indica de la 'a' a la 'z' minúsculas o mayúsculas.

3. Meta-caracteres.

El poder de las expresiones regulares viene dado por la capacidad de incluir alternativas y repeticiones en el patrón. Éstos están codificados en el patrón por el uso de meta-caracteres, los cuales no se representan a sí mismos, sino que son interpretados de una forma especial. Como ejemplo de meta-caracter podemos nombrar los corchetes '[ ]' que como ya hemos visto se utilizan para especificar una clase carácter.

Como iremos viendo, contamos con otros meta-caracteres:

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

Meta-caracter \

Quizás el metacarácter *más importante es la barra invertida*, \. Al igual que en los literales de cadena de Python, la barra invertida puede ir seguida de varios caracteres para señalar varias secuencias especiales. También se usa para escapar otros meta-caracteres, por ejemplo, si necesita hacer coincidir un "[" o "\", se los precede con una barra invertida para eliminar su significado especial: \ [o \\.

Algunas de las secuencias especiales que comienzan con '\' representan conjuntos predefinidos de caracteres que a menudo son útiles, como el conjunto de dígitos, el conjunto de letras o el conjunto de cualquier cosa que no sea un espacio en blanco.

Por ejemplo: \w coincide con cualquier carácter alfanumérico, lo cual sería equivalente a [a-zA-Z0-9_]. Podemos hacer un resumen de esta finalidad en la siguiente tabla:

Expresión	Definición
\A	Coincide sólo al comienzo de la cadena.
\b	Marca la posición de una palabra limitada por espacios en blanco, puntuación o el inicio/final de una cadena.
\d	Coincide con cualquier dígito decimal; esto es equivalente a la clase [0-9].
\D	Coincide con cualquier carácter que no sea un dígito; esto es equivalente a la clase [^ 0-9].
\s	Coincide con cualquier carácter de espacio en blanco; esto es equivalente a la clase [\ t \ n \ r \ f \ v].
\S	Coincide con cualquier carácter que no sea un espacio en blanco; esto es equivalente a la clase [^ \ t \ n \ r \ f \ v].

\w	Coincide con cualquier carácter alfanumérico; esto es equivalente a la clase [a-zA-Z0-9_].
\W	Coincide con cualquier carácter no alfanumérico; esto es equivalente a la clase [^ a-zA-Z0-9_].

Como se indicó anteriormente, las expresiones regulares usan el carácter de barra diagonal inversa ('\') para indicar formas especiales o para permitir que se utilicen caracteres especiales sin invocar su significado especial.

Meta-caracter ^

Este meta-carácter llamado *circunflejo* tiene dos usos:

Fuera de una clase caracter, en el modo de comparación por defecto, el caracter **circunflejo** (^) es una declaración que es verdadera sólo si el punto de coincidencia actual está en el inicio de de la cadena objetivo. Dentro de una clase caracter, circunflejo (^) tiene un significado totalmente diferente (véase más adelante).

Circunflejo (^) no necesita ser el primer caracter del patrón si están implicadas varias alternativas, pero debería ser la primera cosa en cada alternativa en la que aparece si el patrón es comparado siempre con esa rama. Si todas las posibles alternativas comienzan con un circunflejo (^), es decir, si el patrón es obligado a coincidir sólo con el comienzo de la cadena objetivo, se dice que el patrón está "anclado". (También hay otras construcciones que pueden causar que un patrón esté anclado.)

Meta-caracter $

Un caracter pesos (*$*) es una declaración, la cual es TRUE sólo si el punto actual de coincidencia está al final de la cadena objetivo, o inmediatamente antes de un caracter de nueva línea que es el último caracter en la cadena (por defecto). Pesos (*$*) no necesita ser el último caracter del patrón si están implicadas varias alternativas, pero debería ser el último elemento en cualquier rama en la que aparezca. Pesos no tiene un significado especial en una clase caracter.

Meta-caracter .

Este meta-caracter puede ser utilizado para sustituirlo por cualquier caracter.

Meta-caracter |

El carácter barra vertical se usa para separar patrones alternativos. Por ejemplo, el patrón “pera|manzana” coincide con "pera" o con "manzana". Pueden aparecer cualquier número de alternativas, y se permite una alternativa vacía (coincidiendo con la cadena vacía).

regex2.py

```
import re

patron = re.compile('pera"|manzana')
string = "manzana"

print(patron.match(string))
```

4. Repetición

La repetición se especifica mediante cuantificadores.

Por ejemplo:

z{2,4} coincide con "zz", "zzz", o "zzzz". Una llave de cierre por sí misma no es un caracter especial.
[aeiou]{3,} coincide al menos con 3 vocales sucesivas, pero puede coincidir con muchas más
\d{8} coincide exactamente con 8 dígitos.

Cuantificadores de carácter simple	
*	equivale a *{0,}*
+	equivale a *{1,}*
?	equivale a *{0,1}*

colou?r	identifica colour o color
\b[1-9][0-9]{3}\b	Identifica un número entre 1000 y 9999
\b[1-9][0-9]{2,4}\b	Identifica un número entre 100 y 99999
<[A-Za-z][A-Za-z0-9]*>	Identifica una etiqueta html sin atributos
<[A-Za-z0-9]+>	Puede identificar etiquetas inválidas como <1>

5. Sub-patrones – ( )

Los sub-patrones están delimitados por paréntesis, los cuales pueden estar anidados. Localiza un conjunto de alternativas.

Por ejemplo: *cata(rata|pulta|)*

Con paréntesis coincide con una de las palabras "cata", "catarata", o "catapulta".
Sin los paréntesis, coincidiría con "catarata", "pulta" o la cadena vacía.

Si incluimos dentro del sub patrón un signo de interrogación a continuación de la apertura del paréntesis (?...), el primer caracter luego del signo de interrogación, determina cual es el significado y la sintaxis adicional. Las extensiones generalmente no crean un nuevo grupo.

(?aiLmsux)

(Una o más letras del conjunto 'a', 'i', 'L', 'm', 's', 'u', 'x'.) El grupo coincide con la cadena vacía; las letras establecen los indicadores correspondientes:
re.A (coincidencia sólo ASCII),
re.I (ignorar mayúsculas y minúsculas),
re.L (depende del entorno local),
re.M (multilínea), re.S (el punto coincide con todos) ,
re.U (coincidencia de Unicode) y
re.X (verbose), para toda la expresión regular.

Esto es útil si desea incluir los indicadores como parte de la expresión regular, en lugar de pasar un argumento de indicador a la función re.compile (). Las banderas deben usarse al inicio de la expresión regular. Por ejemplo, el siguiente código nos da error, ya que 'manzana' es diferente de 'MANZANA':

regex3.py

```
import re

patron = re.compile('pera"|manzana')
string = "MANZANA"

print(patron.match(string))
```

Sin embargo si cambiamos la expresión regular por:

```
patron = re.compile('(?i)pera|manzana')
```

Nos retorna un objeto de coincidencia.

(?:…)

En el caso de utilizar un flag dentro de esta expresión, éste se aplica sólo a la expresión que se encuentra entre paréntesis pero no para lo que está fuera del paréntesis.

regex4.py

```
import re

patron = re.compile(r'(?i:pera) y manzana')
string1 = "pera y manzana"
string2 = "PERA y manzana"
string3 = "Pera y manzana"
string4 = "PERA Y manzana"
print(patron.match(string1))
print(patron.match(string2))
print(patron.match(string3))
print(patron.match(string4))
```

Retorna:

```
<re.Match object; span=(0, 14), match='pera y manzana'>
<re.Match object; span=(0, 14), match='PERA y manzana'>
<re.Match object; span=(0, 14), match='Pera y manzana'>
None
```

La última opción retorna None ya que no hay coincidencia pues la letra ‘Y’ está en mayúscula y no le aplica.

(?=…)

Este tipo de búsqueda es lo que se llama una afirmación de búsqueda anticipada, es decir que lo que estoy buscando coincide sólo si a continuación de lo que estoy buscando aparece esta estructura. Por ejemplo, si la expresión regular es r'color (?=rojo)' solo voy a encontrar la palabra color seguida de un espacio, si luego se encuentra la palabra rojo.

regex5.py

```
import re
patron = re.compile(r'color (?=rojo)')
string1 = "El auto de color rojo."
print(patron.search(string1))
```

Retorna:

```
<re.Match object; span=(11, 17), match='color '>
```

(?=!...)

Este es el caso opuesto del anterior, en este caso encontraría la palabra auto seguida de un espacio si no estuviera seguida de la palabra rojo

Regex6.py

```
import re
patron = re.compile(r'color (?!rojo)')
string1 = "El auto de color azul."
print(patron.search(string1))
```

Retorna:

```
<re.Match object; span=(11, 17), match='color '>
```

Búsqueda hacia atrás (?<= (?<!

Las declaraciones de *búsqueda hacia atrás* comienzan con (?<= para declaraciones positivas y con (?<! para declaraciones negativas.

Por ejemplo,

(?<!fcolor)azul encuentra una incidencia de "azul" que no esté precedida por "color ". El contenido de una declaración de búsqueda hacia atrás está restringido de tal manera que todas las cadenas que se comparen con ella deben tener una longitud fija. Si solo nos interesa recuperar la coincidencia podemos guardar el resultado de la búsqueda en una variable y utilizar .group(0) como se muestra a continuación.

regex7.py

```
import re

patron = re.compile(r'(?<=color )azul')
string1 = "El auto de color azul."

print(patron.search(string1))

m = re.search(r'(?<=color )azul', 'El auto de color azul.')
print(m.group(0))
```

Retorna:

```
<re.Match object; span=(17, 21), match='azul'>

azul
```

Condicional

Podemos utilizar una condición dentro de la estructura regular:

```
(?(id/name)yes-pattern|no-pattern)
```

En la cual si la condición se cumple, en tal caso se trata de realizar la búsqueda, el segundo término es opcional. Veamos un par de ejemplos para clarificar la estructura.

regex8.py

```
import re
patron1 = re.compile(r'((?P<name>ap8)(?(name)9|))')

mail = 'ap89 ap8 rp8 rp4 w'
print(patron1.search(mail))
```

Retorna:

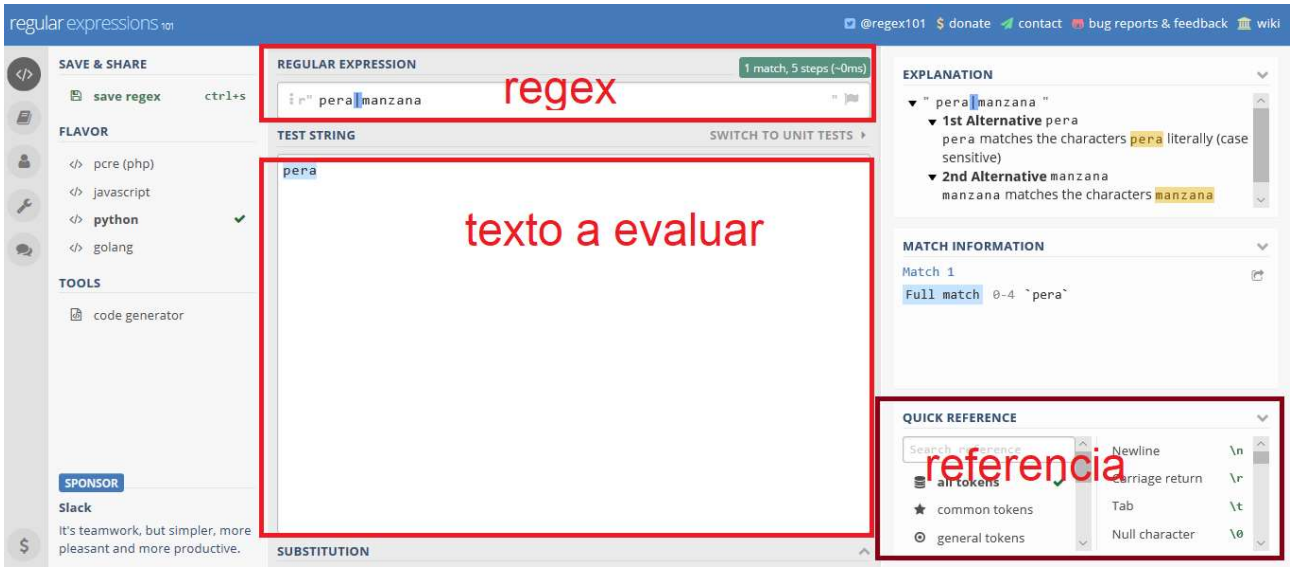
```
<re.Match object; span=(0, 4), match='ap89'>
```

6. Herramienta de trabajo

Existen muchas herramientas de trabajo online que nos ayudan a la hora de desarrollar una expresión regular, un ejemplo es la plataforma “regex101.com”:

https://regex101.com/

La cual nos permite introducir un código de testeo y una expresión regular y evaluar si cumple con nuestras expectativas. La herramienta cuenta con una librería de expresiones regulares y patrones y puede adaptarse a otros lenguajes como javascript.



7. Descripción del módulo.

El módulo “re” cuenta con muchas otras opciones de trabajar con expresiones regulares que pueden aportar mucho a nuestro trabajo, y con las que nos encontraremos muchas veces al buscar información en diferentes referencias. Por ejemplo en lugar de indicar dentro de la expresión regular que un texto puede encontrarse en mayúscula o minúscula mediante (?i:…) podríamos indicarlo agregando una bandera como segundo parámetro, de esta forma los siguientes casos son análogos.

regex9.py

```
import re
patron1 = re.compile(r'(<)?(\w+@\w+(?:\.[a-z]+)+)(?(1)>|$)', re.I)

mail = "<user@host.Com>"
print(patron1.search(mail))

patron2 = re.compile(r'(<)?(\w+@\w+(?i:\.[a-z]+)+)(?(1)>|$)')
mail1 = "<user@host.Com>"
print(patron2.search(mail1))
```

De igual forma podemos utilizar re.A, re.I, re.L, re.M y re.X. En el caso de re.X escribir el código de esta forma puede facilitar mucho la legibilidad de nuestro código así como la posibilidad de compartir nuestro código con descripciones adicionales. Veamos un ejemplo útil:

regex10.py

```
import re
patron1 = re.compile(r"""\d +  # Parte entera
                \.    # Punto decimal
                \d *  # Parte de fracción""", re.X)

mail = "3.3"
print(patron1.search(mail))
```

Este ejemplo nos muestra cómo agregarle descripción a nuestra expresión regular, el código podría ser equivalente simplemente a: re.compile(r"\d+\.\d*")

Métodos del módulo

re.match()

re.match (patrón, cadena, banderas = 0)

Si cero o más caracteres al principio de la cadena coinciden con el patrón de expresión regular, retorna un objeto de coincidencia.

Nota: Hay que tener en cuenta que incluso en el modo MULTILINE, re.match() sólo coincidirá al principio de la cadena y no al principio de cada línea.

re.search()

re.search (patrón, cadena, banderas = 0)

Permite escanear a través de la cadena buscando la primera ubicación donde el patrón de expresión regular produce una coincidencia y retorna un objeto de coincidencia. Retorna "None" si no existe coincidencia.

regex11.py

```
import re
patron1 = re.search(r'un.','Hola esto es una prueba, se escribe de nuevo un hola',
flags=0)

print(patron1)
```

Retorna:

```
<re.Match object; span=(13, 16), match='una'>
```

Nota: si quitamos "una" del texto, encontraría el "un" que está antes del último "hola":

re.fullmatch()

re.fullmatch(patrón, cadena, banderas = 0)

Encuentra todas las subcadenas donde la expresión regular coincida, y retorna las coincidencias en una lista.

re.split()

re.split(patrón, cadena, maxsplit=0, banderas = 0)

Permite dividir la cadena según un determinado patrón. Si se utilizan paréntesis de captura en el patrón, entonces el texto de todos los grupos en el patrón también se devuelve como parte de la lista resultante. Si maxsplit es distinto de cero, como máximo se producen las divisiones maxsplit, y el resto de la cadena se devuelve como el elemento final de la lista.

regex12.py

```
import re
retorno1 = re.split(r'\W+', 'Palabras, palabras, palabras.')
retorno2 = re.split(r'(\W+)', 'Palabras, palabras, palabras.')
retorno3 = re.split(r'\W+', 'Palabras, palabras, palabras.', 1)
retorno4 = re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)

print(retorno1)
print(retorno2)
print(retorno3)
print(retorno4)
```

Retorna:

```
['Palabras', 'palabras', 'palabras', '']
['Palabras', ', ', 'palabras', ', ', 'palabras', '.', '']
['Palabras', 'palabras, palabras.']
['0', '3', '9']
```

Nota: Recordar que en \W como la w está en mayúscula se refiere a un patrón no alfanumérico, por lo que hace referencia a la coma (,).

Nota: En el uso de banderas podemos utilizar en lugar de las letras, las palabras que representan:

Abreviación	Palabra
A	ASCII
I	IGNORECASE
L	LOCALE
M	MULTILINE
S	DOTALL
X	VERBOSE

Si hay grupos de captura en el separador y coinciden al comienzo de la cadena, el resultado comenzará con una cadena vacía. Lo mismo vale para el final de la cadena:

regex13.py

```
print("--------------------------")
retorno5 = re.split(r'\W+', '...Palabras..., palabras, palabras...')
retorno6 = re.split(r'(\W+)', '...Palabras..., palabras, palabras...')
print(retorno5)
print(retorno6)
```

Retorna:

```
--------------------------
['', 'Palabras', 'palabras', 'palabras', '']
['', '...', 'Palabras', '...', ', 'palabras', ', ', 'palabras', '...', '']
```

re.findall()

re.findall(patrón, cadena, banderas = 0)

Devuelve todas las coincidencias no superpuestas del patrón en cadena, como una lista de cadenas. La cadena se escanea de izquierda a derecha y las coincidencias se devuelven en el orden encontrado. Si uno o más grupos están presentes en el patrón, devuelva una lista de grupos; esta será una lista de tuplas si el patrón tiene más de un grupo. Las coincidencias vacías se incluyen en el resultado.

re.finditer()

re.finditer(patrón, cadena, banderas = 0)

Devuelve un iterador que arroja objetos de coincidencia sobre todas las coincidencias no superpuestas para el patrón RE en cadena. La cadena se escanea de izquierda a derecha y las coincidencias se devuelven en el orden encontrado. Las coincidencias vacías se incluyen en el resultado.

Cambiado en la versión 3.7: las coincidencias no vacías ahora pueden comenzar justo después de una coincidencia vacía anterior.

Bibliografía utilizada y sugerida

Libros

Programming Python 5th Edition – Mark Lutz – O'Reilly 2013

Programming Python 4th Edition – Mark Lutz – O'Reilly 2011

Manual online

https://docs.python.org/3.7/tutorial/

https://docs.python.org/3.7/library/index.html

Lo que vimos

En esta unidad comenzamos a trabajar con expresiones regulares.

Lo que viene:

En la siguiente unidad seguiremos aprendiendo sobre expresiones regulares.