Expresiones Regulares

Antonio Espín Herranz

Expresiones regulares

- Las expresiones regulares, también llamadas ***regex*** *o* ***regexp,*** consisten en patrones que **describen conjuntos de cadenas de caracteres***.*

- El módulo **re** disponible desde python 1.5.

- Métodos de **re**:
  - **search**
    - Busca patrones dentro de una cadena.
  - **match**
    - Comprueba si una cadena se ajusta a un determinado patrón.
  - **split**
    - Divide una cadena utilizando coincidencias de un patrón.
  - **sub**
    - Sustituir todas las ocurrencias de una cadena por otra.

Expresiones regulares

- La expresión regular más sencilla sería una cadena tal cual.

```
import re
if re.match(“python”, “python”):
    print (“cierto”)
```

- **Parámetros de match**:
  - El **primer parámetro** representa el **patrón**.
  - Y el **segundo la cadena** a evaluar.

- Se pueden utilizar caracteres especiales para indicar el patrón a localizar.

Patrones

- El “**.**” (**punto**) se utiliza como un carácter comodín.

- **Un carácter cualquiera y solo uno**.

- Localizar cualquier palabra que termine en “**ython**”.

- **Ejemplo**:

  re.match(“**.**ython”, “**p**ython”)

  re.match(“**.**ython”, “**j**ython”)

Patrones

- Si necesitamos buscar el **‘.’** hay que escaparlo,
- Ejemplo:
  - Para comprobar si la cadena consiste en 3 caracteres seguidos de un punto.

    re.match(“...**\.**”, “abc.”)

- La **barra vertical |** indica alternativa:
  - re.match(“python**|**jython**|**cython”, “python”)

- También sería equivalente:
  - re.match(“(p|j|c)ython”, “python”)

Patrones

- También se pueden formar clases de caracteres encerrando estos entre **[]**

  re.match(“**[pjc]**ython”, “python”)

- Se pueden especificar **rangos** dentro de los corchetes:

  [0-9]
  - Del 0 al 9.

  [a-z]
  - Letras minúsculas.

  [a-zA-Z0-9]
  - Letras mayúsculas, minúsculas y dígitos del 0 al 9.

Patrones

- Podemos buscar terminaciones colocando el patrón al final de la expresión regular:

  re.match(“python**[0-9]**”, “python**0**”)

- **Dentro de las clases de caracteres los caracteres especiales no necesitan ser escapados**.

- Para comprobar si la cadena es “**python.**” o “**python,**”, entonces, escribiríamos:

  re.match(“python**[.,]**”, “python.”)

- y no

  re.match(“python**[\.,]**”, “python.”)

- Con esta opción estamos comprobando si la cadena es “python.”, “python,” o “python\”.

Patrones

- Los conjuntos de caracteres se pueden negar utilizando **^**

- La expresión "**python[^0-9a-z]**", por ejemplo, indicaría que nos interesan las cadenas que comiencen por "python" y tengan como ***último carácter algo que NO sea ni una letra minúscula ni un número***.

Patrones

- Secuencias disponibles:
- **\d**
  - Un dígito. Equivale a **[0-9]**
- **\D**
  - Cualquier carácter que no sea un dígito. Equivale a **[^0-9]**
- **\w**
  - Cualquier carácter alfanumérico. Equivale a **[a-zA-Z0-9_]**
- **\W**
  - Cualquier carácter no alfanumérico. Equivale a **[^a-zA-Z0-9_]**
- **\s**
  - Cualquier carácter en blanco. Equivale a **[\t\n\r\f\v]**
- **\S**
  - Cualquier carácter que no sea un espacio en blanco.
  - Equivale a **[^ \t\n\r\f\v]**

Patrones

- Repeticiones de secuencias:
- ? Opcional. 0 ó 1 vez.
- * 0 ó más repeticiones.
- + 1 o más repeticiones.
- {n} n repeticiones.

- El carácter **+** indica que lo que tenemos a la izquierda, sea un carácter como ‘a’, una clase como ‘[abc]’ o un subpatrón como (abc), puede encontrarse una o mas veces. Por ejemplo la expresión regular “**python+**” describiría las cadenas “**python**”, “**pythonn**” y “**pythonnn**”,

Patrones

- En el ejemplo anterior con “pytho*” la cadena pytho sería válida.

- {n}
  - Indicar el número de veces exacto que puede aparecer el carácter de la izquierda, o bien un rango de veces que puede aparecer.

- Por ejemplo:
  - {3}
    - Indicaría que tiene que aparecer exactamente 3 veces.
  - {3,8}
    - Indicaría que tiene que aparecer de 3 a 8 veces.
  - {,8}
    - De 0 a 8 veces.
  - {3,}
    - Tres veces o mas (las que sean).

Patrones

- **^** y **$**
  - Indican, respectivamente, que el elemento sobre el que actúan debe ir al **principio** de la cadena o al **final** de esta.

- La cadena “http://mundogeek.net”, por ejemplo, se ajustaría a la expresión regular “**^http**”, mientras que la cadena “El protocolo es http” no lo haría, ya que el http no se encuentra al principio de la cadena.

- Pero “**net$**” si se ajustaría a la cadena.

Módulo re

- La función match:

  re.match(patrón, cadena, [flags])

  – La función match se puede completar con una serie de **flags**:
    - **re.IGNORECASE**
      – Que hace que no se tenga en cuenta si las letras son mayúsculas o minúsculas.
    - **re.VERBOSE**
      – Que hace que se ignoren los espacios y los comentarios en la cadena que representa la expresión regular.

Módulo re

- La función re.match

- El valor de retorno de la función puede ser:
  - **None** en caso de que la cadena no se ajuste al patrón.
  - Un objeto de tipo **MatchObject** en caso contrario.
    - Este objeto MatchObject cuenta con **métodos start** y **end** que **devuelven la posición en la que comienza y finaliza la subcadena reconocida** y
    - Métodos **group** y **groups** que permiten acceder a los grupos que propiciaron el reconocimiento de la cadena.

Módulo re

- **Ejemplos**:
  mo = re.match(“http://.+**\.**net”, “http://mundogeek.net”)
  print (mo.group())
  http://mundogeek.net
  El **\.** es para escapar el punto. Es el **.** fijo de la extensión.

- Crear grupos utilizando los paréntesis:
  mo = re.match(“http://(.+)**\.**net”, “http://mundogeek.net”)
  print (mo.group(0))
  http://mundogeek.net
  print (mo.group(1))
  mundogeek

- Groups devuelve la lista de grupos:
  mo = re.match(“http://(.+)**\.**(.{3})”, “http://mundogeek.net”)
  print (mo.groups())
  (‘mundogeek’, ‘net’)

Módulo re

- **re.match vs re.search:**
  - La función search del módulo re funciona de forma similar a match;
  - Mismos parámetros y el mismo valor de retorno.
  - Con match la cadena se tiene que ajustar desde el primer carácter.
  - Con search NO tiene porque, puede empezar en otra posición.
  - El método **start** de un objeto **MatchObject** obtenido mediante la función match siempre devolverá 0, mientras que en el caso de search esto no tiene por qué ser así.

Módulo re

- **re.findall(patrón, cadena, [flags])**
  - Devuelve una lista con las subcadenas que cumplieron el patrón.

- **re.finditer(patrón, cadena, [flags])**
  - Que devuelve un iterador con el que consultar uno a uno los distintos MatchObject.

- **re.sub(patrón, reemplazar, cadena, [count],[flags])**
  - La función sub toma como parámetros un patrón a sustituir, una cadena que usar como reemplazo cada vez que encontremos el patrón, la cadena sobre la que realizar las sustituciones.

Módulo re

- **re.split(patrón, cadena)**
  - Utiliza el patrón a modo de puntos de separación para la cadena, devolviendo una lista con las subcadenas.

```
>>> re.split('\W+', 'Words, words, words.')
['Words', 'words', 'words', '']

>>> re.split('(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']

>>> re.split('\W+', 'Words, words, words.', 1)
['Words', 'words, words.']

>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

Notas

- Si da problemas el patrón, precederlo de una **b**.
- Por ejemplo,
  - for match2 in re.finditer(**b**'<td.*?>(.***?**)</td>', linea,re.DOTALL):
  - s = match2.group(1).decode('cp850')
  - OJO, el no poner la **?** hace que no detecte los finales de los <td>
  - Es posible que haya que decodificar, cp850 consola de Windows.