EXPRESSÕES REGULARES

Expressões regulares é basicamente estabelecer padrões onde strings devem seguir, ou mesmo padrões de digitações que o usuário deve seguir para poder fornecer uma informação, servindo também como uma espécie de validação de informações, além de servir para substituir palavras/padrões dentro de uma string

Regular Expressions

Dentro do conceito desse biblioteca temos alguns termos que precisamos ficar atentos como:

No exemplo ao lado, criamos uma variável chamada padrão que recebe como objeto a palavra “ra”, ou seja, dentro de um texto ele buscaria qualquer correspondência que atenda ao objeto padrão

```
import re

padrao = re.compile(r'ra')
```

💡 A letra `r` passada antes de apontar o parâmetro está anulando qualquer código ASCII que possa ser usado dentro de uma string, uma forma de evitar possíveis falhas dos programa (No caso de uma string podemos passar comando como `\n ou \r` para o programa realizar certas ações, porém quando introduzimos a letra r antes da string, esses comando não são mais validos)

METACARACTERES

`.` → Dentro dessa biblioteca o ponto quando é **passado como parâmetro quer dizer que no local onde ele se encontra, pode ser qualquer coisa**

```
texto = 'Arara\n Abrara\n Brasil'

padrao = re.compile(r'A...a')
verificar = padrao.findall(texto)
```

Dentro desse exemplo, será retornado uma lista com **todas as palavras que começam com A e terminam com a com três possíveis letras preenchendo os espaços** representados pelo pontos **não podendo ser por exemplo uma palavra com mais de 5 letras**

```
texto = 'Eu amo Python'
padrao = re.compile(r'.')
verificar = padrao.findall(texto)
```

Nesse outro exemplo me **retorna uma lista com todas as letras presentes dentro da variável texto onde cada letra representa um item da lista,** onde eu basicamente faço uma separação/quebra de todos os caracteres da frase separadamente, parecido com escrever por extenso

---

`^` → Esse símbolo é usado para **verificar se o inicio de uma linha tem a correspondência passada como parâmetro,** podendo por exemplo verificar se um determinado input começar com letras ou alguma coisa do tipo

```
texto = 'Python é bom de programar'
texto2 = 'Programar em Python é muito bom'
#-----------------------------------------
padrao = re.compile(r'^Python')

verifica = padrao.findall(texto)
verifica = padrao.findall(texto2)

# Saida1 = True
# Saida2 = False
```

Devido ao caractere ^ antes da palavra Python, **estou forçando a busca no inicio da string, e só me retorna True caso o inicio da string começar com o definido na variável padrão,** no caso me retornaria uma lista com um item `['Python']`

Caso ambas começassem com Python o retorno seria

`['Python', 'Python']`

💡 Em caso de um texto com várias linhas seria acrescentado o seguinte parâmetro

`resultado = re.findall(r"^Python", texto, re.M)` onde o `re.M` indica para o método buscar em todas as linhas se começa com o padrão

---

`[^]` → Essa sequência de caracteres define uma **negação da expressão**, retornar a expressão eliminando a negação

- Podemos usar uma variável do tipo bool para poder verificar e validar um input

```
texto = 'Ola Mundo prazer'

p = re.compile(r'[^" "]')

verifica = p.findall(texto)
```

💡 Um dos pontos mais importantes é usar para poder negar uma expressão (Que pode ser usado como forma de validação da expressão

---

`$` → Quando colocado esse sinal ao final de uma frase, estamos buscando o padrão que antecipa esse sinal

Nesse caso estamos procurando pela palavra Python no final de uma string ou frase

```
padrao = re.compile(r'Python$')
```

💡 Quando combinado o metacaractere de `^` (Precisa começar com) e o `$` (Precisa terminar com) indicia especificamente que o valor final tem que ser exatamente o padrão

---

`\` → Dentro da biblioteca de REGEX, a barra anula qualquer outro símbolo usado dentro de alguma expressão
Ex:
`['Ar...']` aqui estamos procurando uma correspondência com 5 letras onde comece com Ar e tem mais três outros caracteres (o ponto assume o lugar de qualquer outro), porém quando passamos a `\` fica `['Arara\.']` nesse caso, estamos procurando a correspondência exata de `Arara.`, com o ponto representando um caractere

```
padrao = re.compile(r'arara\.')

texto = 'Eu amo arara.'
```

Nesse caso a saída é `true`, pois foi encontrado a correspondência exata

💡 Não servindo somente para o `.` mais sim para qualquer outro símbolo que precise de procurar

---

```
texto = "Brasil 1954"

p = re.compile(r'\d')
p2 = re.compile(r'\D')

verifica = p.findall(texto)
verifica2 = p.findall(texto)

print(verifica, '\n', verifica2)
```

`\d` → Esse conjunto de caractere retorna **qualquer número ou algarismo de 0 a 9** presente dentro da frase

`\D` → Por outro lado quando é desse modo, é retornado **qualquer coisa menos números presentes dentro da frase ou texto (Mostra também caracteres de formatação e tabulação)**

o primeiro retorna um lista com todos os números encontrados dentro da frase, já o segundo retorna uma lista retirando os números da frase

---

`\s` → Retornar qualquer caractere vazio ou relacionado a formatação de um texto presente dentro da frase

`\S` → Faz o oposto do anterior, retornando todos os caracteres não vazios presentes dentro da frase

No primeira exemplo será retornado toda quebra de linha, todo espaço vazio presente dentro da frase, já no segundo exemplo será retornado todos os caracteres não vazio presentes dentro da frase

▼ Possíveis retornos do `\s`

Caractere Capturado	Representação	Nome	Descr
Espaço comum	`' '`	Espaço	Um es simple para s palavr
Tabulação horizontal	`'\\t'`	Tabulação horizontal	Cria u espaç horizo maior um es comu
Nova linha	`'\\n'`	Linha nova	Indica de um mover cursor linha s
Retorno de carro	`'\\r'`	Retorno de carro	Move para o linha a (usad sistem antigo combi com
Tabulação vertical	`'\\v'`	Tabulação vertical	Move para a posiçã vertic
Form feed	`'\\f'`	Avanço de formulário	Indica de um págin sistem impres antigo

```
import re

texto = '''

arara 1992

'''
t = re.compile(r'\s')
q = re.compile(r'\S')
check = t.findall(texto)
check1= q.findall(texto)
print(check,'\n',check1
```

```
texto = '''

_arara@ 1992_

'''
t = re.compile(r'\w')
q = re.compile(r'\W')
```

`\w` → Esse já verifica se o valor passado para análise é algum caractere alfa numérico (Ou seja, se é um número ou uma letra)

`\W` → Verifica tudo aquilo que não é alfa numérico presente dento do valor passado para análise, quase parecido com o \s, porém além de retorna objetos e

```
check = t.findall(texto)
check1= q.findall(texto)
print(check,'\n',check1)
```

comando usados para formatação, esse retorna acentos sinais com @ ou algo parecido

Nesse exemplo será retornado no primeiro comando a palavra `arara1992` sem o espaço e no segundo vai me retornar o `espaço` e o `@`

> 💡 Caso precise incluir o espaço como um caractere válido dentro da validação devemos colocar assim [\w] com um espaço dentro do colchete após o w

`\b` → Significa borda, ou seja seu padrão busca por uma borda nas frases (Em resumo , o motor de procura do REGEX busca por um espaço) ex:

`\B` → Retorna exatamente a correspondência indicada como parâmetro

```
texto = '''
Flores para Maria
'''
padrao = re.compile(r'\w+e\b')
padrao2 = re.compile(r'flor\B')
```

O primeiro retorno será uma palavra que termina com a letra `e` (**A borda seria como se fosse o espaço que separa as palavras**) já o segundo padrão retorna exatamente a correspondência para a palavra `flor` independente se em espaço ou não (**Ele captura um peço da palavra que forma a cadeia de caractere passada como parâmetro**)

> 💡 Mesmo que existe a palavra flor isolada no texto analisado, a negação de \B não retornaria, pois flor após a letra r tem uma borda e como ele está negando as bordas não encaixa

---

```
print(re.findall(r'João|Maria', texto))
```

`\` → Sinal de `OU`, quando o programa pode fazer uma escolha entre os valores que foram definidos como parâmetros **(Pode haver mais de duas opções para escolha do programa)**

Quantificadores

Esse tipo de símbolo permite especificarmos quantidade de vezes que um determinado caractere ou grupo de caractere aparece

Gulosos

O quantificador está sempre verificando até onde consegue fechar uma expressão regular ou seja até onde ele consegue satisfazer o padrão informado

Para diminuir ou deixar ele não guloso devemos passar o ? após informa um quantificador ex

```
import re

texto = '''
<p>Frase 1</p> <p>Eita</p> <p>Qualquer frase</p> <div></div>
'''

print(re.findall(r'<[pdiv]{1,3}>.*?<\/[pdiv]{1,3}>', texto))
```

Dentro desse exemplo estamos informando ao programa que capture a menor quantidade de texto possível entre a abertura e fechamento da tag, vamos detalhar passo a passo

`<[pdiv]{1,3}>` → Aqui estamos procurando uma tag de abertura para poder começar as buscas e verificação do padrão

`.*?` → Estou buscando uma cadeia de possíveis caracteres que se repitam n vezes ou até mesmo nenhuma porém coma diferença de

💡 Tem que retornar a menor correspondência possível até a tag de fechamento limitando assim a busca a abertura e fechamento de uma tag
Caso não tivesse o sinal de
`?` ele ia encontrar a tag de fechamento, e continuaria verificando até encontrar outra tag de fechamento (Formando a maior cadeia possível) ou até finalizar o texto analisado

`<\/[pdiv]{1,3}>` → Aqui é informado a tag de fechamento ou seja até onde o padrão deve se encerrar

> Em resumo o padrão fala assim: “Preciso da menor correspondência possível de um conjunto de caracteres que estiver entre a tag de abertura e fechamento

```
re.compile(r'c[aeiou]*t')

texto = "ct cat cot cooot cxt c-t cbt cut cueit"
resultado = padrao.findall(texto)
# Saída:'ct', 'cat', 'cot', 'cooot', 'cut', 'cueit'
```

`*` → Nesse quantificados estamos dizendo que o caractere que antecede o `*` pode aparecer **zero ou mais de uma vez**

O exemplo pede uma **palavra que começa com c, seguido ou não por uma ou mais vogais e finalizando com a letra t.**

💡 Estou falando que qualquer um dos caracteres presentes dentro do grupo pode aparecer nenhuma vez ou mais de uma vez (inclusive intercalando = cueit)

`+` → Nesse quantificador estamos dizendo que o caractere que antecede o mesmo **deve aparecer pelo menos uma vez,** diferente do anterior onde não precisava de aparecer

```
re.compile(r'c[aeiou]+t')

texto = "ct cat cot cooot cxt c-t cbt cut cueit"
resultado = padrao.findall(texto)
# Saída:'cat', 'cot', 'cooot', 'cut', 'cueit
```

Nesse exemplo estamos dizendo que tem que haver pelo menos uma vogal entre as letras c e t (Ou seja a palavra deve começar o c seguida **obrigatoriamente** por uma ou mais vogais finalizando com t)

```
re.compile(r'c[aeiou]?t')

texto = "ct cat cot cooot cxt c-t cbt cut cueit"
resultado = padrao.findall(texto)
# Saída:'cat', 'cot', 'cut'
```

`?` → Esse quantificador é mais restrito, onde o elemento que antecede o mesmo **tem que aparecer zero ou uma ocorrência.** Ou seja a palavra tem que começar com a letra `c`, seguida ou não de uma `(Apenas uma) vogal` e finalizando com a letra `t`

Outro exemplo que podemos passar seria com relação a variação na escrita de uma palavra, onde algumas pessoas usam espaço e outra hífen e desejamos buscar a correspondência para ambas

```
re.compile(r'home[- ]? brew, re.IGNORECASE')
```

Nesse exemplo estamos procurando pelas variações **home-brew ou home brew,** além de desconsiderar se está em maiúsculo ou minúsculo

```
padrao = re.compile(r'^Python')
```

`^` → Esse quantificador ou sinalizador passa a informação a expressão que deve obrigatoriamente começar com o padrão informado, caso o contrário não será aceito pelo padrão ex:

No exemplo pedimos obrigatoriamente que a frase analisada comece com a palavra 'Python' , porém pode se aplicado a outros exemplos `(^[0-9]{3}\.[0-9]{3}\.[0-9]{3}-[0-9]{2})` onde pedimos que input comece com um CPF (Ou no caso padrão do CPF, sem validação se é valido ou não)

> 💡 Se for usado `^` para sinalizar o inicio de uma frase e `$` para o final do padrão, pedimos que o valor informado ou mesmo o valor procura tem que ser aquele sem que haja mais nada antes ou depois (Dependendo do método de busca)

`{}` → Nesse quantificador, podemos passar uma espécie de range para que um determinado padrão aconteça limitando a quantidade inicial e final

```
re.compile(r'c[aeiou]{2,5}t')

texto = "ct caat cooot coooot caaaaaat cxt cbt cut cet"
resultado = padrao.findall(texto)
# Saída: ['caat', 'cooot', 'coooot']
```

Nesse padrão queremos que a palavra comece com c, tenhas entre 2 a 5 vogais e termine com a letra t

OBSERVAÇÕES

Podemos omitir um dos argumentos ex.

`{,5}` → Está dizendo que pode aparecer nenhuma ou no máximo 5 vezes o caractere que antecede o quantificador

`{2,}` → Está dizendo que tem que aparecer pelo menos 2x e sem um limite para a quantidade final

`{5}` → Caso esteja assim, estamos pedindo que o caractere tem que aparecer exatamente 5x

---

Parâmetros/Padrão

Caractere Literal

Dentro das expressões de regex, **temos alguns caracteres que são literais,** ou seja corresponde ao seu valor = `U→` Se o U for passado como um parâmetro ele **corresponde a letra U com em caixa alta** (Em outras palavras ao valor **85 da tabela ASCII**) por sua vez o `u` corresponde a mesma coisa porém em minúscula (**Valor 117 da tabela ASCII**)

Porém temos **alguns valores de uso exclusivo dentro do contexto do regex** que são os mencionados nessa página como `+*?` que são para representa um determinado acontecimento dentro do regex (Onde o mesmo perde o valor literal)

Dentro do universo de REGEX, temos os **padrões que queremos observar e pesquisar dentro de um texto,** para passar um padrão, podemos usar qualquer dos símbolos passados para formar um contexto a ser pesquisado

> 💡 Ou seja se quisermos considerar apenas número podemos usar \d se quisermos só formatação usamos \s e assim por diante

Porém temos algumas diferenças a serem pontuadas que fazer a diferença na hora de passar esses parâmetros

`[]` → **Character Set** → Nesse modo passamos tudo que estiver dentro do colchete como se fosse uma lista de possíveis caracteres (Porém os mesmo devem seguir um padrão pré-definido dentro do colchete) no exemplo abaixo podemos observar

```
padrao = re.compile(r'[a-zA-Z] [0-9]')
```

- Nesse exemplo o padrão desejado é uma letra de A a Z independe se é maiúscula ou minúscula seguido de uma espaço e um número. Logo vai me retornar qualquer padrão dentro de um texto seguido por espaço e um número

- [a-zA-Z] → Esse trecho do padrão se refere a letra (Maiúscula ou Minúscula)
- “ “ → Esse pedaço se refere ao espaço entre os caracteres
- [0-9] → Se refere a qualquer algarismo entre 0 e 9

- **O espaço dentro do padrão é literal, ou seja corresponde a um espaço dentro do texto**

Caso fosse alterado o padrão para:

```
padrao = re.compile(r'[aeiou]+ [0-9], re.IGNORECASE')
```

Nesse exemplo queremos um pedaço do texto que seja uma vogal (pelo sinal de + obrigatoriamente tem que aparecer pelo menos uma vez) seguida por um espaço e algum caractere numérico

Ao final temos uma ***flag*** que indica para o **regex** ignorar a diferença entre maiúscula e minúscula

Nesse exemplo foi aplicado um filtro onde estou selecionando qualquer letra indiferente se é maiúscula ou minúscula que se repita uma ou mais vezes no meio de números, após retornando um objeto match com a posição e correspondência

Podemos dizer que esse método trás uma **variação de uma palavra** (Onde a mesma poder ser escrita de formas diferentes) pois você pode **fazer com que um caractere tenha valores diferentes**

```
import re

texto = '''
123 456 789 Estou escondio 987 654 321
'''

padrao = re.compile(r'[a-zA-Z]+')

verifica = re.finditer(padrao, texto)

for correspondendia in verifica:
    print(correspondendia)

#<re.Match object; span=(13, 18), match='Estou'>
#<re.Match object; span=(19, 27), match='escondio'>
```

Uma ressalva final que podemos fazer sobre o conjunto de caracteres é que podemos comparar com o sinal de | que corresponde a porta lógica OR|OU que pode ser uma coisa ou outra

`()` → Quando o argumento ou expressão é passada dentro de parênteses estamos criando um grupo de seleção ou um grupo de captura

Em resumo, um grupo de captura ajuda na hora da organização dos valores obtidos, além de poder escolher oque realmente importa da expressão ou oque pode ser descartado. Também auxilia no momento de exibição, pois você pode mostrar somente um determinado grupo

```
padrao = re.compile(r'<([a-z]+?)>(.*?)<\/\1')

for texto in final:
    print(texto.group(0))
    print(texto.group(1))
    print(texto.group(2))
#Faz interação mostrando o valor de cada grupo
```

Nesse exemplo temos um total de 2 grupos, onde o primeiro identifica uma TAG HTML (No caso identifica oque está escrito dentro da TAG e armazena esse valor) o segundo faz a busca pelo conteúdo que está dentro da TAG (Se for um parágrafo mostraria o conteúdo dentro do parágrafo) por fim temo `\1` esses caracteres significam

`\/` → Escapa a barra / para poder usar (A barra invertida é usada para escapar o caractere) não precisa, porém como forma de garantia

`\1` → Retrovisor do grupo 1, ou seja, durante a iteração do regex com o texto ele vai encontrar o valor presente em `<([a-z]+?)>` e o valor encontrado aqui, será usado para encontrar o fechamento da TAG

Essa é outra vantagens dos grupos, armazenar e reutilizar o valor encontrado dentro da própria expressão

> 💡 Ao final do grupo, após sua definição e fechamento dos parênteses, podemos indexar o mesmo para ficar mais fácil de usar seu retrovisor posteriormente ou mesmo para poder ficar mais fácil de identificar qual grupo será exibido ou manipulado `|(r'<([a-z]+?)-\1>(.*?)-\2<\/\1')|` **o -\1 e -\2 é usado para fazer a indexação**

Quando não queremos capturar o conteúdo dento de um grupo usamos ?: para indicar (Quando é feito isso o grupo passa a não ter uma referência logo seu retrovisor passar para o próximo grupo)

`"..."` → quando é informado os Caracteres de forma direta sem estar dentro do colchete, estamos buscando correspondência clara e exata do argumento passado, onde `"Arara"` seria exclusivamente a palavra `Arara` com a primeira letra a em maiúsculo, cose fosse arara não seria reconhecido

> 💡 Quando colocamos assim [A][r][a] é mesmo de dizer "Arara" onde tem que ser exatamente a mesma correspondência Temos também as FLAGS que servem como

---

Funções

Podemos usar algumas funções para realizar buscas e retornar certos valores ou certos padrões de valores dentro do REGEX, temos as seguintes funções para busca

Findall

Retorna uma lista com todos os caracteres encontrados podendo também retornar uma lista vazia caso nenhuma correspondência for encontrada

> 💡 Caso seja **passado um grupo de caracteres** como parâmetro igual o exemplo da Arara **será retornado uma lista com o grupo encontrado**
>
> ```
> lista = ['Arara', 'Arara']
> ```
>
> Caso tivesse duas palavras arara e fosse passado como parâmetro a função retornara o código acima

---

Match

> 💡 Esse método faz a busca no inicia da string apenas. Em outras palavras **valida se uma string segue o padrão desde o inicio**

```
texto = 'arara'

padrao = re.compile(r'ar')
consuta = padrao.match(texto)

#<re.Match object; span=(0, 2), match='a'>
```

Retorna a posição onde foi encontrado o padrão analisada, ou seja o range onde foi encontrado o padrão (2, 8) - lembrado que o range não leva em conta o ultimo valor passado, ou seja indo até `i-1` ou no exemplo 7, além de retornar também o valor encontrado

- Retorna a posição em formato de range

Esse métodos é usado quando você sabe que uma string deve começar com um certo padrão e deseja buscar por ele

- Retorna o valor encontrado

Search

> 💡 Diferente do anterior, esse método faz a busca na string por completo, ou seja ira busca a correspondência em qualquer ponto da string, porém para ao encontrar a primeira correspondência ao padrão

Retorna a principio o mesmo padrão do match, **onde é retornado o valor e a posição/range do objeto dentro da string**

Nesse caso o método Search para ao encontrar a primeira correspondência, retornando a posição e o valor dela no console

```
texto = 'arara'

padrao = re.compile(r'a')
consuta = padrao.search(texto)

#re.Match object; span=(0, 1), match='a'>
```

Finditer

Faz um processo de busca dentro do texto e retorna um **iterador como resposta contendo todas as correspondências encontradas dentro do texto** (Em **formado de objeto Match**, ou seja gerando um span contendo a posição do item e o match que seria a correspondência) isso para todas as correspondências encontradas pelo método

Com esse iterador podemos usar dentro de um laço de repetição que mostra todas as correspondências encontradas para o padrão

Nesse exemplo será retornado um iterador contendo **todas as correspondências da letra a dentro da variável texto,** após será percorrido esse iterador por um **loop que mostra todas as correspondências em forma de objeto Match**

```
texto = 'arara'
p = re.compile(r'a')
verifica = p.finditer(texto)

correspondencias = verifica
for correspondencia in correspondencias:
    print(correspondencia)
```

Sub

Esse módulo permite buscar certas cadeia de caracteres e fazer a substituição por outras

- Dentro desse método, podemos passar para o programa qual palavra queremos trocar, ou seja em casos de repetição se queremos trocar a primeira segundo e assim por diante, usando o parâmetro count

```
import re

string = 'Esse é um Teste de expressões TESTE regulares.'

padrao = re.compile(r'teste', re.IGNORECASE)
print(re.sub(padrao, r'testado', string))

#Esse é um testado de expressões testado regulares.
```

> 💡 Nesse caso, é realizado uma busca completa por toda a string e faz as devidas substituições de acordo com os padrões

Lookarounds

São métodos de aplicar um novo filtro a expressão, similares a condicional de IF dentro dos códigos, trazendo mais robustez e eficácia ao código

💡 Aplicando um exemplo prático:

- **Lookahead** → Queremos encontrar uma mensagem de erro **após** um endereço IP, logo colocamos a expressão `(?=ERROR`
- **Lookbehind** → Quando queremos encontrar dentro de um texto referências a valores monetários, como por padrão escrevemos sempre colocamos `$` e depois o valor, usamos uma expressão que verifica a existência do símbolo de `$` **antes** de algum valor `(?<=\$)`

Ambos os filtros (lookahead e lookbehind) podem ser negados usando o sinal de ! ao invés do =, passando o resultado de, retorne valores que não apresentam o filtro (antes ou depois) do padrão

Lookahead

Essa função é usada dentro de uma expressão regular (REGEX) para poder aplicar um **"filtro"** a expressão que **além de encontrar o padrão desejado, ainda adiciona uma nova verificação,** como se fosse uma pergunta **"Depois do padrão encontrado tem essa condição"** caso seja verdadeiro retorna o padrão anterior ao lookahead (Vale lembra também que **o padrão de verificação** ou validação dentro do lookahead **não é retornado e nem consumido pelo motor de busca**)

- Também podemos ressaltar que por padrão ele busca unicamente na posição que foi inserido dentro da expressão, ou seja, caso for criado uma expressão que busque pelo padrão de um endereço **IP (192.168.0.1)** ou em REGEX `(\d{3}\.\d{3}\.\d{1}\.\d{1})` e depois for acrescentado o lookahead `(?=Error)` formando algo como `(\d{3}\.\d{3}\.\d{1}\.\d{1})(?=Error)` ele vai buscar uma correspondência para o padrão do endereço seguido por uma mensagem de erro e somente isso
- Podemos **também usar a negação,** ou seja tudo aquilo que não apresente o padrão de verificação usando a expressão `(?!Error)` estamos procurando por tudo aquilo que após o padrão principal não tenha a palavra erro

💡 Quando colocamos a expressão somente de busca pelo padrão do endereço IP ele retorna todos os endereços o match encontrados, já quando acrescentamos o filtro, **restringimos a busca a somente aquele padrão específico, deixa a expressão mais precisa a depender da situação**

Vale mencionar que ele **não exibe/retorna o padrão de validação**, ou seja quando colocamos que ele valide a partir de uma mensagem de erro, ele não retorno o endereço IP e depois a mensagem de erro usada para validação, somente o endereço IP que é o padrão desejado **(Por isso é mencionado que não consome e nem alterar os caracteres, apenas espiona de certa forma e trás uma validação) como se fosse o modo leitura de alguns editores de texto**

💡 "O lookahead **não altera a string original e não captura o padrão de validação.** Ele funciona como um filtro, verificando apenas se o critério logo à frente do padrão principal é atendido.

Passo a Passo

- Busca o padrão
- Após encontrar valida se realmente atende ao padrão principal
- Verifica se logo a frente existe a mensagem de erro
  - Caso seja verdadeiro retorno o valor encontrado
  - Caso não para a busca pois já não corresponde ao padrão total (Endereço IP com a mensagem de erro, logo não tem porque de continuar)
- Próximo padrão encontrado

Lookbehind

Nesse método de verificação segue o mesmo princípio do lookahead, porém com a diferença que ele olha para trás, enquanto o lookahead olha para frente **(Em resumo, enquanto um confirma se existe uma confirmação após o padrão o outro verifica se há uma confirmação antes do padrão)**

Nesse caso colocamos o “filtro” antes da expressão com o sinal de (?<=) e depois o padrão analisado, por exemplo:

```
import re

padrao = re.compile(r'(?<=\$)\d+)
```

Nesse exemplo buscamos dígitos numéricos onde antes dele haja o sinal de $ simbolizando dinheiro por exemplo. Então é feito uma pergunta **(Dentro dos dígitos encontrados algum tem o sinal de $ antes ?)**

Objeto Match

Sinalizadores/FLAGS

São usadas como modificares dentro de uma expressão, com a possibilidade alterar a saída ou mesmo a forma de compilação de um padrão

GLOBAL

Habilita a possibilidade de encontrar mais de uma correspondência ao padrão

ASCII

Essa flag restringe o Python a apenas caracteres da tabela ASCII, desconsiderando os caracteres unicode, por exemplo o `\w` **normal sem a flag consiste em letras com acentuação** por exemplo, **com a flag ativada corresponde somente a letras sem acentos**

IGNORECASE

Usado para não diferenciar minúscula de maiúscula, ou seja teste seria a mesa coisa de TESTE ou exemplos semelhantes

```
import re

string = 'Esse é um TESTE de expressões teste TesTe regulares.'

print(re.search(r'teste',string, re.IGNORECASE))
print(re.findall(r'teste',string, re.IGNORECASE))

#<re.Match object; span=(10, 15), match='TESTE'>
#['TESTE', 'teste', 'TesTe'
```

MULTILINE

É usado para indicar que os metacaracteres como `^` e `$` possar ser procurados em múltiplas linhas, por padrão os metacaracteres indicados acima verificam o texto como um todo, ou seja necessariamente a primeira linha tem que começar com o padrão e finalizar com o padrão independente da quantidade de linhas do texto. Quando ativado essa flag, cada linha passa a ser verificada de forma individual, verificando se ela começa com o padrão ou termina com o padrão e depois passa a para a próxima linha

```
import re

texto = '''
131.768.460-53
055.123.060-50
955.123.060-90
'''

print(re.findall(r'^\d{3}.\d{3}.\d{3}-\d{2}$', texto, flags=re.M))
```

No exemplo acima vai verificar se cada linha começa ou não com o padrão informado

Dotall

É usado para fazer com que os metacaracteres possam verificar em mais de uma linha, por padrão eles verificam até uma quebra de linha ou até o final de uma linha, com essa flag, passa a verificar em todas as linhas até encontrar o padrão