Programación Avanzada
Daniel Gregorio Longino
Tarea 8

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*.

Las expresiones regulares (llamadas RE, o regex, o patrones de regex) son esencialmente en un lenguaje de programación diminuto y altamente especializado incrustado dentro de Python y disponible a través del módulo `re`. Usando este pequeño lenguaje, especificas las reglas para el conjunto de cadenas de caracteres posibles que deseas hacer coincidir; este conjunto puede contener frases en inglés, o direcciones de correo electrónico, o comandos TeX, o cualquier cosa que desee. A continuación, puede hacer preguntas como ¿Coincide esta cadena con el patrón? o ¿Hay alguna coincidencia con el patrón en alguna parte de esta cadena?. También puede utilizar RE para modificar una cadena de caracteres o dividirla de varias formas.

1. El módulo `re`

La funcionalidad de Regex en Python reside en un módulo llamado `re`. El módulo re contiene muchas funciones y métodos útiles. Por ahora únicamente nos enfocaremos en una función, `re.search(<regex>, <string>)`.

`re.search(<regex>, <string>)` escanea `<string>` buscando la primera ubicación donde coincide el patrón `<regex>`. Si se encuentra una coincidencia, `re.search()` devuelve un objeto de coincidencia. De lo contrario, devuelve `None`.

Debido a que `search()` reside en el módulo `re`, debe importarlo antes de poder usarlo. Una forma de hacer esto es importar todo el módulo y luego usar el nombre del módulo como prefijo al llamar a la función:

Python

```python
import re
re.search(...)
```

Alternativamente, puede importar la función desde el módulo por nombre y luego referirse a ella sin el prefijo del nombre del módulo:

Python

```
from re import search
search(...)
```

Veamos ahora un primer ejemplo de coincidencia de patrones:

Python >>>

```
>>> s = 'foo123bar'

>>> # One last reminder to import!
>>> import re

>>> re.search('123', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>
```

Aquí, el patrón de búsqueda `<regex>` es `123` y `<string>` es `s`. El objeto de coincidencia devuelto aparece en la línea 7. Los objetos de coincidencia contienen una gran cantidad de información útil que exploraremos pronto.

Por el momento, el punto importante es que `re.search()` de hecho devolvió un objeto de coincidencia en lugar de `None`. Eso nos dice que encontró una coincidencia. En otras palabras, el patrón `<regex>` especificado `123` está presente en `s`.

Un objeto de coincidencia es de verdad por lo que podemos usarlo en un contexto booleano como una declaración condicional:

Python >>>

```
>>> if re.search('123', s):
...     print('Found a match.')
... else:
...     print('No match.')
...
Found a match.
```

El intérprete muestra el objeto de coincidencia como `<_sre.SRE_Match object; span=(3, 6), match='123'>`. Esto contiene información útil. `span=(3, 6)` indica la parte de `<string>` en la que se encontró la coincidencia. En este ejemplo,

la coincidencia comienza en la posición del carácter 3 y se extiende hasta la posición 6, pero sin incluirla.

Este es un buen comienzo. Pero en este caso, el patrón `<regex>` es simplemente la cadena simple `'123'`. La coincidencia de patrones aquí sigue siendo solo una comparación de carácter por carácter.

1.1. Metacaracteres

El verdadero poder de la coincidencia de expresiones regulares en Python surge cuando `<regex>` contiene caracteres especiales llamados metacarateres. Estos tienen un significado único para el motor de coincidencia de expresiones regulares y mejoran enormemente la capacidad de búsqueda.

Considere nuevamente el problema de cómo determinar si una cadena contiene tres dígitos decimales consecutivos.

En una expresión regular, un conjunto de carácteres especificados entre corchetes `[]` constituye una clase de caracteres. Esta secuencia de metacaracteres coincide con cualquier carácter individual que esté en la clase, como se demuestra en el siguiente ejemplo:

Python >>>

```
>>> s = 'foo123bar'
>>> re.search('[0-9][0-9][0-9]', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>
```

`[0-9]` coincide con cualquier carácter de un solo dígito decimal, cualquier carácter entre `'0'` y `'9'`, inclusive. La expresión completa `[0-9][0-9][0-9]` coincide con cualquier secuencia de tres dígitos decimales. En este caso, `s` coincide porque contiene tres dígitos decimales consecutivos, `'123'`.

Por otro lado, una cadena que no contenga tres dígitos consecutivos no coincidirá:

Python

```
>>> print(re.search('[0-9][0-9][0-9]', '12foo34'))
None
```

Con expresiones regulares en Python, puede identificar patrones en una cadena que no podría encontrar con el operador `in` o con métodos de cadena.

Eche un vistazo a otro metacarácter de expresiones regulares. El metacarácter de punto `(.)` coincide con cualquier carácter excepto una nueva línea.

En el primer ejemplo, la expresión regular `1.3` coincide con `'123'` porque el `'1'` y el `'3'` coinciden literalmente, y el `.` coincide con el `'2'`. Aquí, básicamente estás preguntando: '¿Contiene `s` un `'1'`, luego cualquier caracter (excepto una

Python

```
>>> s = 'foo123bar'
>>> re.search('1.3', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>

>>> s = 'foo13bar'
>>> print(re.search('1.3', s))
None
```

nueva línea), luego un `'3'`? La respuesta es sí para `'foo123bar'` pero no para `'foo13bar'`.

Estos ejemplos proporcionan una ilustración rápida del poder de los metacaracteres de expresiones regulares. La clase de carácter y el punto son solo dos de los metacaracteres admitidos por el módulo `re`. Hay muchos más.

Los caracteres contenidos entre corchetes `([])` representan una clase de caracteres: un conjunto enumerado de caracteres para buscar coincidencias. Una secuencia de metacaracteres de clase de caracteres coincidirá con cualquier carácter individual contenido en la clase.

Puede enumerar los caracteres individualmente así:

Python

```
>>> re.search('ba[artz]', 'foobarqux')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
>>> re.search('ba[artz]', 'foobazqux')
<_sre.SRE_Match object; span=(3, 6), match='baz'>
```

La secuencia de metacaracteres `[artz]` coincide con cualquier carácter `'a'`, `'r'`, `'t'` o `'z'`. En el ejemplo, la expresión regular `ba[artz]` coincide con `'bar'` y `'baz'` (y también coincidiría con `'baa'` y `'bat'`).

Una clase de caracteres también puede contener un rango de caracteres separados por un guión `(-)`, en cuyo caso coincide con cualquier carácter dentro del rango. Por ejemplo, `[a-z]` coincide con cualquier carácter alfabético en minúscula entre `'a'` y `'z'`, inclusive:

Python

```
>>> re.search('[a-z]', 'FOObar')
<_sre.SRE_Match object; span=(3, 4), match='b'>
```

`[0-9]` coincide con cualquier carácter de dígito:

Python

```
>>> re.search('[0-9][0-9]', 'foo123bar')
<_sre.SRE_Match object; span=(3, 5), match='12'>
```

En los ejemplos anteriores, el valor de retorno es siempre la coincidencia posible más a la izquierda. `re.search()` escanea la cadena de búsqueda de izquierda a derecha, y tan pronto como encuentra una coincidencia para `<regex>`, deja de escanear y devuelve la coincidencia.

Puede complementar una clase de carácter especificando ^ como el primer carácter, en cuyo caso coincide con cualquier carácter que no esté en el conjunto. En el siguiente ejemplo, `[^0-9]` coincide con cualquier carácter que no sea un dígito:

Python

```
>>> re.search('[^0-9]', '12345foo')
<_sre.SRE_Match object; span=(5, 6), match='f'>
```

\w coincide con cualquier carácter de palabra alfanumérico. Los caracteres de las palabras son letras mayúsculas y minúsculas, dígitos y el carácter de subrayado (_), por lo que \w es esencialmente una abreviatura de `[a-zA-Z0-9_]`:

Python

```
>>> re.search('\w', '#(.a$@&')
<_sre.SRE_Match object; span=(3, 4), match='a'>
>>> re.search('[a-zA-Z0-9_]', '#(.a$@&')
<_sre.SRE_Match object; span=(3, 4), match='a'>
```

\W es lo contrario. Coincide con cualquier carácter que no sea una letra y es equivalente a `[[^a-zA-Z0-9_]`:

Python

```
>>> re.search('\W', 'a_1*3Qb')
<_sre.SRE_Match object; span=(3, 4), match='*'>
>>> re.search('[^a-zA-Z0-9_]', 'a_1*3Qb')
<_sre.SRE_Match object; span=(3, 4), match='*'>
```

\d coincide con cualquier carácter de dígito decimal. \D es lo contrario. Coincide con cualquier carácter que no sea un dígito decimal:

Python

```
>>> re.search('\d', 'abc4def')
<_sre.SRE_Match object; span=(3, 4), match='4'>

>>> re.search('\D', '234Q678')
<_sre.SRE_Match object; span=(3, 4), match='Q'>
```

\s coincide con cualquier carácter de espacio en blanco.

Python

```
>>> re.search('\s', 'foo\nbar baz')
<_sre.SRE_Match object; span=(3, 4), match='\n'>
```

\S es lo contrario de \s. Coincide con cualquier carácter que no sea un espacio en blanco:

Python

```
>>> re.search('\S', '  \n foo  \n  ')
<_sre.SRE_Match object; span=(4, 5), match='f'>
```

De nuevo, \s y \S consideran que una nueva línea es un espacio en blanco. En el ejemplo anterior, el primer carácter que no es un espacio en blanco es 'f'.

Las secuencias de clases de caracteres \w, \W, \d, \D, \s y \S también pueden aparecer dentro de una clase de caracteres de corchetes:

Python

```
>>> re.search('[\d\w\s]', '---3---')
<_sre.SRE_Match object; span=(3, 4), match='3'>
>>> re.search('[\d\w\s]', '---a---')
<_sre.SRE_Match object; span=(3, 4), match='a'>
>>> re.search('[\d\w\s]', '--- ---')
<_sre.SRE_Match object; span=(3, 4), match=' '>
```

En este caso, `[\d\w\s]` coincide con cualquier dígito, palabra o carácter de espacio en blanco. Y dado que \w incluye \d, la misma clase de caracteres también podría expresarse un poco más corta como `[\w\s]`

Otros metacaracteres son:

1. ^ - Intercalación. El símbolo de intercalación `(^)` coincide con el comienzo de la cadena, es decir, comprueba si la cadena comienza con los caracteres dados o no. Por ejemplo:
   a) `^g` verificará si la cadena comienza con `g`, como `geeks, globe, girl, g`, etc.
   b) `^ge` verificará si la cadena comienza con `ge`, como `geeks`, `geeksforgeeks`, etc.
2. $-Dolar .El símbolo de dólar ($) coincide con el final de la cadena, es decir, comprueba si la cadena termina con los caracteres dados o no. Por ejemplo, s$ buscará la cadena que termina con s como geeks, extremos, s, etc. ks$ buscará la cadena que termina con ks, como geeks, geeksforgeeks, ks, etc.
3. `.` - Punto. El símbolo de punto `(.)` coincide con un solo carácter excepto el carácter de nueva línea `\n`. Por ejemplo `a.b` buscará la cadena que contiene cualquier carácter en el lugar del punto, como `acb`, `acbd`, `abbb`, etc.
4. | - O. El símbolo `or` funciona como el operador `or`, lo que significa que verifica si el patrón antes o después del símbolo `or` está presente en la cadena o no. Por ejemplo `a|b` coincidirá con cualquier cadena que contenga `a` o `b`, como `acd`, `bcd`, `abcd`, etc.
5. `?` – Signo de interrogación. El signo de interrogación `(?)` comprueba si la cadena antes del signo de interrogación en la expresión regular aparece al menos una vez o no aparece en absoluto. Por ejemplo `ab?c` coincidirá con la cadena `ac`, `acb`, `dabc` pero no coincidirá con `abbc` porque hay dos `b`. De manera similar, no coincidirá con `abdc` porque `b` no va seguido de `c`.
6. `*` - Estrella. El símbolo de estrella `(*)` coincide con cero o más apariciones de la expresión regular que precede al símbolo `*`. Por ejemplo `ab*c` coincidirá con la cadena `ac`, `abc`, `abbbc`, `dabc`, etc. pero no coincidirá con `abdc` porque `b` no va seguida de `c`.
7. + – Más. El símbolo más (+) coincide con una o más apariciones de la expresión regular que precede al símbolo +. Por ejemplo `ab+c` coincidirá con la cadena `abc`, `abbc`, `dabc`, pero no con `ac`, `abdc` porque no hay `b` en `ac` y `b` no va seguida de `c` en `abdc`.
8. {`m, n`} – Llaves. Las llaves coinciden con las repeticiones que preceden a la expresión regular de `m` a `n`, ambas inclusive. Por ejemplo `a{2, 4}` coincidirá con la cadena `aaab`, `baaaac`, `gaad`, pero no coincidirá con cadenas como `abc`, `bc` porque solo hay una `a` o ninguna `a` en ambos casos.

1.2. Funciones de busqueda

Las funciones de búsqueda escanean una cadena de búsqueda para una o más coincidencias de la expresión regular especificada:

1. `re.search()`. Escanea una cadena en busca de una coincidencia de expresiones regulares.

Python

```
>>> re.search(r'(\d+)', 'foo123bar')
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> re.search(r'[a-z]+', '123FOO456', flags=re.IGNORECASE)
<_sre.SRE_Match object; span=(3, 6), match='FOO'>

>>> print(re.search(r'\d+', 'foo.bar'))
None
```

2. `re.match()`. Busca una coincidencia de expresiones regulares al comienzo de una cadena.

Python

```
>>> re.search(r'\d+', '123foobar')
<_sre.SRE_Match object; span=(0, 3), match='123'>
>>> re.search(r'\d+', 'foo123bar')
<_sre.SRE_Match object; span=(3, 6), match='123'>

>>> re.match(r'\d+', '123foobar')
<_sre.SRE_Match object; span=(0, 3), match='123'>
>>> print(re.match(r'\d+', 'foo123bar'))
None
```

3. `re.fullmatch()`. Busca una coincidencia de expresiones regulares en una cadena completa.

Python

```
>>> print(re.fullmatch(r'\d+', '123foo'))
None
>>> print(re.fullmatch(r'\d+', 'foo123'))
None
>>> print(re.fullmatch(r'\d+', 'foo123bar'))
None
>>> re.fullmatch(r'\d+', '123')
<_sre.SRE_Match object; span=(0, 3), match='123'>

>>> re.search(r'^\d+$', '123')
<_sre.SRE_Match object; span=(0, 3), match='123'>
```

4. `re.findall()`. Devuelve una lista de todas las coincidencias de expresiones regulares en una cadena.

Python

```
>>> re.findall(r'\w+', '...foo,,,,bar:%$baz//|')
['foo', 'bar', 'baz']
```

5. `re.finditer()`. Devuelve un iterador que produce coincidencias de expresiones regulares de una cadena.

```
>>> for i in re.finditer(r'\w+', '...foo,,,,bar:%$baz//|'):
...     print(i)
...
<_sre.SRE_Match object; span=(3, 6), match='foo'>
<_sre.SRE_Match object; span=(10, 13), match='bar'>
<_sre.SRE_Match object; span=(16, 19), match='baz'>
```