David Gopar Morales

Tarea 8
Entrega 12 de mayo de 2022
Hacer un resumen sobre expresiones regulares en Python

Expresiones regulares o regex en Python son una secuencia especial de caracteres que definen un patrón para así manipular de manera compleja cadenas de caracteres.

Recordando que vimos en el curso de computación que un lenguaje regular es el lenguaje que es reconocible por un autómata finito y formalmente puede ser expresado usando expresiones regulares.

Tenemos que la funcionalidad de regex en python se encuentra en el módulo llamado re, en este módulo se encuentran muchas funciones y métodos muy útiles.

Comenzaremos entendiendo la función re.search()

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

Busca en <string> la primera locación donde el patrón <regex> coincida, si se encuentra nos regresa un objeto match, en caso de no encontrarlo nos regresa none.
Para importar esta función podemos hacerlo con

```
>>> import re
>>> re.search(...)
```

O como

```
>>> from re import search
```

Ahora, para comprender mejor esto, veamos el siguiente ejemplo

```
>>> import re
>>> s='sasa123aSS'
>>> re.search('123',s)
<re.Match object; span=(4, 7), match='123'>
```

Dónde span(4,7) nos dice en dónde se encuentra lo que buscamos, y match=’123’ nos indica que encontró lo que buscábamos
Otro ejemplo puede ser:

```
>>> if re.search('123',s):
...     print('se encontro')
... else:
...     print('no se encontro')
...
se encontro
```

Pasamos ahora a un ejemplo más interesante.
Tenemos que un conjunto de caracteres específicos se nos indicará entre corchetes ([]).

```
>>> re.search('[0-9][0-9][0-9]', 'foo456bar')
<re.Match object; span=(3, 6), match='456'>
>>> re.search('[0-9][0-9][0-9]', '234baz')
<re.Match object; span=(0, 3), match='234'>
>>> re.search('[0-9][0-9][0-9]', 'qux678')
<re.Match object; span=(3, 6), match='678'>
```

Como podemos apreciar, [0-9] nos indica que se busca un caracter de un número entre el 0 y el nueve y al poner [0-9][0-9][0-9], nos indica que se quiere buscar un patrón de 3 números consecutivos entre el 0 y el 9 enteros.
Y como podemos ver, cuando no se tengan esos 3 dígitos consecutivos no se hará un match.

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

Ahora, un ejemplo de algo que puede hacer re.search() pero no se puede hacer con el operador in es lo siguiente:

```
>>> re.search('1.3','foo123bar')
<re.Match object; span=(3, 6), match='123'>
>>> print(re.search('1.3','foo13bar'))
None
```

Donde el '.' funciona como cualquier caracter que pueda aparecer entre el 1 y el 3, es por eso que en el primer caso si lo encuentra, pues el 2 es cualquier caracter, pero en el segundo caso como no hay ninguno ente el 1 y el 3, no se hace el match.

A continuación mostramos los 'methacaracteres' que son aceptados por el módulo re y qué función tienen.

Character(s)	Meaning
.	Matches any single character except newline
^	· Anchors a match at the start of a string · Complements a character class
$	Anchors a match at the end of a string
*	Matches zero or more repetitions
+	Matches one or more repetitions
?	· Matches zero or one repetition · Specifies the non-greedy versions of *, +, and ? · Introduces a lookahead or lookbehind assertion · Creates a named group
{}	Matches an explicitly specified number of repetitions
\	· Escapes a metacharacter of its special meaning · Introduces a special character class · Introduces a grouping backreference

[]	Specifies a character class
|	Designates alternation
()	Creates a group
: # = !	Designate a specialized group
<>	Creates a named group

Ahora pasamos a ver otros ejemplos para ir comprendiendo mejor el uso de esto.
[] lo que pongamos dentro de esto será una lista de caracteres que estamos buscando y cuando se encuentre alguno de estos se hará el match.
Por ejemplo:

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

Aquí se busca la cadena ba y despuestres alguno de los otros caracteres que se encuentran entre llaves.

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

En el mostrado anteriormente se busca la primera letra minúscula en la cadena.

[0-9a-fA-F] busca cualquier digito hexadecimal:

```
>>> re.search('[0-9a-fA-f]', '--- a0 ---')
<re.Match object; span=(4, 5), match='a'>
>>> re.search('[0-9a-fA-f]', '--- 0a ---')
<re.Match object; span=(4, 5), match='0'>
```

Para buscar el complemento de cierto conjunto se puede usar ^ al inicio del [], es decir como el primer caracter.

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

Y cuando no aparece al inicio simplemente se busca el caracter deseado.

```
>>> re.search('[#:^]', 'foo^bar:baz#qux')
<re.Match object; span=(3, 4), match='^'>
```

Si se quiere buscar ‘-’ se puede poner al final o al inicio del conjunto que buscamos o utilizar \-
Ejemplo:

```
>>> re.search('[-abc]', '123-456')
<_sre.SRE_Match object; span=(3, 4), match='-'>
```

```
>>> re.search('[abc-]', '123-456')
<_sre.SRE_Match object; span=(3, 4), match='-'>
>>> re.search('[ab\-c]', '123-456')
<_sre.SRE_Match object; span=(3, 4), match='-'>
```

Otros ejemplos, son que si se quiere buscar], se puede poner [\]], y que otros methacaracteres pierden su significado especial dentro de [].

Tenemos que buscar \w es el equivalente a buscar [a-zA-Z0-9_] y \W es lo contrario, es decir, como buscar [^a-zA-Z0-9_]
Ejemplo:

```
>>> 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'>

>>> 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 nos busca el primer caracter que sea un dígito, y \D todo lo contrario, es decir.

```
>>> 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 busca el caracer que sea un espacio en blanco, y \S lo contrario, el que no sea un espacio en blanco, como ejemplo:

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

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

Estos \s\w\d pueden aparecer dentro de [].

Y como ya se ha mencionado ‘\’, quita el significado especial de los caracteres.

Aunque algunas veces esto puede dar problemas, como para buscar \, pero se puede realizar como\\\\.

Cuando uno pone ^ al inicio de lo que quiere buscar, se buscara la cadena que se pone después solo al inicio de la palabra y no en ningún otro lugar, por ejemplo:

```
>>> re.search('^foo', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> print(re.search('^foo', 'barfoo'))
None

>>> re.search('\Afoo', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> print(re.search('\Afoo', 'barfoo'))
None
```

Y con $, o \z pasa lo contrario, se busca al final de la palabra.

Con \b, busca los caracteres al final o al inicio tomando en cuenta al rededor letras, colocar \b...\b pondrá a buscar entre dos cadenas de letras.

\B ocurre que se busca pero ahora no considerando letras en las fronteras a buscar.

```
>>> re.search(r'\bbar\b', 'foo bar baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>
>>> re.search(r'\bbar\b', 'foo(bar)baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>

>>> print(re.search(r'\bbar\b', 'foobarbaz'))
None



>>> print(re.search(r'\Bfoo\B', 'foo'))
None

>>> print(re.search(r'\Bfoo\B', '.foo.'))

None



>>> re.search(r'\Bfoo\B', 'barfoobaz')

<_sre.SRE_Match object; span=(3, 6), match='foo'>
```

Cuantificadores

*

Si recordamos, en las expresiones regulares a* se utilizaba para indicar el conjunto de la palabra vacía y cualquier concatenación finita de ‘a’ posible. En Nuestro caso indicará exactamente lo mismo, como lo vemos en los ejemplos siguientes:

```
>>> re.search('foo-*bar', 'foobar')                # Zero dashes
<_sre.SRE_Match object; span=(0, 6), match='foobar'>

>>> re.search('foo-*bar', 'foo-bar')               # One dash
<_sre.SRE_Match object; span=(0, 7), match='foo-bar'>
```

```
>>> re.search('foo-*bar', 'foo--bar')                # Two dashes
<_sre.SRE_Match object; span=(0, 8), match='foo—bar'>
```

+

El caracter +, nos sirve para poder detectar una o mas repeticiones del regex que le preceda, por ejemplo.

```
>>> print(re.search('foo-+bar', 'foobar'))           # Zero dashes
None

>>> re.search('foo-+bar', 'foo-bar')                 # One dash
<_sre.SRE_Match object; span=(0, 7), match='foo-bar'>

>>> re.search('foo-+bar', 'foo--bar')                # Two dashes
<_sre.SRE_Match object; span=(0, 8), match='foo--bar'>
```

?

El caracter ? Es para identificar una o ninguna repetición del regex que le preceda, ejemplo:

```
>>> re.search('foo-?bar', 'foobar')                  # Zero dashes
<_sre.SRE_Match object; span=(0, 6), match='foobar'>

>>> re.search('foo-?bar', 'foo-bar')                 # One dash
<_sre.SRE_Match object; span=(0, 7), match='foo-bar'>

>>> print(re.search('foo-?bar', 'foo--bar'))         # Two dashes
None
```

Notemos que los regex previos también pueden estar dentro de [].
Cabe resaltar que si los usamos así, obtendremos la identificación más larga posible, pero si ponemos *?, +?, ??, obtendremos la identificación más corta.

{m}

En este caso buscará exactamente m-veces el regex que le preceda a {m}, ejemplo:

```
>>> print(re.search('x-{3}x', 'x--x'))               # Two dashes
None

>>> re.search('x-{3}x', 'x---x')                     # Three dashes
<_sre.SRE_Match object; span=(0, 5), match='x---x'>

>>> print(re.search('x-{3}x', 'x----x'))             # Four dashes
None
```

{m,n}

Identifica cualquier cantidad de repeticiones entre m y n del regex que le preceda, ejemplo:

```
>>> for i in range(1, 6):
...     s = f"x{'-' * i}x"
...     print(f'{i}  {s:10}', re.search('x-{2,4}x', s))
...
1  x-x        None
2  x--x       <_sre.SRE_Match object; span=(0, 4), match='x--x'>
3  x---x      <_sre.SRE_Match object; span=(0, 5), match='x---x'>
4  x----x     <_sre.SRE_Match object; span=(0, 6), match='x----x'>
5  x-----x    None
```

También se tiene lo siguiente:

Regular Expression	Matches	Identical to
`<regex>{,n}`	Any number of repetitions of `<regex>` less than or equal to `n`	`<regex>{0,n}`
`<regex>{m,}`	Any number of repetitions of `<regex>` greater than or equal to `m`	`----`
`<regex>{,}`	Any number of repetitions of `<regex>`	`<regex>{0,}` `<regex>*`

Además {m,n} busca la cadena más grande que se pueda identificar y {m,n}? la cadena más corta, es decir, se aplica lo que mencionamos anteriormente.

Cómo agrupar
Se agrupa lo que queremos poniéndolo entre paréntesis (), esto se entiende mejor con la siguiente tabla.

Regex	Interpretation	Matches	Examples
`bar+`	The + metacharacter applies only to the character `'r'`.	`'ba'` followed by one or more occurrences of `'r'`	`'bar'` `'barr'` `'barrr'`
`(bar)+`	The + metacharacter applies to the entire string `'bar'`.	One or more occurrences of `'bar'`	`'bar'` `'barbar'` `'barbarbar'`

Y como ejemplo de como aplicarlo tenemos lo siguiente:

```
>>> re.search('(bar)+', 'foo bar baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>

>>> re.search('(bar)+', 'foo barbar baz')
<_sre.SRE_Match object; span=(4, 10), match='barbar'>

>>> re.search('(bar)+', 'foo barbarbarbar baz')
<_sre.SRE_Match object; span=(4, 16), match='barbarbarbar'>
```

Y tenemos más ejemplos de qué cosas podemos hacer

```
>>> re.search('(ba[rz]){2,4}(qux)?', 'bazbarbazqux')
<_sre.SRE_Match object; span=(0, 12), match='bazbarbazqux'>
>>> re.search('(ba[rz]){2,4}(qux)?', 'barbar')
<_sre.SRE_Match object; span=(0, 6), match='barbar'>

>>> re.search('(foo(bar)?)+(\d\d\d)?', 'foofoobar')
<_sre.SRE_Match object; span=(0, 9), match='foofoobar'>
>>> re.search('(foo(bar)?)+(\d\d\d)?', 'foofoobar123')
<_sre.SRE_Match object; span=(0, 12), match='foofoobar123'>
>>> re.search('(foo(bar)?)+(\d\d\d)?', 'foofoo123')
<_sre.SRE_Match object; span=(0, 9), match='foofoo123'>
```

Y más ejemplos:

Regex	Matches
`foo(bar)?`	`'foo'` optionally followed by `'bar'`
`(foo(bar)?)+`	One or more occurrences of the above
`\d\d\d`	Three decimal digit characters
`(\d\d\d)?`	Zero or one occurrences of the above

Obtener los grupos capturados

Otra cosa muy útil es poder obtener los grupos de una cadena que se quería identificar, estos objetos match tienen un atributo .groups() que veremos a continuación en un ejemplo.

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

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

```
>>> m.group(0)
'foo,quux,baz'

>>> m.group(1)
'foo'
>>> m.group(2)
'quux'
>>> m.group(3)
'baz'

>>> m.group(2, 3)
('quux', 'baz')
>>> m.group(3, 2, 1)
('baz', 'quux', 'foo')

>>> m.group(3, 2, 1)
('baz', 'qux', 'foo')

>>> (m.group(3), m.group(2), m.group(1))
('baz', 'qux', 'foo')
```

En lo anterior podemos ver ejemplos tambien del atributo .group() y como es posible utilizarlo.

Referencias anteriores
\<n>
Estos símbolos nos ayudan a volver a identificar el contenido del <n>th grupo capturado, es decir.

```
>>> regex = r'(\w+),\1'
>>> m = re.search(regex, 'foo,foo')
>>> m
<_sre.SRE_Match object; span=(0, 7), match='foo,foo'>

>>> m.group(1)
'foo'

>>> m = re.search(regex, 'qux,qux')
>>> m
<_sre.SRE_Match object; span=(0, 7), match='qux,qux'>

>>> m.group(1)
'qux'

>>> m = re.search(regex, 'foo,qux')
>>> print(m)
None
```

Y como es de suponerse todo lo que hemos estado mencionando se puede combinar para buscar ciertos regex, a continuación mostramos otros ejemplos:

Regex	Matches
`^`	The start of the string
`(?P<ch>\W)`	A single non-word character, captured in a group named `ch`
`(?P<ch>\W)?`	Zero or one occurrences of the above
`foo`	The literal string `'foo'`
`(?(ch)(?P=ch)|)`	The contents of the group named `ch` if it exists, or the empty string if it doesn't
`$`	The end of the string

Ver adelante y ver atrás

Estos nos sirven para saber que debe ir antes o después de cierto regex, pero sin guardar esa parte, para comprenderlo mejor observemos.

(?=<lookahead_regex>)

<lookahead_regex> es lo que queremos que vaya después pero que no queremos guardar, a continuación vemos un ejemplo:

```
>>> re.search('foo(?=[a-z])', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
```

(?!<lookahead_regex>)
En este caso nos sirve para que lo que siga no sea lo que está en <lookahead_regex>, es decir,

```
>>> re.search('foo(?=[a-z])', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>

>>> print(re.search('foo(?![a-z])', 'foobar'))
None

>>> print(re.search('foo(?=[a-z])', 'foo123'))
None
>>> re.search('foo(?![a-z])', 'foo123')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
```

(?<=<lookbehind_regex>)
Nos sirve para ahora decirle que tiene que ir antes del regex y que guarde solo el regex, es decir.

```
>>> re.search('(?<=foo)bar', 'foobar')
<_sre.SRE_Match object; span=(3, 6), match='bar'>

>>> print(re.search('(?<=qux)bar', 'foobar'))
None
```

(?<!--<lookbehind_regex-→)
Concluimos con este,que es la negación del ver atrás, del anterior comando y ponemos un ejemplo para identificar.

```
>>> print(re.search('(?<!foo)bar', 'foobar'))
None

>>> re.search('(?<!qux)bar', 'foobar')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
```

Referencias:

https://realpython.com/regex-python/
https://www.youtube.com/watch?v=UQQsYXa1EHs