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
```

Nota: La expresión regular debemos indicarla entre comillas simples o dobles.

Como podemos ver estamos buscando coincidencia del patrón formado a partir de la expresión regular con los componentes de una lista. La primera búsqueda nos retorna coincidencia mientras que la segunda no. En los siguientes puntos que trataremos, veremos cómo interpretar los patrones, y tomaremos la siguiente convención:

- Expresión regular en rojo regex.
- Texto al cual le aplico la expresión regular en verde regex.
- Coincidencia en celeste regex.

Nota: En las siguientes secciones veremos como se expresan las expresiones regulares.

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	Especifica 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 guion.

[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 preceder 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 solo 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. Esto entra en conflicto con el uso de Python del mismo carácter para el mismo propósito en cadenas literales.

Digamos que queremos escribir una expresión regular que coincida con la cadena **\ sección**, que podría encontrarse en un archivo LaTeX (en este tipo de archivos podemos encontrar muchas líneas que inician con una barra invertida). Para averiguar qué escribir en el código del programa, podemos comenzar con la cadena de deseamos hacer coincidir. A continuación, se debe escapar de las barras invertidas y otros meta caracteres precediéndolos con una barra diagonal inversa, lo que da como resultado la sección de cadena \\. La cadena resultante que se debe pasar a re.compile () debe ser **\\ sección**. Sin embargo, para expresar esto como un literal de cadena de Python, ambas barras invertidas deben volver a escaparse.

En resumen, para hacer coincidir una barra invertida literal, uno tiene que escribir '\\\\' como expresión regular, ya que la expresión regular debe ser \\, y cada barra invertida debe expresarse como \\ dentro de una cadena de texto literal en Python. En las expresiones regulares que deben tener varias barras diagonales, esto conduce a un montón de barras invertidas repetidas y hace que las cadenas resultantes sean difíciles de entender. La solución es usar la notación de Python literal utilizando el prefijo ‘r’ en expresiones regulares, de esta forma **r "\ n"** es una cadena de dos caracteres que contiene '\' y 'n', mientras que **"\ n"** es una cadena de un solo carácter que contiene una nueva línea.

Además, las secuencias de escape especiales que son válidas en expresiones regulares, pero no válidas como literales de cadenas de Python, ahora dan como resultado una advertencia de desaprobación y eventualmente se convertirán en un SyntaxError, lo que significa que las secuencias no serán válidas si la notación de cadena sin formato o el escape de las barras diagonales no se están utilizando.

Expresión regular	Expresión regular con (r)
"ab*"	r"ab*"
"\\\\section"	r"\\section"
"\\w+\\s+\\1"	"\w+\s+\1"

Meta-caracter ^

Este meta-carácter llamado circunflejo tiene dos usos:

Fuera de una clase carácter, en el modo de comparación por defecto, el carácter **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 carácter, circunflejo (^) tiene un significado totalmente diferente (véase más adelante).

Circunflejo (^) no necesita ser el primer carácter 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 carácter pesos (*$*) es una declaración la cual es TRUE sólo si el punto actual de coincidencia está al final de la cadena objetivo, o inmediatamente antes de un carácter de nueva línea que es el último carácter en la cadena (por defecto). Pesos (*$*) no necesita ser el último carácter 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 carácter.

Meta-caracter .

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

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 carácter 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 invalidas 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 carácter 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, este se aplica solo 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.