Resumen acerca de Expresiones Regulares

Escrito por: Daniela Martínez Madrid.

Este resumen es creado con la intención de recopilar información básica y relevante, fruto de un primer acercamiento a los conceptos de expresiones regulares, que además es un requerimiento del curso de programación avanzada del centro de investigación CINVESTAV, dirigido por el profesor Dr. Feliu Sagols.

Ciudad de México, mayo 2022.

Una expresión regular (conocida más comúnmente por regex o RE) es una secuencia de carácteres que definen un patrón de búsqueda, es decir, especificar funcionalidades para el conjunto de cadenas de carácteres que se desea hacer coincidir. Podemos hacer uso de estas expresiones haciendo uso del modulo re en Python.

Las expresiones regulares son comúnmente usadas para el trabajo y procesamiento de texto, por ejemplo:

- Encontrar y reemplazar.
- Coincidencia de texto.
- Buscar texto que se ajusta a un formato determinado (como lo es el correo electrónico).
- Buscar si existe o no una cadena de carácteres dentro de un texto.
- Contar el número de coincidencias en un texto.
- Verificar input.

Echamos un vistazo a algunas de los métodos y funciones más usados del módulo re.

Función re.search (<regex>, <string>). Recorre <string> buscando la primera ubicación donde el patrón <regex> coincide. En el caso de que la coincidencia es encontrada, la función retorna el objeto que coincidieron, en caso contrario, retorna None.

```
import re

cadena = 'vamos a aprender expresiones regulares'

print(re.search('aprender', cadena))
```

Return:

```
<re.Match object; span=(8, 16), match='aprender'>
```

Notemos que adicionalmente nos brinda información de los lugares donde se encuentra ubicado, en este caso desde el espacio 8 hasta el 16.

En el caso de no coincidir, si:

```
import re

cadena = 'vamos a aprender expresiones regulares'

print(re.search('aprder', cadena))
```

Return: `None`

Observación: El return de el objeto se puede interpretar como un valor de verdad, por tanto, podemos usarlo en el contexto de una variable Booleana, por ejemplo, así:

```
import re

cadena = 'vamos a aprender expresiones regulares'

Texto_buscado = 'aprender'

if re.search(Texto_buscado, cadena):
    print('He encontrado el texto')
else:
    print('No he encontrado el texto')
```

Metacarácteres en expresiones regulares

Los métodos del módulo re por si solos tienen limitantes cuando queremos encontrar patrones con carácterísticas particulares sin recordar en específicamente cual es la subcadena, por ejemplo, si queremos encontrar los formatos tipo correos, sin conocer los nombres de usuarios. Por eso hacemos uso de los metacarácteres.

Carácter ([]): Se pide encontrar el objeto de tres dígitos consecutivos, sin conocer cuales son estos.

En este caso, consideraremos las clases de carácteres, que son un conjunto de carácteres específicos descritos entre corchetes [].

```python
import re

cadena = 'hola235mundo'

encontrado = re.search('[0-9][0-9][0-9]', cadena)

print(encontrado)
```

Return:

```
<re.Match object; span=(4, 7), match='235'>
```

En este caso, las clases [0-9] representa coincidir con algún número del 0 al 9.

En el caso de no recordar algún carácter se puede usar el comodín punto (.), así:

```python
import re

cadena = 'hola235mundo'

encontrado_1 = re.search('.[0-9][0-9]', cadena)

print(encontrado_1)
```

Return:

```
<re.Match object; span=(3, 6), match='a23'>
```

Carácter (|): Funciona como el operador “or”, retornara cualquier de las dos subcadenas con que coincidan.

```python
import re

cadena = 'hola235mundo'

encontrado_2 = re.search('7|[0-9]', cadena)

print(encontrado_2)
```

Return:

```
<re.Match object; span=(4, 5), match='2'>
```

Carácter (^): Únicamente coincide cuando el patrón aparece en la primera línea de la cadena.

```
import re

cadena = 'hola235mundo, como estas'

encontrado_3 = re.search('^hola235m', cadena)

print(encontrado_3)
```

Return:

```
<re.Match object; span=(0, 8), match='hola235m'>
```

Metacarácteres soportados por el módulo re

Carácter (.): Coincide con algún carácter, excepto nueva línea, se puede entender como comodín.

Carácter (^): Ancla la coincidencia en el inicio de una cadena, y además complementa clase de carácteres.

Carácter ($): Ancla la coincidencia al final de la cadena.

Carácter (*): Coincide con cero o más repeticiones. Por ejemplo, 12*3 coincide con 1223 y 123, pero no con 223 ni con 23.

Carácter (+): Coincidencias con una o más repeticiones. Por ejemplo, [0–9] + coincide con 1, 11, 456.

Carácter(?): Coincidencias con cero o más repeticiones. Introduce una coincidencia con valores hacia adelante o hacia atrás. Por ejemplo, sept? coincide con sept y septiembre, pero no con diciembre.

Carácter({}): Coincide con un número especifico de repeticiones. Por ejemplo ‘Hooola’ coincide con ‘Ho{3}la’.

Carácter ([]): Especifica una clase de carácter. Puede ser específicamente con algo que se quiera coincidir o un grupo de carácteres. Por ejemplo, ‘h[aeiou]la’, coincide con ‘hala’, ‘hela’, ‘hila’, ‘hola’ y ‘hula’.

Carácter(()): Crean un grupo. Por ejemplo, ‘(and|vol)ando’ coincide con ‘andando’ y ‘volando’.

Carácteres (:, #,=,!): Designa un grupo especializado.

Carácter (<>): Crea un grupo nombrado.

Algunas observaciones sobre los carácteres anteriores.

Ya habíamos visto la utilidad del carácter [ ], y los beneficios al usarlo con grupos de carácteres. En el caso especial que queramos que aparezca el primer carácter diferente a los que están agrupados por [ ], vamos a presidir el grupo de carácteres por ^, por ejemplo, ‘[^0-9] coincide con ‘f’ en la cadena ‘12345foo’

También podemos hacer uso de ‘\’ para diferenciar la función de un carácter en su función módulo y cuando lo queremos usar como patrón, por ejemplo si queremos buscar un ‘]’ en la cadena ‘foo[1]’, hacemos uso del patrón ‘[ab\ ]cd]’.

Secuencias Especiales

Las siguientes abreviaturas son atajos para los rangos de valores, son expresiones que comienzan ‘\’ y logran reducir el tamaño de las expresiones regulares.

\d. Hace coincidir cualquier carácter número, es equivalene a usar [0-9]

\D. Hace coincidir cualquier carácter no numérico, es equivalente a usar [^0-9]

\s. Hace coincidir cualquier espacio. Es equivalente a usar [\t\n\r\f\v]

\S. Hace coincidir cualquiera que no sea espacio. Es equivalente a usar [^ \t\n\r\f\v]

\w. Hace coincidir cualquier carácter alfanúmerico. Es equivalente a usar [a-zA-Z0-9_]

\W. Hace coincidir cualquier carácter que no sea alfanúmerico. Es equivalente a usar [^a-zA-Z0-9_]

Declaraciones

Una declaración determina el limite de la ubicación especifica donde debe de ocurrir la coincidencia, no consumen ningún carácter y se incluyen dentro de la expresión regular.

En este caso es importante anteceder los carácteres con r'.

\b. Limita la palabra.

\B. No limita la palabra.

\A o ^. Concide con el comienzo de cadena.

\Z o $. Coincide con el final de un cadena.

Cuantificadores

Un cuantificador de un metacarácter está incluido en una expresión regular, para indicar el número de veces que debe ocurrir para se pueda dar la coincidencia.

* . Coincide con cero o mas repeticiones del precedente carácter.

+. Coincide con una o más repeticiones del precedente carácter.

?. Coincide con cero o una repetición del precedente carácter.

{n}. Coincide con el el carácter precedente exactamente *n* veces.

{m,n}. Coincide un número de repeticiones del carácter anterior que este entre n y m (incluyéndolos).

{m, }. Coincide con cualquier número de repeticiones del carácter anterior que sean mas grandes o iguales a m. Similarmente se define {,n}, pensando en el operador menor o igual.

Cuando usamos unicamente alguno de los metacarácteres *, + y ?, se denominan los cuantificadores codiciosos, pues ellos brindan la mas larga posible coincidencia. Para encontrar coincidencias mas cortas posibles usamos los cuantificadores perezosos o no codiciosos, como lo son las combinaciones: *?, +?, ??.

*?: Coincide con el carácter anterior 0 o más veces (el menor número posible).
+?: Coincide con el carácter anterior 1 o más veces (lo menos posible).
??: Haga coincidir el carácter anterior 0 o 1 veces (preferiblemente 0).

Agrupando y capturando

Vamos a diferenciar dos tipos de construcciones de agrupación, las cuales tomaran una regex y la tratatan como subexpresiones o grupos.

Agrupamiento. Un grupo que representa única identidad sintactica.

Capturando. Se captura parte de la cadena de búsqueda que coincide con la subepresión en el grupo, cuando pensamos en el proceso de agrupamiento.

Para agrupar una regex, simplemente la escribimos entre paréntesis: (<regex>). Lo que se llamará una subexpresión o un grupo.

Entonces los metacaracteres actuaran en el grupo pensándolo como una unidad.

Por parte de las capturas, existen dos métodos por los que se pueden acceder, que son: groups() y group().

m.groups(). Retorna una tupla que contine los grupos capturados, dada una coincidencia de una regex.

Este sería un ejemplo de agrupamiento:

```
>>> m = re.search('(\w+),(\w+),(\w+)', 'foo,quux,baz')
>>> print(m)
<re.Match object; span=(0, 12), match='foo,quux,baz'>
```

Con sus capturas:

```
>>> m.groups()
('foo', 'quux', 'baz')
```

Referencias inversas(\<n>). Coincide el contenido de un grupo previamente capturado.