

Status	Finished
Started	Monday, 16 December 2024, 6:59 PM
Completed	Monday, 16 December 2024, 7:15 PM
Duration	15 mins 53 secs
Marks	5.75/6.00
Grade	9.58 out of 10.00 (95.83%)

Information

Expresiones regulares

Numerosos problemas de tratamiento de [ristras](#) requieren la localización en un texto de sub[ristras](#) que cumplen un determinado patrón: fechas, números de DNI, números de la seguridad social, e-mails, direcciones web, etc. La búsqueda exacta que proporcionan operaciones como el método *find()* de las [strings](#) resulta insuficiente para resolver este tipo de problemas.

Una [expresión](#) regular es una secuencia de caracteres que define un patrón de búsqueda en una [string](#). Por ejemplo, la secuencia:

```
^r..o$
```

define un patrón para una [string](#) de cuatro caracteres, empezando por la letra *r* (^r), terminando por la letra *o* (o\$) y con dos caracteres en medio (..). [Strings](#) que encajan en este patrón de búsqueda son, por ejemplo: 'ramo', 'rito', 'rato', 'ralo', ...

Python tiene un módulo llamado **re** que proporciona herramientas para trabajar con expresiones regulares:

```
import re
pattern = '^r..o$'
test_string = 'ramo'
result = re.match(pattern, test_string)
```

```
if result:
    print("Búsqueda exitosa")
else:
    print("Búsqueda fallida")
```

Nótese que las expresiones regulares, al ser secuencias de caracteres, se representan en un programa usando [valores](#) del tipo [string](#) (la [variable](#) *pattern* en el ejemplo). La [función](#) *match()* del módulo *re* devuelve un [objeto](#) de tipo *match* (del que hablaremos más adelante) si encuentra una coincidencia, o [None](#) si no la encuentra.

Information

Metacaracteres

Las expresiones regulares son una combinación de caracteres normales y metacaracteres. Estos últimos tienen un significado especial, distinto de su [valor](#) facial. En la [expresión](#):

```
^r..o$
```

los caracteres '^', '.' y '\$' son metacaracteres; '^' significa "principio de línea", '.' significa "cualquier caracter" y '\$' significa "final de línea".



Si necesitamos que un metacaracter se busque por su [valor](#) como carácter, tenemos que "escaparlo" con el carácter '\':

```
^r..o\. $
```

La [expresión](#) significa "una [string](#) que ocupa una línea, empieza por la letra 'r', seguida de dos caracteres cualesquiera y termina con la secuencia 'o.'" ("ramo.", "reto.", "ralo.", ...):



Information

Metacaracteres - clases de caracteres

El carácter '.' es un metacarácter que se usa como comodín para indicar que el patrón admite cualquier carácter (salvo fin de línea) en esa posición.

Varias secuencias de caracteres concretas, que empiezan con el carácter '\' se usan para indicar distintas clases de caracteres.

La secuencia '\d' significa "cualquier dígito decimal":

\d\d-\d\d-\d\d\d\d ● digit — digit — "-" — digit — digit — "-" — digit — digit — digit — digit ●

"19-12-1978", "22-12-2008", "41-20-9999", ...

La secuencia '\D' significa "cualquier carácter que no sea un dígito decimal":

\d\D\d ● digit — non-digit — digit ●

"1A2", "3-5", "1 5",...

La secuencia '\s' significa "un carácter espaciador", que puede ser p.e. un '\t' - tabulador, '\n' - salto de línea, '\r' - carriage return, o un ' ' - espacio.

\d\d\s\d\d ● digit — digit — white space — digit — digit ●

"12 32", "12\t21", "33\n35",...

La secuencia '\S' significa "cualquier carácter que no sea un espaciador".

La secuencia '\w' significa "un carácter de palabra" (la 'w' es de *word*). Se consideran "caracteres de palabra" las letras mayúsculas y minúsculas, los dígitos decimales, del '0' al '9', y el guión bajo o underscore '_'.

\w\w\w ● word — word — word ●

"abc", "123", "A_1",...

La secuencia '\W' significa "cualquier carácter que no sea de palabra".

Question 1

Complete

Mark 0.75 out of 1.00

Empareje cada string con la expresión regular con la que encaja (cada expresión regular debe emparejarse con una única string).

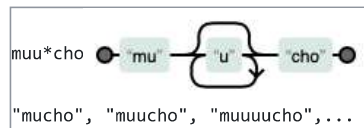
- 28-a3
- 28-ad
- 2.-a3
- 28-w3

Information

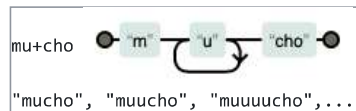
Metacaracteres - cuantificadores

Algunos metacaracteres sirven como cuantificadores, indicando repeticiones en el patrón.

El carácter asterisco '*' indica cero o más ocurrencias del patrón que le precede:



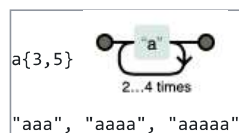
El carácter '+' indica una o más ocurrencias del patrón que le precede:



El carácter '?' indica cero o una ocurrencia del patrón que le precede:

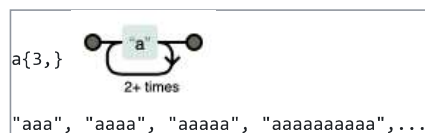


Unas llaves con dos números en medio separados por comas, '{m,n}' se usan para indicar un rango de ocurrencias del patrón que las precede; significa "como mínimo m ocurrencias y como máximo n":

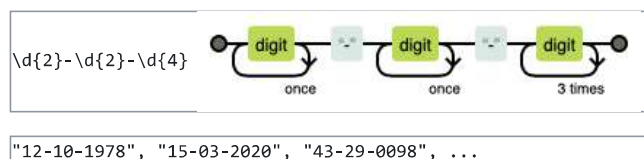


Atención a cómo se muestra en el diagrama, que especifica "2...4 times". Con ello se pretende indicar que ese camino de "vuelta atrás" se ha de recorrer, como mínimo, 2 veces y, como máximo, 4 veces, por lo que el número de veces que hemos pasado por la "a" cuando recorremos el diagrama desde su extremo izquierdo al derecho será entre 3 y 5 veces, que son los números que indicamos en la expresión regular. Por tanto, las strings mostradas son las únicas que encajan con ese patrón, ya que cualquier otra string no permitiría hacer un recorrido como se ha descrito.

Unas llaves con un número en medio seguido de una coma, '{m,}' indica un número mínimo de apariciones del patrón precedente:



Unas llaves con un número en medio, '{m}' se usa para indicar un número exacto de apariciones:



Atención de nuevo, por ejemplo "once": se ha de recorrer el camino de vuelta atrás una vez (y no más) por lo que deberán haber exactamente dos dígitos seguidos.

Question 2

Complete

Mark 1.00 out of 1.00

Marque las **strings** que encajan en el patrón:

Select one or more:

- ☐ "casa"
- ☒ "casta"
- ☒ "casiita"
- ☒ "casita"

Information

Greedy vs nongreedy

Los cuantificadores repetitivos, como '+' o '*', son "avariciosos" (greedy), esto quiere decir que intentan tomar el máximo número de repeticiones posible, lo cual puede ser problemático en ciertas circunstancias. En el siguiente ejemplo, se intenta obtener, separadas, todas las etiquetas de un documento html; una etiqueta del lenguaje html empieza con el carácter '<', seguido de cualquier número de caracteres y finaliza con el carácter '>':

```
import re
pattern = '<.*>'
test_string = '<html><head><title>Título</title></head><body>Cuerpo del documento</body></html>'
print(re.findall(pattern, test_string))
```

Aunque el patrón usado parece correcto (por ejemplo, '<html>' y '<head>' encajan aisladamente), el carácter "avaricioso" del cuantificador '*' hace que todo el texto (hasta el último '>') se considere como una única "etiqueta" html:

```
['<html><head><title>Título</title></head><body>Cuerpo del documento</body></html>']
```

Para evitarlo, en Python se puede incluir una interrogación detrás del '*', para indicarle que se extienda lo menos posible, en vez de lo más posible:

```
import re
pattern = '<.*?>'
test_string = '<html><head><title>Título</title></head><body>Cuerpo</body></html>'
print(re.findall(pattern, test_string))
```

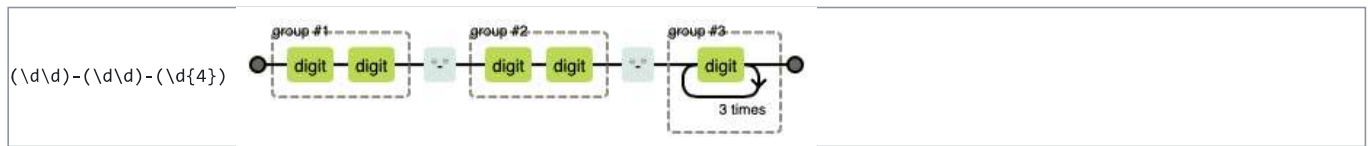
Ahora se obtiene:

```
['<html>', '<head>', '<title>', '</title>', '</head>', '<body>', '</body>', '</html>']
```

Information

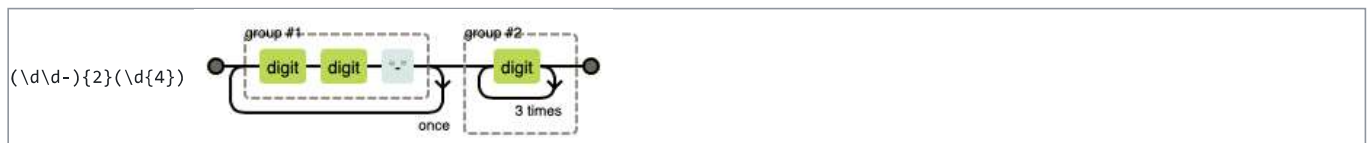
Metacaracteres - grupos

Se pueden agrupar porciones de una [expresión](#) regular usando [paréntesis](#):



El agrupamiento tiene varias ventajas. La primera es que permite identificar subexpresiones (grupos) a las que se puede acceder por separado en las operaciones de tratamiento de [strings](#) que se usen. Suponiendo que el ejemplo anterior se usara para localizar posibles fechas *día-mes-año* en un texto, al usar paréntesis el día queda identificado como grupo 1, el mes como grupo 2 y el año como grupo 3; la fecha [completa](#) sería el grupo 0.

El agrupamiento también permite compactar las expresiones. Las [ocurrencias](#) encontradas por el patrón del ejemplo anterior también pueden ser encontradas por el siguiente patrón (dicho de otra manera, ambos patrones el del ejemplo anterior y el que se muestra a continuación son [equivalentes](#)):



El agrupamiento también permite delimitar un grupo de alternativas. Se usa la barra vertical '|' para separar dos (o más) trozos del patrón indicando así que son alternativos entre sí; de hecho, si no existiese la posibilidad de agrupar, sería difícil o engorroso expresar determinados patrones de alternativas. Obsérvense los dos siguientes patrones:



Al no usarse agrupamiento en este último caso, todo lo que está a la izquierda de la barra es alternativa de lo que está a la derecha. Esto es debido a que el "operador invisible" de [concatenación](#), en este caso "[operando](#)" sobre los caracteres que conforman el subpatrón 'niño' tiene precedencia sobre el operador, este sí expreso, '|', de hecho este operador tiene menor **precedencia** también que '*' y '+', los cuales a su vez tienen mayor precedencia que la [concatenación](#); así pues, como en las expresiones aritméticas, los paréntesis nos sirven para forzar una interpretación distinta de la determinada por las precedencias.

Para encontrar la misma [string](#) que con el primer patrón, sin usar agrupamiento, habría que usar un patrón más largo:

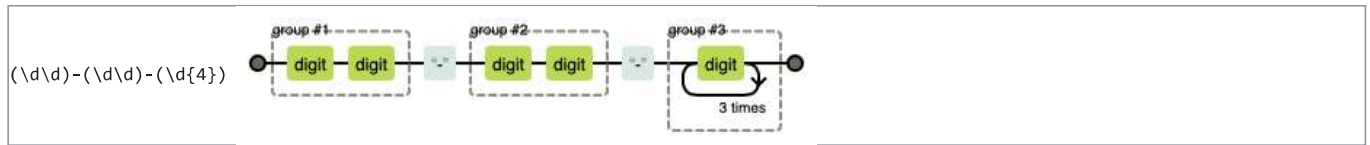


Podemos entender este otro patrón como similar al resultado de "quitar paréntesis" en el caso de una [expresión](#) aritmética donde en lugar del '|' tuviéramos un operador + y donde la multiplicación, que correspondería con la [concatenación](#), es implícita, sin expresar el operador, como muchas veces hacemos en matemáticas: p.e. $abc(x+y) = abcx+abcy$.

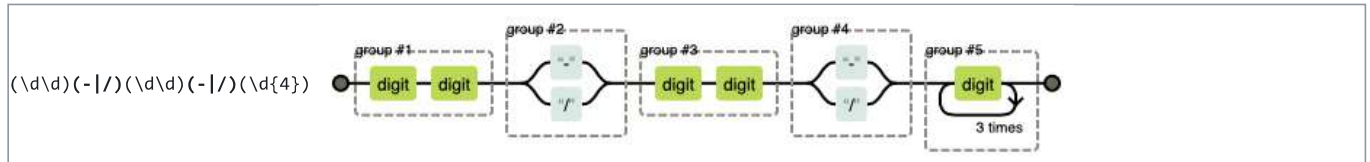
Information

Metacaracteres - backreferences

Otra ventaja de los agrupamientos, es que permiten hacer "backreferences". Supongamos que queremos usar el patrón del primer ejemplo para buscar posibles fechas en un texto:

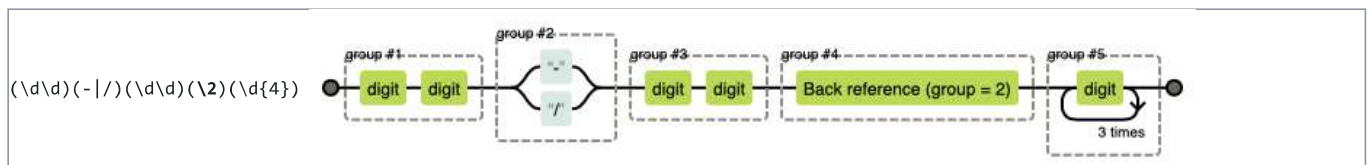


Este patrón permite encontrar fechas que usen guiones como separadores ("30-06-1979"), pero no fechas que usen barras ("30/06/1970"). Podemos usar alternativas para contemplar ambos separadores:



Este patrón acepta como separadores guiones o barras, pero permite mezclarlos en una misma fecha ("30-06/1970", "30/06-1970"), lo que normalmente no consideraríamos correcto.

Una "backreference" consiste en insertar en una [expresión](#) regular una [referencia](#) a un grupo anterior, usando su número precedido de una barra invertida '`\`' (*backslash*):



En este caso, la especificación para el separador de la derecha hace [referencia](#) con `\2` a que debe ser igual a la [sub-string concreta](#) que encajó con el grupo 2, esto es, si el separador de la izquierda es un guión, el otro deberá serlo también, y si es una barra, el otro deberá ser también una barra.

```
import re
pattern = '(\d\d)(-|/)(\d\d)(\2)(\d{4})'
test_string = '30/06/1970'
result = re.match(pattern, test_string)
```

```
if result:
    print("Búsqueda exitosa")
else:
    print("Búsqueda fallida")
```

Obsérvese que hemos tenido que [duplicar](#) la barra antes del 2, de lo contrario, con una sola barra, '`\2`' sería tomado como un carácter expresado mediante una secuencia de escape!

Information

La barra para metacaracteres de expresión regular vs secuencias de escape de string

En el siguiente trozo de código:

```
import re
pattern = '(\d\d)(-|/)(\d\d)(\2)(\d{4})'
test_string = '30/06/1970'
result = re.match(pattern, test_string)
```

```
if result:
    print("Búsqueda exitosa")
else:
    print("Búsqueda fallida")
```

Podemos observar que, aunque la expresión regular que representa lo que queremos buscar es:

```
'(\d\d)(-|/)(\d\d)(\2)(\d{4})'
```

A la variable *pattern*, que representa en el código el patrón de búsqueda, le hemos asignado:

```
'(\d\d)(-|/)(\d\d)(\2)(\d{4})'
```

Nótese, que hemos **duplicado** la barra que está antes del dos en la backreference.

La razón es que, mientras que en una expresión regular la barra se usa para representar ciertos metacaracteres, la misma barra se usa en los literales normales de string para expresar ciertas secuencias de escape. Así, en una expresión regular '\2' es una referencia al grupo 2 previo pero '\n' es una secuencia de dos caracteres (barra seguido de 'n'). Pero al usar un literal normal de string para emplearlo como expresión regular tenemos el problema de que '\2' será tomado por una secuencia de escape que representa el carácter cuyo código en octal es 2 y '\n' como el carácter de nueva línea (*newline*). Para solucionarlo, en estos casos concretos de posible interpretación como secuencia de escape deberemos duplicar la barra, ya que en un literal de string '\\' se toma como una sola barra. En las secuencias de metacaracteres que no coincidan con secuencias de escape no hace falta duplicar la barra.

Una alternativa es indicar que el literal de string es una **raw string**, en la que los caracteres se toman tal cual (aunque por tanto si lo necesitásemos no podríamos expresar caracteres especiales mediante secuencias de escape) anteponiendo una letra **r** a las comillas (simples o dobles):

```
import re
pattern = r'(\d\d)(-|/)(\d\d)(\2)(\d{4})'
test_string = '30/06/1970'
result = re.match(pattern, test_string)
```

```
if result:
    print("Búsqueda exitosa")
else:
    print("Búsqueda fallida")
```

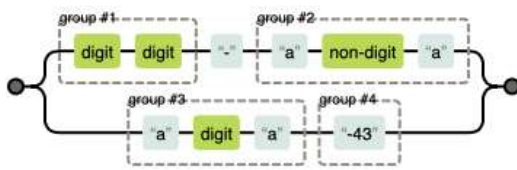
Por lo demás, la string resultante es del mismo tipo de datos *str* que el de un literal normal, y podemos concatenar combinando ambas modalidades, a conveniencia.

Question 3

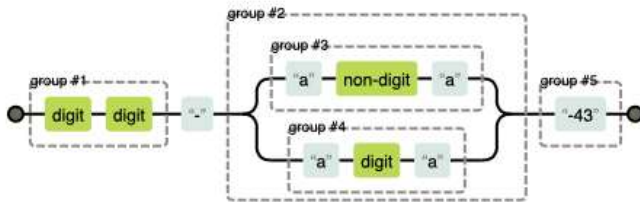
Complete

Mark 1.00 out of 1.00

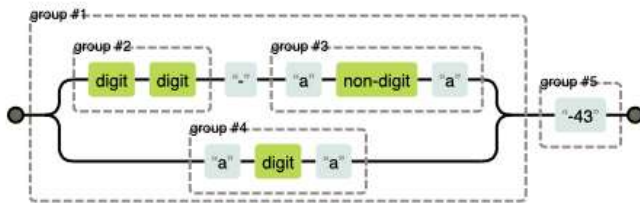
Empareje cada gráfico con la [expresión](#) regular que representa.



$(\backslash d\backslash d) - (a\backslash Da) (a\backslash da) (-43)$



$(\backslash d\backslash d) - ((a\backslash Da) (a\backslash da)) (-43)$

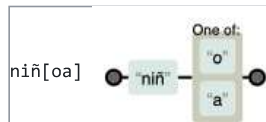


$((\backslash d\backslash d) - (a\backslash Da) (a\backslash da)) (-43)$

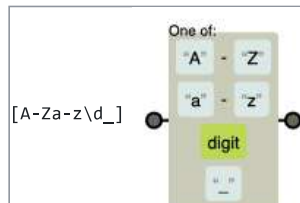
Information

Metacaracteres - conjuntos y rangos de caracteres

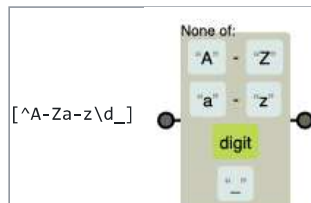
Los corchetes se pueden usar en una [expresión](#) regular para expresar "cualquier carácter de un conjunto". Por ejemplo, el siguiente patrón permite que el último carácter sea una 'a' o una 'o':



Se puede usar el guión '-' entre dos caracteres dentro de los corchetes para indicar un rango. El siguiente patrón admite una letra mayúscula (rango A-Z), una letra minúscula (rango a-z), un dígito, o un guión bajo:



Un carácter '^' (acento circunflejo) como [primer carácter](#) entre los corchetes sirve para negar su contenido (conjunto complementario), es decir indica "cualquier carácter que no esté en el conjunto indicado a continuación". Nótese la [expresión](#) "None of" en la parte superior de la imagen adjunta al siguiente patrón:



Question 4

Complete

Mark 1.00 out of 1.00

Marque las [strings](#) que encajan con el patrón:

Select one or more:


- ☒ "aaaaa"
- ☒ "hasta"
- ☐ "zapeo"
- ☒ "papeo"

Information

Metacaracteres - delimitadores


Algunos metacaracteres sirven como delimitadores, esto es, no representan carácter alguno por sí mismos, sino que expresan requisito que debe cumplir la [string](#) en ese punto del patrón.

El acento circunflejo '^', cuando no está dentro de corchetes, indica que ese punto del patrón debe corresponder con [principio de línea](#). Dualmente, el signo de dólar '\$' indica [fin de línea](#). El siguiente patrón abarca una línea completa, de principio a fin:


`^r..o$` 

También se puede usar la secuencia '\A' que indica [principio de string](#). Nótese que esto es distinto de principio de línea sólo en el caso de [strings](#) multilínea.

La secuencia '\b' indica [extremo de "palabra"](#) (ésta en el sentido de '\w' visto anteriormente). En los siguientes ejemplos se muestran en azul el tramo de la [string](#) en cuestión que encaja con el patrón de la izquierda conteniendo '\b' o su opuesto '\B':

`\bser`  p.e. en "al **ser**vir"

`ser\b`  p.e. en "al cos**er**"

`\bser\b`  p.e. en "el **ser** o la nada"

La secuencia '\B', por el contrario, indica que ese punto no debe ser extremo de palabra:

`\Bcos`  p.e. en "des**cos**er", "cas**cos**"

`cos\b`  p.e. en "des**cos**er", "**cos**er"

`\Bcos\b`  p.e. en "des**cos**er"

Question 5

Complete

Mark 1.00 out of 1.00

Empareje cada flag del módulo re de Python con su descripción.

DOTALL	Hace que el metacarácter '.' incluya los caracteres de salto de línea
MULTILINE	Hace que los metacaracteres ^ y \$ coincidan, respectivamente, con el comienzo y fin de cada línea en una string multilinea
LOCALE	Tiene en cuenta las especificidades del idioma local
VERBOSE	Facilita usar expresiones más legibles, permitiendo usar espacios para darles formato e introducir comentarios para explicarlas
IGNORECASE	Se busca sin diferenciar entre mayúsculas y minúsculas
ASCII	Hace que las secuencias de escape \w, \b, \s y \d busquen solo caracteres ASCII

Question 6

Complete

Mark 1.00 out of 1.00

Marque las strings que encajan en el patrón:

Select one or more:

- ☐ "novela"
- ☒ "casino"
- ☒ "penoso"