Tarea 8: Expresiones regulares en Python.

Nombre: German Jordi Arreortua Reyes.
Materia: Programación Avanzada
Fecha: jueves, 12 de mayo de 2022.

Las expresiones regulares en Python son una secuencia especial de caracteres que define un patrón para la coincidencia de cadenas.

A continuación se muestra el uso de expresiones regulares en Python.

Sea s una cadena y si queremos comprobar si una cadena es subcadena de s podemos usar el operador in, el cual devolverá True si es subcadena y False en caso contrario, como a continuación se muestra

```
>>> s = 'Abcd123fgh456'
>>> '123' in s
True
```

Si además de eso, queremos conocer donde está dicha subcadena podemos usar .find() o .index(), las cuales devuelven la posición del carácter dentro de s donde empieza la subcadena, como a continuación se muestra.

```
>>> s.find('123')
4
>>> s.index('123')
4
```

Si la cadena ingresada no es subcadena de s devuelve ValueError: substring not found.

Pero si en lugar de buscar la cadena ‘123’ queremos conocer si la cadena contiene una subcadena de tres números enteros entre 0 y 9 consecutivos entonces usaremos las expresiones regulares.

Función search del módulo re.

re.search(<regex>, <string>, <flags>)

re.search(<regex>, <string>, <flags>), tiene tres argumentos, <regex> una expresión regular, <string> una cadena y <flags>, este argumento es opcional y lo veremos más adelante. Así pues re.search() escanea una cadena en busca de una coincidencia de expresiones regulares, es decir, recibe una cadena (<string>) y busca la primera ubicación donde coincida con <regex>, si se encuentra una coincidencia devuelve un objeto de coincidencia, de lo contrario, devuelve None.

Ejemplo:

```
>>> s = 'Abcd123fgh456'
>>> import re
>>> re.search('123',s)
<re.Match object; span=(4, 7), match='123'>
```

En lo anterior la cadena (<string>) es s y <regex> es ‘123’, lo obtenido es el objeto de coincidencia <re.Match object; span=(4, 7), match='123'>, donde span=(4,7), indica la parte de s en la que encontró la coincidencia y match='123' indica la coincidencia.

Metacaracteres Regex de Python

En el ejemplo anterior <regex> fue la cadena ‘123’, y al aplicar re.search() se realiza una comparación de caracteres que es parecido a lo visto con in, .find e .index. Ahora veremos las coincidencias de expresiones regulares cuando <regex> contiene caracteres especiales llamados metacaracteres.

Por ejemplo consideremos s = ‘Abcd123fgh456’ y comprobaremos si s contiene tres números enteros entre 0 y 9, como a continuación se muestra

```
>>> s = 'Abcd123fgh456'
>>> re.search('[0-9][0-9][0-9]', s)
<re.Match object; span=(4, 7), match='123'>
```

En una expresión regular, un conjunto de caracteres especificados entre corchetes constituyen una clase de caracteres. En este caso ‘[0,9]’, indica que queremos encontrar una coincidencia con cualquier carácter de un solo dígito entre 0 y 9, por lo cual ‘[0-9][0-9][0-9]’ indica que queremos encontrar una coincidencia de una secuencia de tres dígitos entre 0 y 9. Al ingresar ‘[0-9][0-9][0-9]’ y s en re.search() obteneos la primera coincidencia de tres números entre 0 y 9 consecutivos la cual es match='123' y la parte de s donde coincide es span=(4,7).

Así pues con expresiones regulares podemos encontrar patrones en una cadena que no podemos encontrar con el operador in o con algún método de cadena.

Metacaracteres que coinciden con un solo carácter

Ahora veamos todos los metacaracteres compatiblex con el módulo re que coinciden con un solo carácter. Estos metacaracteres intentan hacer coincidir un solo carácter de la cadena de búsqueda.

- **[]:** Especifica un conjunto de caracteres para hacerlo coincidir.

Los caracteres entre corchetes representan una clase de caracteres la cual es un conjunto enumerado de caracteres para buscar coincidencias. Dada una clase de caracteres re.search() devolverá el objeto de coincidencia con match un solo carácter en la clase de caracteres, o nulo en caso de que ninguno de los caracteres de la clase de caracteres este en la cadena.

Ejemplo:

```
>>> re.search('[wxzy]', 'A12arsyzxwrsB78')
<re.Match object; span=(6, 7), match='y'>
```

```
>>> re.search('rs[wxzy]', 'A12arsyzxwrsB78')
<re.Match object; span=(4, 7), match='rsy'>
```

[wxzy] busca la coincidencia de ‘w’, ‘x’, ‘z’ o ‘y’. En el primer ejemplo la coincidencia es ‘y’, pues es el primer carácter de la cadena'A12arsyzxwrsB78' que se encuentra en la clase [wxzy]. En el segundo ejemplo especificamos que debe de encontrase primeramente la coincidencia de ‘rs’ por lo cual match=‘rsy’.
Una clase de caracteres también puede contener un rango de caracteres separados por un guion y se buscara la coincidencia con un único carácter dentro del rango, por ejemplo

```
>>> re.search('[a-z]', 'A12arsyzxwrsB78')
<re.Match object; span=(3, 4), match='a'>
```

- **.** (Punto)**:** Coincide con cualquier carácter individual excepto con una nueva línea.

Ejemplo:

```
>>> re.search('z.2', 'A09zC21')
<re.Match object; span=(3, 6), match='zC2'>

>>> re.search('z..2', 'A09zab21')
<re.Match object; span=(3, 7), match='zab2'>
```

La expresión regular ‘z.2’ coincide con ‘zC2’, porque ‘z’ y ‘2’ están en la cadena 'A09zC21' y solo hay un carácter entre los dos el cual es ‘C’. Básicamente se está preguntado ¿Hay un solo carácter entre ‘z’ y ‘2’ distinto de nueva línea en 'A09zC21'?.
En el ejemplo 2, se está preguntando ¿Hay dos caracteres entre ‘z’ y ‘2’ distinto de nueva línea en 'A09zab21'?.

- **\w, \W**: Coincidencia basada en si es un carácter de una palabra.

\w coincide con cualquier carácter de una palabra, 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_], mientras que \W es lo contrario, coincide con cualquier carácter distinto de letras mayúsculas y minúsculas, dígitos y el carácter de subrayado (_) es decir \W es equivalente a [^a-zA-Z0-9_]. ^ indica el complemento de una clase de caracteres.

Ejemplo:

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

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

- **\d, \D**: Coincidencia basada en si el carácter es un dígito decimal.

\d coincide con cualquier carácter de dígito decimal, mientras que \D coincide con cualquier carácter que no sea un dígito decimal

Ejemplo:

```
>>> re.search('\d', 'A_x45df')
<re.Match object; span=(3, 4), match='4'>

>>> re.search('\D', 'A_x45df')
<re.Match object; span=(0, 1), match='A'>
```

- **\s, \S:** Coincidencia basada en si un carácter representa un espacio en blanco.

\s coincide con cualquier carácter de espacio en blanco o salto de línea mientras que \S coincide con cualquier carácter distinto de espacio en blanco o salto de línea.

```
>>> re.search('\s', 'A_w\naa zz')
<re.Match object; span=(3, 4), match='\n'>

>>> re.search('\S', ' \naa zz')
<re.Match object; span=(2, 3), match='a'>
```

Evitar metacaracteres

Si queremos usar algún metacarácter en nuestra expresión regular, pero sin que tenga el significado especial de metacarácter es decir que se represente literalmente como un carácter, usaremos el siguiente metacarácter

- ****: Elimina el significado especial de un metacarácter.

Ejemplo:

```
>>> re.search('.', 'Hola.123')
<re.Match object; span=(0, 1), match='H'>

>>> re.search('\.', 'Hola.123')
<re.Match object; span=(4, 5), match='.'>
```

Vemos que el primer ejemplo tomo al punto como metacarácter, mientras que en el segundo lo tomo como el carácter '.'.

Anclas

Un ancla dicta una ubicación particular en la cadena de búsqueda donde debe ocurrir una coincidencia. Las anclas no coinciden con ningún carácter en la cadena de búsqueda y no consumen nada de la cadena de búsqueda durante el análisis.

- **^, \A:** Ancla una coincidencia al comienzo de <string>.

Cuando el analizador de expresiones regulares encuentra ^ o \A, la posición actual del analizador debe estar al principio de la cadena de búsqueda para que encuentre una coincidencia.

Ejemplo:

```
>>> re.search('^Hola', 'Hola123asdf')
<re.Match object; span=(0, 4), match='Hola'>

>>> print(re.search('^Hola', '123Holaasdf'))
None
```

- **$, \Z:** Ancla una coincidencia al final de <string>.

Ejemplo:

```
>>> re.search('Hola$', '123asdfHola')
<re.Match object; span=(7, 11), match='Hola'>

>>> print(re.search('Hola$', '123Holaasdf'))
None
```

- **\b**: Ancla una coincidencia al principio o al final de una palabra.

Ejemplo:

```
>>> re.search(r'\bfdg', 'abc*+fdg123')
<re.Match object; span=(5, 8), match='fdg'>

>>> print(re.search(r'\bfdg', 'abcfdg123'))
None

>>> re.search(r'fdg\b', 'abcfdg*qwe')
<re.Match object; span=(3, 6), match='fdg'>
>>> re.search(r'\bbar\b', 'foo(bar)baz')
<re.Match object; span=(4, 7), match='bar'>
```

- **\B:** Ancla una coincidencia a una ubicación que no es el principio o final de una palabra.

Ejemplo:

```
>>> re.search(r'\Bfoo\B', 'qwefoo123')
<re.Match object; span=(3, 6), match='foo'>
```

Cuantificadores

Un metacarácter cuantificador sigue inmediatamente a una parte del <regex> e indica cuántas veces debe ocurrir esa parte para que la coincidencia tenga éxito.

- ***:** Coincide con cero o más repeticiones del carácter o grupo anterior.

Más adelante veremos lo que es un grupo.

Ejemplo:

```
>>> re.search('a*', 'aaaa')
<re.Match object; span=(0, 4), match='aaaa'>
```

```
>>> re.search('abc-*efg', 'abc--efg')
<re.Match object; span=(0, 8), match='abc--efg'>
```

- **+:** Coincide con una o más repeticiones del carácter o grupo anterior.

+ es similar a *, pero la expresión regular cuantificada debe ocurrir al menos una vez.

```
>>> print(re.search('abc-+efg', 'abcefg'))
None

>>> re.search('abc-+efg', 'abc----efg')
<re.Match object; span=(0, 10), match='abc----efg'>
```

- **?** : Coincide con cero o una repetición del carácter o grupo anterior.

? es similar a * y +, pero en este caso solo hay una coincidencia si la expresión regular anterior ocurre una vez o no ocurre en absoluto.

Ejemplo:

```
>>> re.search('abc-?efg', 'abcefg')
<re.Match object; span=(0, 6), match='abcefg'>

>>> print(re.search('abc-?efg', 'abc--efg'))
None
```

- **{m}** : Coincide exactamente con m repeticiones del carácter o grupo anterior.

Es similar a * o +, pero especifica exactamente cuántas veces debe ocurrir la expresión regular anterior para que la coincidencia tenga éxito.

Ejemplo:

```
>>> re.search('a-{3}a', 'a---a')
<re.Match object; span=(0, 5), match='a---a'>

>>> print(re.search('a-{3}a', 'a----a'))
None
```

- **{m,n} :** Coincide entre m a n repeticiones del carácter o grupo anterior.

Ejemplo:

```
>>> re.search('a-{1,3}a', 'a-a')
<re.Match object; span=(0, 3), match='a-a'>
>>> re.search('a-{1,3}a', 'a--a')
<re.Match object; span=(0, 4), match='a--a'>
>>> re.search('a-{1,3}a', 'a---a')
<re.Match object; span=(0, 5), match='a---a'>
>>> print(re.search('a-{1,3}a', 'a----a'))
None
```

Construcciones de agrupación y referencias inversas.

Las construcciones de agrupación dividen una expresión regular en subexpresiones o grupos los cuales tiene como propósito que el grupo sea una sola entidad sintáctica y los metacaracteres se aplican a todo el grupo.
Como vimos en un inicio re.search() toma una expresión regular <regex>, pero ahora si ingresamos (<regex>) este define una subexpresión o grupo. Esta es la construcción de agrupación más básica, y al agrupar de esta forma hace que la expresión entre paréntesis solo coincida con el contenido dentro de los paréntesis.

Ejemplo:

```
>>> re.search('(abc)', 'ABC seabcd_efg')
<re.Match object; span=(6, 9), match='abc'>
```

Tratar un grupo como una unidad.

Un metacarácter cuantificador se aplica a todo el grupo.

Ejemplo:

```
>>> re.search('(abc){3}', 'fghabcabcabcghj')
<re.Match object; span=(3, 12), match='abcabcabc'>
```

Notemos que {3} trata a ‘abc’ como una unidad, pues si no estuviera entre paréntesis tendríamos los siguiente.

```
>>> print(re.search('abc{3}', 'fghabcabcabcghj'))
None
```

Pues en este caso el {3} solo está aplicando al carácter ‘c’.

Capturan de grupos

Si ingresamos más de un grupo en re.search(), entonces podemos ocupar **m.groups()** el cual devuelve una tupla que contiene todos los grupos capturados de una coincidencia.

Ejemplo:

```
>>> p = re.search('(.+),(\w+),(.+)', '&12*,12abd_0,gh_0&')
>>> p
<re.Match object; span=(0, 18), match='&12*,12abd_0,gh_0&'>
>>> p.groups()
('&12*', '12abd_0', 'gh_0&')
```

m.group(<n>): Devuelve una cadena que contiene la n-ésima coincidencia capturada.

Ejemplo:

```
>>> p = re.search('(.+),(\w+),(.+)', '&12*,12abd_0,gh_0&')
>>> p.group(1)
'&12*'
```

```
>>> p.group(2)
'12abd_0'
>>> p.group(3)
'gh_0&'
```

m.group(<n1>, <n2>, ...): Devuelve una tupla que contiene las coincidencias especificadas.

Ejemplo:

```
>>> p = re.search('(.+),(\w+),(.+)', '&12*,12abd_0,gh_0&')
>>> p.group(3,1)
('gh_0&', '&12*')
```

Referencia inversa.

Se puede hacer coincidir un grupo capturado previamente más tarde dentro de la misma expresión regular usando una secuencia especial de metacaracteres llamada referencia inversa.

\<n>: Coincide con el contenido del n-enésimo grupo capturado previamente, donde <n> es un entero entre 1 y 99.

Ejemplo:

```
>>> regex = r'(\w+),\1'
>>> m = re.search(regex, 'abc, abc')
>>> m.group
>>> m.group(1)
'abc'
>>> m = re.search(regex, '123, 123')
>>> m.group(1)
'123'
>>> m = re.search(regex, 'abc, 123')
>>> print(m)
None
```

En el primer ejemplo (\w+) coincide con ‘abc’ luego la coma coincide con la coma, posteriormente puesto que \1 es la referencia inversa del primer grupo vuelve a coincidir con ‘abc’. El segundo ejemplo es similar solo que (\w+) coincide con ‘123’ y finalmente en el último ejemplo tenemos que (\w+) coincide con ‘abc’, pero lo que viene después de la coma es ‘123’ que no coincide con ‘abc’ que es la referencia inversa \1, por lo cual nos devuelve None.

Coincidencias de expresiones regulares modificadas con flags.

Como vimos en la primera parte re.serch(<regex>, <string>, <flags>) el argumento <flags> es opcional y al ingresar <flags> se escanea una cadena en busca de una coincidencia de expresión regular, aplicando el modificador especificado <flags>.

A continuación veremos algunos de los flags admitidos.

re.I o **re. IGNORECASE**: Hace que la coincidencia de caracteres alfabéticos no distinga entre mayúsculas y minúsculas.

Ejemplo:

```
>>> re.search('a+', 'aaaAAA')
<re.Match object; span=(0, 3), match='aaa'>
>>> re.search('a+', 'aaaAAA',re.I)
<re.Match object; span=(0, 6), match='aaaAAA'>
```

re.M o **re.MULTILINE**: Hace que los anclajes de inicio y fin de cadena coincidan con las nuevas líneas.

Ejemplo:

```
>>> s = 'abc\nefg\nhij'
>>> re.search('^abc', s)
<re.Match object; span=(0, 3), match='abc'>
>>> print(re.search('^efg', s))
None
>>> print(re.search('efg$', s))
None
>>> re.search('hij', s)
<re.Match object; span=(8, 11), match='hij'>
```

Al anclar al inicio ‘abc’, notemos que si obtenemos match=’abc’, pero al anclar al inicio ‘efg’ obtenemos None, de igual manera si anclamos al final ‘hij’ obtemos match=’hij’ pero al anclar al final ‘efg’ obtenemos None. Ahora su usamos re.M obtenemos lo siguiente.

```
>>> re.search('^efg', s, re.M)
<re.Match object; span=(4, 7), match='efg'>
>>> re.search('efg$', s, re.M)
<re.Match object; span=(4, 7), match='efg'>
```

re.S o **re.DOTALL:** Hace que el metacarácter punto (.) coincida con una nueva línea.

Como vimos anteriormente el metacarácter punto coincide con cualquier carácter excepto con una nueva línea, así que al agregar re.S quitamos esta restricción.

Ejemplo:

```
>>> print(re.search('a.1', 'a\n1'))
None
>>> re.search('a.1', 'a\n1', re.S)
<re.Match object; span=(0, 3), match='a\n1'>
```

Fuente.
https://realpython.com/regex-python