Introducción a la programación en Python

Tema 11. Expresiones regulares

Autor: Borja Rodríguez Cuenca

Contenido

1. Expresiones regulares

En teoría de lenguajes formales, las expresiones regulares son una secuencia de caracteres que conforma un patrón de búsqueda. Se utilizan principalmente para la búsqueda de patrones de cadenas de caracteres u operaciones de sustituciones.

Las expresiones regulares son patrones utilizados para encontrar una determinada combinación de caracteres dentro de una cadena de texto. Las expresiones regulares proporcionan una manera muy flexible de buscar o reconocer cadenas de texto.

Si en la línea de comandos de Windows escribimos:

dir *.exe

'*.exe" sería una expresión regular que describe todas las cadenas de caracteres que empiezan con cualquier texto seguido de ".exe", es decir, todos los archivos ejecutables .exe.

1.1. Aplicaciones

Su utilidad más inmediata es la de describir un conjunto de cadenas para una determinada función, resultando de utilidad en editores de texto y otras aplicaciones informáticas para buscar y manipular textos.

Numerosos editores de texto y otras herramientas utilizan expresiones regulares para buscar y reemplazar patrones en un texto. Por ejemplo, las herramientas proporcionadas por las distribuciones de Unix popularizaron el concepto de expresión regular entre usuarios no programadores, aunque ya era familiar entre los programadores.

Inicialmente, este reconocimiento de cadenas se programaba para cada aplicación sin mecanismo alguno inherente al lenguaje de programación, pero, con el tiempo, se ha ido incorporando el uso de expresiones regulares para facilitar programar la detección de ciertas cadenas. Por ejemplo, Perl tiene un potente motor de expresiones regulares directamente incluido en su sintaxis. Otros lenguajes lo han incorporado como funciones específicas sin incorporarlo a su sintaxis.

2. Sintaxis de las expresiones regulares

Una expresión regular (o RE, por sus siglas en inglés *regular expression*) especifica un conjunto de cadenas que coinciden con ella; las funciones de este módulo permiten comprobar si una determinada cadena coincide con una expresión regular dada.

A continuación, se explica brevemente el formato de las expresiones regulares. Las expresiones regulares pueden contener tanto caracteres especiales como ordinarios. La mayoría de los caracteres ordinarios, como 'A', 'a', o '0' son las expresiones regulares más sencillas; simplemente se ajustan a sí mismas. Se pueden concatenar caracteres ordinarios, así que last coincide con la cadena 'last'

Algunos caracteres, como '|' o '(', son especiales. Los caracteres especiales representan clases de caracteres ordinarios, o afectan a la forma en que se interpretan las expresiones regulares que los rodean.

La expresión regular más sencilla consiste en una cadena simple que describe un conjunto compuesto tan solo por esa misma cadena. En el siguiente ejemplo podemos ver como la cadena “INAP” coincide con la expresión regular “INAP” usando la función match:

```
import re
if re.match('INAP', 'INAP'):
   print ('cierto')
```

Si quisiéramos comprobar si la cadena es “INAP”, “ANAP”, “ENAP” o cualquier otra cadena que termine en “-NAP”, podríamos utilizar el punto, que es el carácter comodín en la sintaxis de las expresiones regulares:

```
import re
re.match('.NAP', 'ANAP')
re.match('.NAP', 'ENAP')
```

En las expresiones regulares existen una serie de caracteres especiales. A continuación, vamos a ver los delimitadores, que nos permiten delimitar dónde queremos buscar los patrones de búsqueda:

. (punto): en el modo predeterminado, esto coincide con cualquier carácter excepto con una nueva línea.

^ (circunflejo.): limita la búsqueda al inicio de una línea (otra función: se utiliza como elemento de negación en clases de caracteres). La secuencia \n establece el final de una línea. En el siguiente ejemplo se va a buscar el string 'Estamos' en el comienzo de la línea 'Estamos en el curso INAP':

```
import re
if re.search('^Estamos', 'Estamos en el curso INAP\n'):
    print('cierto')
```

$: limita la búsqueda al final de una línea. Similar al caso anterior, pero en este caso se va a buscar el string 'INAP' al final de la línea 'Estamos en el curso INAP':

```
import re
if re.search('INAP$', 'Estamos en el curso INAP\n'):
    print('cierto')
```

\A: Coincide solo al comienzo de la cadena de caracteres.

```
import re
if re.search('\AEstamos', 'Estamos en el curso INAP'):
    print('cierto')
```

\Z: Coincide solo al final de la cadena de caracteres:

```
import re
if re.search('INAP\Z', 'Estamos en el curso INAP'):
    print('cierto')
```

La expresión regular ".NAP" describe todas las cadenas que consistan en un carácter cualquiera (uno y solo uno) seguido de "NAP". En el caso de que necesitáramos el carácter '.' (punto) en la expresión regular, o cualquier otro de los caracteres especiales reservados para las expresiones regulares, habría que escaparlo con la barra invertida, tal como se muestra en el siguiente ejemplo, en el que se comprueba que una cadena consiste en 3 caracteres seguidos de un punto:

```
import re
if re.match('…\.', 'abc'):
    print ('match')
```

Si necesitáramos una expresión que sólo resultara cierta para las cadenas "ENAP", "INAP" y "ONAP" y ninguna otra, podríamos utilizar el carácter "|" para expresar alternativa escribiendo los tres subpatrones completos:

```
import re
if re.match('ENAP | INAP | ONAP', 'INAP'):
    print ('match')
```

O bien solo la parte que pueda cambiar, encerrada entre paréntesis formando un grupo. Los grupos son de gran importancia a la hora de trabajar con expresiones regulares:

```
import re
if re.match("(E|I|O)NAP", "INAP"):
    print ('match')
```

Otra opción es encerrar los caracteres 'E', 'I' y 'O' entre corchetes para formar una clase de caracteres:

```
import re
if re.match('[EIO]NAP', 'INAP'):
    print ('match')
```

Para comprobar si la cadena INAP es seguida de una cifra (INAP1, INAP2…) podríamos encerrar entre corchetes los valores de 0 a 9 o bien indicar un rango a través de un guión ([0-9]) de la siguiente forma:

```
import re
if re.match('INAP[0-9]', 'INAP1'):
    print ('match')
```

Los rangos y grupos en las expresiones regulares se establecen de las siguientes formas:

[ ]: Se utiliza para indicar un conjunto de caracteres. En un conjunto:

- Los caracteres pueden ser listados individualmente: [amk] coincidirá con 'a', 'm', o 'k'.
- Los rangos de caracteres se pueden indicar mediante dos caracteres y separándolos con un guion medio ('-'). Por ejemplo, [a-z] coincidirá con cualquier letra ASCII en minúscula, [0-5][0-9] coincidirá con todos los números de dos dígitos desde el 00 hasta el 59, y [0-9A-Fa-f] coincidirá con cualquier dígito hexadecimal. Si se escapa - (por ejemplo, [a\-z]) o si se coloca como el primer o el último carácter (por ejemplo, [-a] o [a-]), coincidirá con un literal '-'.
- Los caracteres especiales pierden su significado especial dentro de los sets. Por ejemplo, [(+*)] coincidirá con cualquiera de los caracteres literales '(', '+', '*', o ')'.
- Las clases de caracteres como \w o \S (definidas más adelante) también se aceptan dentro de un conjunto

- Para coincidir con un ']' literal dentro de un set, se debe preceder con una barra inversa, o colocarlo al principio del conjunto. Por ejemplo, tanto [()[\][{}] como [](){{}] coincidirá con los paréntesis, corchetes y llaves.

|: A|B, donde *A* y *B* pueden ser RE arbitrarias, crea una expresión regular que coincidirá con *A* or *B*. Un número arbitrario de RE puede ser separado por '|' de esta manera. Esto puede también ser usado dentro de grupos (ver más adelante). Cuando la cadena de destino es procesada, los RE separados por '|' son probados de izquierda a derecha. Cuando un patrón coincide completamente, esa rama es aceptada. Esto significa que una vez que *A* coincida, *B* no se comprobará más, incluso si se produce una coincidencia general más larga.

(...): Coincide con cualquier expresión regular que esté dentro de los paréntesis, e indica el comienzo y el final de un grupo. Para hacer coincidir los literales '('o')', se usa \(o\), o se envuelve dentro de una clase de caracteres: [(], [)].

En las expresiones regulares son utilizadas abreviaturas y expresiones simplificadas que agilizan el chequeo de una determinada expresión. En el último ejemplo, se podría sustituir [0-9] por \d y el resultado no se vería afectado:

```
import re
if re.match(“INAP\d”, “INAP1”):
    print ('match')
```

Las clases predefinidas son abreviaturas que facilitan la utilización de las expresiones regulares. A continuación, se muestran algunas de las clases predefinidas más comúnmente utilizadas:

d: un dígito. Equivale a [0-9]

D: cualquier carácter que no sea un dígito. Equivale a [^0-9]

s: Cualquier carácter en blanco. Equivale a [\t\n\r\f\v]

S: Cualquier carácter que no sea un espacio en blanco; equivalente a la expresión [^ \t\n\r\f\v].

w: Coincide con cualquier carácter alfanumérico. Equivale a [a-zA-Z0-9_]

\W: Cualquier carácter no alfanumérico; equivalente a [^a-zA-Z0-9_]

Con el siguiente código vamos a comprobar que el texto INAP seguido de un carácter no alfanumérico está contenido en el patrón INAP#:

```
import re
if re.match('INAP\W', 'INAP#'):
    print ('match')
```

Si el patrón tuviera varios caracteres no alfanuméricos (INAP#$%) y quisiéramos chequear una cadena de texto, podríamos repetir la expresión \W tantas veces como caracteres quisiéramos cotejar:

```
import re
if re.match('INAP\W\W\W', 'INAP#$%'):
    print ('match')
```

o podríamos hacer uso de los iteradores. Los iteradores son un tipo de metacaracteres con los que se puede especificar el número de ocurrencias del carácter previo. Estos iteradores son los caracteres especiales +, *, ? y las llaves {}.

El carácter "+" indica que lo que aparezca a la izquierda del signo, sea el elemento que sea (un simple carácter, una clase como [abc] o un subpatrón como (abc), puede encontrarse una o más veces. Por ejemplo, la expresión regular "INAP+" describirá las cadenas "INAP", "INAPP" y "INAPPP", pero no "INA", ya que debe haber al menos una P.

El carácter "*" es similar a "+", pero en este caso lo que se sitúa a la izquierda puede encontrarse cero o más veces.

El carácter '?' indica opcionalidad. Lo que tenemos a la izquierda o derecha puede o no aparecer (es decir, puede aparecer 0 o 1 veces).

Las llaves indican el número de veces exacto o el rango de veces que puede aparecer el carácter situado a la izquierda de las llaves. De esta forma:

- {m} indicará que tiene que aparecer exactamente m veces
- {m,n} indica que el elemento en cuestión debe aparecer de m a n veces
- {,m} el elemento debe aparecer entre 0 y m veces
- {m,} el elemento debe aparecer m veces o más
- {m,n}? Se busca el mínimo número de repeticiones posibles. En la cadena de 8 caracteres 'aaaaaaaa', a{3,7} coincidirá con 7 caracteres 'a' mientras que a{3,7}? Solo coincidirá con 3 caracteres.

Podríamos reemplazar el código anterior por la siguiente expresión, incluyendo los iteradores {}:

```
import re
if re.match('INAP\W{3}', 'INAP#$%'):
    print ('match')
```

Ejercicio 1: vamos a desarrollar un código que sirva para comprobar si una fecha dada cumple con el formato dd/mm/aaaa. Las comprobaciones que habría que implementar son las siguientes:

- Comprobar que los días (dd) toman valores de 1 a 31 Chequear que los meses (mm) toman valores de 1 a 12
- En cuanto a los años (aaaa), vamos a restringir la búsqueda desde 1900 a la actualidad
- Además de las anteriores consideraciones, debemos tener en cuenta que los diferentes dígitos de una fecha están separados por el carácter /.

Vamos a desarrollar el código en varios pasos. En primer lugar, debemos comprobar que el dígito de los días consta de 1 o 2 cifras y toma un valor máximo de 31. Esto se comprobará con el siguiente código:

```
if re.search(r'^(0?[1-9]|[12][0-9]|3[01])/', '31/'):
    print('Dia correcto')
```

- 0?[1-9]: La cifra puede tener un solo dígito, de 1 a 9 (y, opcionalmente, puede tener un 0 delante)
- [12][0-9]: De tener dos dígitos, puede tomar valores de 10 a 29
- 3[01]: En los valores de la treintena, la cifra de días puede tomar valores hasta 31

Se puede observar que en el código anterior se ha introducido el carácter '/' al final de los dos dígitos del día. De esta manera se delimita la cifra de días a un máximo de dos dígitos y se obliga a tener el carácter '/' entre las cifras de días, meses y años. Además, se incluye el carácter '^' al principio del código para asegurarnos de comprobar el día al comienzo de la línea de texto.

Para comprobar que el mes es correcto, el código es similar al anterior:

```
if re.search(r'(0?[1-9]|[1][012])/', '11/'):
    print('Mes correcto')
```

- 0?[1-9]: la cifra del mes puede tener un solo dígito, de 1 a 9
- [1][012]: en las decenas, el mes puede tomar valores de 10, 11 y 12

Como en el caso de los días, para los meses introducimos el carácter '/' al final del dígito del mes y en este caso no se incluye el carácter '^'. En el caso de los años, el código es algo diferente y podría ser similar a este:

```
if re.search(r'([1][9][0-9][0-9]|[2][0][01][0-9]|[2][0][2][01])$', '2021'):
    print('Año correcto')
```

- [1][9][0-9][0-9]: se comprueba que el año esté entre 1900 y 1999
- [2][0][01][0-9]: se comprueba que el año tome valores entre 2000 y 2019
- [2][0][2][01]: se comprueba que la cifra del año tome los valores 2020 y 2021

En el caso del año no se incluye el carácter '/' pero sí se incluye el símbolo $ al final del código para realizar esta comprobación con el final de la cadena. Si no incluyéramos este carácter, se considerarían como correctas los años compuestos por más de 4 dígitos que comiencen por cualquier valor entre 1900 y 2021.

Por último, vamos a concatenar las comprobaciones de días, meses y años en un solo código (prestar atención a la colocación de paréntesis y demás caracteres especiales):

```
if re.search(r'^(0?[1-9]|[12][0-9]|3[1])/(0?[1-9]|[1][012])/([1][9][0-9][0-
9]|[2][0][01][0-9]|[2][0][2][01])$', '31/12/2020'):
    print('Fecha correcta!')
else:
    print ('Fecha incorrecta :(')
```

Si la fecha tiene un formato válido (dd/mm/aaaa) y toma unos valores entre los rangos establecidos, al ejecutar el código obtendremos el mensaje "Fecha correcta!". Si por el contrario hay algún error, el mensaje será "Fecha incorrecta :(". Se propone al alumno chequear la validez del código introduciendo diferentes valores de fechas.

Ejercicio propuesto:

Como puedes comprobar, el código desarrollado no hace distinción entre meses y, así, se puede tomar como fecha correcta el día 30 de febrero (30/02/aaaa) o todos los días 31 de los meses que solo tienen 30 días. Partiendo del código anterior, se propone al alumno introducir las modificaciones necesarias para hacer distinción entre los diferentes meses, de tal forma que el valor del campo días (dd) tome valores:

- entre 1 y 31 en los meses de enero, marzo, mayo, julio, agosto, octubre y diciembre (1, 3, 5, 7, 8, 10 y 12)
- de 1 a 30 en los meses de abril, junio, septiembre y noviembre (4, 6, 9 y 11)
- de 1 a 28 en el mes de febrero (2)

3. Expresiones regulares en Python: el módulo RE

El módulo *re* cuenta con funciones para trabajar con expresiones regulares para buscar patrones dentro de una cadena (función search), para comprobar si una cadena se ajusta a un determinado criterio descrito mediante un patrón (match), dividir la cadena (Split) o para sustituir las ocurrencias del patrón por otra cadena (sub). En las próximas secciones vamos a ver en detalle todas estas funciones.

3.1. La función re.search

La función *search(pattern, string)* busca dentro de la *'cadena'* (string) la presencia del texto especificado en el *'pattern'* que satisfaga la expresión regular.

Examina a través de la string («cadena») buscando el primer lugar donde el pattern («patrón») de la expresión regular produce una coincidencia, y retorna un objeto con la coincidencia correspondiente. Retorna None si ninguna posición en la cadena coincide con el patrón.

Vamos a comprobar el funcionamiento de la función re.search con el siguiente ejemplo.

```
import re
text = u'Bienvenidos al módulo de expresiones regulares del curso
"Introducción a Python"'
pattern = r'Python'
result = re.search(pattern, text)

print (result)
```

Con este código se generan las variables "text" y "pattern", que son las entradas en la función re.search. Al ejecutar la función de búsqueda e imprimir su resultado, almacenado en la variable "result", nos informará si la búsqueda ha tenido éxito o no. Como el contenido de la variable "pattern" está contenido en "text", re.search nos informará del lugar en el que se ha producido la coincidencia:

Output: <re.Match object; span=(73, 79), match='Python'>

Podemos comprobar el resultado de la función re.search() con un operador if… else:

```
import re
text = u'Bienvenidos al modulo de expresiones regulares del curso
"Introducción a Python"'
pattern = r'Python'
result = re.search(pattern, text)
if result:
    print ('pattern localizado en el string')
else:
    print ('No se han encontrado coincidencias')
```

Output: pattern localizado en el string

Los atributos 'start' y 'end' devuelven respectivamente el índice del comienzo y final de la coincidencia entre el pattern y el text:

```
print (result.start(), result.end())
```

Output: 73 79

El atributo 'span' retorna una tupla que contiene la posición inicial y final de la coincidencia detectada por la función re.search():

```
print (result.span())
```

Output: (73, 79)

Con estos atributos podemos ofrecer información más precisa sobre una búsqueda, como se puede ver en el siguiente código:

```
import re
pattern = 'este'
text = '¿Coincide este texto con el patrón?'
match = re.search(pattern, text)
s = match.start()
e = match.end()
print('Encontrado "{}"\nen "{}"\ndesde {} hasta {}
("{}")'.format(match.re.pattern, match.string, s, e, text[s:e]))
```

Cuya salida es:

Encontrado "este"
en "¿Coincide este texto con el patrón?"
desde 10 hasta 14 ("este")

Además del *pattern* y el *string*, las funciones de la librería *re* tienen otra entrada opcional: los flags. Estos flags permiten especificar criterios en la búsqueda de patrones y, así, establecer que no haya diferencias entre mayúsculas y minúsculas, que los espacios sean ignorados…:

- re.I o re.IGNORECASE: No se hará diferencia entre mayúsculas y minúsculas

```
import re
text = u'Bienvenidos al modulo de expresiones regulares del curso
"Introducción a Python"'
pattern = r'python'
result = re.search(pattern, text, re.IGNORECASE)
if result:
    print ('pattern localizado en el string')
else:
    print ('No se han encontrado coincidencias')
```

En este caso el patrón es 'python', sin embargo, en el texto aparece la palabra 'Python'. Como se introduce el flag re.IGNORECASE, el resultado que se obtiene al ejecutar el código es 'pattern localizado en el sring'.

- re.X y re.VERBOSE: Este indicador permite escribir expresiones regulares que se ven mejor y son más legibles al facilitar la separación visual de las secciones lógicas del patrón y añadir comentarios. Los espacios en blanco dentro del patrón se ignoran, excepto cuando están en una clase de caracteres, o cuando están precedidos por una barra inversa no escapada, o dentro de fichas como *?, (?: o (?P<...>. Cuando una línea contiene un # que no está en una clase de caracteres y no está precedida por una barra inversa no escapada, se ignoran todos los caracteres desde el más a la izquierda (como #) hasta el final de la línea.
  Esto significa que los dos siguientes objetos expresión regular que coinciden con un número decimal son funcionalmente iguales:

```
import re
a = re.compile(r"""\d +  # la parte entera
                   \.    # el punto decimalt
                   \d *  # la parte decimal""", re.X)

b = re.compile(r"\d+\.\d*")
```

- re.DOTALL: Hace que el carácter especial '.' coincida con cualquier carácter, incluyendo una nueva línea. Sin este indicador, '.' coincidirá con cualquier cosa, excepto con una nueva línea.

3.2. La función re.match

La función match comprueba si una expresión regular tiene coincidencias con el comienzo de una cadena de texto. Si cero o más caracteres en el *beginning* (comienzo) de la *string* (cadena) coinciden con esta expresión regular, retorna un objeto match. Retorna None si la cadena no coincide con el patrón.

Vamos a comprobar el funcionamiento de esta función con el siguiente código:

```
import re
pattern = r"Cookie"
text = "Cookie"
if re.match(pattern, text):
    print("Match!")
else:
    print("No es un match!")
```

Dado que el patrón y el texto coinciden (ambos toman el valor *Cookie*), si ejecutamos el anterior código, obtendremos el resultado “Match”.

La ‘r’ al comienzo del patrón *Cookie* se denomina literal de cadena sin formato. Cambia cómo se interpreta el literal de cadena. Estos literales se almacenan tal como aparecen. Por ejemplo, \ es solo una barra invertida cuando tiene el prefijo r en lugar de ser interpretado como una secuencia de escape.

Veremos en el siguiente código qué ocurre cuando el *pattern* y el *text* no coinciden:

```
import re
pattern = "f"
text1 = "fresa"
text2 = "coliflor"

print("Text1 : ", re.match(pattern, text1))
print("Text 2: ", re.match(pattern,text2))
```

En este ejemplo tenemos el pattern “f” y los textos “fresa” y “coliflor”. En el primer caso el pattern coincide con el comienzo del string, por lo que existe un Match en la posición 0:

Output: Text1 : <re.Match object; span=(0, 1), match='f'>

En el segundo caso, a pesar de que el string ‘coliflor’ incluye el patrón ‘f’, este no está situado al comienzo del patrón y por lo tanto no se produce el match:

Output: None

Las funciones re.search() y re.match() pueden parecer similares, pero la principal diferencia entre ambas es que:

re.match() comprueba si hay una coincidencia sólo al principio de la cadena, mientras que *re.search()* comprueba si hay una coincidencia en cualquier parte de la cadena.

3.3. La función re.findall

Como hemos visto anteriormente, la función re.search() devuelve el primer lugar en el que el *pattern* coincide con el *string*. Sin embargo, en muchas aplicaciones nos interesará más conocer todas las coincidencias entre el pattern y el string. La función re.findall() devuelve una lista de strings que contiene todas las coincidencias entre el pattern y el string.

Vamos a conocer el funcionamiento de la función re.findall() a través del siguiente ejemplo en el que vamos a incluir caracteres propios de la sintaxis de las expresiones regulares. En este caso, vamos a identificar todos los dígitos que hay en el texto "Lunes: 1, Martes: 2, Miércoles: 3, Jueves: 4, Viernes: 5, Sábado: 6, Domingo: 7". Y esto lo vamos a hacer con el pattern '[0-9]', lo que significa que buscaremos en el texto todos los dígitos comprendidos entre 0 y 9:

```
import re
text = u'Lunes: 1, Martes: 2, Miércoles: 3, Jueves: 4, Viernes: 5, Sábado:
6, Domingo: 7'
pattern = r'[0-9]'
result = re.findall(pattern, text)

print (result)
```

La salida de este código será:

Output: ['1', '2', '3', '4', '5', '6', '7']

Es posible realizar la misma consulta utilizando la sintaxis propia de las expresiones regulares que hemos visto en la primera parte de este tema. Recordad que la expresión '\d' coincide con cualquier dígito decimal de Unicode. Si introducimos la sintaxis \d en el código:

```
import re

text = u'Lunes: 1, Martes: 2, Miércoles: 3, Jueves: 4, Viernes: 5, Sábado:
6, Domingo: 7'
pattern = r'\d'
result = re.findall(pattern, text)

print (result)
```

Este nuevo código arroja el mismo resultado que el anterior:

Output: ['1', '2', '3', '4', '5', '6', '7']

En los casos anteriores las cifras contenidas en el *text* son de un solo dígito y la salida del comando re.findall() devuelve los valores tal y como los vemos en la variable "text". Sin embargo, si hubiera

cifras con varios dígitos, la salida consideraría todos los dígitos individualmente. Con el código que se muestra a continuación, vamos a extraer los números que están asociados a cada mes del año:

```
import re
text = u'Enero: 1, Febrero: 2, Marzo: 3, Abril: 4, Mayo: 5, Junio: 6, Julio:
7, Agosto: 8, Septiembre: 9, Octubre: 10, Noviembre: 11, Diciembre:12'
pattern = r'\d'
result = re.findall(pattern, text)

print (result)
```

El resultado de ejecutar este código sería el siguiente:

Output: ['1', '2', '3', '4', '5', '6', '7', '8', '9', '1', '0', '1', '1', '1', '2']

En aquellas cifras que tienen dos dígitos, se están considerando estos como elementos individuales, ya que en los criterios de búsqueda ('pattern') no se ha hecho distinción en cuanto a la longitud de las cifras. Para establecer criterios considerando la longitud de las cadenas hay que utilizar el iterador '{}' que hemos visto previamente:

{m}

Especifica que exactamente *m* copias de la expresión regular deben coincidir; menos coincidencias hacen que la expresión regular entera no coincida. Por ejemplo, *a{6}* coincidirá exactamente con seis caracteres *'a'*, pero no con cinco.

{m,n}

Hace que el RE resultante coincida de m a n repeticiones del RE precedente, tratando de coincidir con el mayor número de repeticiones posible. Por ejemplo, a{3,5} coincidirá de 3 a 5 caracteres 'a'. Omitiendo m se especifica un límite inferior de cero, y omitiendo n se especifica un límite superior infinito. Por ejemplo, a{4,}b coincidirá con 'aaaab' o mil caracteres 'a' seguidos de una 'b', pero no 'aaab'. La coma no puede ser omitida o el modificador se confundiría con la forma descrita anteriormente.

{m,n}?

Hace que el RE resultante coincida de m a n repeticiones del RE precedente, tratando de coincidir con el mínimo de repeticiones posible. Esta es la versión non-greedy (no codiciosa) del delimitador anterior. Por ejemplo, en la cadena de 6 caracteres 'aaaaaaa', a{3,5} coincidirá con 5 caracteres 'a', mientras que a{3,5}? solo coincidirá con 3 caracteres.

Para el caso anterior, si quisiéramos considerar todos los números que constan de 1 o de 2 dígitos, debemos ejecutar el siguiente código:

```
import re
text = u'Enero: 1, Febrero: 2, Marzo: 3, Abril: 4, Mayo: 5, Junio: 6, Julio:
7, Agosto: 8, Septiembre: 9, Octubre: 10, Noviembre: 11, Diciembre:12'
pattern = r'\d{1,2}'
result = re.findall(pattern, text)

print (result)
```

De esta forma estaremos buscando las cifras de 1 y de 2 dígitos y así podemos extraer las cifras asociadas a cada mes:

Output: ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12']

3.4. La función re.split

Todas las funciones que hemos visto hasta ahora permiten determinar la existencia de un patrón en una línea de texto y su posición. Pero en ocasiones es interesante no solo conocer la existencia y posición de un patrón, sino que nos puede interesar hacer una búsqueda por un determinado campo y consultar la información que acompaña a los patrones de búsqueda. Esto es posible realizarlo con la función re.split:

re.split(pattern, string, maxsplit=0, flags=0)

Esta función divide el *string* ('cadena') por el número de ocurrencias del patrón.

Vamos a ver el funcionamiento de la función *re.split()* con el siguiente ejemplo. Dada una lista de las 10 ciudades más pobladas de España, vamos a hacer una consulta para saber qué cifras son superiores a 1 millón a través de una función *re.findall()*:

```
import re
text = u'1. Madrid: 3334730, 2. Barcelona: 1664182, 3. Valencia: 800215, 4.
Sevilla: 691395, 5. Zaragoza: 681877, 6. Málaga: 578460, 7. Murcia: 459403,
8. Palma de Mallorca, 422587, 9. Las Palmas: 381223, 10. Bilbao: 350184'

pattern = r'\d{7}'
result = re.findall(pattern, text)

print (result)
```

Las cifras superiores a 1 millón tendrán 7 dígitos. Vamos a establecer en el pattern una búsqueda de aquellos números que tengan 7 dígitos con la expresión \d{7}. La salida de esta función es:

Output: ['3334730', '1664182']

Comprobando el listado de ciudades podemos comprobar que la selección es correcta, ya que únicamente Madrid y Barcelona tienen más de un millón de habitantes. Si quisiéramos obtener el nombre de las ciudades con más de un millón de habitantes sin tener que acudir a la lista, haríamos uso de la función re.split(). Esta función corta el string '*text*' y forma con su contenido una lista, cuyos elementos están separados en aquellos elementos en los que el pattern y el text han coincidido:

```
import re
text = u'1. Madrid: 3334730, 2. Barcelona: 1664182, 3. Valencia: 800215, 4.
Sevilla: 691395, 5. Zaragoza: 681877, 6. Málaga: 578460, 7. Murcia: 459403,
8. Palma de Mallorca, 422587, 9. Las Palmas: 381223, 10. Bilbao: 350184'

pattern = r'\d{7}'

ciudades = re.split(pattern, text)

print (type(ciudades))
print (len(ciudades))
print (ciudades)
```

En este código estamos almacenando el resultado de re.split(pattern, text) en la variable ciudades. Podemos ver que esta variable se trata de un tipo lista con los siguientes tres elementos:

Output:

<class 'list'>

3

['1. Madrid: ', ', 2. Barcelona: ', ', 3. Valencia: 800215, 4. Sevilla: 691395, 5. Zaragoza: 681877, 6. Málaga: 578460, 7. Murcia: 459403, 8. Palma de Mallorca, 422587, 9. Las Palmas: 381223, 10. Bilbao: 350184']

Podemos comprobar que la lista consta de los siguientes 3 elementos:

a) '1. Madrid: '
b) ', 2. Barcelona: '
c) ', 3. Valencia: 800215, 4. Sevilla: 691395, 5. Zaragoza: 681877, 6. Málaga: 578460, 7. Murcia: 459403, 8. Palma de Mallorca, 422587, 9. Las Palmas: 381223, 10. Bilbao: 350184'

Podríamos restringir la impresión de resultados a las ciudades que satisfacen la condición de tener más de un millón de habitantes introduciendo la siguiente condición:

```
print (ciudades[0:len(result)])
```

Y así obtendremos el listado de aquellas ciudades cuya población es superior a 1 millón de habitantes:

['1. Madrid: ', ', 2. Barcelona: ']

Ejercicio 2: dado un listado de las calificaciones obtenidas por 10 personas en un examen, se pide al alumno extraer:

a) Un listado con todas las calificaciones (un número entero, una coma decimal y otro número entero). Pista: hay que utilizar un código similar al utilizado al explicar el flag re.VERBOSE)
b) Un listado con las calificaciones superiores a 5 (similar al apartado anterior, pero en este caso considerando solo aquellas calificaciones cuya parte entera está entre 5 y 10)
c) Los DNIS de los alumnos (consistentes en una cadena de 8 dígitos y una letra)

Listado de calificaciones:

```
text = "11111111A. 8,3 /n 22222222B. 3,5 /n 33333333C. 5,0 /n 44444444D.
10,0 /n 55555555E. 1,2 /n 66666666F. 4,8 /n 77777777G. 7,2 /n 88888888H. 7,1
/n 99999999I. 3,2 /n 00000000Z 4,8"
```

Ejercicio 3. Extracción de información de una guía telefónica:

Disponemos de una guía telefónica en la que, para cada persona, se conocen sus nombres y apellidos, su código postal y su número de teléfono (toda esta información es ficticia):

```
text = """Jose Manuel Caballero Fernández. CP 28005. 123 670 810\n
Ulises Amores Bastida. CP 33200. 789 879 354\n
Flor Casanovas Cortina. CP 28006. 789 132 321\n
Natividad Egea Teruel. CP 33213. 789 213 909\n
Eva Madrid Albero. CP 33301. 123 215 011\n
David Coloma Rivas. CP 28899. 123 897 678\n
Narciso Soto Sánchez. CP 28024. 123 787 987\n
María Carmen Guerrero Alcalá. CP 33000. 123 478 989\n
Xavier Larrañaga Borrell. CP 28345. 789 578 912\n
Raimundo Cuervo Moya. CP 33010. 789 123 544\n
"""
```

Se propone al alumno generar los códigos que permitan:

a) Realizar una búsqueda *re.search()* para conocer si en el listado está contenido el código postal 33204

b) Extraer todos los códigos postales de Madrid (aquellos que tienen 5 dígitos y comienzan por 28***)

c) Extraer todos los números de teléfono cuyo prefijo es ‘123’

4. Recursos

https://docs.python.org/es/3/library/re.html#search-vs-match

Python para todos, Raúl González Duque

El lenguaje de programación Python de principio a fin, Angel Pablo Hinojosa Gutiérrez