Expresiones regulares

Una de las tareas más utilizadas en la programación es la búsqueda de subcadenas o patrones dentro de otras cadenas de texto.

Las expresiones regulares, también conocidas como 'regex' o 'regexp', son patrones de búsqueda definidos con una sintaxis formal. Siempre que sigamos sus reglas, podremos realizar búsquedas simples y avanzadas, que utilizadas en conjunto con otras funcionalidades, las vuelven una de las opciones más útiles e importantes de cualquier lenguaje.

Sin embargo antes de utilizarlas hay que estar seguros de lo que hacemos, de ahí aquella famosa frase de Jamie Zawinski [https://es.wikipedia.org/wiki/Jamie_Zawinski], programador y hacker:

> *Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.*

Que viene a decir:

> *Hay gente que, cuando se enfrenta a un problema, piensa "Ya sé, usaré expresiones regulares". Ahora tienen dos problemas.*

Métodos básicos

- **re.search**: busca un patrón en otra cadena:

```
import re

texto = "En esta cadena se encuentra una palabra mágica"

re.search('mágica', texto)
```

```
<_sre.SRE_Match object; span=(40, 46), match='mágica'>
```

Como vemos, al realizar la búsqueda lo que nos encontramos es un objeto de tipo *Match* (encontrado), en lugar un simple *True* o *False*.

En cambio, si no se encontrase la palabra, no se devolvería nada (*None*):

```
import re
re.search('hola', texto)
```

Por tanto, podemos utilizar la propia funcionalidad junto a un condicional sin ningún problema:

```
palabra = "mágica"

encontrado = re.search(palabra,  texto)

if encontrado:
    print("Se ha encontrado la palabra:", palabra)
else:
    print("No se ha encontrado la palabra:", palabra)
```

```
Se ha encontrado la palabra: mágica
```

Sin embargo, volviendo al objeto devuelto de tipo *Match*, éste nos ofrece algunas opciones interesantes.

```
# Posición donde empieza la coincidencia
print( encontrado.start() )
# Posición donde termina la coincidencia
print( encontrado.end() )
# Tupla con posiciones donde empieza y termina la coincidencia
print( encontrado.span() )
# Cadena sobre la que se ha realizado la búsqueda
print( encontrado.string )
```

```
40
46
(40, 46)
En esta cadena se encuentra una palabra mágica
```

Como vemos, en este objeto se esconde mucha más información de la que parece a simple vista, luego seguiremos hablando de ellos.

- **re.match**: busca un patrón al principio de otra cadena:

```
texto = "Hola mundo"

re.match('Hola', texto)
```

```
<_sre.SRE_Match object; span=(0, 4), match='Hola'>
```

- **re.split**: divide una cadena a partir de un patrón:

```
texto = "Vamos a dividir esta cadena"

re.split(' ', texto)
```

```
['Vamos', 'a', 'dividir', 'esta', 'cadena']
```

- **re.sub**: sustituye todas las coincidencias en una cadena:

```
texto = "Hola amigo"

re.sub('amigo', 'amiga', texto)
```

```
'Hola amiga'
```

- **re.findall**: busca todas las coincidencias en una cadena:

```
texto = "hola adios hola hola"

re.findall('hola', texto)
```

```
['hola', 'hola', 'hola']
```

Aquí se nos devuelve una lista, pero podríamos aplicar la función *len()* para saber el número:

```
len(re.findall('hola', texto))
```

```
3
```

Patrones con varios valores

Si queremos comprobar varias posibilidades, podemos utilizar una tubería | a modo de OR. Generalmente pondremos el listado de alternativas entre paréntesis ():

```
texto = "hola adios hello bye"

re.findall('hola|hello', texto)
```

```
['hola', 'hello']
```

Patrones con sintaxis repetida

Otra posibilidad que se nos ofrece es la de buscar patrones con letras repetidas, y aquí es donde se empieza a poner interesante. Como podemos o no saber de antemano el número de repeticiones hay varias formas de definirlos.

```
texto = "hla hola hoola hooola hooooola"
```

Antes de continuar, y para aligerar todo el proceso, vamos a crear una función capaz de ejecutar varios patrones en una lista sobre un texto:

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

patrones = ['hla', 'hola', 'hoola']
buscar(patrones, texto)
```

```
['hla']
['hola']
['hoola']
```

- **Con meta-carácter ***

Lo utilizaremos para definir ninguna o más repeticiones de la letra a la izquierda del meta-carácter:

```
patrones = ['ho','ho*','ho*la','hu*la']

buscar(patrones, texto)
```

```
['ho', 'ho', 'ho', 'ho']
['h', 'ho', 'hoo', 'hooo', 'hooooo']
['hla', 'hola', 'hoola', 'hooola', 'hooooola']
['hla']
```

- **Con meta-carácter +**

Lo utilizaremos para definir una o más repeticiones de la letra a la izquierda del meta-carácter:

```
patrones = ['ho*', 'ho+']

buscar(patrones, texto)
```

```
['h', 'ho', 'hoo', 'hooo', 'hooooo']
['ho', 'hoo', 'hooo', 'hooooo']
```

- **Con meta-carácter ?**

Lo utilizaremos para definir una o ninguna repetición de la letra a la izquierda del meta-carácter:

```
patrones = ['ho*', 'ho+', 'ho?', 'ho?la']

buscar(patrones, texto)
```

```
['h', 'ho', 'hoo', 'hooo', 'hooooo']
['ho', 'hoo', 'hooo', 'hooooo']
['h', 'ho', 'ho', 'ho', 'ho']
['hla', 'hola']
```

- **Con número de repeticiones explícito {n}**

Lo utilizaremos para definir 'n' repeticiones exactas de la letra a la izquierda del meta-carácter:

```
patrones = ['ho{0}la', 'ho{1}la', 'ho{2}la']

buscar(patrones, texto)
```

```
['hla']
['hola']
['hoola']
```

- **Con número de repeticiones en un rango {n, m}**

Lo utilizaremos para definir un número de repeticiones variable entre 'n' y 'm' de la letra a la izquierda del meta-carácter:

```
patrones = ['ho{0,1}la', 'ho{1,2}la', 'ho{2,9}la']

buscar(patrones, texto)
```

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

Conjuntos de caracteres

Cuando nos interese crear un patrón con distintos carácteres, podemos definir conjuntos entre paréntesis:

```
texto = "hala hela hila 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']
```

Evidentemente los podemos utilizar con repeticiones:

```
texto = "haala heeela hiiiila hooooola"

patrones = ['h[ae]la', 'h[ae]*la', 'h[io]{3,9}la']
buscar(patrones, texto)
```

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

- **Exclusión en grupos [^]**:

Cuando utilizamos grupos podemos utilizar el operador de exclusión ^ para indicar una búsqueda contraria:

```
```

```
texto = "hala hela hila hola hula"

patrones = ['h[o]la', 'h[^o]la']
buscar(patrones, texto)
```

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

- **Rangos [-]**:

Otra característica que hace ultra potentes los grupos, es la capacidad de definir rangos. Ejemplos de rangos:

- **[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 número).
- **[A-z]**: Cualquier carácter alfabético en minúscula o mayúscula (no especial ni número).
- **[0-9]**: Cualquier carácter numérico (no especial ni alfabético).
- **[a-zA-Z0-9]**: Cualquier carácter alfanumérico (no especial).

Tened en cuenta que cualquier rango puede ser excluido para conseguir el patrón contrario.

```
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']
```

Códigos escapados \

Si cada vez que quisiéramos definir un patrón variable tuviéramos que crear rangos, al final tendríamos expresiones regulares gigantes. Por suerte su sintaxis también acepta una serie de caracteres escapados que tienen un significo único. Algunos de los más importantes son:

Código	Significado
\d	numérico
\D	no numérico
\s	espacio en blanco
\S	no espacio en blanco
\w	alfanumérico
\W	no alfanumérico

El problema que encontraremos en Python a la hora de definir código escapado, es que las cadenas no tienen en cuenta el \ a no ser que especifiquemos que son cadenas en crudo (raw), **por lo que tendremos que precedir las expresiones regulares con una 'r'**.

```
texto = "Este curso de Python se publicó en el año 2016"

patrones = [r'\d+', r'\D+', r'\s', r'\S+', r'\w+', r'\W+']
buscar(patrones, texto)
```

```
['2016']
['Este curso de Python se publicó en el año ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['Este', 'curso', 'de', 'Python', 'se', 'publicó', 'en', 'el', 'año', '2016']
['Este', 'curso', 'de', 'Python', 'se', 'publicó', 'en', 'el', 'año', '2016']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
```

Por mi parte lo vamos a dejar aquí, pero el mundo de las expresiones regulares es gigantesco y daría para un curso entero. Os animo a seguir aprendiendo leyendo documentación y buscando ejemplos, os dejo algunos enlaces que os podrían servir.

Documentación

Hay docenas y docenas de códigos especiales, si queréis echar un vistazo a todos ellos podéis consultar la documentación oficial:

- https://docs.python.org/3.5/library/re.html#regular-expression-syntax [https://docs.python.org/3.5/library/re.html#regular-expression-syntax]

Un resumen por parte de Google Eduactión:

- https://developers.google.com/edu/python/regular-expressions [https://developers.google.com/edu/python/regular-expressions]

Otro resumen muy interesante sobre el tema:

- https://www.tutorialspoint.com/python/python_reg_expressions.htm [https://www.tutorialspoint.com/python/python_reg_expressions.htm]

Un par de documentos muy trabajados con ejemplos básicos y avanzados:

- http://www.python-course.eu/python3_re.php [http://www.python-course.eu/python3_re.php]
- http://www.python-course.eu/python3_re_advanced.php [http://www.python-course.eu/python3_re_advanced.php]

Última edición: 6 de Octubre de 2018