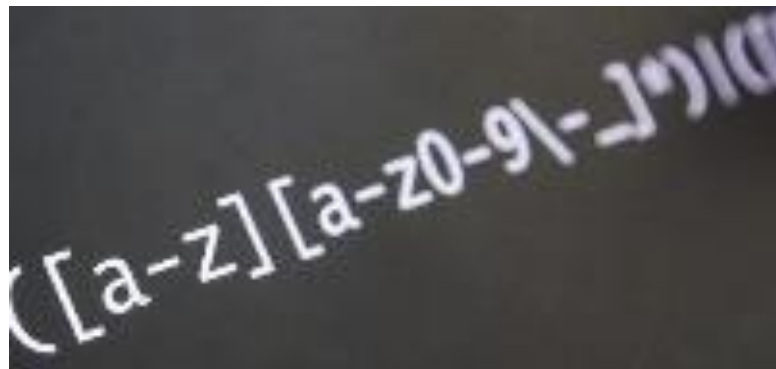


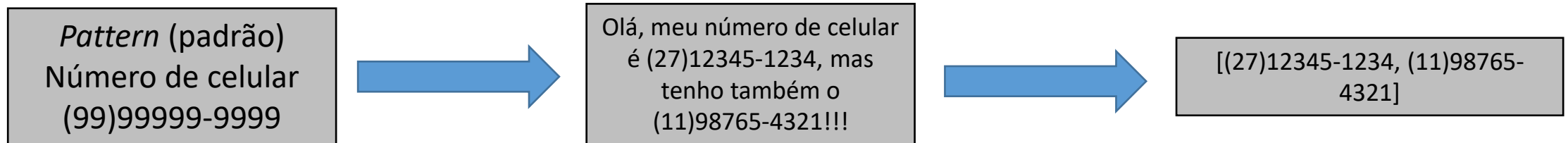
# Expressões Regulares

## Introdução



# Expressões Regulares

Expressões regulares (Regex, Regular Expression) são escritas em uma linguagem formal que pode ser interpretada por um processador de expressão regular, um programa que analisa um texto e identifica as partes que casam com a expressão informada. Expressões regulares são utilizadas para combinar padrões de texto.



# Expressões Regulares

O módulo “re” do Python fornece suporte a expressões regulares.

Em Python, uma pesquisa de expressão regular geralmente é escrita desta forma:

**`variável = re.função(padrão, string, flag)`**

# Expressões Regulares

## Exemplo em Python

```
# Importando o módulo re
import re

# Definindo o padrão usando uma expressão literal
padrao = "aula"

# Definindo o texto
texto = "Esta é uma aula de Python. Nesta aula vamos falar sobre expressões regulares."

# Usando o search para pesquisar o texto que coincide com o padrão
resultado = re.search(padrao, texto)

# Imprime o match object
print(resultado)

# Imprime o texto encontrado
print(resultado.group())
```

# Expressões Regulares

## Exemplo em Python

```
# Importando o módulo re
import re

# Definindo o padrão usando uma expressão literal
padrao = "abacaxi"

# Definindo o texto
texto = "Esta é uma aula de Python. Nesta aula vamos falar sobre expressões regulares."

# Usando o search para pesquisar o texto que coincide com o padrão
resultado = re.search(padrao, texto)

# Imprime o match object
print(resultado)

# Imprime o texto encontrado
print(resultado.group())
```

### None

Traceback (most recent call last):  
File "/exp2.py", line 17, in <module>  
print(resultado.group())  
AttributeError: 'NoneType' object has no attribute 'group'

```
if resultado:
    print(resultado.group())
else:
    print("Nenhuma informação encontrada.")
```

# Expressões Regulares

## Principais caracteres especiais (meta-caracteres):

- Ponto “.”: Corresponde a qualquer caractere, menos o de nova linha “\n”. Se a flag DOTALL for especificada, isso corresponde a qualquer caractere que inclua uma nova linha.

# Expressões Regulares

## Principais caracteres especiais (meta-caracteres):

- Circunflexo “^”: Início da string. No modo MULTILINE também coincide imediatamente após cada nova linha.

Exemplo:

Encontre na string “Olá, esta é uma aula de Python. Olá, Espero que goste desta aula.” ocorrências de “^Olá”.

Vai localizar a palavra aula que está no início da string.

# Expressões Regulares

## Principais caracteres especiais (meta-caracteres):

- Cifrão “\$”: Fim da string.

Exemplo:

Encontre na string “Olá, esta é uma aula de Python. Espero que goste desta aula” ocorrências de “aula\$”.

Vai localizar a palavra aula que está no final da string.



# Expressões Regulares

## **Principais caracteres especiais (meta-caracteres):**

- Contra-barra “\”: Caractere de escape, permite usar caracteres especiais como se fossem comuns.

# Expressões Regulares

## Principais caracteres especiais (meta-caracteres):

- Colchetes “[ ]”: Qualquer caractere dos listados entre colchetes (range). Os caracteres podem ser listados individualmente [abc], combinando “a”, “b” e “c”, ou sequência como [a-z]. Para usar o “-” literalmente, temos que usá-lo no início ou fim da expressão.

# Expressões Regulares

## Principais caracteres especiais (meta-caracteres):

- Asterisco “\*”: Zero ou mais ocorrências da expressão anterior.

# Expressões Regulares

## **Principais caracteres especiais (meta-caracteres):**

- Mais “+”: Uma ou mais ocorrências da expressão anterior.

# Expressões Regulares

## **Principais caracteres especiais (meta-caracteres):**

- Interrogação “?”: Zero ou uma ocorrência da expressão anterior.

Utilizado quando o caractere não é obrigatório.

# Expressões Regulares

## Principais caracteres especiais (meta-caracteres):

- `\d`: Dígito. Equivale a `[0-9]`.
- `\D`: Não dígito. Equivale a `[^0-9]`.
- `\s`: Qualquer caractere de espaçamento “`[ \t\n\r\f\v]`”.
- `\S`: Qualquer caractere que não seja de espaçamento “`[^ \t\n\r\f\v]`”.
- `\w`: Caractere alfanumérico ou sublinhado “`[a-zA-Z0-9_]`”.
- `\W`: Caractere que não seja alfanumérico ou sublinhado “`[^a-zA-Z0-9_]`”.
- Chaves “`{n}`”: “N” ocorrências da expressão anterior.
- Barra vertical ou pipe “`|`”: ‘Ou’ lógico.
- Parênteses “`()`”: Delimitam um grupo de expressões.

# Expressões Regulares

As funções do módulo *re* aceitam uma *string* representando a expressão regular.

Expressões regulares usam a barra invertida ('\') para indicar formas especiais ou para permitir a utilização de caracteres especiais como se fossem caracteres comuns.

Esse comportamento conflita com o uso da ('\') no *Python*, que utiliza este caractere para o mesmo propósito em *strings* literais. Como exemplo, temos o "\b" que é o caractere *ASCII backspace*)

Para resolver este conflito, recomenda-se prefixar a *string* com r'...' para indicar uma *raw string* (*string* crua), evitando estes conflitos entre as sequências de escape de Python.

```
>>> print("\nOlá \bMundo")
Olá Mundo
>>> print(r"\nOlá \bMundo")
\nOlá \bMundo
```

```
>>> print("Expressões\b\b\b Regulares")
Express Regulares
```

Se quiséssemos representar uma barra-invertida teríamos que fazer assim:  
"C:\\Usuários\\Evaldo\\Downloads"  
Com raw string usamos:  
r"C:\\Usuários\\Evaldo\\Downloads"

# Expressões Regulares

## Funções do módulo re:

- ***re.compile(pattern, flags=0)***: Compila o *pattern* (padrão).

pattern: padrão, ou expressão regular avaliada.

Flags: re.IGNORECASE (re.I), re.LOCALE (re.L), re.MULTILINE (re.M), re.DOTALL (re.S), re.UNICODE (re.U), re.VERBOSE (re.X).

Retorna: re.RegexObject

- ***re.findall(pattern, string, flags=0)***: Procura ocorrência do *pattern* dentro da *string* e retorna os valores em forma de lista.

pattern: padrão, ou expressão regular avaliada.

Flags: re.IGNORECASE (re.I), re.LOCALE (re.L), re.MULTILINE (re.M), re.DOTALL (re.S), re.UNICODE (re.U), re.VERBOSE (re.X).

Retorna: lista.



# Expressões Regulares

## Veja algumas flags:

re.IGNORECASE (re.I): Não faz distinção entre maiúsculas e minúsculas.

re.DOTALL (re.S): Faz com que o caractere especial “.” combine qualquer caractere, incluindo uma nova linha, sem esta flag, o “.” irá combinar qualquer coisa, exceto uma nova linha.

re.VERBOSE (re.X): Permite que você escreva expressões regulares mais agradáveis e que sejam mais legíveis, permitindo que adicione comentários. O espaço em branco dentro do padrão é ignorado, exceto quando está em uma classe de caracteres ou quando é precedido por uma barra invertida (“\ ”). Quando uma linha contém um # que não está em uma classe de caracteres e não esteja precedido de uma barra invertida (“\#”), ele próprio e todos caracteres até o final da linha são ignorados.

```
import re
a = re.compile(r"""\d +   # Parte inteira
               \.       # Ponto decimal
               \d *     # Alguns dígitos fracionários""", re.X)
b = re.compile(r"\d+\.\d*")
```

# Expressões Regulares

Exemplos:

“[0-9]” Qualquer número entre 0 e 9 (cada ocorrência individual).

“[0-9]+” Qualquer número entre 0 e 9 (várias ocorrências).

---

```
cmd Command Prompt - python
>>> import re
>>> pattern = re.compile(r"[0-9]")
>>> texto = "Olá, temos 58 números 67 espalhados 47 neste texto 14. Mas tem negativos -7 e mais -48."
>>> re.findall(pattern, texto)
['5', '8', '6', '7', '4', '7', '1', '4', '7', '4', '8']
>>>
>>> pattern = re.compile(r"[0-9]+")
>>> re.findall(pattern, texto)
['58', '67', '47', '14', '7', '48']
>>> █
```

# Expressões Regulares

Exemplos:

“(segunda | terça | quarta | quinta | sexta)-feira”

Qualquer ocorrência de segunda-feira,  
terça-feira...sexta-feira

```
>>> pattern = re.compile(r"(segunda|terça|quarta|quinta|sexta)-feira")
>>> sambadotrabalhador = "Na segunda-feira eu não vou trabalhar Na terça-feira não vou pra poder descansar Na quarta preciso me recuperar Na quinta eu acordo meio-dia, não dá Na sexta viajo pra veranear No sábado vou pra mangueira sambar Domingo é descanso e eu não vou mesmo lá Mas todo fim de mês chego devagar Porque é pagamento eu não posso faltar"
>>> re.findall(pattern, sambadotrabalhador)
['segunda', 'terça']
>>> _
```

# Expressões Regulares

Exemplos:

“ana” Usa a string literal “ana”. Usando a flag IgnoreCase, também localizou a string “Ana”

```
>>> pattern = re.compile('ana')
>>> texto = "Ana adora ouvir chiclete com Banana, gosta de bananada e também banana, ela é irmã da Mariana."
>>> re.findall(pattern, texto)
['ana', 'ana', 'ana', 'ana']
```

```
>>>
>>> pattern = re.compile('ana', re.I)
>>> re.findall(pattern, texto)
['Ana', 'ana', 'ana', 'ana', 'ana']
>>>
```

# Expressões Regulares

Veja um exemplo, utilizando “.” no padrão:  
“.ato” = qualquer\_caracter + ato.

```
import re

padrao = re.compile(r".ato")
texto = ("Eu tenho um gato que corre de rato,"
        "foge para o mato, aí eu pego o sapato "
        "e bato ele no ato.")
resultado = re.findall(padrao, texto)
print(resultado)
```

Resultado:  
['gato', 'rato', 'mato', 'pato', 'bato', ' ato']

# Expressões Regulares

## Exemplo *re.findall()* com *re.IGNORECASE*:

```
import re

padrao = re.compile(r".ato")
padrao2 = re.compile(r".ato", re.I)
texto = ("Eu tenho um Gato que corre de RATO,"
         "foge para o MATO, aí eu pego o sapato "
         "e bato ele no ato.")
resultado = re.findall(padrao, texto)
print("Sem IGNORECASE:", resultado)
resultado = re.findall(padrao2, texto)
print("Com IGNORECASE:", resultado)
```

Sem IGNORECASE: ['Gato', 'pato', 'bato', ' ato']

Com IGNORECASE: ['Gato', 'RATO', 'MATO', 'pato', 'bato', ' ato']

# Expressões Regulares

## Funções do módulo re:

- ***re.search(pattern, string, flags=0)***: Procura a ocorrência do *pattern* (padrão) dentro da *string* e retorna um objeto que corresponda a este padrão (*match object*).

pattern: padrão, ou expressão regular avaliada.

string: texto em que será feita a pesquisa.

Flags: re.IGNORECASE (re.I), re.LOCAL (re.L), re.MULTILINE (re.M), re.DOTALL (re.S), re.UNICODE (re.U), re.VERBOSE (re.X).

Retorna: re.MatchObject ou None

- ***re.match(pattern, string, flags=0)***: Procura a ocorrência do pattern no início da string. Se o pattern casar o início da string, a função retorna o re.MatchObject correspondente, senão, retorna None.

pattern: padrão, ou expressão regular avaliada.

Flags: re.IGNORECASE (re.I), re.LOCAL (re.L), re.MULTILINE (re.M), re.DOTALL (re.S), re.UNICODE (re.U), re.VERBOSE (re.X).

Retorna: re.MatchObject ou None

# Expressões Regulares

## Exemplo *re.search()* e *re.match()*:

```
import re

padrao = re.compile(r".ato")

texto = ("Eu tenho um Gato que corre de RATO,"
         "foge para o MATO, aí eu pego o sapato "
         "e bato ele no ato.")
resultado = re.search(padrao, texto)
print(resultado)

texto = ("Eu tenho um Gato que corre de RATO,"
         "foge para o MATO, aí eu pego o sapato "
         "e bato ele no ato.")
resultado = re.match(padrao, texto)
print(resultado)

texto = ("Gato que corre de RATO,"
         "foge para o MATO, aí eu pego o sapato "
         "e bato ele no ato.")
resultado = re.match(padrao, texto)
print(resultado)
```

Resultado dos prints:

<\_sre.SRE\_Match object; span=(12, 16), match='Gato'>

None

<\_sre.SRE\_Match object; span=(0, 4), match='Gato'>

Veja que na segunda execução retornou None, porque não foi encontrada uma string que correspondesse ao padrão informado logo no início do texto.

Já na terceira execução, foi encontrada.



# Expressões Regulares

## Localizando CPFs/CNPJs em um texto:

Pattern: `([0-9]{2}[\.]?[0-9]{3}[\.]?[0-9]{3}[/]?[0-9]{4}[-]?[0-9]{2})|([0-9]{3}[\.]?[0-9]{3}[\.]?[0-9]{3}[-]?[0-9]{2})`

```
# CNPJ: ([0-9]{2}[\.]?[0-9]{3}[\.]?[0-9]{3}[/]?[0-9]{4}[-]?[0-9]{2})
# Ou: |
# CPF: ([0-9]{3}[\.]?[0-9]{3}[\.]?[0-9]{3}[-]?[0-9]{2})
```

- `[0-9]{2}` = Faixa de caracteres: 0 a 9, quantidade: 2 caracteres;
- `[0-9]{3}` = Faixa de caracteres: 0 a 9, quantidade: 3 caracteres;
- `[0-9]{4}` = Faixa de caracteres: 0 a 9, quantidade: 4 caracteres;
- `[\.]?` = Um ponto, opcional.
- `[-]?` = Um traço, opcional (se acrescentar outros caracteres, comece pelo - sempre);
- `[/]?` = Uma barra, opcional;
- `(grupo1)|(grupo2)` Usando o OU lógico. Se um dos grupos validar, a expressão é válida.

Arquivo exemplo: **`expressao_cpfcnpj.py`**

# Expressões Regulares

## Localizando E-mails em um texto:

Pattern: `[\w.-]+@[\w.-]+`

`\w` -> Caractere alfanumérico ou sublinhado `[a-zA-Z0-9_]`  
`.` -> Considera o ".". Devido ao ponto, foi encontrado "evaldo.wolkers@algumacoisa.com.br", se não permitisse o ponto, seria retornado 'wolkers@algumacoisa.com.br'.  
`-` -> Considera o "-". Devido ao hífen, foi encontrado "evaldo-wolkers@algo.com.br", se não permitisse o hífen, seria retornado 'wolkers@algo.com.br'.  
`+` -> Uma ou mais ocorrências da expressão anterior `[\w.-]`  
`@` -> Considera um caractere arroba.

**Arquivo exemplo: `expressao_email.py` e `expressao_email2.py`**

Os colchetes "[ ]" servem para indicar alternativas.  
`[abcdef]` procura por "a" ou "b"... ou "f".

# Expressões Regulares

## Localizando datas em um texto:

Pattern: `\d{2}/\d{2}/\d{4}`

`\d{x}` - Dois/Quatro números  
/ - Barra (literal)

*Arquivo exemplo: `expressao_data.py`*

# Expressões Regulares

## Usando split

```
import re

texto = "Evaldo|Maria|Joaquina|Cirilo|Didi|Mussum|Tarzan"
print(re.split("\|", texto))
```

```
['Evaldo', 'Maria', 'Joaquina', 'Cirilo', 'Didi', 'Mussum', 'Tarzan']
```

# Expressões Regulares

Utilizando grupos para “quebrar” o e-mail localizado, pegando o usuário e o domínio separados:

`([\w.-]+)@([\w.-]+)`

```
import re
texto = ("Meu principal e-mail é evaldowolkers@gmail.com, mas "
        "tenho também o evaldorw@hotmail.com, e o que dizer do "
        "evaldo.wolkers@algunacoisa.com.br? Este eu ainda não "
        "tenho. Que tal também o evaldo-wolkers@algo.com.br?")
resultado = re.search(r'([\w.-]+)@([\w.-]+)', texto)
print("resultado.group():", resultado.group())
print("resultado.group(1):", resultado.group(1))
print("resultado.group(2):", resultado.group(2))

resultado = re.findall(r'([\w.-]+)@([\w.-]+)', texto)
print(resultado)
for email in resultado:
    print(email)
```

Tudo que estiver entre (), quando executarmos o `re.search()` pode ser acessado com `groups()`.

```
resultado.group(): evaldowolkers@gmail.com
resultado.group(1): evaldowolkers
resultado.group(2): gmail.com
```

```
[('evaldowolkers', 'gmail.com'), ('evaldorw', 'hotmail.com'), ('evaldo.wolkers',
'algunacoisa.com.br'), ('evaldo-wolkers', 'algo.com.br')]
```


```
('evaldowolkers', 'gmail.com')
('evaldorw', 'hotmail.com')
('evaldo.wolkers', 'algunacoisa.com.br')
('evaldo-wolkers', 'algo.com.br')
```

**Arquivo exemplo: `expressao_group.py`**

# Expressões Regulares

Editor de expressões regulares online

Testando expressões regulares em <https://pythex.org>

 | pythex.org

## pythex

Your regular expression:

/

`(([0-9]{3}[\.]?[0-9]{3}[\.]?[0-9]{3})[-]?[0-9]{2})`

/

IGNORECASE

MULTILINE

DOTALL

VERBOSE

Your test string:

Olá, testando expressão regular, meu CPF é 000.111.222-33, mas poderia ser 012.345.678-90. Veja em Match result que os dois CPFs foram localizados e veja em Match captures os dois CPFs localizados.

Match result:

Olá, testando expressão regular, meu CPF é 000.111.222-33, mas poderia ser 012.345.678-90. Veja em Match result que os dois CPFs foram localizados e veja em Match captures os dois CPFs localizados.

Match captures:

**Match 1**

1. 000.111.222-33

**Match 2**

1. 012.345.678-90

Regular expression cheatsheet

Inspired by [Rubular](#). For a complete reference, see the official [re module documentation](#).

Made by [Gabriel Rodríguez](#). Powered by [Flask](#) and [jQuery](#).

# Expressões Regulares

Editor de expressões regulares online

Testando expressões regulares em <https://regex101.com/#python>

The screenshot shows the regex101.com website interface. The browser address bar displays `regex101.com/#python`. The page title is "regular expressions 101". The left sidebar contains sections for "SAVE & SHARE" (with a "save regex" button and "ctrl+s" shortcut), "FLAVOR" (with options for pcre (php), javascript, python (selected with a green checkmark), and golang), and "TOOLS" (with a "code generator" button). The main area is divided into three sections: "REGULAR EXPRESSION" (containing the regex `[\w.-]+@[ \w.-]+`), "TEST STRING" (containing the text "Meu principal e-mail é evaldowolkers@gmail.com, mas tenho também o evaldorw@hotmail.com, e o que dizer do evaldowolkers@algumacoisa.com.br? Este eu ainda não tenho."), and "SUBSTITUTION" (empty). The "REGULAR EXPRESSION" section shows "3 matches, 121 steps (~112ms)". The "TEST STRING" section shows three matches highlighted in blue. The right sidebar contains an "EXPLANATION" section (with a dropdown arrow) detailing the regex components: "Match a single character present in the list below" (showing `[\w.-]`), "Quantifier" (showing `+`), "Global pattern flags" (showing `g`), and "MATCH INFORMATION" (showing three matches with their full match and position). Below the "EXPLANATION" section is a "QUICK REFERENCE" section (with a dropdown arrow) containing a search bar and a list of tokens: "all tokens", "common tokens" (selected with a green checkmark), "general tokens", "anchors", "meta sequences", and "quantifiers". The "common tokens" list includes: "a single character of: a, b o... [abc]", "a character except: a, b or c [^abc]", "a character in the range: a-z [a-z]", "a character not in the ran... [^a-z]", "a character in the rang... [a-zA-Z]", "any single character .", and "any whitespace character \s".

# FIM