Validação de dados

Prof. Alysson Messias

Descrição

Conceitos de expressões regulares, apresentação do seu funcionamento e demonstração de aplicações para validação de dados.

Propósito

Aprender a validar conteúdos, realizar substituições de texto e extrair informações. Com expressões regulares, é possível atestar se um e-mail é válido ou não, verificar contagem de letras e uso de caracteres especiais em senhas. Tem a finalidade de facilitar vários tipos de validações de dados, além de estar presente em diversas linguagens de programação.

Preparação

No decorrer deste estudo, utilizaremos ferramentas web, sem necessidade de instalação ou criação de conta.

Objetivos

Módulo 1

Expressões regulares

Demonstrar a montagem de expressões regulares.

Módulo 2

Grupos e quantificadores

Relacionar quantificadores e grupos.

Módulo 3

Regex com JavaScript

Elaborar regex com JavatScript.

Módulo 4

Regex com Python

Elaborar regex com Python.

Introdução

Vivemos em um mundo em que os dados são o novo petróleo, quanto melhor cuidarmos deles mais valiosos serão. Nesse contexto, podemos entender a importância em validar os dados, que também podem ser representados como sanitizar, realizar substituições ou apenas extrair informações. As expressões regulares são grandes aliadas nesse cenário, trazendo velocidade e produtividade nessa tarefa de validação.

Apesar de a maioria das linguagens de programação, programas e editores de texto mais utilizados possuir recursos para interpretar expressões regulares, poucas pessoas os dominam. Isso ocorre devido ao fato de a documentação sobre o assunto, quando existente, ser pouco didática ou se resumir a listagens ou exemplos muito específicos sem descrever os conceitos.

No decorrer deste estudo, serão apresentados os conceitos básicos para compreender a criação, manipulação e aplicação de expressões regulares no cotidiano. Esse conhecimento, com certeza, vai melhorar a sua produtividade na manipulação e validação de dados.

1 - Expressões regulares

Ao final deste módulo, você será capaz de demonstrar a montagem de expressões regulares.

Conceitos básicos

Expressões regulares são strings de texto especialmente codificadas e utilizadas como padrões para corresponder a conjuntos de strings. Podem ser consideradas como um método formal de se especificar um padrão de texto.

As expressões regulares surgiram na década de 1940 como uma maneira de descrever linguagens comuns, mas passaram realmente a ter destaque no mundo da programação na década de 1970.

> “
>
> Uma expressão regular é um padrão que especifica um conjunto de strings de caracteres; diz-se que ela corresponde a determinadas strings.
>
> (THOMPSON, s. d., n. p. apud FITZGERALD, 2012, p. 16)

Essas expressões contêm diversas terminologias, sendo algumas delas: **ER, ExpReg, Regular Expression, RegExp e regex**. A terminologia **regex** é mais popular na internet, por isso a utilizaremos no decorrer de nosso estudo.

E para quais situações podemos utilizar regex?

Na prática, servem para uma infinidade de tarefas, sendo úteis sempre que for necessário **buscar ou validar um padrão de texto que pode ser variável**, como uma data, horário, email, CPF, RG, cartão de crédito, número de telefone, CEP e outras possibilidades. Para tornar tudo isso possível, vamos primeiro conhecer os metacaracteres.

Metacaracteres

Definições

Os **metacaracteres** são caracteres especiais identificados pelo regex. Cada um deles tem uma função específica, sendo utilizados para identificar padrões dentro da string, como números, letras minúsculas ou maiúsculas. Uma vez identificados, existe a possibilidade de validar ou substituir determinado caractere ou um conjunto deles.

Suas funções podem mudar dependendo do contexto. Também podem ser combinados uns com os outros. A sintaxe de utilização não é igual para todos os programas que suportam regex. Pode parecer um pouco complicado, mas vamos explicar cada um deles e os exemplificar.

Comentário

Antes de tudo, vamos conhecer os principais metacaracteres:
`. ? * + ^ $ | [] {} () \`

Os metacaracteres vistos anteriormente são os **símbolos** que, quando combinados, definem um padrão (pattern) para casar (match) em um texto. Observe a tabela a seguir:

Metacaracteres	Nome	Metacaracteres	Nome	Metacaractere	Nome
.	Ponto	[ ]	Lista	[^]	Lista Negada
?	Opcional	*	Asterisco	+	Mais
{ }	Chaves	^	Circunflexo	$	Cifrão
\b	Borda	\	Escape	|	Ou
( )	Grupo	\1	Retrovisor		

Tabela: Metacaracteres.
Alysson Messias.

Existem **metacaracteres estendidos** para atender a tarefas mais complexas e com funções mais específicas que estão divididos em quatro grupos distintos por características. Existem vários motores de interpretação regex, os quais podem variar dependendo do programa ou da linguagem em que será utilizado. Nos exemplos deste estudo, vamos utilizar o *motor perl compatible regular expressions* (PCRE), uma biblioteca escrita em C que

implementa um mecanismo de expressão regular inspirado nos recursos da linguagem de programação perl. Existem algumas ferramentas web que fazem a validação regex:

- https://regex101.com
- https://www.regexpal.com

Representantes ou especificadores

O **ponto (.)** é um curinga utilizado para buscar qualquer caractere. A lista ([ ]) só aceita o caractere descrito dentro dela. O circunflexo ([^]) nega os caracteres que forem especificados logo após. Observe a seguir a tabela com o uso e as funções do ponto.

Metacaractere	Nome	Função que representa
.	Ponto	Um caractere qualquer
[...]	Lista	Lista de caracteres permitidos
[^...]	Lista negada	Lista de caracteres proibidos

Tabela: Representação e função do ponto (.).
Alysson Messias.

Com as definições desse grupo, vamos colocar a mão na massa analisando exemplos a seguir.

Para começar, vamos abrir o site da **regex101** e supor que tenhamos um campo de input em um formulário de dados que precise validar algumas situações, podendo ou não permitir a inserção de alguns caracteres. Para isso, será necessário passar algumas informações dentro da lista [ ] de caracteres.

Problema 1: Validar apenas números

Regex: [0-9]
String de teste: abc123DEF
Resultado: abc**123**DEF

O trecho destacado é a parte em que houve incidência (match) do regex. A expressão 0-9 corresponde a qualquer ocorrência que apareça entre os números 0 até 9.

Problema 2: Permitir apenas letras no campo

Regex: [a-zA-Z]
String de teste: Coração Valente 123
Resultado: Coração **Valente** 123

A expressão a-z e A-Z corresponde a qualquer ocorrência que apareça entre as letras a até Z, sendo minúsculas ou maiúsculas. Note que nem as letras com acento ou número entraram na validação.

Problema 3: Permitir letras com e sem acentos

Regex: [A-zÀ-ú]
String de teste: Coração Valente 123
Resultado: Coração Valente 123

Veja que, nesse exemplo, usamos outra maneira de expressar caracteres minúsculos ou maiúsculos A-z, porém essa definição abrange também os caracteres [\]^_`.Então, quando quiser permitir apenas letras, é melhor utilizar A-Za-z. Também incluímos a validação À-ú, que será o nosso range para caracteres acentuados.

Problema 4: Não pode inserir letras

Regex: [^A-zÀ-ú]
String de teste: Coração Valente 123
Resultado: Coração Valente **123**

Quando o circunflexo é incluído em uma lista, a expressão vira uma negação, e tudo que vier depois será para **não** permitir.

O **POSIX** é um padrão internacional que define grupos por tipo de regra e facilita a montagem do regex.

Veja na tabela a seguir a representação e função do POSIX:

Classe POSIX	Similar	Significado	Classe POSIX	Similar	Significado
[:upper:]	[A-Z]	Letras maiúsculas	[:xdigit:]	[0-9A-Fa-f]	Números hexadecimais
[:lower:]	[a-z]	Letras minúsculas	[:punct:]	[.!?: ...]	Sinais de pontuação
[:alpha:]	[A-Za-z]	Maiúsculas/ minúsculas	[:blank:]	[\t]	Espaço e Tab
[:alnum:]	[A-Za-0-9]	Letras e números	[:space:]	[\t\n\r\f \ v]	Caracteres brancos
[:digit:]	[0-9]	Números	[:cntrl:]		Caracteres de controle
[:graph:]	[^\t\n\r\f\v]	Caracteres imprimíveis	[:print:]	[^\ t \ n\ r\f\v]	Imprimíveis e o espaço

Classe POSIX	Similar	Significado	Classe POSIX	Similar	Significado

Tabela: Representação e função do POSIX.
Alysson Messias.

Quantificadores

Os **quantificadores** indicam o número de repetições permitidas para a entidade imediatamente anterior, sendo um caractere ou metacaractere. Os quantificadores não podem vir seguidos, pois isso causaria um erro de regex, salvo uma exceção que será mostrada mais à frente. Observe a tabela a seguir:

Metacaractere	Nome	Função que representa
?	Opcional	Zero ou um
*	Asterisco	Zero, um ou mais
+	Mais	Um ou mais
{n,m}	Chaves	De n até m

Tabela: Representação de quantificadores.
Alysson Messias.

O **grupo opcional (?)** pode ter nenhuma ou uma ocorrência da entidade anterior definida. O **asterisco (*)** é mais abrangente, aceitando nenhuma, uma ou diversas ocorrências. O **mais (+)** tem uma função parecida com a do asterisco, porém não é opcional, sendo necessária ao menos uma ocorrência. Na **chave ({n,m})**, é passado um delimitador de quantidades de ocorrências, logo {n,m} significa de n até m vezes. Para compreendermos melhor, vamos praticar utilizando alguns exemplos.

Problema 1: Validar código que tenha as duas primeiras letras “BR” e após 4 dígitos

Regex: BR[0-9?][0-9?][0-9?][0-9?]
Casa com os casos: BR0001, BR2354, BR9999

Problema 2: Validar o formato de CEP

Regex: [0-9?][0-9?][0-9?][0-9?][0-9?]-[0-9?][0-9?][0-9?]
Casa com os casos: 49000-400, 72000-200, 57035-300

Problema 3: Encontrar o radical “cert” no texto

Regex: cert*
String de teste: certo, certeza e incerteza.
Resultado: **cert**o, **cert**eza e in**cert**eza.

Problema 4: Encontrar caractere “c” seguido de uma ou mais vogais minúsculas

Regex: c[aeiou]+
String de teste: É encontrado em cooperativas secundárias.
Resultado: É en**co**ntrado em **coo**perativas se**cu**ndárias.

Problema 5: Encontrar ditongos

Regex: [aeiouAEIOU]{2}[^aeiouAEIOU]
String de teste: INÍCÍO: É encontrado em cooperativas secundárias no PARAGUAI.
Resultado: INÍCÍO: É encontrado em c**oop**erativas secundár**ias** no PARAGU**AI**. }

A primeira parte da expressão “[aeiouAEIOU]{2}” busca qualquer duas vogais com duas incidências. A segunda parte desconsidera os tritongos, que teriam mais uma vogal junta. Perceba que o ditongo na palavra “INÍCÍO” não foi encontrado pelo regex porque uma das vogais está acentuada. Para solucionarmos essa questão, basta realizarmos um pequeno ajuste:
Regex: [aeiouAEIOUÀ-ú]{2}[^aeiouAEIOU]
String de teste: INÍCÍO: É encontrado em cooperativas secundárias no PARAGUAI.
Resultado: INÍC**ÍO**: É encontrado em c**oop**erativas secundár**ias** no PARAGU**AI**.

E caso quiséssemos marcar ditongos e tritongos, bastaria fazer a seguinte modificação:
Regex: [aeiouAEIOUÀ-ú]{2,3}
String de teste: INÍCÍO: É encontrado em cooperativas secundárias no PARAGUAI.
Resultado: INÍC**ÍO**: É encontrado em c**oop**erativas secundár**ia**s no PARAG**UAI**.

Para melhor compreensão, veja com atenção a tabela com as possibilidades de uso das chaves:

Metacaractere	Repetições
{1,3}	De 1 a 3
{3}	Pelo menos 3 (3 ou mais)
{0,3}	Até 3
{3}	Exatamente 3
{1}	Exatamente 1
{0,1}	Zero ou 1 (igual ao opcional)

Metacaractere	Repetições
{0,}	Zero ou mais (igual ao asterisco)
{1,}	Um ou mais (igual ao mais)

Tabela: Tabela com possibilidades de usos para as chaves.
Alysson Messias.

Âncora

Os metacaracteres do **grupo âncora** estabelecem posições de referência para o casamento do restante da regex. Eles não casam com caracteres específicos no texto ou definem quantidades, mas, sim, com posições antes, depois ou entre os caracteres. Marcam uma posição específica na linha. Veja na tabela a seguir:

Metacaractere	Nome	Função que representa
^	Circunflexo	Início da linha
$	Cifrão	Fim da linha
\b	Borda	Início ou fim de palavra

Tabela: Quantificadores do grupo âncora.
Alysson Messias.

O **circunflexo (^)** marca o começo de uma linha, porém, como vimos, esse símbolo também é o marcador de uma lista negada quando está no começo da lista. Por exemplo: [^0-9]. Nesse caso, como âncora, o circunflexo estaria no começo e fora da lista: ^[0-9].'

Problema 1: Começo de linha que comece com a letra "A".

Regex: ^[A]
String de teste:
A escola é demais.
O hospital está com problemas.
As lojas estão caras.

Resultado:
A escola é demais.
O hospital está com problemas.
As lojas estão caras.

Problema 2: Código começa com “BR”.

Regex: ^(BR)
String de teste:
BR992810
QBR992810
0BR992810
992810BR

Resultado:
BR992810
QBR992810
0BR992810
992810BR

No outro extremo, encontramos o símbolo cifrão, que marca o fim de uma linha e somente será válido se for incluído no fim da linha de um regex. Por exemplo: [0-9]$.

Problema 3: Os últimos caracteres do código devem obrigatoriamente ser dígitos

Regex: [0-9][0-9][0-9]$
String de teste:
BR9928A10
QBR992810
0BR992810

Resultado:
BR9928A10
QBR992**810**
0BR992**810**

Por marcar os limites de uma palavra, a borda é muito útil para buscar termos exatos. Quando usada do lado direito, como em “dia\b”, a borda encontra determinado trecho no final das palavras. Quando usada do lado esquerdo, como em “\bdia”, a borda encontra determinado trecho no começo das palavras.

Problema 4: Encontrar palavras no infinitivo (terminadas com ar, er ou ir)

Regex: [aeiouAEIOU][rR]\b
String de teste: Vamos falar de Tecpix, melhor marca para escolher.
Resultado: Vamos fal**ar** de Tecpix, melh**or** marca para escolh**er**.

Problema 5: Descobrir as quebras de linhas

Regex: \n
String de teste:
Aqui é melhor,
ou será pior
que melhor.

Resultado:
Aqui é melhor **[ENTER]**
ou será pior **[ENTER]**
que melhor.

Usamos ainda outros metacaracteres, tais como: **\n** para retornar uma nova linha; **\r** para retornar uma linha; **\s** para retornar espaço em branco; **\d** para localizar números; **\A** para marcar o início do texto; **\Z** para marcar o fim do texto; e **\B** para retornar as posições de não borda.

Outros metacaracteres

Observe a tabela a seguir que indica outros metacaracteres.

Metacaractere	Nome	Função que representa
\	Escape	Torna os metacaracteres em texto quando os antecede.
|	Ou	Ou um outro.
( )	Grupo	Delimita um grupo.
\1...\9	Retrovisor	Textos casados nos grupos 1...9.

Tabela: Outros metacaracteres.
Alysson Messias.

Existem outros **metacaracteres com funções específicas e não relacionadas entre si** e que não podem ser agrupados em outra classe. A seguir veremos alguns exemplos de uso desse tipo de metacaracteres.

Problema 1: Encontrar radical "pedr" nas palavras

Regex: (in)?pedr
String de teste: pedra pedreiro pedregulho pedrada apedrejar
Resultado: pedra **pedr**eiro **pedr**egulho **pedr**ada a**pedr**ejar

Problema 2: Encontrar as palavras "casado" e "casada"

Regex: casad(o|a)
String de teste: O homem é casado e a mulher é casada.
Resultado: O homem é **casado** e a mulher é **casada**.

Problema 3: Encontrar o radical (a ou e ou i)ndo]

Regex: (a|e|i)ndo
String de teste: caminhando, conversando, sofrendo, sorrindo
Resultado: caminh**ando**, convers**ando**, sofr**endo**, sorr**indo**.

Problema 4: Localize os números

Regex: \d
String de teste: O número 3 é menor do 5, porém maior que 2.
Resultado: O número **3** é menor do **5**, porém maior que **2**.

Problema 5: Localize palavras que tenham “ss” ou “rr”

Regex: (s)\1|(r)\2
String de teste: O frango assa lentamente enquanto o carro chega.
Resultado: O frango a**ss**a lentamente enquanto o carro chega.

Problema 6: Localize todos os tipos de mercado (supermercado, hipermercado, mercado) começando com maiúsculo ou minúsculo

Regex: (([sS]u|[hH]i)per)?[mM]ercado
String de teste: O sonho do mercado é virar Supermercado e depois hipermercado.
Resultado: O sonho do **mercado** é virar **Supermercado** e depois **hipermercado**.

Antes de encerrarmos este assunto, vamos analisar com detalhes o exemplo anterior:

(su|hi)permercado – nessa expressão, já retornaria o supermercado e o hipermercado, porém o mercado ficaria de fora. Por esse motivo, vamos melhorar o regex.

((su|hi)per)?mercado – retirando o trecho “per” e deixando o trecho “mercado” como opcional, conseguimos abranger as três palavras. Vamos, agora, ajustar para localizar as palavras que começam com minúsculas e minúsculas.

(([sS]u|[hH]i)per)?[mM]ercado – agora conseguimos atender a todos os critérios!

Validando um código de rastreio do correio brasileiro

Confira agora a expressão regular que reconhece como válido o código de rastreio de uma correspondência postada no correio do Brasil.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

As expressões regulares são muito utilizadas para realizar formatação de campos de input de dados em formulário de páginas html. Qual dos itens abaixo contém o regex correto para validar o CPF com ou sem formatação?

A [0-9][0-9][0-9]|\d{2}

B \d{3}\.\d{3}\.\d{3}-\d{2}

C [0-9][0-9][0-9].[0-9][0-9][0-9].[0-9][0-9][0-9]-[0-9][0-9]

D \d{3}[.-]?\d{3}[.-]?\d{3}[.-]?\d{2}

E	[0-9.0-9.0-9-0-9.0-9]

Parabéns! A alternativa D está correta.

Opção A está ERRADA - A última parte aceita tanto 2 quanto 3 dígitos e não aceita o CPF formatado.
Opção B está ERRADA - Valida apenas CPF formatado.
Opção C está ERRADA - Valida apenas CPF formatado.
Opção D está CERTA - Valida tanto CPF formatado com 14 dígitos como sem formatação com 11 dígitos.
Opção E está ERRADA - Casa com o CPF com e sem formatado, porém não delimita a quantidade de dígitos.

Questão 2

Suponha que vamos criar um regex para validar uma hora e minuto “00:00” no formato válido e respeitando apenas horários e minutos válidos. Qual dos itens abaixo contém o regex correto para realizar essa validação?

A	[0-9][0-9]:[0-9][0-9]
B	([01]?2[0-3]|[0-9]):[0-5][0-9]
C	[012][0-9]:[0-5][0-9]
D	[\d][\d]:[\d][\d]
E	\d{2}:\d{2}

Parabéns! A alternativa B está correta.

Opções A, D e E estão ERRADAS – Aceitam valores corretos como “08:30” e “22:50”, porém não validam horas como “99:00” e “99:99”.
Opção B está CERTA – A formatação respeita a quantidade de caracteres e define apenas os valores possíveis para de uma hora correta.
Opção E está ERRADA – Melhorou a validação, porém aceita horas maiores que 23 como “29:00”.

2 - Grupos e quantificadores

Ao final deste módulo, você será capaz de relacionar quantificadores e grupos.

Grupos

O objetivo dos **grupos** é juntar vários tipos de metacaracteres e caracteres em um mesmo lugar. Desse modo, podem conter um ou mais caracteres, metacaracteres e inclusive outros grupos. Em uma correlação, os grupos são como expressões matemáticas, sendo definidos por parênteses, ou seja, **"("** e **")"**. O seu conteúdo por ser visto como um bloco na expressão.

Por meio dos grupos, todos os metacaracteres e quantificadores podem ter seu poder ampliado, pois serão mais abrangentes. Vamos analisar alguns exemplos de uso a seguir.

Para **Regex: (?<=>)([\w\s]+)(?=<\ /)**, teremos a String de teste e o resultado:

REGEX

```
<h1>Identificar tags</hi>
<html>
   <div>teste</div>
   <div>teste</div>
   <h1>teste</h1>
</html>
```

Strring de teste.

REGEX

```
<h1>Identificar tags</hi>
<html>
   <div>teste</div>
   <div>teste</div>
   <h1>teste</h1>
</html>
```

Resultado.

Nesse caso, temos três grupos e cada um possui uma definição: identificar aberturas de tags, identificar o fechamento de tags, e buscar o que estiver dentro das tags; esse caso funciona para qualquer tag, mas pode ser definido para determinada tag realizando alterações.

Para **Regex: (?<=>)([\w\s]+)(?=<\ /h1)** , teremos:

```
REGEX

<h1>Identificar tags</hi>
<html>
    <div>teste</div>
    <div>teste</div>
    <h1>teste</h1>
</html>
```

String de teste.

```
REGEX

<h1>Identificar tags</hi>
<html>
    <div>teste</div>
    <div>teste</div>
    <h1>teste</h1>
</html>
```

Resultado.

Os **retrovisores "\"** são muito úteis nos grupos. Como o nome diz, servem para **"olhar para trás"** e buscar um trecho que case com o regex. Retrovisores são úteis em trechos repetidos em uma mesma linha.

Para compreender melhor a respeito dos retrovisores, que tal revisar um texto e descobrir quais são as palavras que se repetem para melhorar a escrita?

Regex: (quero)-\1

Descrição: Encontra duas palavras "quero" com um "-" entre elas.
String de teste: Hoje levei um susto de um quero-quero.
Resultado: Hoje levei um susto de um **quero-quero**.

Regex: ([A-Za-z]+)-?\1

Descrição: Qualquer palavra repetida com um "-" entre elas.
String de teste: Qualquer palavra repetida: quero-quero, bora-bora, vira-vira, Zé le-le.
Resultado: Qualquer palavra repetida: **quero-quero, bora-bora, vira-via**, Zé **le-le**.

Regex: ([A-Za-z]+) \1

Descrição: Encontra letras repetidas com um espaço entre elas.
String de teste: No dia a dia, estamos consultando a documentação ou saída de comando.
Resultado: No di**a a** dia, estamos consultando a documentaçã**o o**u saída de comando.

Regex: \b([A-Za-z]+) \1\b

Descrição: Encontra palavras repetidas com um espaço entre elas.
String de teste: Qualquer palavra palavra repetida deve deve ser encontrada.
Resultado: Qualquer **palavra palavra** repetida **deve deve** ser encontrada.

O regex somente pode ter, no máximo, nove retrovisores, então \10 será o retrovisor 1 seguido de um zero. Vamos ver alguns exemplos que utilizam mais de uma possibilidade de retrovisor.

Regex: (lenta)(mente)

Descrição: Encontra “lenta” ou “mente” em toda a string seguidos. Caso com \2 \1
String de teste: lentamente é mente lenta.
Resultado: **lentamente** é mente lenta.

Regex: ((band)eira)nte

Descrição: Encontra os dois grupos destacados mais um sufixo. \1 \2a
String de teste: bandeirante bandeira banda.
Resultado: **bandeirante** bandeira banda.

Regex: in(d)ol(or)

Descrição: Encontra os dois grupos mas com um radical. Sem \1\2
String de teste: indolor é sem dor.
Resultado: **indolor** é sem dor.

Regex: ((((a)b)c)d)-1 =

Descrição: Aqui temos 4 grupos que aparecem em sequência. \1,\2,\3,\4
String de teste: abcd-1 = abcd,abc,ab,a
Resultado: **abcd-1** = abcd,abc,ab,a

Quantificadores

Definição

Como já sabemos, os quantificadores indicam o número de repetições permitidas para entidade imediatamente anterior. Dito isso, vamos dividir os quantificadores em dois tipos:

Gulosos

Eles **não** são quantificáveis, logo não é possível utilizar dois deles seguidamente na regex.

Não gulosos

Eles são quantificáveis e buscam capturar o máximo possível de instâncias.

Quantificadores gulosos

Antes de prosseguirmos, vamos relembrar quais são os quantificadores:

Metacaractere	Nome	Função que representa
?	Opcional	Zero ou um
*	Asterisco	Zero, um ou mais
+	Mais	Um ou mais
{n,m}	Chaves	De n até m

Tabela: Tabela de quantificadores gulosos.
Alysson Messias.

Sabendo que os quantificadores gulosos **não** são quantificáveis, vamos entender na prática a razão de serem nomeados assim.

REGEX

```
Regex: <.*>
Descrição: Veja que o asterisco, como todo quantificador, é guloso e casou o máximo que conseguiu.
String de teste: Um <b>negrito</b> aqui.
Resultado: Um <b>negrito</b> aqui.
```

Para entendermos melhor essa gulodice dos quantificadores, vejamos o passo a passo desse exemplo e como o compilador executa a regex. Ao ler da esquerda para a direita, o regex procura primeiro o **“<”** . Vamos simular que, na leitura, **“.”** significará que não houve correspondência; e **“x”** significará que houve correspondência. Vamos lá.

REGEX

```
Regex: <.*>
um <b>negrito</b> aqui.
^
.^
..^
...^
...x^

Aqui tivemos o primeiro casamento, encontramos o “<”. Vamos passar para “.” que pode casar com qualquer caractere

...xx^
...xxx^
...xxxx^
...xxxxx^
```

Por causa dessa gulodice e da subsequente procura de trás para frente, acaba “se casando” além do desejado. Assim também funcionam todos os outros quantificadores: mais, chaves e opcional. Sempre casam o máximo possível.

Quantificadores não gulosos

Em alguns casos, a gulodice pode não atender ao que se espera. Por vezes, precisamos de um menor casamento possível. Dependendo da linguagem, essa opção pode variar na sua definição, mas, basicamente, basta acrescentar uma interrogação logo após os quantificadores normais, conforme apresentado na tabela a seguir:

Metacaractere	Nome
??	Opcional não guloso
*?	Asterisco não guloso

Metacaractere	Nome
+?	Mais não guloso
{n,m}?	Chaves não guloso

Tabela: Tabela de quantificadores não gulosos.
Alysson Messias.

O comportamento será ao contrário daquele que já detalhamos no tópico anterior. Para encerrar, vamos fazer uma comparação de comportamentos de alguns regex com as duas opções de quantificadores para o texto **“abbbb”**.

Gulosos	Resultado	Não gulosos	Resultado
ab*	**abbbb**	ab*?	**a**bbbb
ab+	**abbbb**	ab+?	**ab**bbb
ab?	**ab**bbb	ab??	**a**bbbb
ab{1,3}	**abbb**b	ab{1,3}?	**ab**bbb

Tabela: Tipos de quantificadores e comportamento de alguns regex.
Alysson Messias.

Encontrando a tag < P > < /P > com grupos e quantificadores.

Confira agora a expressão regular que destaca a TAG <P> </P> e todo seu conteúdo em um arquivo HTML.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Joana precisa encontrar as palavras que contêm dígrafos como “rr”,“ss”,“sc”,“ch” e “lh” para um trabalho escolar. Qual seria a regex mais adequada para resolver esse problema?

A (rr|ss|sc|ch|lh)

B (*.rr|ss|sc|ch|lh.*)

C ([A-z]rr|ss|sc|ch|lh[A-z])

D ([A-Za-z]*rr[A-Za-z])|([A-Za-z]*ss[A-Za-z])|([A-Za-z]*lh[A-Za-z])|([A-Za-z]*sc[A-Za-z])|([A-Za-z]*ch[A-Za-z])

E

([A-z]*)((rr)|(ss)|(sc)|(ch)|(lh))([A-z]*)

Parabéns! A alternativa E está correta.

Opção A está ERRADA – Encontra apenas os trechos que contêm os dígrafos.
Opção B está ERRADA – Até parece que vai funcionar, mas não casa com nenhum dígrafo.
Opção C está ERRADA – Pois da mesma forma como na letra A, somente apresenta dígrafos.
Opção D está ERRADA – Apesar de encontrar todas as palavras que contêm dígrafos, não busca todo o sufixo da palavra encontrada.
Opção E está CERTA – Atende a todas.

Questão 2

Uma empresa de desenvolvimento de aplicativos precisa fazer uma refatoração em seu código devido a alguns problemas de encode em sua página html. Para isso, ela precisa encontrar o que está dentro das tags . Qual a regex poderia ajudar os funcionários da empresa a encontrar apenas o conteúdo das tags div na seguinte página html?

<html>
<head>
<meta charset="UTF-8"/>
<title>Document</title>
</head>
<body>
<h1>Identificar tags</hi>
Agora®ï┼F
□Ñ*ª°.Ú
</body>
</html>

A (?<=>)([A-zÁ-ü])(?=<\/div)

B (?<=>)(.*)(?=<\/div)

C (?<=>)([*A-zÁ-ü*])(?=<\/div)

D (?<=>)([*.A-zÁ-ü.*])(?=<\/div)

E (?<=>)(A-zÁ-ü.*)(?=<\/div)

Parabéns! A alternativa B está correta.

As opções A, C, D e E estão ERRADAS - As definições de abertura e fechamento das tags estão corretas, porém o regex de pegar o conteúdo interno não alcança nenhuma ocorrência.
Opção B está CERTA – Pois casa com todo conteúdo dentro da tag div.

3 - Regex com JavaScript

Ao final deste módulo, você será capaz de elaborar regex com JavaScript.

Validação de dados com JS

Aprendemos até este momento que infinitas possibilidades podem ser alcanças utilizando o Regex. O suporte às expressões regulares, incluído na versão 1.2 de 1997, está presente na maioria dos navegadores, como Firefox, Google Chrome, Safari, Internet Explorer e Microsoft Edge.

Tanto o **JavaScript** quanto a linguagem **ActionScript** implementam o padrão ECMA-262 para expressões regulares. As definições abordadas no primeiro módulo servirão como base para ambas as linguagens, mas terão algumas modificações.

O JavaScript aceita vários elementos dos regex que foram definidos no início deste estudo, porém contém funções específicas e tratamentos particulares. As expressões mais populares são os métodos de string **search()** e **replace()**.

avaScript

É uma linguagem de programação para desenvolvimento web.

ctionScript

É uma linguagem de programação utilizada pelo Adobe Flash.

Comentário

O método **search( )** usa uma expressão para procurar uma correspondência e retorna a posição da correspondência. Por sua vez, o método **replace( )** retorna uma string modificada na qual o padrão é substituído.

No decorrer deste estudo, serão demonstrados os métodos e as funções mais utilizados e relevantes dentro da linguagem JavaScript, juntamente com as suas peculiaridades. Alguns browsers, como Chrome e Firefox, contêm um modo de debug para desenvolvedor com um console para acompanhar as saídas dos resultados das funções Javascript.

Para realizar testes e praticar o uso das funções, iremos demonstrar os exemplos utilizando o Quokkajs. Essa ferramenta interpreta o código e já traz o resultado. Pode ser instalada como extensão em alguns editores de desenvolvimento web, tais como: VSCode, JetBrainsIDEs, Atom Editor e Sublime Text. Para instalar o plugin do Quokkajs, basta acessar o endereço on-line: https://quokkajs.com/docs/#getting-started

Métodos em JS que utilizam regex

Conforme mencionado anteriormente, há uma série de métodos JavaScript que utilizam regex, cada qual com suas particularidades e retornos esperados. Observe a tabela a seguir.

Método	Objeto	Descrição	Retorno
exec	RegExp	Executa uma pesquisa por uma correspondência em uma string.	Array de informações.
test	RegExp	Testa uma correspondência em uma string.	True ou false.
match	String	Executa uma pesquisa por uma correspondência em uma string.	Array de informações ou null caso não haja uma correspondência.
search	String	Testa uma correspondência em uma string.	Índice da correspondência ou -1 se o teste falhar.
replace	String	Executa uma pesquisa por uma correspondência em uma string, e substitui a substring correspondente por uma substring de substituição.	String alterada.
split	String	Usa uma expressão regular ou uma string fixa para quebrar uma string dentro de um array de substrings.	Array de substrings.

Tabela: Tabela de JavaScript com regex.
Alysson Messias.

No Javascript, o objeto específico para tratar expressões regex é o objeto do tipo **RegExp**. Para usá-lo, basta criar uma nova instância e informar a expressão desejada. Nesse caso, temos duas opções. A **primeira**, é definir a expressão regular com uma variável utilizando os delimitadores "/", conforme o exemplo:

TUTORIAL COPIAR

Javascript

```
var texto = "TestanDo";//Texto alvo
var er = /[a-z]/; //Regex - letras minúsculas
let result = texto.match(er);//Função string para pesquisar o regex
console.log(result); //['e', index: 1, input: 'Testando', groups: undefined]
```

null

null

Exemplo 1.

A **segunda** opção é instanciar o objeto RegExp com a string da regex escolhida, utilize o trecho de código abaixo para atualizar o exemplo executável anterior e veja o resultado.

Javascript

```
er = new RegExp('[A-Z]'); //Regex - letras maiúsculas
result = texto.match(er);
console.log(result); //['T', index: 0, input: 'Testando', groups: undefined]
```

Exemplo 2.

A função **match( )** testa a expressão e, ao mesmo tempo, devolve informações sobre o casamento e um array com as informações.

Note que, nos exemplos anteriores, mesmo havendo mais de um caso de letras minúsculas e maiúsculas na string, apenas a primeira ocorrência foi retornada. Para mudar o comportamento e retornar todas as ocorrências, é necessário incluir no regex o casamento global, incluindo a letra g após a "/", ou adicionar um parâmetro no objeto RegExp:

TUTORIAL COPIAR

Javascript

```
texto = "TestanDo"; //Texto alvo
er = /[a-z]/g;
result = texto.match(er);
console.log(result); // ['e', 's', 't', 'a', 'n', 'o']
er = new RegExp('[A-Z]','g');
```

null

null

Exemplo 3.

O objeto RegExp tem apenas dois métodos, os quais são mais limitados que as opções dos métodos de string. Esses métodos são **test( )**, que retorna true/false ao casar a expressão, e **exec( )**, que retorna o trecho casado.

TUTORIAL COPIAR

Javascript

```
er = new RegExp('[A-Z]');
const regex1 = RegExp('foo*' ,"g");

const str1 = 'table football, foosball';
let array1;
```

null

null

Exemplo 4.

Para casos em que se deseja saber o índice no qual houve o casamento do regex para realizar alguma transformação da string, utilizamos a função **search( )**.

TUTORIAL COPIAR

Javascript

```
const texto='table football, foosball';

er = /[a-z]/;
result = texto.search(er);
if(texto.search(er) != -1) { //Retorna o index caso dê case com o alvo
```

null

```
} else {
    console.log("Não casou")
}
```

null

Exemplo 5.

Uma das funções mais populares é a **replace( )** que executa uma pesquisa por correspondência em uma string e substitui a substring correspondente por uma substring de substituição.

TUTORIAL COPIAR

Javascript

```
texto = "Podemos ir Andando e andando chegaremos mais longe"
result = texto.replace(/andando/gi,"correndo");
console.log(result); // Podemos ir correndo e correndo chegaremos mais longe
```

null

null

Exemplo 6.

Note que só foi possível o regex casar com a string “Andando” e “andando” por ter passado o parâmetro g (global), que retornou todos os casos, e o parâmetro i (ignore case), que trouxe letras maiúsculas e minúsculas. Esse exemplo demonstra uma possibilidade de realizar uma troca simples de um trecho de texto em uma string, mas o maior uso dessa função é realizar a formatação e apresentar o dado em uma informação mais amigável.

TUTORIAL COPIAR

Javascript

```
texto = "12:34"
result = texto.replace(/(..):(..)/,'$1h $2min'); //Formatação de horário
console.log(result); // 12h 34min
```

null

null

Exemplo 7.

Veja, a seguir, um exemplo combinando um trecho de código para tratamento do resultado. A função definida vai receber um número variável de argumentos que, dependendo do número de grupos de sua expressão, deve retornar uma string.

TUTORIAL COPIAR

Javascript

```
function data_por_extenso(texto_casado, grupo1, grupo2, grupo3) {

    var dia = grupo1;
    var mes = grupo2;
    var ano = grupo3;
```

null

null

Exemplo 8.

Devemos tomar cuidado com a acentuação, pois no Javascript não há suporte às classes POSIX como **[:upper:]** e **[:digit:]**. Para se adequar, pode-se utilizar o \w que casa com letras e números, mas sozinho não vai resolver. Nesse caso, é necessário incluir mais algumas definições:

Javascript

```
texto= "Amanhã é dia alegria.";
regex = /\w/g  //Apenas o barra-letra não é suficiente para casa com letras acentuadas
result = texto.replace(regex, '-');
console.log(result);  // -----ã é --- -------.

regex = /[\wÁ-ü]/g //Dessa forma vai casar com os caracteres acentuados
result = texto.replace(regex, '-');
console.log(result); // ------ - --- -------.
```

Exemplo 9.

A função **split( )** recebe um regex ou uma string fixa para quebrar a string alvo em um array de substrings. Como segundo argumento opcional, pode ser delimitada a quantidade de itens a serem devolvidos.

TUTORIAL COPIAR

Javascript

```
texto= "Comando 1;Ação 2;Tarefa 3;Resultado 4";
regex = /;/;
result = texto.split(regex);
console.log(result); // [ 'Comando 1', 'Ação 2', 'Tarefa 3', 'Resultado 4' ]
```

null

null

Exemplo 10.

Vamos agora analisar alguns casos misturando as funções vistas neste módulo. O exemplo a seguir utiliza a formação de expressões regulares para limpar uma string de entrada formatada com nomes (primeiro nome e sobrenome) separados por espaço em branco, tabulações, um ponto e vírgula e sendo acentuados ou não. Por fim, inverte a ordem do nome (sobrenome e primeiro nome) e ordena a lista.

TUTORIAL COPIAR

Javascript

```
// A cadeia de nomes contém vários espaços e guias,
// e pode ter vários espaços entre o nome e o sobrenome.
var names = "José Silva ; Heitor Vasconcelos; Cristina Magalhães ; Abel Belarmino ; João Pitágoras ";
var output = ["---------- String original\n", names + "\n"];
// pattern: possível espaço em branco, em seguida, ponto e vírgula, em seguida, possível espaço em branco
```

null

```
var nameList = names.split(pattern)
// novo pattern: um ou mais caracteres, espaços, caracteres acentuados ou não.
// Use parênteses para "memorizar" partes do padrão.
// As partes memorizadas são referenciadas mais tarde.
pattern = /([\wÁ-ü]+)\s+([\wÁ-ü]+)/
// Nova matriz para armazenar nomes sendo processados.
var bySurnameList = []
```

null

Exemplo 11.

No próximo exemplo, será verificado se é válida a formatação de um número de telefone informado juntamente com o seu dígito de área. Para isso, a expressão regular deverá casar com os seguintes formatos: ##-#####-#### ou ##-#####-####.

TUTORIAL COPIAR

Javascript

```
//A expressão regular procura por zero ou uma ocorrência de parênteses de abertura êíêê?,//seguidodetrêsdígitos
//guarda o caractere ([-\/\.]), seguido de três dígitos \d{4,5}, seguido por um caractere
//de hífen, barra ou ponto decimal que fora guardado \1, seguido por quatro dígitos \d{4}.
var re = /?\d2?([-\/\.])\d{4,5}\1\d{4}/;
function formataTelefone(phoneInput){
```

null

null

Exemplo 12.

Para finalizarmos este esudo, vamos observar um último exemplo que valida a formação de uma url e monta um regex para checar se a formação está correta. Importante recordar que estamos nos referindo a diversos de url, tais como: http, https e ftp.

Javascript

```
function is_url(str)
{
  regexp =  /^(?:(?:https?|ftp):\/\/)?(?:(?!(?:10|127)(?:\.\d{1,3}){3})(?!(?:169\.254|192\.168)(?:\.\d{1,3}){2})(
      if (regexp.test(str))
      {
        return true;
      }
      else
```

```
        {
            return false;
        }
}

console.log(is_url("http://www.google.com")); // true
```

Exemplo 13.

Para maiores detalhes, consulte a página do Mozilla que mantém uma documentação detalhada sobre o assunto: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Regular_Expressions

Em projetos maiores, nos quais esses tratamentos se repetem, é muito comum que sejam implementadas bibliotecas úteis, as quais são reaproveitadas por todo o projeto para não haver necessidade de escrever o código de validação novamente. No mercado, já existem várias bibliotecas com funções de formatação encapsuladas que resolvem muitos desses problemas, como as seguintes:

- MommentJS – para formatação de Datas.
- You Dont Need MomentJS – para formatação de Datas.
- Mask JS – formatação de CPF, telefone, CNPJ e outros dados.

Validando um CPF

Confira agora o código que valida um CPF utilizando a biblioteca Mask.js.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

O bloco de código javaScript abaixo possui a função contaVogais e a saída do seu resultado. Qual alternativa contém o trecho de código adequado que substitui a linha 2 e cujo método pode contabilizar todas as vogais, tanto as acentuadas quanto as não acentuadas?

```
1. function contaVogais(str) {
2. ...
3. }
4. console.log(contaVogais('O conteúdo desse curso está incrível!')); // 14
```

A return str.match(/[aeiou]/gi).length

B return str.match(/[\wÁ-ü]/gi).length

C return str.match(/[\w]/gi).length

D return str.match(/[aeiouÁ-ü]/g).length

E return str.match(/[aeiouÁ-ü]/gi).length

Parabéns! A alternativa E está correta.

Opção A está ERRADA – Retorna as vogais não acentuadas, maiúsculas e minúsculas.
Opção B está ERRADA – Retorna todos os caracteres acentuados, não acentuados, maiúsculos e minúsculos.
Opção C está ERRADA – Retorna todos os caracteres não acentuados, maiúsculos e minúsculos.
Opção D está ERRADA – Retorna as vogais acentuadas, não acentuadas e apenas minúsculas.
Opção E está CERTA – Retorna as vogais acentuadas, não acentuadas, maiúsculas e minúsculas. O argumento i inclui ambas as caixas.

Questão 2

A empresa XPTO contém um formulário para cadastro de seus usuários, porém a verificação do campo e-mail está com problema. Qual regex deve ser atribuído à variável mailformat na linha 3 para corrigir o problema dessa validação?

```
1. function emailValido(str)
2. {
3. var mailformat = ...
4. if(mailformat.test(str)){
5. console.log("Email válido!");
6. } else {
7. console.log("O email informado não é um email válido!");
8. }
9. }
```

A	/^[\wÁ-ü]+([\.-]?[\wÁ-ü]+)*@[\wÁ-ü]+([\.-]?[\wÁ-ü]+)*(\.[\wÁ-ü]{2,3})+$/
B	/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\;\w{2,3})+$/
C	/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/
D	/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)/
E	/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\d{2,3})+$/

Parabéns! A alternativa C está correta.

Opção A está ERRADA – Permite e-mail com caracteres acentuados.
Opção B está ERRADA – Permite e-mail com ; dividindo a url.
Opção C está CERTA – Obrigação de inclusão do @ e limitação do tamanho do domínio entre 2 e 3 caracteres não dígitos.
Opção D está ERRADA – Aceita que o último caractere de domínio tenha uma quantidade ilimitada de caracteres. Nesse caso, o tamanho só pode ser 2 ou 3 (br,com).
Opção E está ERRADA – Obriga a colocar dígitos no domínio, na última parte.

4 - Regex com Python

Ao final deste módulo, você será capaz de elaborar regex com Python.

Conceitos básicos

Python é uma linguagem de programação de alto nível, dinâmica, interpretada, modular, multiplataforma e orientada a objetos. Foi idealizada e desenvolvida no início dos anos 90 por Guido Van Rossum, matemático holandês.

Comentário

O objetivo do Python é otimizar a leitura de códigos e estimular a produtividade de quem os cria, seja um programador seja qualquer outro profissional. Possui um dos mais completos suportes para as expressões regulares, com objetos e métodos já prontos para obter diversas informações sobre os casamentos.

As expressões estudadas anteriormente serão aproveitadas aqui. A proposta deste estudo é demonstrar a aplicação dos regexs em harmonia com os comandos específicos da linguagem Python. Para trabalhar com expressões regulares, Python contém um pacote específico e disponível no módulo re, que para ser utilizado dentro do código precisa, primeiramente, ser importado:

Python

```
1 import re
```

Esse pacote contém os métodos mais utilizados para manipular expressões regulares na linguagem: **re.match, re.search, re.findall, re.finditer e re.sub**. Os padrões das expressões regulares são compilados em uma série de bytecodes, que são então executadas por um mecanismo de combinação escrito em C.

Recomendação

Para realizar os testes e praticar o uso das funções, existem algumas opções. Você pode instalar o Python na sua máquina seguindo os seguintes tutoriais de instalação encontrados nos sites da Linux, Windows ou Mac.

Alguns editores de código de desenvolvimento, como VSCode e Sublime, possuem plugins ou extensões que dão suporte à linguagem. Você pode utilizar o Jupyter Notebook ou, caso apenas queira ver os resultados e praticar os regex, outras opções são os editores Python web, como Replit, Online Python e Programiz.

Métodos que utilizam regex

Para definir as regex, é possível utilizar strings normais, porém é recomendável colocar suas expressões dentro de **raw strings (r' ... ')** para torná-las cruas, evitando problemas com escapes. Observe a tabela a seguir:

Método	Descrição	Retorno
match	Determina se a expressão regular combina com o início da string.	Posição da string buscada.
findall	Encontra todas as substrings que têm correspondência com a expressão regular, e as retorna como uma lista.	Lista com as strings encontradas.
finditer	Encontra todas as substrings que têm correspondência com a expressão regular.	Lista das strings como um iterador.
search	Varre toda a string, procurando o primeiro local em que essa expressão regular tem correspondência.	Posição da string buscada.
sub	Procura o local em que essa expressão regular tem correspondência e faz as substituições desejadas.	A string com a substituição realizada.

Tabela: Tabela de Pynthon e função regex.
Alysson Messias.

Essa notação trata as barras invertidas (\) como caracteres literais. Acostume-se a sempre usar essa notação mesmo quando sua expressão for simples e não possuir nenhuma contrabarra.

TUTORIAL COPIAR

Python3

```
import re

string = "frase que será analisada pelo regex"
result = re.match(r'frase', string)
print(result)  # <_sre.SRE_Match object; span=(0, 5), match='frase'>
```

null

null

Exemplo 1.

Note que o regex pode ser informado tanto com string quanto com as raw strings. Importante notar que a função **match( )** só busca a correspondência do regex no início da string.

Caso queira utilizar o regex para encontrar em qualquer lugar no texto, a opção a ser utilizada será a função **search( )**.

TUTORIAL COPIAR

Python3

```
import re

string = 'Teste regex Python'
m = re.search(r'Py', string);
if m:
```

null

null

Exemplo 2.

Lembre-se de que a função **search( )** retorna apenas a primeira ocorrência.

TUTORIAL COPIAR

Python3

```
import re

string = 'Teste Python regex Python'
m = re.search(r'Py', string);
print(m)  # <_sre.SRE_Match object; span=(6, 8), match='Py'>
```

null

null

Exemplo 3.

O método **search( )**, quando não casa com o regex, retorna None se nenhuma posição na string corresponder ao padrão. Observe que isso é diferente de encontrar uma correspondência de comprimento zero em algum ponto da string. Quando encontra, retorna um objeto correspondência MatchObject, o qual, se utilizado em uma expressão ternária, tem um valor booleano true.

Com o objeto de retorno em mãos, como manipular os valores? Na tabela a seguir vamos detalhar os principais métodos de utilização **group( ), start( ), end( ) e span( )**.

Método MatchObject	Descrição	Retorno
group( )	Traz o trecho de texto casado pela expressão.	Um ou mais subgrupos da correspondência.
start( )	Obtém a posição de início do trecho casado na string original.	Índice da substring encontrada e -1 se não houver correspondência.
end( )	Obtém a posição de fim do trecho casado na string original.	Índice da substring encontrada e -1 se não houver correspondência.
span( )	Traz ambas as posições dentro de uma tupla.	Índice da substring encontrada e 0 se não houver correspondência.
expand( )	Converte caracteres apropriados, referências anteriores numéricas (\1, \2) e referências	String obtida fazendo a substituição da

Método MatchObject	Descrição	Retorno
	anteriores nomeadas (\g<1>, \g<nome>) em conteúdo do grupo correspondente.	contrabarra na string modelo template.

Tabela: Tabela método matchObject.
Alysson Messias.

Vale lembrar que os índices na linguagem Python iniciam em zero. Vamos ver alguns exemplos de uso.

TUTORIAL COPIAR

Python3

```
import re

string = 'De grão em grão, a galinha enche o papo'
m = re.search(r'g[A-z]l[A-z]+', string)
if m:
```

null

null

Exemplo 4.

Veja algumas variações do uso dos métodos:

TUTORIAL COPIAR

Python3

```
import re

m = re.search(r'(..)/(..)/(....)','05/05/2022')
if m:
  print(m.group(0))  # 05/05/2022
```

null

null

Exemplo 5.

Se o objetivo for capturar todas as incidências do regex na string, o método escolhido deverá ser o **findall( )**, o qual, ao encontrar correspondência da expressão, devolverá uma lista de **strings [ ]**, mas não retorna os índices onde foram encontrados na string

TUTORIAL COPIAR

Python3

```
import re

string = 'De médico e Louco, todo mundo tem um pouco'
m = re.findall(r'[A-z]ou[A-z]+', string);
print(m)              # ['Louco', 'pouco']
```

null

null

Exemplo 6.

Em casos de múltiplas ocorrências e em que seja necessário realizar alguma lógica ou conversão a cada resultado encontrado, a melhor opção é a função **finditer( )**. Você pode inspecionar cada ocorrência de métodos já estudados, como **group( )** e **span( )**.

TUTORIAL COPIAR

Python3

```
import re

texto= "Acordei às 08:00, comi 12:30, dormi às 23:59."
for m in re.finditer(r'(\d\d):(\d\d)', texto):
 hora = m.group(1)
```

null

null

Exemplo 7.

Para o exemplo anterior, pensando em formatação de apresentação dos dados, ficaria mais simples se fosse utilizado o método **expand( )**, no qual houve uma formatação dentro do laço de repetição. Vamos substituir e verificar o resultado:

TUTORIAL COPIAR

Python3

```
import re
texto= "Acordei às 08:00, comi 12:30, dormi às 23:59."
for m in re.finditer(r'(\d\d):(\d\d)', texto):
 print(m.expand(r'\1 horas, \2 minutos.'))

```

null

null

Exemplo 8.

Faz-se um uso mais comum da regex para realizar formatação em um campo. Nesse caso, o método **sub( )** será valioso porque substitui todas as ocorrências encontradas e aceita um terceiro argumento opcional para limitar o número de substituições a serem feitas.

TUTORIAL COPIAR

Python3

```
import re

texto= "08:00"
print(re.sub(r'\w', '.', 'Python'))     # ......
print(re.sub(r'\w', '.','Python', 2))   # ..thon

```

null

null

Exemplo 9.

Observe agora um exemplo que demonstra o uso dentro do código sendo aplicado com demais lógicas.

TUTORIAL COPIAR

Python3

```
import re

def data_por_extenso(m):
 dia = m.group(1)
 mes = m.group(2)
```

null

null

Exemplo 10.

Um problema recorrente quando se trabalha com dados é receber arquivos de texto no formato CVS. Normalmente, as informações são provenientes de algum dataset ou de uma planilha. Em comum, essas informações vêm tabuladas e com delimitadores entre as colunas para dividi-las. Para extrair as informações e manipular os dados, a função mais apropriada é a **split( )**.

TUTORIAL COPIAR

Python3

```
import re

texto = '''Método, descrição
teste(),Testa valores
split(),Separa strings''' #''' são usados para declarar string com quebra de linha
```

null

null

Exemplo 11.

Com relação à otimização do código, no caso de utilizar o regex várias vezes, é possível compilá-lo para garantir uma execução mais rápida e menos uso de memória. O objeto retornado da compilação tem os mesmos métodos do módulo re, então você pode casar, substituir e dividir os textos usando expressões regulares compiladas.

TUTORIAL COPIAR

Python3

```
import re

#Primeiro compile as expressões
er_hora = re.compile(r'(\d\d):(\d\d)')
er_separador = re.compile(r'[/.:]')
```

null

null

Exemplo 12.

Existem algumas flags que ajudam na montagem dos regex. Por exemplo, **\i** retorna a string independentemente se for minúscula ou maiúscula; **\w** retorna as letras acentuadas. Veja alguns exemplos de flags que podem ser úteis.

TUTORIAL COPIAR

Python3

```
import re

# Números de telefone no formato internacional
texto = '''
+554798765432
```

null

null

Exemplo 13.

Os métodos de **MatchObject** aceitam o nome ou o número do grupo. Desse modo, você também pode dar nomes aos grupos de sua expressão. Assim, além dos números, é possível referenciar esses grupos usando o seu nome.

TUTORIAL COPIAR

Python3

```
import re

# Converter datas de DD/ MM/ AAAA para AAAA-MM-DD
data= '10/05/2022'
# Substituição usando grupos normais
```

null

null

Exemplo 14.

Validando um CNPJ

Confira agora a valiadação de um CNPJ utilizando expressões regulares em Python.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Veja abaixo um bloco de código Python que verifica se uma string contém apenas o seguinte conjunto de caracteres (a-Z,A-Z e 0-9). Qual das alternativas abaixo pode ser utilizada para substituir a variável regex na linha 3 para que essa verificação aconteça de forma correta?

```
1. import re
2. def validaExpressao(string):
3.     charRe = re.compile(regex)
4.     string = charRe.search(string)
5.     return not bool(string)
6.
7. print(validaExpressao("ABCDEFabcdef123450")) # True
8. print(validaExpressao("*&%@#!}{"))       # False
```

A r'[a-zA-Z0-9]'

B r'[^a-zA-Z0-9]'

C r'[A-z0-9]'

D r'[\w0-9]'

E r'[\w\d]'

Parabéns! A alternativa B está correta.

As opções A, C, D e E estão ERRADAS – Contêm todos os conjuntos detalhados. A linha 5 tem uma negação na saída, o que faz com que essa expressão não atenda à questão.
Opção B está CERTA – Contém todos os conjuntos detalhados. A linha 5 tem uma negação na saída do array contido no regex, que significa uma negação ^.

Questão 2

Maria precisa criar um campo de código que permita apenas conter no início a letra a seguida de 3 letras b (abbb). Qual das alternativas abaixo pode ser utilizada para substituir a variável regex na linha 3 para validar que essa formação seja respeitada?

```
1. import re
2.    def text_match(text):
3.        patterns = regex
4.        if re.search(patterns, text):
5.            return 'Válido'
6.        else:
7.            return('Inválido')
8.
9. print(text_match("abbb"))
10. print(text_match("aabbbbbc"))
```

A r'ab{3}?'

B r'^ab'

C r'^ab{3}?'

D r'^ab? '

E r'^ab*'

Parabéns! A alternativa C está correta.

Opção A está ERRADA – Obriga o uso da letra a na presença de 3 b em sequência, porém não força estar no começo da string. Pode estar em qualquer parte do alvo.
Opção B está ERRADA – Obriga começar com ab, porém não obriga o uso da letra b 3x em sequência.

Opção C está CERTA – Além de obrigar o uso da letra a na presença de 3 b em sequência, também força estar no começo da string ^.

Opções D e E estão ERRADAS – Situação semelhante à da letra B.

Considerações finais

As expressões regulares não são complicadas, mas podem ficar complexas para alcançar os objetivos esperados. O importante é decompor o problema e encontrar pequenas expressões para resolver determinadas partes e, em seguida, montar e agrupar as expressões. Dessa maneira, fica bem mais fácil entender a montagem do regex.

A melhor forma de fixar a aprendizagem é praticando. Para isso, é preciso montar o regex para diversas situações e utilizar cada vez mais os recursos oferecidos pelas expressões regulares.

Para encerrar, ouça sobre a importância das expressões regulares na programação do front-end.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.

Referências

EXPRESSÕES regulares. Mozilla. Mdn. Consultado na internet em: 5 maio 2022.

FITZGERALD, M. **Introdução às expressões regulares**. São Paulo: Novatec, 2012.

JARGAS, A. M. **Expressões regulares**: uma abordagem divertida. Novatec, 2012.

KUCHLING, A. M. **Expressões regulares HOWTO**. PYTHON. Consultado na internet em: 5 maio 2022.

Explore +

Acesse os seguintes sites de referência para consulta e apoio para realizar a montagem das expressões regulares:

- Aurelio.net

- W3resource.com
- Python.org
- Developer.mozilla.org