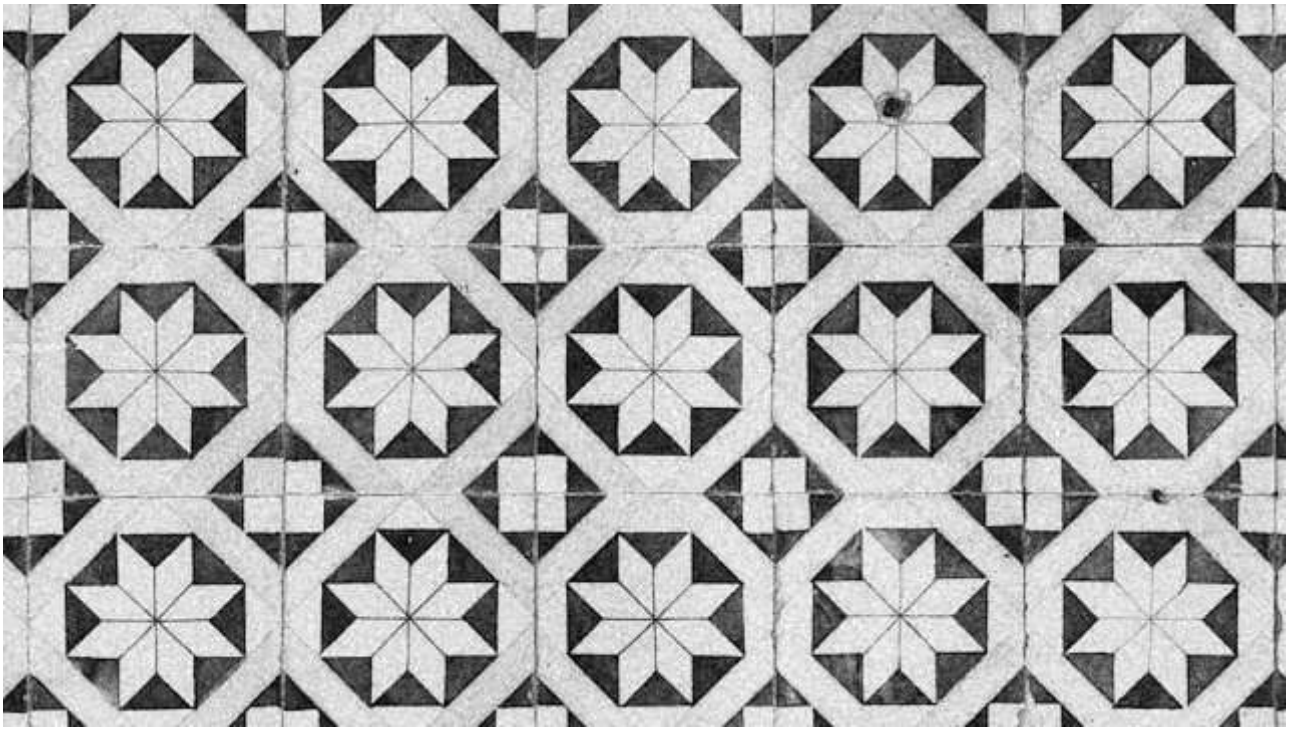
re



El módulo re permite trabajar con expresiones regulares. [1]

> **Consejo**
>
> Si tienes un problema y lo intentas resolver con expresiones regulares, entonces tienes dos problemas 😅 – Anónimo (Nótese la ironía)

¿Qué es una expresión regular?

Una **expresión regular** (también conocida como **regex** o **regexp** por su contracción anglosajona **reg**-ular **exp**-ression) es una cadena de texto que conforma un **patrón de búsqueda**. Se utilizan principalmente para la *búsqueda de patrones* en cadenas de caracteres u *operaciones de sustituciones*.

Se trata de una herramienta ampliamente utilizada en las ciencias de la computación y necesaria para multitud de aplicaciones que traten con información textual.

Pero... ¿qué pinta tiene una expresión regular?

```
>>> regex = '^\d{8}[A-Z]$'
```

La expresión regular anterior nos permite comprobar que una cadena de texto dada es un DNI válido. Si analizamos parte por parte tendríamos lo siguiente:

- `^` comienzo de línea.
- `\d{8}` dígito que se repite 8 veces.
- `[A-Z]` letra en mayúsculas.
- `$` final de línea.

Sintaxis

Las expresiones regulares pueden contener tanto **caracteres especiales** como caracteres ordinarios. La mayoría de los caracteres ordinarios, como `'A'`, `'b'`, o `'0'` son las expresiones regulares más sencillas; simplemente se ajustan a sí mismas.

Caracteres especiales

Existen una serie de caracteres que tienen un significado especial dentro de una expresión regular:

Caracter	Descripción
`.`	Coincide con cualquier carácter excepto con una nueva línea
`^`	Coincide con el comienzo de la cadena/línea
`$`	Coincide con el final de la cadena/línea
`*`	Coincide con 0 o más repeticiones de la expresión regular precedente
`+`	Coincide con 1 o más repeticiones de la expresión regular precedente
`?`	Coincide con 0 o 1 repetición de la expresión regular precedente
`{m}`	Coincide con exactamente `m` copias de la expresión regular precedente
`{m,n}`	Coincide de `m` a `n` copias de la expresión regular precedente, tratando de coincidir con el mayor número de repeticiones posibles. Omitiendo `m` se especifica un límite inferior de cero, y omitiendo `n` se especifica un límite superior infinito
`\`	Permite "escapar" el caracter que le sigue. Es decir, quitarle el significado especial que tiene
`[]`	Coincide con el conjunto de caracteres indicados dentro de los corchetes. Existe la variante `[a-f]` que coincide con el rango de caracteres entre la `a` y la `f`. Y también existe la variante `[^abc]` que coincide con todos los caracteres salvo los indicados dentro del conjunto.
`|`	Coincide con una expresión regular u otra, separadas por este símbolo
`(...)`	Coincide con cualquier expresión regular que esté dentro de los paréntesis, e indica el comienzo y el final de un grupo; el contenido de un grupo puede ser recuperado después de que se haya realizado una coincidencia

Caracter	Descripción
`(?P<name>...)`	Coincide con cualquier expresión regular que esté dentro de los paréntesis; el contenido del grupo es accesible por `name`

`\b`	Coincide con el comienzo o el final de una palabra
`\B`	Coincide con cualquier caracter que no sea cmienzo o final de una palabra
`\d`	Coincide con cualquier dígito decimal. Equivalente a `[0-9]`
`\D`	Coincide con cualquier carácter que no sea un dígito decimal. Equivalente a `[^0-9]`
`\s`	Coincide con cualquier espacio en blanco. Equivalente a `[ \t\n\r\f\v]`
`\S`	Coincide con cualquier carácter que no sea un espacio en blanco. Equivalente a `[^ \t\n\r\f\v]`
`\w`	Coincide con cualquier carácter alfanumérico. Equivalente a `[a-zA-Z0-9_]`
`\W`	Coincide con cualquier carácter que no sea un carácter alfanumérico. Equivalente a `[^a-zA-Z0-9_]`

Expresiones en crudo

Cuando definimos una expresión regular es conveniente utilizar el formato raw en las cadenas de texto para que los caracteres especiales no pierdan su semántica.

Veamos un ejemplo con el tabulador:

```
>>> regex = '\t[abc]$'
>>> print(regex)
    [abc]$

>>> regex = r'\t[abc]$'  # formato crudo
>>> print(regex)
\t[abc]$
```

Operaciones

Buscar

La búsqueda de patrones es una de las principales utilidades de las expresiones regulares.

Supongamos que queremos buscar un número de teléfono dentro de un texto. Para ello vamos a utilizar la función search():

```
>>> import re

>>> text = 'Estaré disponible en el +34755142009 el lunes por la tarde'

>>> regex = r'\+?\d{2}\d{9}'
>>> re.search(regex, text)
<re.Match object; span=(24, 36), match='+34755142009'>
```

Esta función devuelve un objeto de tipo Match en cuya representación podemos ver un campo `span` que nos indica el alcance de la concidencia:

```
>>> text[24:36]
'+34755142009'
```

En el ejemplo anterior estamos buscando un solo elemento. Imaginemos un caso en el que queremos buscar todas las cantidades de dinero que aparecen en un determinado texto. Para ello vamos a utilizar la función findall():

```
>>> text = 'El coste ascendió a 36€ más un 12% de impuestos para un total de 40€'

>>> re.findall(r'\d+€', text)
['36€', '40€']
```

Si utilizamos un **grupo de captura** (paréntesis) la función `findall()` sólo nos devolverá aquellas coincidencias del grupo de captura:

```
>>> re.findall(r'(\d+)€', text)
['36', '40']
```

En el caso de que queramos agrupar expresiones regulares con `findall()` sin que se capturen estos grupos debemos utilizar la sintaxis: `(?:...)`

> **Atención**
>
> La función `findall()` no devuelve un objeto `Match` sino que retorna una lista con las cadenas de texto coincidentes.

COINCIDENCIA

El tipo de objeto Match es el utilizado en este módulo para representar una coincidencia.

Retomando el ejemplo anterior de la búsqueda del teléfono, veamos qué podemos hacer con este tipo de objetos:

```
>>> text = 'Estaré disponible en el +34755142009 el lunes por la tarde'

>>> regex = r'\+?\d{2}\d{9}'
>>> m = re.search(regex, text)
```

```
>>> m
<re.Match object; span=(24, 36), match='+34755142009'>
```

Si queremos acceder al texto completo coincidente, tenemos dos alternativas equivalentes:

```
>>> m[0]
'+34755142009'

>>> m.group(0)
'+34755142009'
```

Podemos conocer dónde empieza y dónde acaba el texto coincidente de la siguiente manera:

```
>>> m.span()  # equivale a m.span(0)
(24, 36)
```

Incluso hay una manera de acceder a estos índices por separado:

```
>>> m.start()
24

>>> m.end()
36
```

Si hubiera algún **subgrupo de búsqueda** podríamos acceder con los índices subsiguientes. Para ejemplificar este comportamiento vamos a modificar ligeramente la expresión regular original y capturar también el prefijo y el propio número de teléfono:

```
>>> m = re.search(r'\+?(\d{2})(\d{9})', text)
```

Ahora podemos acceder a los grupos capturados de distintas maneras:

```
>>> m.groups()
('34', '755142009')

>>> m[0]
'+34755142009'
>>> m[1]
'34'
>>> m[2]
'755142009'

>>> m.group()  # equivale a m.group(0)
'+34755142009'
>>> m.group(1)
'34'
>>> m.group(2)
'755142009'
```

Igualmente podemos acceder a los índices de comienzo y fin de cada grupo capturado:

```
>>> m.span(0)  # equivale a m.span()
(24, 36)

>>> m.span(1)  # '34'
(25, 27)

>>> m.span(2)  # '755142009'
(27, 36)
```

Por tanto, se cumple lo siguiente:

```
>>> for group_id in range(len(m.groups()) + 1):
...     start, end = m.span(group_id)
...     print(text[start:end])
...
+34755142009
34
755142009
```

Ahora vamos a **añadir nombres** a los **grupos de captura** para poder explicar otras funcionalidades de este objeto `Match`:

```
>>> regex = r'\+?(?P<prefix>\d{2})(?P<number>\d{9})'
>>> m = re.search(regex, text)
```

Tras este código, todo lo anterior sigue funcionando igual:

```
>>> m.groups()
('34', '755142009')

>>> m[1]
'34'

>>> m[2]
'755142009'
```

La diferencia está en que ahora podemos acceder a los grupos de captura por su nombre:

```
>>> m.group('prefix')
'34'
>>> m['prefix']
'34'

>>> m.group('number')
'755142009'
```

```
>>> m['number']
'755142009'
```

Y también existe la posibilidad de obtener el diccionario completo con los grupos capturados:

```
>>> m.groupdict()
{'prefix': '34', 'number': '755142009'}
```

IGNORAR MAYÚSCULAS Y MINÚSCULAS

Supongamos que debemos encontrar todas las vocales que hay en un determinado nombre. La primera aproximación sería la siguiente:

```
>>> name = 'Alan Turing'
>>> regex = r'[aeiou]'

>>> re.findall(regex, name)
['a', 'u', 'i']
```

Aparentemente está bien pero nos damos cuenta de que la primera `A` mayúscula no está entre los resultados.

Este módulo de expresiones regulares establece una serie de «flags» que podemos pasar a las distintas funciones para modificar su comportamiento. Uno de los más importantes es el que nos permite ignorar mayúsculas y minúsculas: `re.IGNORECASE`.

Veamos su aplicación con el ejemplo anterior:

```
>>> re.findall(regex, name, re.IGNORECASE)
['A', 'a', 'u', 'i']
```

Podemos «abreviar» esta constante de la siguiente manera:

```
>>> re.findall(regex, name, re.I)
['A', 'a', 'u', 'i']
```

Separar

Otras de las operaciones más usadas con expresiones regulares es la separación o división de una cadena de texto mediante un separador.

En su momento vimos el uso de la función split() para cadenas de texto, pero era muy limitada al especificar patrones avanzados. Veamos el uso de la función `re.split()` dentro de este módulo de expresiones regulares.

Un ejemplo muy sencillo sería **separar la parte entera de la parte decimal** en un determinado número flotante:

```
>>> regex = r'[.,]'

>>> re.split(regex, '3.14')
['3', '14']

>>> re.split(regex, '3,14')
['3', '14']
```

Vemos que la función devuelve una lista con los distintos elementos separados.

> **Prudencia**
>
> Aunque parezca muy sencillo, este ejemplo no se puede resolver de manera «directa» usando la función `split()` de cadenas de texto.

Reemplazar

Este módulo de expresions regulares también nos ofrece la posibilidad de reemplazar ocurrencias dentro de un texto.

A vueltas con el ejemplo del nombre de una persona, supongamos que recibimos la información en formato `<nombre> <apellidos>` y que la necesitamos en formato `<apellidos>, <nombre>`. Veamos cómo resolver este problema con la operación de reemplazar:

```
>>> name = 'Alan Turing'

>>> regex = r'(\w+) +(\w+)'

>>> repl = r'\2, \1'

>>> re.sub(regex, repl, name)
'Turing, Alan'
```

Hemos utilizado la función `re.sub()` que recibe 3 parámetros:

1. La expresión regular a localizar.
2. La expresión de reemplazo.
3. La cadena de texto sobre la que trabajar.

Dado que hemos utilizado *grupos de captura* podemos hacer referencia a ellos a través de sus índices mediante `\1`, `\2` y así sucesivamente.

Al igual que veíamos previamente, existe la posibilidad de nombrar los grupos de captura, y así facilitar la escritura de las expresiones de reemplazo:

```
>>> name = 'Alan Turing'

>>> regex = r'(?P<name>\w+) +(?P<surname>\w+)'
```

```
>>> repl = r'\g<surname>, \g<name>'

>>> re.sub(regex, repl, name)
'Turing, Alan'
```

Esta función admite un uso más avanzado ya que podemos **pasar una función** en vez de una cadena de texto de reemplazo, lo que nos abre un mayor rango de posibilidades.

Siguiendo con el caso anterior, supongamos que queremos hacer la misma transformación pero convirtiendo el apellido a mayúsculas, y asegurarnos de que el nombre queda como título:

```
>>> name = 'Alan Turing'

>>> regex = r'(\w+) +(\w+)'

>>> re.sub(regex, lambda m: f'{m[2].upper()}, {m[1].title()}', name)
'TURING, Alan'
```

> **Ver también**
>
> Existe una función `re.subn()` que devuelve una tupla con la nueva cadena de texto reemplazada y el número de sustituciones realizadas.

Casar

Si lo que estamos buscando es ver si una determinada cadena de texto «casa» (coincide) con un patrón de expresión regular, podemos hacer uso de la función `re.fullmatch()`.

Veamos un ejemplo en el que comprobamos si un texto dado es un DNI válido:

```
>>> regex = r'\d{8}[A-Z]'

>>> text = '54632178Y'

>>> re.fullmatch(regex, text)  # devuelve un objeto Match
<re.Match object; span=(0, 9), match='54632178Y'>
```

Si el patrón no casa la función devuelve `None`:

```
>>> text = '87896532$'

>>> re.fullmatch(regex, text)  # devuelve None

>>> re.fullmatch(regex, text) is None
True
```

Todo esto lo podemos poner dentro una sentencia condicional haciendo uso además del operador morsa para aprovechar la variable creada:

```
>>> def check_id_card(id_card: text) -> None:
...     if m := re.fullmatch(regex, id_card):
...         print(f'{text} es un DNI válido')
...         print(m.span())
...     else:
...         print(f'{text} no es un DNI válido')
...

>>> check_id_card('54632178Y')
54632178Y es un DNI válido
(0, 9)

>>> check_id_card('87896532$')
87896532$ no es un DNI válido
```

Hay una **variante más «flexible»** para casar que es `re.match()` y comprueba la existencia del patrón **sólo desde el comienzo de la cadena**. *Es decir, que si el final de la cadena no coincide sigue casando.*

Continuando con el caso anterior de comprobación de los DNI, podemos ver que añadir caracteres al final del documento de identidad no modifica el comportamiento de `re.match()`:

```
>>> regex = r'\d{8}[A-Z]'
>>> text = '54632178Y###'

>>> re.match(regex, text)
<re.Match object; span=(0, 9), match='54632178Y'>
```

Sin embargo no sucede lo mismo si añadimos caracteres al principio de la cadena:

```
>>> regex = r'\d{8}[A-Z]'
>>> text = '&&&54632178Y###'

>>> re.match(regex, text)
# None!
```

En cualquier caso podemos hacer que `re.match()` se comporte como `re.fullmatch()` si especificamos los **indicadores de comienzo y final de línea** en el patrón:

```
>>> regex = r'^\d{8}[A-Z]$'
>>> text = '54632178Y'

>>> re.match(regex, text)
<re.Match object; span=(0, 9), match='54632178Y'>
```

> **Truco**
>
> Tanto `re.fullmatch()` como `re.match()` devuelven un objeto de tipo Match con lo que podemos hacer uso de todos sus métodos y atributos.

Compilar

Si vamos a utilizar una expresión regular una única vez entonces no debemos preocuparnos por cuestiones de rendimiento. Pero si repetimos su aplicación, sería más recomendable **compilar** la expresión regular a un patrón para mejorar el rendimiento:

```
>>> regex = r'\d+'

>>> r = re.compile(regex)

>>> type(r)
re.Pattern

>>> re.search(r, '1:abc;10:def;100;ghi')
<re.Match object; span=(0, 1), match='1'>
```