Guía de expresiones regulares en Python

soporte-platzi 537 Puntos hace 5 años

Cuando manejamos texto en Python, una de las operaciones más comunes es la búsqueda de una subcadena; ya sea para obtener su posición en el texto o simplemente para comprobar si está presente. Si la cadena que buscamos es fija, los métodos como *find()*, *index()* o similares nos ayudarán. Pero **si buscamos una subcadena con cierta forma**, este proceso se vuelve más complejo.

Al buscar direcciones de correo electrónico, números de teléfono, validar campos de entrada, o *una letra mayúscula seguida de dos minúsculas y de 5 dígitos entre 1 y 3*; es necesario recurrir a las **Expresiones Regulares**, también conocidas como **Patrones**.

Puedes aprender más en el Curso de Python y luego profesionalizarte en el Curso de Django.

Patrones

Las expresiones regulares son un **potente lenguaje de descripción de texto**. Y no existe un lenguaje moderno que no permita usarlas. Las reglas con las que se forman son bastante simples. Pero aprender a combinarlas correctamente requiere de práctica. Utilizándolas podemos **buscar una subcadena al principio o al final del texto**. Incluso si queremos que se repita cierta cantidad de veces, si queremos que algo NO aparezca, o si debe aparecer una subcadena entre varias posibilidades. Permite, además, capturar aquellos trozos del texto que coincidan con la expresión para guardarlos en una variable o reemplazarlos por una cadena predeterminada; o incluso una cadena formada por los mismos trozos capturados. Estos son algunos aspectos básicos de las expresiones regulares:

Metacaracteres

Se conoce como metacaracteres a aquellos que, dependiendo del contexto, tienen un **significado especial** para las expresiones regulares. Por lo tanto, los debemos escapar colocándoles una contrabarra () delante para buscarlos explícitamente. A continuación listaré los más importantes:

- **Anclas**: Indican que lo que queremos encontrar se encuentra al principio o al final de la cadena. Combinándolas, podemos buscar algo que represente a la cadena entera:
  - `patron`: coincide con cualquier cadena que comience con ***patron.***
  - `patron$`: coincide con cualquier cadena que termine con *patron*.
  - `patron$`: coincide con la cadena exacta *patron*.
- **Clases de caracteres**: Se utilizan cuando se quiere buscar un caracter dentro de varias posibles opciones. Una clase se delimita entre corchetes y lista posibles opciones para el caracter que representa:
  - `[abc]`: coincide con *a*, *b*, o *c*
  - [387ab]: coincide con *3*, *8*, *a* o *b*
  - `niñ[oa]s`: coincide con *niños* o *niñas*.
  - Para evitar errores, en caso de que queramos crear una clase de caracteres que contenga un corchete, debemos escribir una barra \ delante, para que el motor de expresiones regulares lo considere un caracter normal: la clase `[ab\[]` coincide con *a*, *b* y *[*.

Rangos

Si queremos encontrar un número, podemos usar una clase como `[0123456789]`, o podemos utilizar un **rango**. Un rango es una **clase de caracteres abreviada** que se crea escribiendo el primer caracter del rango, un guión y el último caracter del rango. Múltiples rangos pueden definirse en la misma clase de caracteres.

- `[a-c]`: equivale a `[abc]`
- `[0-9]`: equivale a `[0123456789]`
- `[a-d5-8]`: equivale a `[abcd5678]`

Es importante notar que **si se quiere buscar un guión debe colocarse al principio o al final de la clase**. Es decir, inmediatamente después del corchete izquierdo o inmediatamente antes del corchete derecho; o, en su defecto, escaparse. Si no se hace de esta forma, el motor de expresiones regulares intentará crear un rango y la expresión no funcionará como debe (o dará un error). Si queremos, por ejemplo, crear una clase que coincida con los caracteres a, 4 y -, debemos escribirla así:

- `[a4-]`
- `[-a4]`
- `[a\-4]`

Rango negado

Así como podemos listar los caracteres posibles en cierta posición de la cadena, **también podemos listar caracteres que no deben aparecer**. Para lograrlo, debemos **negar** la clase, colocando un circunflejo inmediatamente después del corchete izquierdo:

- `[^abc]`: coincide con cualquier caracter distinto a *a*, *b* y *c*

Clases predefinidas

Existen algunas clases que se usan frecuentemente y por eso existen **formas abreviadas** para ellas. En Python, así como en otros lenguajes, se soportan las clases predefinidas de Perl y de POSIX. Algunos ejemplos de expresiones regulares son:

- `\d` (POSIX `[[:digit:]]`): equivale a `[0-9]`
- `\s` (POSIX `[[:space:]]`): caracteres de espacio en blanco (espacio, tabulador, nueva línea, etc)
- `\w` (POSIX `[[:word:]]`): letras minúsculas, mayúsculas, números y guión bajo (_)

Además existe una clase de caracteres que coincide con cualquier otro caracter. Ya sea letra, número, o un caracter especial. Esta clase es el **punto**:

- `"."` : coincide con cualquier caracter.

Cuantificadores

Son conjuntos de caracteres que multiplican el patrón que les precede. Mientras que con las clases de caracteres podemos buscar un dígito, o una letra; con los cuantificadores **podemos buscar cero o más letras, al menos 7 dígitos, o entre tres y cinco letras mayúsculas**. Los cuantificadores son:

- `**?**`: coincide con cero o una ocurrencia del patrón. Dicho de otra forma, hace que el patrón sea opcional
- `**+**`: coincide con una o más ocurrencias del patrón
- `*****`: coincide con cero o más ocurrencias del patrón.
- `**{x}**`: coincide con exactamente *x* ocurrencias del patrón
- `**{x, y}**`: coincide con al menos *x* y no más de *y* ocurrencias. Si se omite *x*, el mínimo es cero, y si se omite *y*, no hay máximo. Esto permite especificar a los otros como casos particulares: *?* es *{0,1}*, *+* es *{1,}* y *** es *{,}* o *{0,}*.

Ejemplos:

- `.*` : cualquier cadena, de cualquier largo (incluyendo una cadena vacía)
- `[a-z]{3,6}`: entre *3* y *6* letras minúsculas
- `\d{4,}`: al menos 4 dígitos
- `.*hola!?`: una cadena cualquiera, seguida de *hola*, y terminando (o no) con un *!*

Otros metacaracteres

Existen otros metacaracteres en el lenguaje de las expresiones regulares:

- `**?**`: Además de servir como cuantificador, **puede modificar el comportamiento de otro**. De forma predeterminada, un cuantificador coincide con la *mayor* cadena posible. Cuando se le coloca un ?, se indica que se debe coincidir con la *menor* cadena posible. Esto es: dada la cadena *bbbbb*, `b+` coincide con la cadena entera, mientras que `b+?` coincide solamente con *b*. Es decir, la menor cadena que cumple el patrón.
- `**()**`: agrupan patrones. Sirven para que aquel pedazo de la cadena que coincida con el patrón **sea capturado**; o para delimitar el alcance de un cuantificador. Ejemplo: `ab+` coincide con *ab*, *abb*, *abbbbb*, ..., mientras que `(ab)+` coincide con *ab*, *abab*, *abab*...
- `**|**` : permite definir opciones para el patrón: `perro|gato` coincide con *perro* y con *gato*.

Módulo re

Para utilizar Expresiones Regulares, Python provee el módulo **re**. Importando este módulo podemos **crear objetos de tipo patrón y generar objetos tipo *matcher***, que son los que contienen la información de la coincidencia del patrón en la cadena.

Creando un patrón

Para crear un objeto patrón, debemos importar el módulo *re* y utilizamos la función **compile**:

```
import re

patron = re.compile('a[3-5]+') # coincide con una letra, seguida de al menos 1 dígit
```

Desde este momento, podemos usar el objeto <var>patron</var> para comparar cadenas con la expresión regular.

Buscar el patrón en la cadena

Para buscar un patrón en una cadena, Python provee los métodos search y match. La diferencia entre ambos es que, mientras **search** busca en la cadena alguna ocurrencia del patrón, **match** devuelve **None**si la ocurrencia no se da al principio de la cadena:

```
cadena = 'a44453'
patron.match(cadena)  # <_sre.SRE_Match object at 0x02303BF0>
patron.search(cadena) # <_sre.SRE_Match object at 0x02303C28>
cadena = 'ba3455' # la coincidencia no está al principio!
patron.search(cadena)  #  <_sre.SRE_Match object at 0x02303BF0>
print patron.match(cadena) # None
```

Si sabemos que **obtendremos más de una coincidencia**, podemos usar el método findall, que recorre la cadena y devuelve una lista de coincidencias:

```
patron.findall('a455 a333b435') # ['a455', 'a333']
```

O el método finditer, que **devuelve un iterador** que podemos usar en el bucle **for**:

```
for m in patron.finditer('a455 a333b435'): # cada m es un objeto tipo matcher ... pr
```

Objetos matcher

Ya hablamos del uso del los paréntesis en un patrón. Cuando se obtiene una coincidencia del patrón en una cadena, cada grupo delimitado por paréntesis captura el texto que haya coincidido con él. **Estos grupos son accesibles** a través de un objeto tipo **matcher** devuelto por **search** o **match**. Los grupos se numeran de izquierda a derecha según su orden de aparición en el patrón, y podemos usar este número para acceder al contenido del grupo con el método group del objeto **matcher**. De forma alternativa, podemos usar el método groups que devuelve una lista de grupos:

```
patron = re.compile('([ab])([3-5]+)') # ahora la letra se capturará en el grupo 1, y
matcher = patron.search('a455 a333b435')
matcher.group(0) # el grupo 0 es el trozo de cadena que coincidió con el patrón comp
matcher.group(1) #  'a'
matcher.group(2) # '455'
matcher.groups() # groups() no incluye el grupo 0 ('a', '455')
```

Los objetos *matcher* **guardan más información sobre la coincidencia**. Por ejemplo, la posición de la cadena en la que se produjo. En este caso, al principio de la cadena:

```
matcher.pos 0
```

También permiten **sustituir los grupos capturados en una cadena cualquiera**, mediante el uso de referencias de la forma `\g`, donde *x* es el número de grupo:

```
print matcher.expand('La cadena que coincidió fue \g<0>, el grupo 1 es \g<1> y el gr
```

Es importante mencionar que, si bien **findall** no devuelve objetos tipo **matcher**, sí proporciona los grupos de forma similar, como una lista de **tuplas**:

```
patron.findall('a455 a333b435') # [('a', '455'), ('a', '333'), ('b', '435')]
```

Reemplazo de cadenas

Similar a la combinación **search + expand**, existe el método sub; cuya función es encontrar todas las coincidencias de un patrón y sustituirlas por una cadena. Este **recibe dos parámetros**: el primero es la cadena con la que se sustituirá el patrón y el segundo es la cadena sobre la que queremos aplicar la sustitución. Y también se pueden utilizar referencias:

```
patron.sub("X", 'a455 a333b435') # sustituye todas las ocurrencias por X 'X XX'
patron.sub("LETRA(\g<1>), NUMERO(\g<2>)", 'a455 a333b435') # El reemplazo depende de
```

Grupos con nombre

De la misma forma en la que podemos usar grupos numerados, también podemos usar **grupos con nombre**. Esto hace más cómodo el **manejo de patrones complejos**; ya que siempre es más natural manejar un nombre que un número. Además, si sólo usamos números de grupo, podemos tener errores si modificamos el patrón para agregar algún grupo. Es decir, si lo agregamos podríamos estar cambiando el índice de otro posterior. Los nombres de grupo se definen agregando `?P` al paréntesis de apertura del grupo:

```
patron = re.compile('(?P<letra>[ab])(?P<numero>[3-5]+)') # defino dos grupos con nom
matcher = patron.search('a455 a333b435') # busco en la misma cadena de antes
matcher.groups() # groups y group(n) funcionan igual ('a', '455')
matcher.group(1) # 'a'
 matcher.group('letra')  # pero además ahora puedo acceder por nombre 'a'
matcher.group('numero') # '455'
matcher.expand('La letra es \g<letra>') # las referencias se usan con el nombre en v
```

Otra ventaja de utilizar nombres de grupo es que podemos usar el método groupdict para obtener un diccionario de pares **nombre-contenido** de cada grupo:

```
matcher.groupdict() {'letra': 'a', 'numero': '455'}
```

Modificadores para el patrón

Existen varios modificadores que podemos pasar al método compile para **modificar el comportamiento del patrón**. Los más usados son:

- `re.I` o `re.IGNORECASE`: hace que el patrón no distinga entre minúsculas y mayúsculas.
- `re.M` o `re.MULTILINE`: modifica el comportamiento de `^` y `<letra para que coincidan con el comienzo y final de cada línea de la cadena, en lugar de coincidir con el comienzo y final de la cadena entera
- `re.S` o `re.DOTALL`: hace que el punto (.) coincida además con un salto de línea. Sin este modificador, el punto coincide con cualquier caracter excepto un salto de línea

Cada modificador se usa como segundo parámetro de la función. Podemos unir los efectos de más de un modificador separándolos con `|`. Por ejemplo:

```
patron = re.compile('el patron', re.I | re.MULTILINE)
```

Estos son sólo algunos ejemplos de expresiones regulares. Si te interesa conocer la lista completa, puedes encontrarla en el sitio de Python. En la siguiente entrega de este artículo trabajaremos con **Diccionarios, funciones y archivos** en Python. Aunque si quieres conocer las **mejores técnicas de desarrollo de aplicaciones web** con Python, Django, Node.js y otras tecnologías, regístrate al Curso de Backend en Platzi. Ahí lo aprenderás todo de aplicaciones en tiempo real, administración, APIs REST y despliegue en servidores.</letra>
</numero></letra>

Platzi
@soporte-platzi 537 Puntos hace 5 años Todas sus entradas