# Expressões regulares para iniciantes

Qualquer solução de descoberta e classificação de dados depende bastante de expressões regulares (comumente chamados de RegExes, Res ou padrões de RegEx) para identificar dados sensíveis. Mas em suma, o que são RegExes e como eles podem ser usados para descobrir dados sensíveis? Vamos descobrir.

Expressões regulares são pequenas, mas altamente especializadas na linguagem de programação; eles funcionam como coringas. Utilizando esta pequena linguagem, você especifica regras que define as strings que você quer encontrar. Por exemplo, você pode definir uma RegEx que irá combinar  endereços de e-mail, PII, PHI ou números de cartão de crédito.

## Componentes do RegEx

Um RegEx pode incluir literais e Metacaracteres.

### Literais

Qualquer caractere único, exceto aqueles reservados para metacaracteres, é uma expressão regular por si só. Por exemplo, www é resultado de busca para [www.netwrix.com](http://www.netwrix.com/) mas wwz não. Note que as expressões regulares diferenciam maiúsculas e minúsculas então www não irá dar resultado para WWW ou wWw.

### Metacaracteres

Os caracteres a seguir não são interpretados como literais e tem significados especiais

* . ^ $ \* + ? { } [ ] \ | ( )

A tabela a seguir descreve como cada um desses metacaracteres funciona

|  |  |  |  |
| --- | --- | --- | --- |
| Tipo | Metacaractere | Descrição | Exemplos |
| O ponto | . | O ponto quer dizer qualquer caractere | Net.rix trará resultados para ambos, [www.netwrix.com](http://www.netwrix.com/) e [www.netfrix.com](http://www.netfrix.com/) |
| Colchete | [] | Encontra qualquer coisa dentro dos colchetes com exceção  do caractere ^. Dentro de um colchete, o ^ no começo significa  a exceção em uma busca.  Por exemplo [^n] trará qualquer caractere exceto n, isso é  chamado de negação de caractere.  Note que os metacaracteres (com uma única exceção) não  estão ativos dentro de colchetes. Por exemplo, [net$] trará  qualquer um dos caracteres n, e, t ou $ ($ é um metacaractere,  porém dentro dos colchetes ele apenas traz resultados para $) | Você pode listar os caracteres individualmente; por exemplo, net[wrx] irá trazer resultados como netw, netr e netx mas não netz.  Ou você pode ver um grupo de caracteres dando dois caracteres e separando-os com um hífen; por exemplo net[a-z] trará neta, netw e netf mas não trará net1. |
| Ancoras | ^ (circunflexo) | Usado para trazer resultados de caracteres no começo de uma  string. | ^https trará resultados como [https://netwrix.com](https://netwrix.com/) mas não trará netwrix.com ou [http://netwrix.com](http://netwrix.com/) |
| $ (cifrão) | Usado para trazer resultados no final de uma string | com$ trará resultados como netwrix.com e telecom mas não computador |
| Iteração/quantificadores | ? (interrogação) | Traz o resultado precedente uma ou nenhuma vez. (sempre  trará resultado se o caractere não seja encontrado). Ele é ótimo  para encontrar caracteres opcionais | Colou?r trará resultando tanto para color e colour |
| \*(asterisco) | Traz o resultado zero ou mais vezes ao invés de zero ou uma vez.  Ele é ótimo para encontrar uma série de caracteres opcionais | Ne\*t trará resultados para nt (nenhum caractere e) net (um e), neeet (três e) e assim por diante. |
| + (mais) | Traz o resultando de um elemento uma ou mais vezes. Não  confundir \* e +.  \*  traz zero ou mais resultados, logo qualquer repetição pode  não ser representada; + requer ao menos uma ocorrência | Ne+t trará resultados para net e neeet mas não nt. |
| | (barra vertical) | Traz resultados das expressões entre a barra. | Net|wrix trará os resultados net e wrix. |
| {} (chaves) | {x} traz resultados de um elemento que é encontrado exatamente  x vezes.  {x,y} traz resultado se o elemento for encontrado x vezes, mas  não mais do que y vezes. | N{3} encontrará nnn, nnnn e nnnd (por que eles encontram 3 n em sequência) porém ele não trará resultado para nnw.  9{3} encontrará 999, 12349991234 e texto999texto, mas não encontrará 84299238, 9909 ou página992.  N{3,5} encontrará nnn, nnnn e nnnnn. |
| Bloqueio e captura | () (parênteses) | Define uma sub expressão que pode ser chamada novamente  usando uma abreviação: A primeira sub expressão pode ser  chamado por /1, a segundo por /2 e assim por diante. Os  parênteses normalmente são usados com a | (barra vertical)  dentro ou com os quantificadores de fora. | GR(a|e)y encontrará tanto Gray ou Grey.  [0-9]([-])[0-9]\1[0-9]  Encontrará 3-4-2 e 4-6-1 mas não encontrará 1-23, 42-1 ou 234. |
| Sequencias literais | \ (barra invertida) | Os metacaracteres que forem usados depois da barra invertida  serão usados como literais.  Note que há algumas sequencias que se inicial em \ e não são  sequencias literais. Ao invés disso, elas representam um conjunto  pré-definido de caracteres que podem ser úteis como um conjunto  de dígitos, letras ou espaço em branco. Os mais populares estão  listados abaixo como metacaracteres especiais | www\.netwrix\.com trará o resultado [www.netwrix.com](http://www.netwrix.com/) mas não trará www,netwrix,com |
| Metacaracteres especiais | \s | Traz resultado em qualquer caractere em branco (um espaço, um  tab, uma quebra de linha, etc) | Netwrix\sAuditor trará os resultados Netwrix Auditor e Netwrix(tab)Auditor, mas não Netwrix<5 espaços>Auditor ou NetwrixAuditor |
| \S | Traz o resultado de qualquer espaço que não esteja em branco. | \Snetwrix trará os resultados Xnetwrix e 1netwrix. |
| \w | Traz o resultado de qualquer caractere que seja alfa-numérico. | \w\w\w trará resultados como net, dfw e Netwrix. |
| \W | Traz o resultado de qualquer caractere que não seja alfa-numérico. | Netwrix\W trará resultados como netwrix! e netwrix?. |
| \d | Traz o resultado de qualquer digito decimal. | Netwrix\d\d traz resultados como Netwrix80 e Netwrix90. |
| \D | Traz o resultado de qualquer caractere que não seja um digito. | Netwrix\D trará resultados como Netwrix) e Netwrix. |
| \a | Traz o resultado de qualquer caractere alfabético esteja maiúscula  ou minúscula | Net\arix trará resultados como netWrix, netfrix e netarix. |
| \b | Define o ponto inicial de uma palavra | \brix trará resultados como rixon mas não trará netwrix |
| \B | Define um ponto que não o inicial de uma palavra. | \Brix trará resultados como Netwrix e trix mas não trará rixon |

## Combinações de metacaracteres

Agora que sabemos quase todos os metacaracteres, estamos prontos para combiná-los.

#### **Exemplo**: Procurando por placas de carro

Vamos supor que precisemos procurar por placas de carro no formato aaa-nnnn, onde os primeiros três dígitos precisam ser alfanuméricos e os 4 últimos precisam ser numéricos. O hífen pode ser substituído por qualquer caractere ou simplesmente não estar presente.

Essa RegEx será formada da seguinte forma:

* \b[0-9A-Z]{3}([^ 0-9A-Z]|\s)?[0-9]{4}\b

Vamos desvendar essa RegEx:

* \b Define o primeiro limite de palavra, para que essas strings não se tornem parte de uma string maior
* [0-9A-Z]{3} significa que os primeiros caracteres devem ser alfa numéricos
* ([^ 0-9A-Z]|\s)? significa que a próxima parte da string deve ser um delimitador, um caractere não alfanumérico ou um caractere de espaço em branco, ou nada.
* [0-9]{4} Diz que a próxima parte da string deve conter 4 digitos
* \b Define outro limite de palavra

Esta RegEx trará resultados para placas como essas: NT5-6345, GH3 9452, XS83289

Entretanto, ela não trará essas placas: ZNT49371, HG3-29347, nt4-9371

#### **Exemplo**: Procurando por número de segurança social

Nos EUA, existe um número de segurança social (SSN, na sigla em inglês) que tem o seguinte formato: nnn-nn-nnn.

A RegEx mais fácil para essa finalidade seria:

* [0-9]{3}-[0-9]{2}-[0-9]{4}

Entretanto, isso irá gerar falsos positivos, já que nem todos os números nesse formato são SSN legítimos. Além disso, alguns SSNs poderão ser perdidos, como os que forem escritos sem hífen. Para resultados mais precisos, devemos montar uma RegEx mais complexa.

Nós sabemos que:

* Nenhum grupo de dígitos pode ser composto apenas de zeros
* O primeiro bloco não pode ser 666 ou 900-999
* Os SSNs podem ser escritos com espaço em branco ao invés de hífens, ou até mesmo sem delimitador.
* Caso o primeiro bloco comece com um 7, ele deve ser seguido por um número entre 0 e 6 e então qualquer terceiro digito.

Logo, a RegEx avançada será algo parecido com isso:

* \b(?!000|666|9\d{2})([0-8]\d{2}|7([0-6]\d))([-]?|\s{1})(?!00)\d\d\2(?!0000)\d{4}\b

Como antes, o \b na frente e no fim pra especificar os limites da string. Vamos analisar mais profundamente cada número entre os blocos.

O primeiro bloco

* (?!000|666|9\d{2}) é uma negação que especifica que os números não devem começar com 000, 666 ou 9 seguido de quaisquer 2 dígitos
* ([0-8]\d{2} especifica que a string deve começar com um digito entre 0 e 8 e que tenha mais dois dígitos de (0-9) após ele.
* |7[0-6]\d)) diz que caso começa com o número 7, o próximo digito deve ser entre 0 e 6 seguido por qualquer digito
* ([-]?|\s{1}) diz que após 3 dígitos, deve haver um hífen, um espaço em branco ou nada para marcar o fim do primeiro bloco.

O segundo bloco

* (?!00) é outra negação que diz que o segundo block não pode ser 00
* \d\d diz que deve haver quaisquer dígitos no segundo bloco
* \2 chama o mesmo texto do segundo grupo de captura que é ([-]?|\s{1}), que especifica que o segundo bloco pode acabar em um hífen, com um espaço em branco ou sem nenhum caractere adicional.

O terceiro bloco

* (?!0000) é outra negação que diz que não pode haver 4 zeros no terceiro bloco
* \d{4} requer quaisquer 4 dígitos no terceiro bloco SSN

## Exemplos de RegExes populares

|  |  |  |
| --- | --- | --- |
| Para encontrar | Use esta RegEx | Exemplo de resultado |
| Endereço de e-mail | ^[\w\.=-]+@[\w\.-]+\.[\w]{2,3}$ | [fulano@site.com](mailto:fulano@site.com) |
| CPF | /(?!(\d)\1{2}.\1{3}.\1{3}-\1{2})\d{3}\.\d{3}\.\d{3}\-\d{2}/gm | 061.381.209-30 |
| Endereços IPV4 | ^\d{1,3}[.]\d{1,3}[.]\d{1,3}[.]\d{1,3}$ | 192.168.1.1 |
| Datas no formato MM/DD/YYYY | ^([1][12]|[0]?[1-9])[\/-]([3][01]|[12]\d|[0]?[1-9])[\/-](\d{4}|\d{2})$   |  | | --- | | 05/05/2018 | |  | |  |
| Números MasterCard | ^(?:5[1-5][0-9]{2}|222[1-9]|22[3-9][0-9]|2[3-6][0-9]{2}|27[01][0-9]|2720)[0-9]{12}$ | 5258704108753590 |
| Números do cartão Visa | \b([4]\d{3}[\s]\d{4}[\s]\d{4}[\s]\d{4}|[4]\d{3}[-]\d{4}[-]\d{4}[- ]\d{4}|[4]\d{3}[.]\d{4}[.]\d{4}[.]\d{4}|[4]\d{3}\d{4}\d{4}\d{4})\b | 4563-7568-5698-4587 |
| Números do cartão AmericanExpress | ^3[47][0-9]{13}$ | 34583547858682157 |
| Código postal dos EUA | ^((\d{5}-\d{4})|(\d{5})|([A-Z]\d[A-Z]\s\d[A-Z]\d))$ | 97589 |
| Caminho de arquivos | \\[^\\]+$ | \\fs1\shared |
| URLs | (?i)\b((?:[a-z][\w-]+:(?:\/{1,3}|[a-z0-9%])|www\d{0,3}[.]|[a-z0-9.\-]+[.][a-z]{2,4}\/)(?:[^\s()<>]+|\(([^\s()<>]+|(\([^\s()<>]+\)))\*\))+(?:\(([^\s()<>]+ |(\([^\s()<>]+\)))\*\)|[^\s`!()\[\]{};:'”.,<>?«»“”‘’])) | [www.aiqon.com.br](http://www.aiqon.com.br/) |

### Páginas na web que podem te ajudar com RegEx

* [https://regexr.com](https://regexr.com/) e[https://regex101.com](https://regex101.com/) podem te ajudar a checar as suas RegExes destacando sintaxes e dando dicas
* [https://regexcrossword.com](https://regexcrossword.com/) é um jogo de palavras cruzadas nos quais as pistas são definidas usando expressões regulares.
* [https://www.regular-expressions.info](https://www.regular-expressions.info/) é um ótimo site para informação sobre as expressões regulares. Além disso, o Notepad++ contém uma extensão para ajudar no RegEx que pode te servir bem enquanto você trabalha com as expressões regulares

https://aiqon.com.br/blog/expressoes-regulares-para-iniciantes/

# Expressões Regulares - Um guia para iniciantes

12 de janeiro de 2021  |

10 min de leitura

[Tweet](https://twitter.com/intent/tweet?text=Expressões Regulares - Um guia para iniciantes %0ABy @Gonkristiano%0Ahttps%3A%2F%2Fwww.crisgon.dev%2Fexpressões-regulares-um-guia-para-iniciantes%2F)

[Compartilhar](https://www.facebook.com/sharer/sharer.php?u=https://www.crisgon.dev/expressões-regulares-um-guia-para-iniciantes/)

[Compartilhar](https://www.linkedin.com/sharing/share-offsite/?url=https://www.crisgon.dev/expressões-regulares-um-guia-para-iniciantes/)

Antes de aprender uma tecnologia é interessante entender que tipo de problema ela se propõe a resolver, então vamos lá... Imagine que você trabalha em uma aplicação que precisa receber o CPF com 11 dígitos numéricos. Qual seria sua abordagem para fazer uma validação? Acredito que caso não conheça regex, sua primeira opção seria verificar se o tamanho da string com os números digitados é 11, mas quem garante que tudo que foi passado são números?

"12345678909".lenght === 11; // Pode ser um CPF válido...

"123456789".lenght === 11; // Não seria um CPF válido...

"1234567890B".lenght === 11; // Aqui temos uma string com tamanho 9, mas é um CPF válido?

No código acima vimos 3 situações que provam que verificar apenas o tamanho da string não é garantia de que teremos um CPF válido, pois essa é uma verificação muito rasa que não garante que todos os caracteres passados são numéricos. Para conseguir ter a certeza de que os caracteres atendem aos requisitos que desejamos é preciso definir e verificar padrões e isso pode ser feito com uso de expressões regulares.

### O que é uma expressão regular?

Em ciência da computação, uma expressão regular ou "Regex" (ou os estrangeirismos regex ou regexp) , abreviação do inglês regular expression) provê uma forma concisa e flexível de identificar cadeias de caracteres de interesse, como caracteres particulares, palavras ou padrões de caracteres. Expressões regulares são escritas numa linguagem formal que pode ser interpretada por um processador de expressão regular, um programa que serve um gerador de analisador sintático ou examina o texto e identifica as partes que casam com a especificação dada.

[Wikipédia](https://pt.wikipedia.org/wiki/Expressão_regular)

Diversas linguagens de programação tem sua própria implementação de expressões regulares, geralmente com pequenas diferenças e em alguns casos com diferenças bem significativas. Mas não se preocupe, pois, a base que você irá aprender a seguir vai ser de grande valor em qualquer linguagem de programação que você for se aventurar.

Agora você já sabe que uma expressão regular permite que palavras ou padrões de caracteres sejam encontrados em um texto, então vamos partir do exemplo mais básico que são os caracteres literais.

#### Antes de prosseguir recomendo que visite o site [*regex101*](https://regex101.com/) e vá reproduzindo os exemplos que forem sendo abordados ao longo do artigo.

### Caracteres literais

Antes de prosseguir precisamos combinar que todas as nossas regex(expressões regulares) serão escritas entra duas barras(/SUA REGEX AQUI/). Agora que estamos na mesma página, qual seria a forma de encontrar todas as palavras "doce" que aparecem no nosso trava-línguas?

O doce perguntou pro doce  
Qual é o doce mais doce  
Que o doce de batata-doce.  
O doce respondeu pro doce  
Que o doce mais doce que  
O doce de batata-doce  
É o doce de doce de batata-doce.

Já que queremos encontrar a palavra "doce" vamos fazer um uso literal e nossa regex vai ficar assim: /doce/

PS: Tudo que for encontrado por nossa regex vai ser destacado com a cor azul.

Parabéns! Você acabou de fazer sua primeira regex, mas note que apenas a primeira palavra doce ficou azul... Isso acontece porque por padrão nossa expressão regular busca apenas a primeira ocorrência e para de procurar quando encontra algo que atenda o seu padrão. Tá, mas como resolvemos isso? Vamos resolver utilizando flags!

### Flags

Flags são informações adicionais que passamos para mudar o comportamento da nossa regex. A estrutura de uma regex com flag seria assim: /doce/SUA FLAG

Existem diversos tipos de flags, mas por agora vamos focar apenas na g que significa global e indica que a busca só deve parar após encontrar todas as ocorrências do texto. Nossa regex ficaria assim /doce/g e esse seria o resultado das ocorrências:

Show de bola! Encontramos todas as ocorrências para a palavra doce, mas geralmente você não vai usar uma regex para buscar por uma palavra específica... O grande poder de uma regex é permitir que padrões sejam encontrados, então vamos "dificultar" um pouco mais as coisas. Preciso de uma expressão regular que seja capaz de encontrar todos os números do texto a seguir:

0 N4rut0 p0de 5er um p0uc0 dur0 às veze5, t4lvez v0cê nã0 s41b4 d1550, m4s 0 N4rut0 t4mbém cresceu sem p41. N4 verd4de ele nunc4 c0nheceu nenhum de seus p41s, e nunc4 teve nenhum 4m1g0 em n0ss4 4lde14. Mesm0 4ss1m eu nunc4 v1 ele ch0r4r, f1c4r z4ng4d0 0u se d4r p0r venc1d0, ele está sempre d1sp0st0 4 melh0r4r, ele quer ser respe1t4d0, é 0 s0nh0 dele e 0 N4rut0 d4r14 4 v1d4 p0r 1ss0 sem hes1t4r. Meu p4lp1te é que ele se c4ns0u de ch0r4r e dec1d1u f4zer 4lgum4 c01s4 4 respe1t0!

Para resolver esse problema precisamos aprender mais uma funcionalidade de expressões regulares.

### Classes de caracteres ou conjunto de caracteres

Um conjunto de caracteres é uma lista de possíveis ocorrências que é passada entre []. No nosso caso queremos encontrar qualquer número no nosso texto, então podemos passar uma lista numérica da seguinte forma: [0123456789]  
É como se estivéssemos falando o seguinte para a nossa regex: eu quero que você encontre o digito 0, ou o digito 1, ou digito 2...

Outra forma de escrever um conjunto de caracteres é usando os intervalos. No nosso exemplo queremos encontrar os números de 0 até 9, logo podemos escrever um conjunto da seguinte forma: [0-9]. Se o objetivo fosse encontrar os números de 5 até 8 o conjunto seria [5-8].

Resumindo, nossa regex ficou assim/[0-9]/ e os números encontrados foram os seguintes:

A mesma lógica serve para conjuntos de letras ou de outros caracteres. Podemos usar uma sequência de letras assim [abcdefghijklmnopqrstuvwxyz] ou abreviar usando a sequência [a-z].

Um ponto de atenção!! A sequência de [a-z] é diferente de [A-Z]. A primeira vai encontrar apenas letras minusculas e a segunda apenas letras maiúsculas.

"Mas e se eu quiser encontrar as letras minusculas e maiúsculas?"

Vou te dar duas opções!

1. Utilizando uma flag de case insensitivo(i) que seria assim /[a-z]/i
2. Usando um conjunto com a sequência de letras minusculas ou a sequência de letras maiúsculas que seria assim /[a-zA-Z]/

As duas opções vão reproduzir o resultado abaixo:

Se você analisar o resultado com cuidado vai ver que temos um pequeno problema... Não conseguimos encontrar todas as letras do nosso texto. Todas que possuem algum caractere especial(è,ã) foram ignoradas.

### Classes de caracteres abreviados: o famoso shorthand

Ter que sempre escrever [0-9]ou [a-zA-z] seria um saco, mas felizmente existem maneiras de fazer as mesmas coisas escrevendo menos. Vou listar alguns das principais formas abreviadas

* \d corresponde a qualquer caractere numérico. O mesmo que [0-9]
* \D corresponde a tudo que não é um digito, incluindo caracteres especiais como !, ç~ e etc.
* \scorresponde a espaços em branco, tabs, quebra de linhas.
* \Scorresponde a tudo que não for espaços em branco, tabs, quebras de linha.
* \w corresponde a tudo que é alfanumérico. O mesmo que [a-zA-z0-9\_]
* \W corresponde a tudo que não é alfanumérico. Pode ser utilizado em conjunto com \w para encontrar caracteres alfanuméricos e não alfanuméricos.[\w\W]
* . é considerado um coringa, pois com ele podemos encontrar qualquer caractere, menos quebras de linha.

### Quantificadores

Os quantificadores, também podem ser conhecidos como repetidores, dizem quantas vezes determinada ocorrência pode se repetir. Até o momento fizemos buscas por letras ou números, mas não chegamos a especificar quantas letras seguidas queremos encontrar, ou quantos números seguidos.

Pense na seguinte situação, precisamos encontra o número de uma placa de carro em um texto qualquer.

Uma placa de carro tem a seguinte estrutura: ABC-1234 (3 letras 1 hífen e 4 números). Note que apenas com o que aprendemos até agora é impossível encontrar o número da placa no texto abaixo:

O carro com a placa de número ABC-1234 foi furtado na manhã de domingo.

É ai que os quantificadores entram! Nós vamos dizer exatamente quantas vezes cada caractere vai se repetir para atender ao nosso padrão. Logo, nossa regex ficaria assim: /[a-zA-Z]{3}-\d{4}/g

A expressão regular é bem simples! Podemos separar ela em 3 partes para ficar mais fácil de entender:

* [a-zA-Z]{3} Essa parte diz que qualquer coisa de a-z tem que aparecer 3 vezes
* - Aqui indica que é preciso ter um hífen
* \d{4} Essa parte diz que 4 números quaisquer devem aparecer seguidos.

Agora que você já tem uma ideia de como um quantificador pode ser usado, veja uma lista com outros quantificadores importantes:

* {n} exatamente n repetições/ocorrências.
* {n,m} no mínimo n ocorrências e no máximo m ocorrências.
* {n, } no mínimo n ocorrências e o limite máximo não existe.
* {, m} no máximo m ocorrências. Nesse caso o mínimo é 0.
* ? 0 ou 1 ocorrência; a mesma coisa que {, 1}
* + 1 ou mais ocorrências; mesma coisa que {1, }
* \* 0 ou mais ocorrências.

### Âncoras

Âncoras definem posições para que os caracteres sejam correspondidos. Uma âncora pode definir o início, o fim ou um limite para que o match seja feito. Vamos a seguinte situação: temos que encontrar arquivos que tenham um formato .gif. Pra que isso funcione precisamos garantir que o nome do arquivo sempre vai terminar com .gif. Nossa regex ficaria assim: /\w+.gif$/

Explicando nossa regex...

* \w+ essa parte quer dizer para encontrar um ou mais letrar, números e underline.
* .gif$ essa parte quer dizer que a ocorrência deve terminar com .gif

Agora que você já tem uma ideia de como uma âncora pode ser usado, veja uma lista com outras âncoras importantes:

* ^ inicio de um texto ou de uma linha, caso esteja utilizando a flag para multiline(m)
* \A inicio de um texto
* $ fim de um texto ou de uma linha, caso esteja utilizando a flag para multiline(m)
* \Z fim de um texto
* \b indica uma "borda" para a ocorrência, ou no inicio da palavra, ou no fim.

Um pequeno exemplo utilizando a âncora \b:

Nossa missão é encontrar todas às vezes que a preposição de aparece na frase abaixo.

Note que com a expressão /de/ conseguimos pegar tudo que tem de, incluindo as palavras dez e desânimo. Uma solução é mudar nossa expressão para /\bde\b/.

### Alternância

Esse é bem simples, pois existem casos onde queremos capturar uma ou outra ocorrência. Onde queremos encontrar o número 1 ou o número 2, a palavra rei ou a palavra rato. Pra isso só precisamos colocar que queremos encontrar entre parenteses e separar por uma |.

Vamos ao exemplo onde queremos encontrar as palavras rato e rei no texto abaixo. A expressão regular vai ficar assim: /(rato|rei)/

### Isso é tudo pessoal!

Obrigado por chegar até aqui!! Espero que tenha conseguido te ajudar de alguma forma. 😊

Tudo que abordei até aqui é apenas o começo desse assunto tão vasto , mas acredito que seja o suficiente para que você dê um pontapé inicial e não fique tão perdido quando o assunto for expressões regulares. Recomendo dar uma conferida nos links que vou deixar mais abaixo, caso queira se aprofundar mais no assunto.

### Links importantes

* [Expressões Regulares](https://www.regular-expressions.info/index.html)
* [Expressões Regulares: introdução](http://turing.com.br/material/regex/introducao.html)
* [Regex 101](https://regex101.com/)
* [Expressões Regulares: Mozila Developers](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Regular_Expressions)

https://www.crisgon.dev/express%C3%B5es-regulares-um-guia-para-iniciantes/

# O guia do iniciante para expressões regulares

[Linux](https://www.routech.ro/pt-br/linux-pt-br/), [Uncategorized @ro](https://www.routech.ro/pt-br/uncategorized-ro-pt-br-2/) / novembro 20, 2020

Você já tentou encontrar um padrão recorrente em um trecho de texto? Você pode ter usado algo como a função de pesquisa em seu navegador ou processador de texto, mas quando precisa encontrar algo mais complexo, pode ser como encontrar uma agulha no palheiro proverbial.

Felizmente, há uma maneira de escolher padrões precisos no texto até o caractere. É chamado de expressões regulares (RegEx) e permite que você se torne um mestre na pesquisa de texto.

Conţinut [[ascunde](https://www.routech.ro/pt-br/o-guia-do-iniciante-para-expressoes-regulares/)]

* [1 Onde posso usar o RegEx?](https://www.routech.ro/pt-br/o-guia-do-iniciante-para-expressoes-regulares/" \l "Onde_posso_usar_o_RegEx)
* [2 Encontrando Padrões](https://www.routech.ro/pt-br/o-guia-do-iniciante-para-expressoes-regulares/" \l "Encontrando_Padroes)
* [3 Gamas](https://www.routech.ro/pt-br/o-guia-do-iniciante-para-expressoes-regulares/" \l "Gamas)
* [4 MetaCaracteres](https://www.routech.ro/pt-br/o-guia-do-iniciante-para-expressoes-regulares/" \l "MetaCaracteres)
* [5 Escapar](https://www.routech.ro/pt-br/o-guia-do-iniciante-para-expressoes-regulares/" \l "Escapar)
* [6 Âncoras e limites](https://www.routech.ro/pt-br/o-guia-do-iniciante-para-expressoes-regulares/" \l "Ancoras_e_limites)

## Onde posso usar o RegEx?

Embora o Unix e o Linux os tenham tornado populares, as expressões regulares estão disponíveis em uma variedade de pacotes, incluindo o Microsoft Word.

As expressões regulares são usadas principalmente em vários programas Linux notáveis, incluindo grep, Awk e Sed.

Por exemplo, você pode querer verificar os dispositivos USB em seu PC. Usando lspci, você verá uma lista de todos os dispositivos e terá que localizar as entradas USB sozinho. Em vez disso, você pode usar o seguinte para mostrar apenas os dispositivos USB:

**lspci** **|** **grep** "USB"

Este é o exemplo mais simples de RegEx em ação. É a forma mais popular de usar expressões regulares no terminal, mas não a única. Hoje você pode encontrar suporte RegEx em muitos tipos diferentes de software, de editores de texto a gerenciadores de arquivos.

## Encontrando Padrões

Você provavelmente já usou o \* caractere, que atua como um curinga ao selecionar arquivos ou pastas no terminal. Por exemplo, para listar todos os arquivos JPG em uma pasta, você pode usar:

**ls** **\***.jpg

O RegEx equivalente ao acima seria:

**ls** **|** **grep** -E ".jpg"

Para pesquisar arquivos jpg e png, use:

**ls** **|** **grep** -E "(.jpg|.png)"

## Gamas

Se você deseja pesquisar um intervalo específico de caracteres em vez de um padrão, pode fazer isso definindo-o entre colchetes. Se, por exemplo, você usa [a-z] como seu padrão, ele corresponderia a qualquer string consistindo em quaisquer letras minúsculas do alfabeto.

Como você deve ter adivinhado, [A-Z] selecionaria apenas letras maiúsculas. Para escolher qualquer intervalo de letras, tanto maiúsculas como minúsculas, a expressão mudaria para [a-zA-Z].

Para localizar um número específico de instâncias de seu padrão, você pode colocá-lo entre colchetes. {5} retornaria cinco ocorrências de seu padrão. Você também pode usar intervalos de números, então {5,10} apresentaria cinco a dez instâncias.

## MetaCaracteres

Em expressões regulares, você também pode pesquisar partes de uma string com dois caracteres chamados metacaracteres. Eles são semelhantes às correspondências de curinga que você pode ter usado no shell.

O principal é o ponto simples, que representa qualquer outro caractere único. Se você usou o padrão c.ll, corresponderia a “celular”, mas também “abate” e “chamada”

Ao inserir um asterisco após um ponto, você pode usá-lo para corresponder a um número infinito de caracteres. Por exemplo, .\*board será uma combinação para “teclado” e “skate”. mesmo que “chave” e “skate” tenham um número diferente de letras.

## Escapar

Você deve ter notado que em nosso exemplo, onde selecionamos diferentes tipos de arquivos de imagem, usamos barras invertidas antes do ponto final (“ .jpg”). É assim que você evita caracteres especiais no RegEx.

Se não os usássemos, nosso padrão não corresponderia apenas às extensões dos arquivos, strings como “.jpg” e “.png”, mas também corresponderia a “ajpg” e “opng”. Lembrar, . é um curinga que corresponde a qualquer caractere.

## Âncoras e limites

Âncoras e limites permitem que você defina com mais precisão o que você está procurando.

Para encontrar apenas a palavra individual “computador”, sem outros caracteres anexados antes ou depois, você deve definir o padrão como <computer>.

Você também pode pesquisar especificamente os padrões que aparecem no início ou no final da linha. Isso é alcançado com o ^ e $ caracteres respectivamente.

Então, se você quiser encontrar apenas as entradas onde a palavra “computador” aparece no início de uma linha, seu padrão seria semelhante ^computer. Pelo contrário, quando está no final da linha, o padrão mudaria para computer$.

Essas são regras simples do RegEx, que você também pode misturar para encontrar precisamente os padrões que procura. Você pode pesquisar intervalos de caracteres no início de uma linha ou palavras alternadas no final, datas específicas ou um intervalo de anos, usando uma única sequência de texto.

Não se esqueça de verificar nossa cheatsheet de Expressões regulares para dominar as expressões regulares.

https://www.routech.ro/pt-br/o-guia-do-iniciante-para-expressoes-regulares/

# Learn Regex: A Beginner’s Guide

By [Michael Wanyoike](https://www.sitepoint.com/author/mwanyoike)

[JavaScript](https://www.sitepoint.com/javascript)

In this guide, you’ll learn regex, or regular expression syntax. By the end, you’ll be able to apply regex solutions in most scenarios that call for it in your web development work.

Regular expressions have many uses cases, which include:

* form input validation
* web scraping
* search and replace
* filtering for information in massive text files such as logs

Regular expressions, or regex as they’re commonly called, look complicated and intimidating for new users. Take a look at this example:

/^[a-zA-Z0-9.!#$%&’\*+/=?^\_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)\*$/

It just look like garbled text. But don’t despair, there’s method behind this madness.



Credit: [*xkcd*](https://xkcd.com/208/)

I’ll show you how to master regular expressions in no time. First, let’s clarify the terminology used in this guide:

* pattern: regular expression pattern
* string: test string used to match the pattern
* digit: 0-9
* letter: a-z, A-Z
* symbol: !$%^&\*()\_+|~-=`{}[]:”;'<>?,./
* space: single white space, tab
* character: refers to a letter, digit or symbol

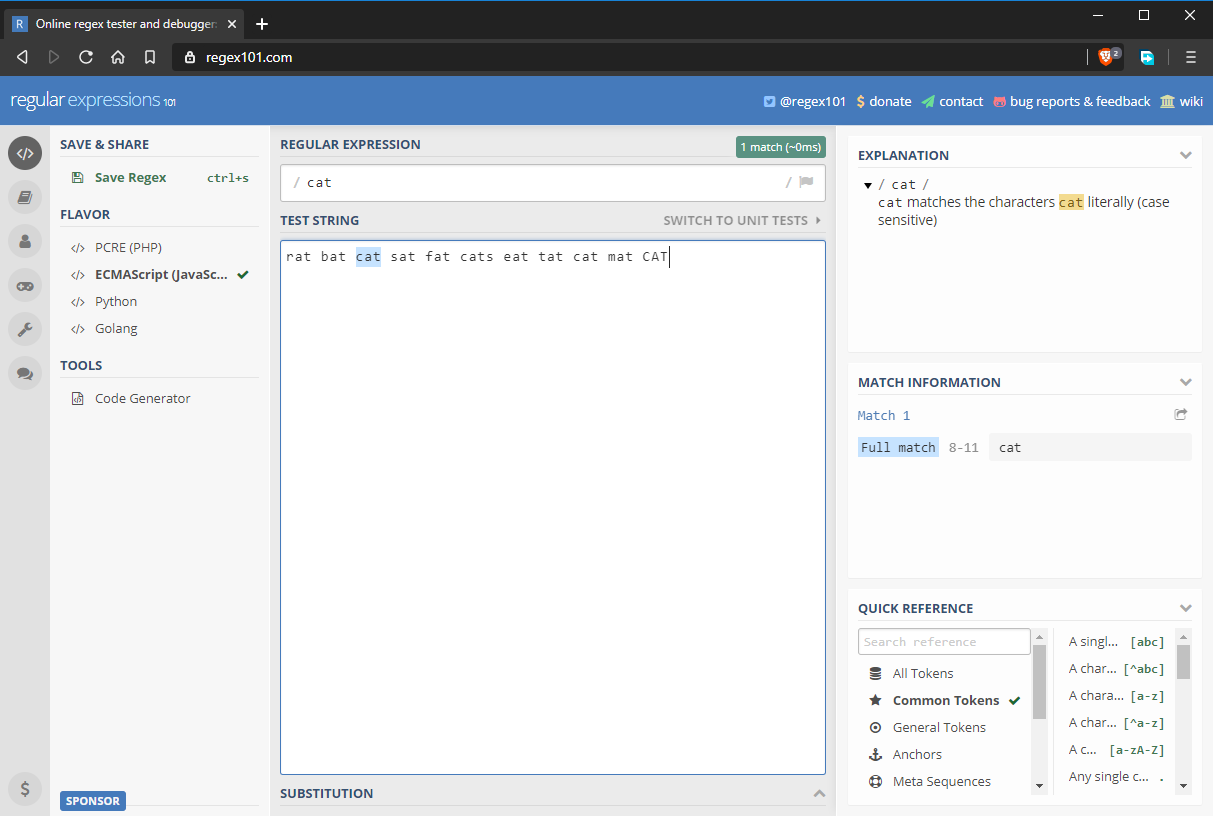
## Basics

To learn regex quickly with this guide, visit [Regex101](https://regex101.com/), where you can build regex patterns and test them against strings (text) that you supply.

When you open the site, you’ll need to select the JavaScript flavor, as that’s what we’ll be using for this guide. (Regex syntax is mostly the same for all languages, but there are some minor differences.)

Next, you need to disable the global and multi line flags in Regex101. We’ll cover them in the next section. For now, we’ll look at the simplest form of regular expression we can build. Input the following:

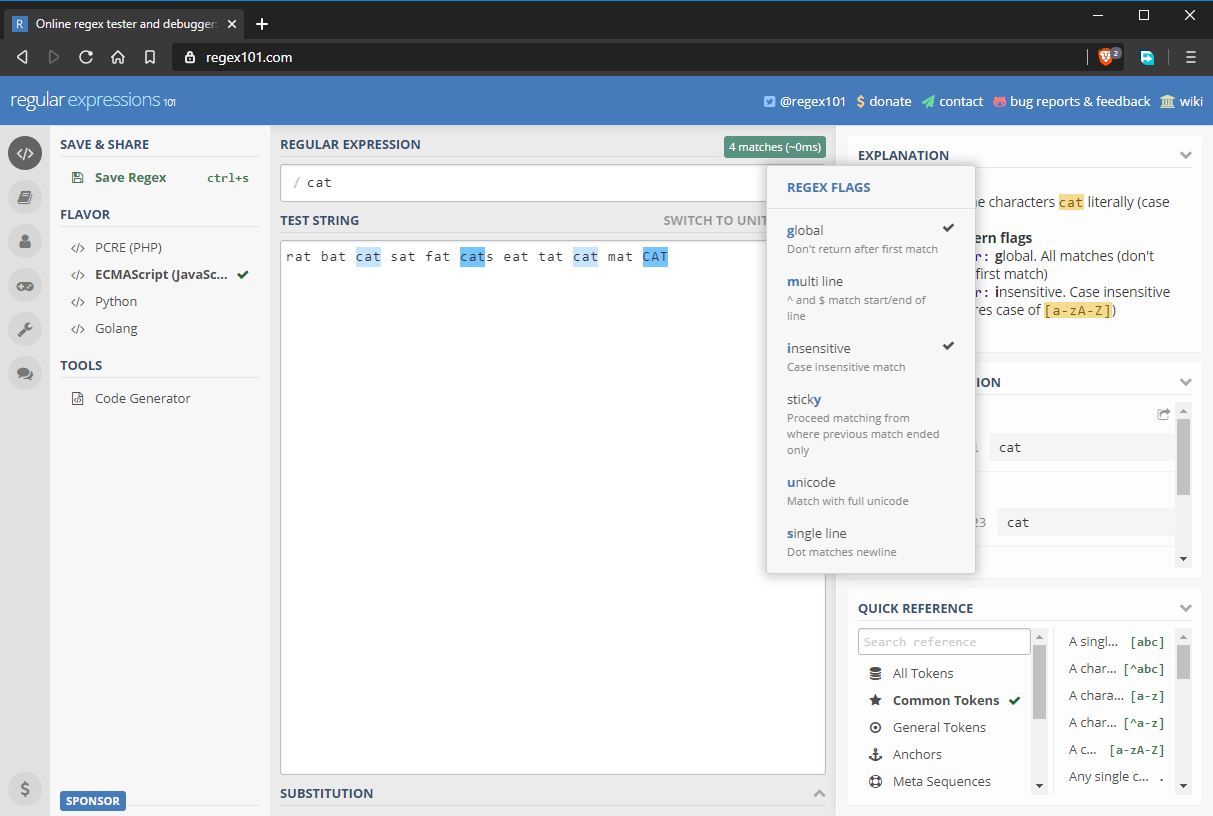
* regex input field: cat
* test string: rat bat cat sat fat cats eat tat cat mat CAT



Take note that regular expressions in JavaScript start and end with /. If you were to write a regular expression in JavaScript code, it would look like this: /cat/ without any quotation marks. In the above state, the regular expression matches the string “cat”. However, as you can see in the image above, there are several “cat” strings that are not matched. In the next section, we’ll look at why.

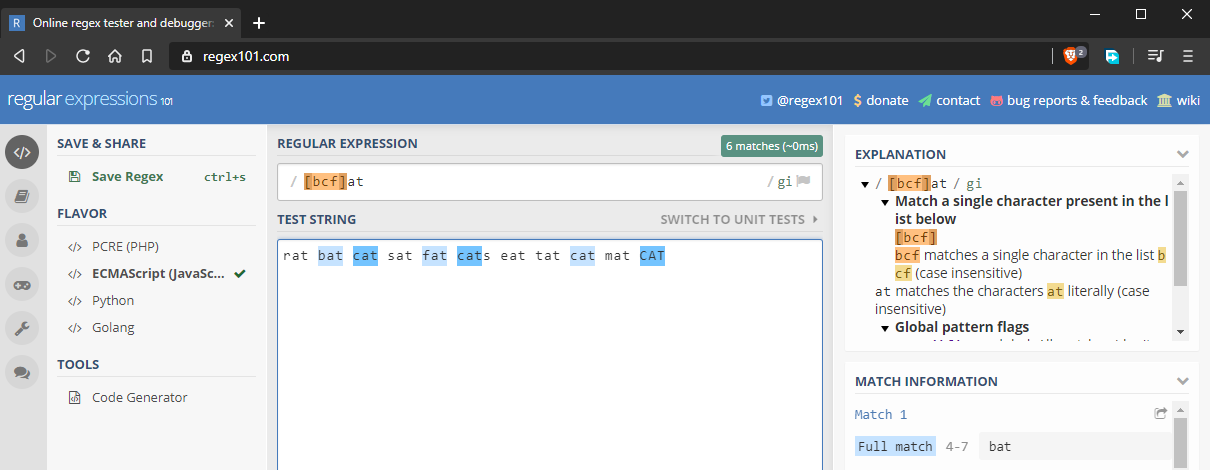
## Global and Case Insensitive Regex Flags

By default, a regex pattern will only return the first match it finds. If you’d like to return additional matches, you need to enable the global flag, denoted as g. Regex patterns are also case sensitive by default. You can override this behavior by enabling the insensitive flag, denoted by i. The updated regex pattern is now fully expressed as /cat/gi. As you can see below, all “cat” strings have been matched including the one with a different case.



## Character Sets

In the previous example, we learned how to perform exact case-sensitive matches. What if we wanted to match “bat”, “cat”, and “fat”. We can do this by using character sets, denoted with []. Basically, you put in multiple characters that you want to get matched. For example, [bcf]at will match multiple strings as follows:



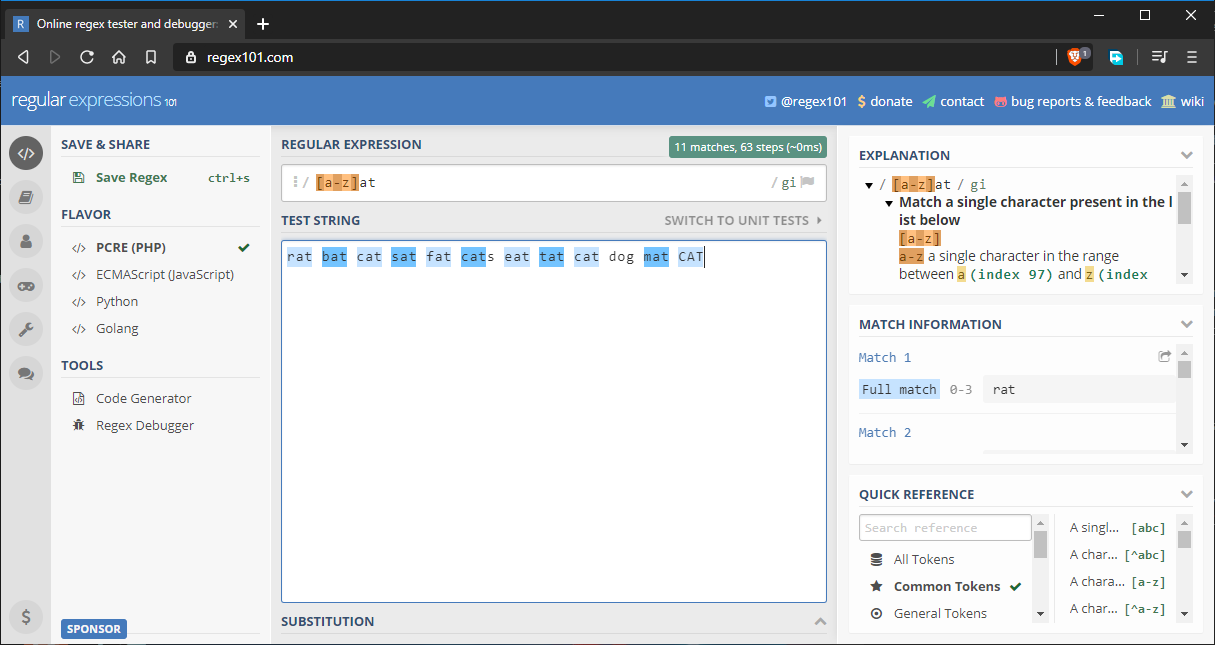
Character sets also work with digits.

## Ranges

Let’s assume we want to match all words that end with at. We could supply the full alphabet inside the character set, but that would be tedious. The solution is to use ranges like this [a-z]at:

### Learn PHP for free!

Make the leap into server-side programming with a comprehensive cover of PHP & MySQL.  
  
Normally ~~RRP $39.99~~ Yours absolutely free



Here’s the full string that’s being tested: rat bat cat sat fat cats eat tat cat dog mat CAT.

As you can see, all words are matching as expected. I’ve added the word dog just to throw in an invalid match. Here are other ways you can use ranges:

* Partial range: selections such as [a-f] or [g-p].
* Capitalized range: [A-Z].
* Digit range: [0-9].
* Symbol range: for example, [#$%&@].
* Mixed range: for example, [a-zA-Z0-9] includes all digits, lower and upper case letters. Do note that a range only specifies multiple alternatives for a single character in a pattern.

To further understand how to define a range, it’s best to look at the full ASCII table in order to see how characters are ordered.



## Repeating Characters

Let’s say you’d like to match all three-letter words. You’d probably do it like this:

[a-z][a-z][a-z]

This would match all three-letter words. But what if you want to match a five- or eight-character word. The above method is tedious. There’s a better way to express such a pattern using the {} curly braces notation. All you have to do is specify the number of repeating characters. Here are examples:

* a{5} will match “aaaaa”.
* n{3} will match “nnn”.
* [a-z]{4} will match any four-letter word such as “door”, “room” or “book”.
* [a-z]{6,} will match any word with six or more letters.
* [a-z]{8,11} will match any word between eight and 11 letters. Basic password validation can be done this way.
* [0-9]{11} will match an 11-digit number. Basic international phone validation can be done this way.

## Metacharacters

Metacharacters allow you to write regular expression patterns that are even more compact. Let’s go through them one by one:

* \d matches any digit that is the same as [0-9]
* \w matches any letter, digit and underscore character
* \s matches a whitespace character — that is, a space or tab
* \t matches a tab character only

From what we’ve learned so far, we can write regular expressions like this:

* \w{5} matches any five-letter word or a five-digit number
* \d{11} matches an 11-digit number such as a phone number

## Special Characters

Special characters take us a step further into writing more advanced pattern expressions:

* +: One or more quantifiers (preceding character must exist and can be optionally duplicated). For example, the expression c+at will match “cat”, “ccat” and “ccccccccat”. You can repeat the preceding character as many times as you like and you’ll still get a match.
* ?: Zero or one quantifier (preceding character is optional). For example, the expression c?at will only match “cat” or “at”.
* \*: Zero or more quantifier (preceding character is optional and can be optionally duplicated). For example, the expression c\*at will match “at”, “cat” and “ccccccat”. It’s like the combination of + and ?.
* \: this “escape character” is used when we want to use a special character literally. For example, c\\* will exactly match “c\*” and not “ccccccc”.
* [^]: this “negate” notation is used to indicate a character that should not be matched within a range. For example, the expression b[^a-c]ld will not match “bald” or “bbld” because the second letters a to c are negative. However, the pattern will match “beld”, “bild”, “bold” and so forth.
* .: this “do” notation will match any digit, letter or symbol except newline. For example, .{8} will match a an eight-character password consisting of letters, numbers and symbols. for example, “password” and “P@ssw0rd” will both match.

From what we’ve learned so far, we can create an interesting variety of compact but powerful regular expressions. For example:

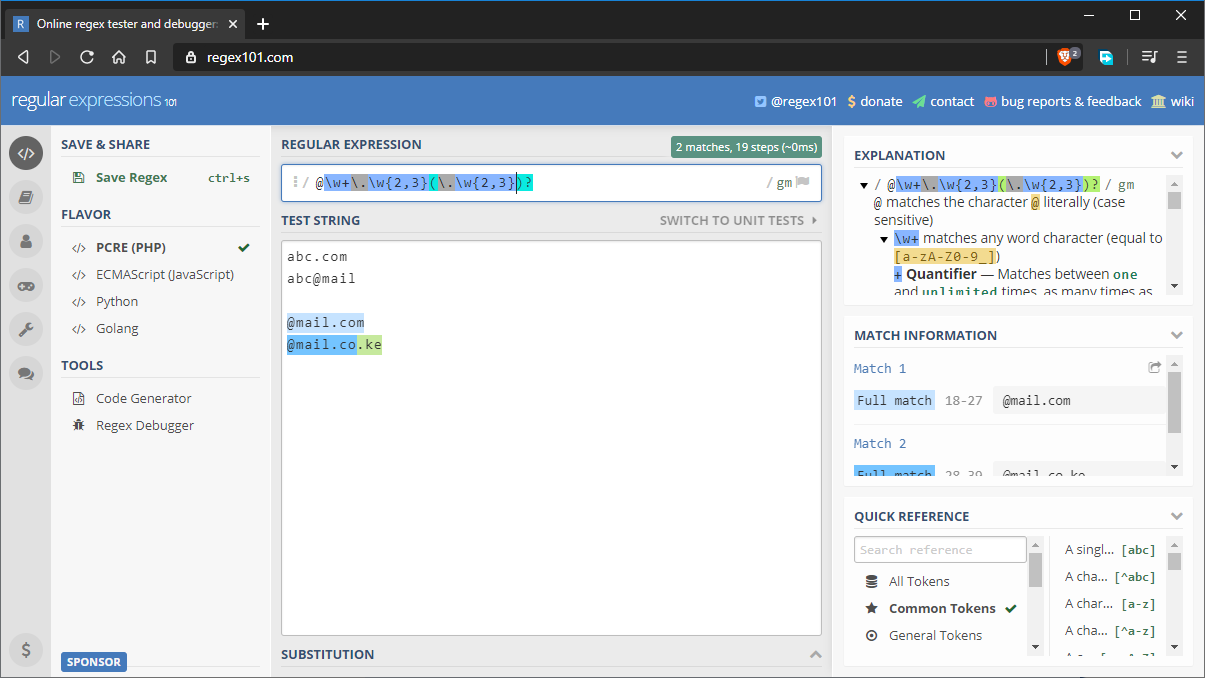
* .+ matches one or an unlimited number of characters. For example, “c” , “cc” and “bcd#.670” will all match.
* [a-z]+ will match all lowercase letter words irrespective of length, as long as they contain at least one letter. For example, “book” and “boardroom” will both match.

## Groups

All the special characters we just mentioned only affect a single character or a range set. What if we wanted the effect to apply to a section of the expression? We can do this by creating groups using round brackets — (). For example, the pattern book(.com)? will match both “book” and “book.com”, since we’ve made the “.com” part optional.

Here’s a more complex example that would be used in a realistic scenario such as email validation:

* pattern: @\w+\.\w{2,3}(\.\w{2,3})?
* test string: abc.com abc@mail @mail.com @mail.co.ke



## Alternate Characters

In regex, we can specify alternate characters using the “pipe” symbol — |. This is different from the special characters we showed earlier as it affects all the characters on each side of the pipe symbol. For example, the pattern sat|sit will match both “sat” and “sit” strings. We can rewrite the pattern as s(a|i)t to match the same strings.

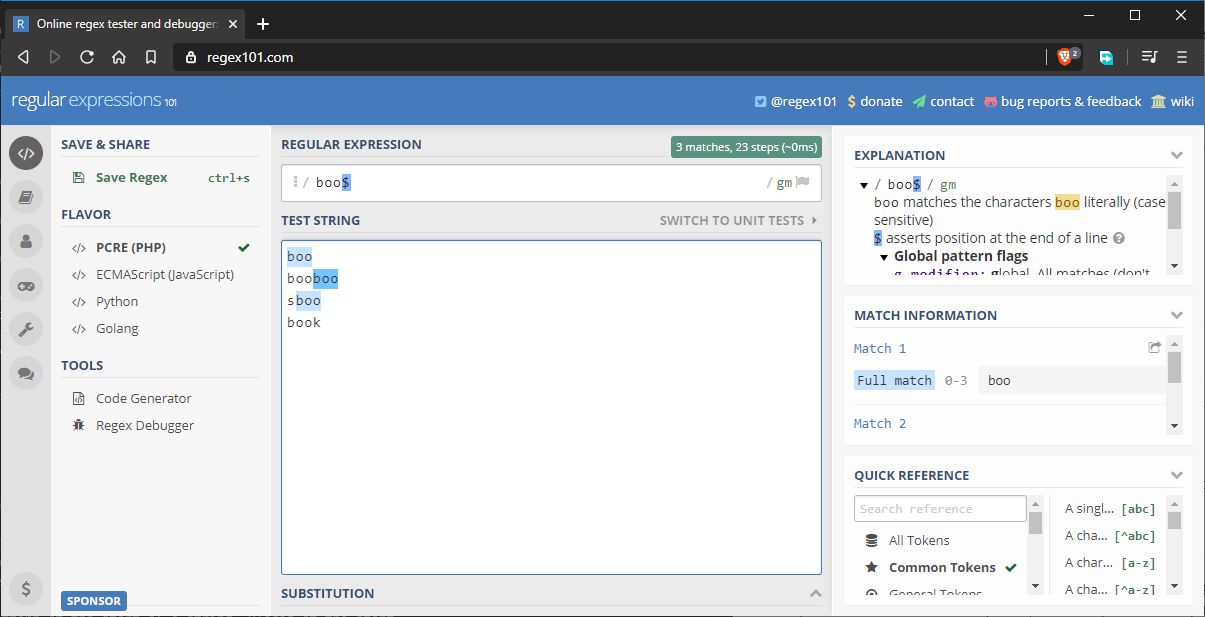
The above pattern can be expressed as s(a|i)t by using () parentheses.

## Starting and Ending Patterns

You may have noticed that some positive matches are a result of partial matching. For example, if I wrote a pattern to match the string “boo”, the string “book” will get a positive match as well, despite not being an exact match. To remedy this, we’ll use the following notations:

* ^: placed at the start, this character matches a pattern at the start of a string.
* $: placed at the end, this character matches a pattern at the end of the string.

To fix the above situation, we can write our pattern as boo$. This will ensure that the last three characters match the pattern. However, there’s one problem we haven’t considered yet, as the following image shows:



The string “sboo” gets a match because it still fulfills the current pattern matching requirements. To fix this, we can update the pattern as follows: ^boo$. This will strictly match the word “boo”. If you use both of them, both rules are enforced. For example, ^[a-z]{5}$ strictly matches a five-letter word. If the string has more than five letters, the pattern doesn’t match.

## Regex in JavaScript

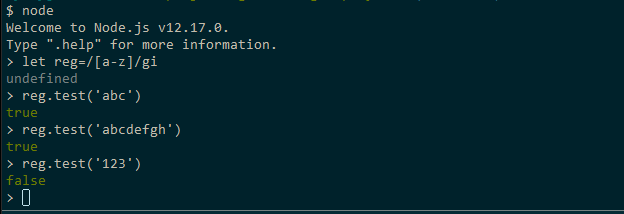
// Example 1

const regex1=/a-z/ig

//Example 2

const regex2= new RegExp(/[a-z]/, 'ig')

If you have Node.js installed on your machine, open a terminal and execute the command node to launch the Node.js shell interpreter. Next, execute as follows:



Feel free to play with more regex patterns. When done, use the command .exit to quit the shell.

## Real World Example: Email Validation

As we conclude this guide, let’s look at a popular usage of regex, email validation. (For example, we might want to check that an email address a user has entered into a form is a valid email address.)

This subject is more complicated than you might think. The email address syntax is quite simple: {name}@{domain}. In theory, an email address can contain a limited number of symbols such as #-@&%. etc. However, the placement of these symbols matters. Mail servers also have different rules on the use of symbols. For example, some servers treat the + symbol as invalid. In other mail servers, the symbol is used for [email subaddressing](https://tools.ietf.org/id/draft-newman-email-subaddr-01.html).

As a challenge to test your knowledge, try to build a regular expression pattern that matches only the valid email addresses marked below:

# invalid email

abc

abc.com

# valid email address

abc@mail.com

abc@mail.nz

abc@mail.co.nz

abc123@mail.com

abc.def@music.com

# invalid email prefix

abc-@mail.com

abc..def@mail.com

.abc@mail.com

abc#def@mail.com

# valid email prefix

abc-d@mail.com

abc.def@mail.com

abc@mail.com

abc\_def@mail.com

# invalid domain suffix

abc.def@mail.c

abc.def@mail#archive.com

abc.def@mail

abc.def@mail..com

# valid domain suffix

abc.def@mail.cc

abc.def@mail-archive.com

abc.def@mail.org

abc.def@mail.com

fully-qualified-domain@example.com

Do note some email addresses marked as valid may be invalid for certain organizations, while some that are marked as invalid may actually be allowed in other organizations. Either way, learning to build custom regular expressions for the organizations you work for is paramount in order to cater for their needs. In case you get stuck, you can look at the following possible solutions. Do note that none of them will give you a 100% match on the above valid email test strings.

* Possible Solution 1:

^\w\*(\-\w)?(\.\w\*)?@\w\*(-\w\*)?\.\w{2,3}(\.\w{2,3})?$

* Possible Solution 2:

^(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)\*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$

## Summary

I hope you’ve now learned the basics of regular expressions. We haven’t covered all regex features in this quick beginner guide, but you should have enough information to tackle most problems that call for a regex solution. To learn more, read our guide on best practices for the [practical application of regex](https://www.sitepoint.com/demystifying-regex-with-practical-examples/) in real-world scenarios.

https://www.sitepoint.com/learn-regex/

# Demystifying RegEx with Practical Examples

By [Nicola Pietroluongo](https://www.sitepoint.com/author/npietroluongo)

[PHP](https://www.sitepoint.com/php)

A regular expression is a sequence of characters used for parsing and manipulating strings. They are often used to perform searches, replace substrings and validate string data. This article provides tips, tricks, resources and steps for going through intricate regular expressions.

If you don’t have the basic skillset under your belt, you can [learn regex with our beginner’s guide](https://www.sitepoint.com/learn-regex-a-beginners-guide). As arcane as regular expressions look, it won’t take you long to learn the concepts.



There are many books, articles, websites and the [PHP official documentation](http://php.net/manual/en/regexp.reference.meta.php) that explain regular expressions, so instead of writing another explanation I’d prefer to go straight to more practical examples. You can find a useful cheat sheet at [this link](https://github.com/niklongstone/regular-expression-cheat-sheet).

Along with a host of useful resources, there is also a conference video by Lea Verou at the bottom of this post – it’s a bit long, but it’s excellent in breaking down RegEx.

## How to build a good regex

Regular expressions are often used in the developer’s daily routine – log analysis, form submission validation, find and replace, and so on. That’s why every good developer should know how to use them, but what is the best practice to build a good regex?

### 1. Define a scenario

Using natural language to define the problem will give you a better idea of the approach to use. The words could and must, used in a definition, are useful to describe mandatory constraints or assertions.

Below is an example:

* The string must start with ‘h’ and finish with ‘o’ (e.g. hello, halo).
* The string could be wrapped in parentheses.

### 2. Develop a plan

After having a good definition of the problem, we can understand the kind of elements that are involved in our regular expression:

* What are the types of characters allowed (word, digit, new line, range, …)?
* How many times must a character appear (one or more, once, …)?
* Are there some constraints to follow (optionals, lookahead/behind, if-then-else, …)?

### 3. Implement/Test/Refactor

It’s very important to have a real-time test environment to test and improve your regular expression. There are websites like [regex101.com](https://regex101.com/), [regexr.com](http://www.regexr.com/) and [debuggex.com](https://www.debuggex.com/) that provide some of the best environments.

To improve the efficiency of the regex, you could try to answer some of these additional questions:

* Are the character classes correctly defined for the specific domain?
* Should I write more test strings to cover more use cases?
* Is it possible to find and isolate some problems and test them separately?
* Should I refactor my expression with subpatterns, groups, conditions, etc., to make it smaller, clearer and more flexible?

## Practical examples

The goal of the following examples is not to write an expression that will only solve the problem, but to write the most effective expression for the specific use cases, using important elements like character ranges, assertions, conditions, groups and so on.

### Matching a password



Scenario:

* 6 to 12 characters in length
* Must have at least one uppercase letter
* Must have at least one lower case letter
* Must have at least one digit
* Should contain other characters

Pattern:

^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d).{6,12}$

This expression is based on multiple positive lookahead (?=(regex)). The lookahead matches something followed by the declared (regex). The order of the conditions doesn’t affect the result. Lookaround expressions are very useful when there are several conditions.   
We could also use the negative lookahead (?!(regex)) to exclude some character ranges. For example, I could exclude the % with (?!.\*#).

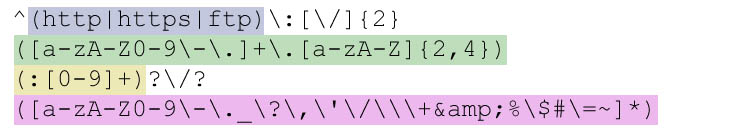
Let’s explain each pattern of the above expression:

### Learn PHP for free!

Make the leap into server-side programming with a comprehensive cover of PHP & MySQL.  
  
Normally ~~RRP $39.99~~ Yours absolutely free

1. ^ asserts position at start of the string
2. (?=.\*[a-z]) positive lookahead, asserts that the regex .\*[a-z] can be matched:
   * .\* matches any character (except newline) between zero and unlimited times
   * [a-z] matches a single character in the range between a and z (case sensitive)
3. (?=.\*[A-Z]) positive lookahead, asserts that the regex .\*[A-Z] can be matched:
   * .\* matches any character (except newline) between zero and unlimited times
   * [A-Z] matches a single character between A and Z (case sensitive)
4. (?=.\*\d) positive lookahead, asserts that the regex \*\dcan be matched:
   * .\* matches any character (except newline) between zero and unlimited times
   * \d matches a digit [0-9]
5. .{6,12} matches any character (except newline) between 6 and 12 times
6. $ asserts position at end of the string

### Matching URL



Scenario:

* Must start with http or https or ftp followed by ://
* Must match a valid domain name
* Could contain a port specification (http://www.sitepoint.com:80)
* Could contain digit, letter, dots, hyphens, forward slashes, multiple times

Pattern:

^(http|https|ftp):[\/]{2}([a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,4})(:[0-9]+)?\/?([a-zA-Z0-9\-\.\_\?\,\'\/\\\+&amp;%\$#\=~]\*)

The first scenario is pretty easy to solve with ^(http|https|ftp):[\/]{2}.   
To match the domain name we need to bear in mind that to be valid it can only contain letters, digits, hyphen and dots. In my example, I limited the number of characters after the punctuation from 2 to 4, but could be extended for new domains like .rocks or .codes. The domain name is matched by ([a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,4}).

The optional port specification is matched by the simple (:[0-9]+)?.

A URL can contain multiple slashes and multiple characters repeated many times (see [RFC3986](http://tools.ietf.org/html/rfc3986" \l "section-2)), this is matched by using a range of characters in a group ([a-zA-Z0-9\-\.\_\?\,\'\/\\\+&amp;%\$#\=~]\*).   
It’s really useful to match every important element with a group capture (), because it will return only the matches we need. Remember that certain characters need to be escaped with \.

Below, every single subpattern explained:

1. ^ asserts position at start of the string
2. capturing group (http|https|ftp), captures http or https or ftp
3. : escaped character, matches the character : literally
4. [\/]{2} matches exactly 2 times the escaped character /
5. capturing group ([a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,4}):
   * [a-zA-Z0-9\-\.]+ matches one and unlimited times character in the range between a and z, A and Z, 0 and 9, the character - literally and the character . literally
   * \. matches the character . literally
   * [a-zA-Z]{2,4} matches a single character between 2 and 4 times between a and z or A and Z (case sensitive)
6. capturing group (:[0-9]+)?:
   * quantifier ? matches the group between zero or more times
   * : matches the character : literally
   * [0-9]+ matches a single character between 0 and 9 one or more times
7. \/? matches the character / literally zero or one time
8. capturing group ([a-zA-Z0-9\-\.\_\?\,\'\/\\\+&amp;%\$#\=~]\*):
   * [a-zA-Z0-9\-\.\_\?\,\'\/\\\+&amp;%\$#\=~]\* matches between zero and unlimited times a single character in the range a-z, A-Z, 0-9, the characters: -.\_?,'/\+&amp;%$#=~.

### Matching HTML TAG



Scenario:

* The start tag must begin with < followed by one or more characters and end with >
* The end tag must start with </ followed by one or more characters and end with >
* We must match the content inside a TAG element

Pattern:

<([\w]+).\*>(.\*?)<\/\1>

Matching the start tag and the content inside it’s pretty easy with <([\w]+).\*> and (.\*?), but in the pattern above I have added a useful thing: the reference to a capturing group.   
Every capturing group defined by parentheses () could be referred to using its position number, (first)(second)(third), which will allow for further operations.   
The expression above could be explained as:

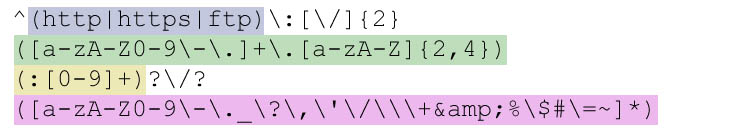
* Start with <
* Capture the tag name
* Followed by one or more chars
* Capture the content inside the tag
* The closing tag must be </tag name captured before>

Including only two capture groups in the expression, the tag name and the content, will return a very clear match, a list of tag names with related content.

Let’s dig a little deeper and explain the subpatterns:

1. < matches the character < literally
2. capturing group ([\w]+) matches any word character a-zA-Z0-9\_ one or more times
3. .\* matches any character (except newline) between zero or more times
4. > matches the character > literally
5. capturing group (.\*?), matches any character (except newline), zero and more times
6. < matches the characters < literally
7. \/ matches the character / literally
8. \1 matches the same text matched by the first capturing group: ([\w]+)
9. > matches the characters > literally

### Matching duplicated words



Scenario:

* The words are space separated
* We must match every duplication – non-consecutive ones as well

Pattern:

\b(\w+)\b(?=.\*\1)

This regular expression seems challenging but uses some of the concept previously shown.   
The pattern introduces the concept of word boundaries.

A word boundary \b mainly checks positions. It matches when a word character (i.e.: abcDE) is followed by a non-word character (Ie: -~,!).   
Below you can find some example uses of word boundary to make it clearer:   
– Given the phrase Regular expressions are awesome   
– The pattern \bare\b matches are   
– The pattern \w{3}\b could match the last three letters of the words: lar, ion, are, ome

The expression above could be explained as:

* Match every word character followed by a non-word character (in our case space)
* Check if the matched word is already present or not

Below you will find the explanation for each sub pattern:

1. \b word boundary
2. capturing group ([\w]+) matches any word character a-zA-Z0-9\_
3. \b word boundary
4. (?=.\*\1) positive lookahead assert that the following can be matched:
   * .\* matches any character (except newline)
   * \1 matches same text as first capturing group

The expression will make more sense if we return all the matches instead of returning only the first one. See the PHP function [preg\_match\_all](http://php.net/manual/en/function.preg-match-all.php) for more information.

## Final thoughts

Regular expressions are double-edged swords. The more complexity is added, the more difficult it is to solve the problem. That’s why, sometimes, it’s hard to find a regular expression that will match all the cases, and it’s better to use several smaller regex instead.

Having a good scenario of the problem could be very helpful, and will allow you to start thinking of the character range, constraints, assertions, repetitions, optional values, etc. Paying more attention to group captures will make the matches useful for further processing. Feel free to improve the expressions in the examples, and let us know how you do!

## Useful resources

Below you can find further information and resources to help your regex skills grow.   
Feel free to add a comment to the article if you find something useful that isn’t listed.

### Lea Verou – /Reg(exp){2}lained/: Demystifying Regular Expressions

<https://www.youtube.com/watch?v=EkluES9Rvak>

### PHP libraries

| Name | Description |
| --- | --- |
| [RegExpBuilder](https://github.com/gherkins/regexpbuilderphp) | Creates regex using human-readable chains of methods |
| [NooNooFluentRegex](https://github.com/tomgray15/NooNooFluentRegex) | Builds Regex expressions using fluent setters and English language terms like above |
| [Hoa\Regex](https://github.com/hoaproject/Regex) | Provides tools to analyze regex and generate strings |
| [Regex reverse](https://github.com/niklongstone/regex-reverse) | Given a regular expression will generate a string |

### Websites

| URL | Description |
| --- | --- |
| [regex101.com](https://regex101.com/) | PCRE online regex tester |
| [regextester.com](http://www.regextester.com/) | PCRE online regex tester |
| [rexv.org](http://www.rexv.org/) | PCRE online regex tester |
| [debuggex.com](https://www.debuggex.com/) | Supports PCRE and provides a very useful visual regex debugger |
| [regexper.com](http://regexper.com/) | Javascript style regex, but useful for debug |
| [phpliveregex.com](http://www.phpliveregex.com/) | Online tester for preg functions |
| [regxlib.com](http://www.regxlib.com/) | Database of regular expressions ready to use |
| [regular-expressions.info](http://www.regular-expressions.info/) | Regex tutorials, books review, examples |

### Books

| Title | Description | Author | Editor |
| --- | --- | --- | --- |
| Mastering Regular Expressions | The must have regex book | Jeffrey Friedl | O’Reilly |
| Regular Expression Pocket Reference | Regular Expressions for Perl, Ruby, PHP, Python, C, Java and .NET | Tony Stubblebine | O’Reilly |

https://www.sitepoint.com/demystifying-regex-with-practical-examples/

# Mini Tutorial RegEx

Sumário:

* [Expressões regulares: introdução](http://turing.com.br/material/regex/introducao.html)
  + [Idéia básica](http://turing.com.br/material/regex/introducao.html" \l "ideia-basica)
  + [Metacaracteres](http://turing.com.br/material/regex/introducao.html" \l "metacaracteres)
  + [Gula × moderação](http://turing.com.br/material/regex/introducao.html" \l "gula-moderacao)
* [Expressões regulares em Python: módulo re](http://turing.com.br/material/regex/python_re.html)
  + [Conceitos](http://turing.com.br/material/regex/python_re.html" \l "conceitos)
  + [funções × métodos](http://turing.com.br/material/regex/python_re.html" \l "funcoes-metodos)
  + [match](http://turing.com.br/material/regex/python_re.html" \l "match-search) × search
  + [Flags: ajustes no funcionamento das regex](http://turing.com.br/material/regex/python_re.html" \l "flags-ajustes-no-funcionamento-das-regex)
* [Exercícios](http://turing.com.br/material/regex/exercicios.html)
  + [1. Especificadores e quantificadores](http://turing.com.br/material/regex/exercicios.html" \l "especificadores-e-quantificadores)
  + [2. Âncoras](http://turing.com.br/material/regex/exercicios.html" \l "ancoras)
  + [3. Agrupamento](http://turing.com.br/material/regex/exercicios.html" \l "agrupamento)
* [Onde aprender mais](http://turing.com.br/material/regex/referencias.html)
  + [Documentação oficial do Python](http://turing.com.br/material/regex/referencias.html" \l "documentacao-oficial-do-python)
  + [Livros](http://turing.com.br/material/regex/referencias.html" \l "livros)
  + [Resumos](http://turing.com.br/material/regex/referencias.html" \l "resumos)
  + [Software](http://turing.com.br/material/regex/referencias.html" \l "software)
* [Glossário](http://turing.com.br/material/regex/glossary.html)

# Expressões regulares: introdução

## Idéia básica

Uma expressão regular é uma notação para representar padrões em strings. Serve para validar entradas de dados ou fazer busca e extração de informações em textos.

Por exemplo, para verificar se um dado fornecido é um número de 0,00 a 9,99 pode-se usar a expressão regular "\d,\d\d", pois o símbolo \d é um curinga que casa com um dígito.

O verbo *[casar](http://turing.com.br/material/regex/glossary.html" \l "term-casar)* aqui está sendo usado tradução para match, no sentido de combinar, encaixar, parear. Dizemos que a expressão \d,\d\d casa com 1,23 mas não casa com 123 (falta a vírgula) nem com 1,2c (“c” não casa com \d, porque não é um dígito).

O termo em inglês é regular expression de onde vem as abreviações regex e re (o nome do módulo Python). Na ciência da computação, o termo tem um significado bem específico.

### Algums exemplos

Veja alguns exemplos com breves explicações para ter uma idéia geral:

\d{5}-\d{3}

O padrão de um CEP como 05432-001: 5 dígitos, um - (hífen) e mais 3 dígitos. A sequência \d é um *[metacaractere](http://turing.com.br/material/regex/glossary.html" \l "term-metacaractere)*, um curinga que casa com um dígito (0 a 9). A sequência {5} é um *[quantificador](http://turing.com.br/material/regex/glossary.html" \l "term-quantificador)*: indica que o padrão precedente deve ser repetido 5 vezes, portanto \d{5} é o mesmo que \d\d\d\d\d.

[012]\d:[0-5]\d

Semelhante ao formato de horas e minutos, como 03:10 ou 23:59. A sequência entre colchetes [012] define um *[conjunto](http://turing.com.br/material/regex/glossary.html" \l "term-conjunto)*. Neste caso, o conjunto especifica que primeiro caractere deve ser 0, 1 ou 2. Dentro dos [] o hífen indica uma faixa de caracteres, ou seja, [0-5] é uma forma abreviada para o conjunto [012345]; o conjunto que representa todos os dígitos, [0-9] é o mesmo que \d. Note que esta expressão regular também aceita o texto 29:00 que não é uma hora válida (horas válidas serão o tema de um dos *[Exercícios](http://turing.com.br/material/regex/exercicios.html" \l "exercicios)*).

[A-Z]{3}-\d{4}

É o padrão de uma placa de automóvel no Brasil: três letras de A a Z é seguidas de um - (hífen) seguido de quatro dígitos, como CKD-4592.

### Sobre os sinais «» usados neste texto

Ao descrever de modo genérico alguma parte da sintaxe das expressões regulares usamos neste documento os símbolos «», para indicar uma parte que deve ser fornecida pelo usuário.

Por exemplo, a referência a um grupo tem a sintaxe \«n» onde «n» é o número do grupo a ser recuperado. Os sinais «» não fazem parte da sintaxe, então a referência ao terceiro grupo escreve-se como \3.

De modo semelhante, a sintaxe de quantificador moderado é «q»?, onde «q» é qualquer quantificador, como \* em \*? ou {1,3} no caso de {1,3}?.

## Metacaracteres

Um *[metacaractere](http://turing.com.br/material/regex/glossary.html" \l "term-metacaractere)* é um caractere ou sequência de caracteres com significado especial em expressões regulares. Os metacaracteres podem ser categorizados conforme seu uso.

### Especificadores

Especificam o conjunto de caracteres a casar em uma posição.

| metacaractere | conhecido como | significado |
| --- | --- | --- |
| . | curinga | qualquer caractere, exceto a quebra de linha \n (ver flag\_dotall) |
| [...] | conjunto | qualquer caractere incluido no conjunto |
| [^...] | conjunto negado | qualquer caractere não incluido no conjunto |
| \d | dígito | o mesmo que [0-9] |
| \D | não-digíto | o mesmo que [^0-9] |
| \s | branco | espaço, quebra de linha, tabs etc.; o mesmo que [ \t\n\r\f\v] |
| \S | não-branco | o mesmo que [^ \t\n\r\f\v] |
| \w | alfanumérico | o mesmo que [a-zA-Z0-9\_] (mas pode incluir caracteres Unicode; ver flag\_unicode) |
| \W | não-alfanumérico | o complemento de \w |
| \ | escape | anula o significado especial do metacaractere seguinte; por exemplo, \. representa apenas um ponto, e não o curinga |

### Quantificadores

Definem o número permitido repetições da expressão regular precedente.

| metacaractere | significado |
| --- | --- |
| {n} | exatamente n ocorrências |
| {n,m} | no mínimo n ocorrências e no máximo m |
| {n,} | no mínimo n ocorrências |
| {,n} | no máximo n ocorrências |
| ? | 0 ou 1 ocorrência; o mesmo que {,1} |
| + | 1 ou mais ocorrência; o mesmo que {1,} |
| \* | 0 ou mais ocorrência |
| «q»? | modera qualquer um dos quantificadores acima (ver *[Gula × moderação](http://turing.com.br/material/regex/introducao.html" \l "gula)*) |

Veja o grupo de exercícios *[1. Especificadores e quantificadores](http://turing.com.br/material/regex/exercicios.html" \l "exer-espec)*.

### Âncoras

Estabelecem posições de referência para o casamento do restante da regex. Note que estes metacaracteres não casam com caracteres no texto, mas sim com posições antes, depois ou entre os caracteres.

| metacaractere | significado |
| --- | --- |
| ^ | início do texto, ou de uma linha com o flag re.MULTILINE |
| \A | início do texto |
| $ | fim do texto, ou de uma linha com o flag re.MULTILINE; não captura o \n no fim do texto ou da linha |
| \Z | fim do texto |
| \b | posição de borda, logo antes do início de uma palavra, ou logo depois do seu término; o mesmo que a posição entre \W e \w ou vice-versa |
| \B | posição de não-borda |

Veja o grupo de exercícios *[2. Âncoras](http://turing.com.br/material/regex/exercicios.html" \l "exer-ancoras)*.

### Agrupamento

Definem ou grupos ou alternativas.

| metacaractere | significado |
| --- | --- |
| (...) | define um *[grupo](http://turing.com.br/material/regex/glossary.html" \l "term-grupo)*, para efeito de aplicação de quantificador, alternativa ou de posterir extração ou re-uso |
| ...|... | alternativa; casa a regex à direita ou à esquerda |
| \«n» | recupera o texto casado no n-ésimo grupo |

## Gula × moderação

Por default, todos os quantificadores são gulosos: tentam casar a maior quantidade possível de caracteres.

Para entender o que isso significa, considere que desejamos capturar o nome do primeiro tag (h1) no fragmento de HTML abaixo:

>>> html = '<h1>Alan Turing: 100 anos</h1>'

Usando o quantificador guloso +, acabamos por capturar o elemento inteiro, e não apenas o tag:

>>> res = re.match('<.+>', html)

>>> res.group()

'<h1>Alan Turing: 100 anos</h1>'

O resultado acima ocorre porque o sinal > casa em duas posições no texto, e casando na segunda posição o curinga guloso .+ captura mais caracteres.

Se usamos o quantificador moderado +?, a expressão .+? fica satisfeita em capturar apenas os caracteres até o primeiro casamento de >:

>>> res = re.match('<.+?>', html)

>>> res.group()

'<h1>'

# Regex cookbook — Top 10 Most wanted regex

The most commonly used (and most wanted) regexes

After some time I thought about publishing this article ([see my previous tutorial](https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285)) with the regex I used most in the projects on which I was involved.

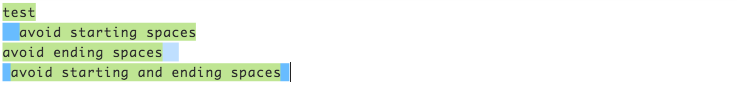
Write in the comments any regex that you would like to add and I will try (if I can) to implement them 😅

Happy coding!!!

## Trim spaces — [try it!](https://regex101.com/r/r4nexX/1)

Matches text avoiding additional spaces





^[\s]\*(.\*?)[\s]\*$

## HTML Tag — [try it!](https://regex101.com/r/tyHta0/2)

Matches any valid HTML tag and the corresponding closing tag





<([a-z]+)([^<]+)\*(?:>(.\*)<\/\1>|\s+\/>)

## Hexadecimal value — [try it!](https://regex101.com/r/4x757Y/1/)

Matches any valid hex color inside text



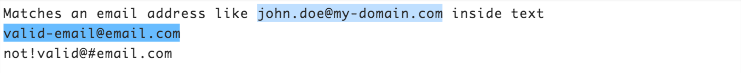


\B#(?:[a-fA-F0–9]{6}|[a-fA-F0–9]{3})\b

## Valid email (RFC5322) — [try it!](https://regex101.com/r/70ARRh/3)

Matches any valid email inside text





\b[\w.!#$%&’\*+\/=?^`{|}~-]+@[\w-]+(?:\.[\w-]+)\*\b

## Username (simple) — [try it!](https://regex101.com/r/yOtMX7/1)

Minimum length of 3, maximum length of 16, composed by letters, numbers or dashes.



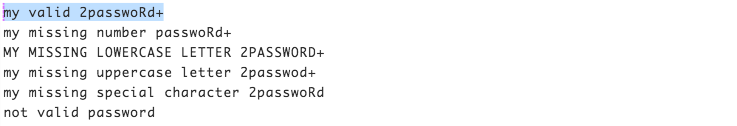


/^[a-z0-9\_-]{3,16}$/

## Strong password — [try it!](https://regex101.com/r/59wfy8/2)

Minimum length of 6, at least one uppercase letter, at least one lowercase letter, at least one number, at least one special character



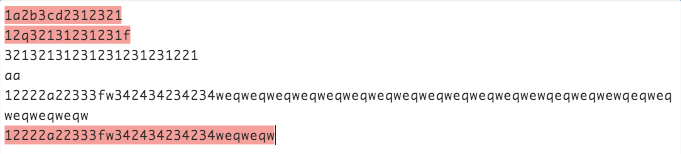


(?=^.{6,}$)((?=.\*\w)(?=.\*[A-Z])(?=.\*[a-z])(?=.\*[0-9])(?=.\*[|!"$%&\/\(\)\?\^\'\\\+\-\\*]))^.\*

## 2 of a kind— [try it!](https://regex101.com/r/euOmcI/2)

At least 2 letters (uppercase or lowercase) at any index, minimum length of 8, maximum length of 32

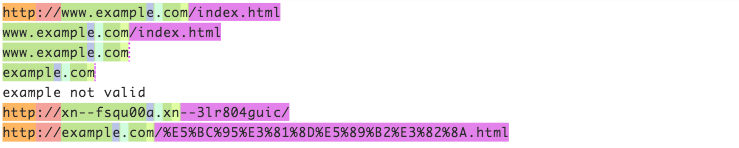




^(?=([0-9]\*[a-z]){2,})([a-zA-Z0-9]{8,32})$

If you want to use capturing groups to get scheme, path, etc. (or add user-info, host, port…) feel free to ask it in comments!



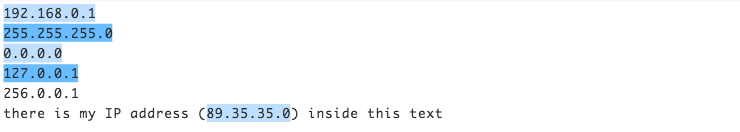


^(((https?|ftp):\/\/)?([\w\-\.])+(\.)([\w]){2,4}([\w\/+=%&\_\.~?\-]\*))\*$

## IPv4 address — [try it!](https://regex101.com/r/y6DWDT/1)

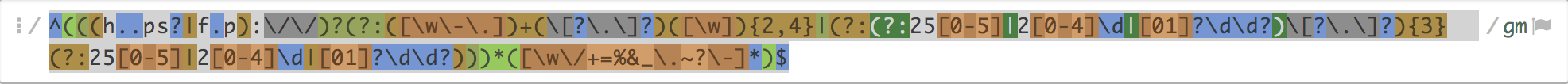
Matches any valid IPv4 address inside text





\b(?:(?:25[0-5]|2[0-4]\d|[01]?\d\d?)\.){3}(?:25[0-5]|2[0-4]\d|[01]?\d\d?)\b

## “Defanged” URL or IPv4 address — [try it!](https://regex101.com/r/6qUtv2/1/)





^(((h..ps?|f.p):\/\/)?(?:([\w\-\.])+(\[?\.\]?)([\w]){2,4}|(?:(?:25[0–5]|2[0–4]\d|[01]?\d\d?)\[?\.\]?){3}(?:25[0–5]|2[0–4]\d|[01]?\d\d?)))\*([\w\/+=%&\_\.~?\-]\*)$

## SSN — Social Security Number (simple) — [try it!](https://regex101.com/r/YGZPLx/1)

If you want to check the [validity of an SSN](https://en.wikipedia.org/wiki/Social_Security_number) feel free to ask in comments!





^((?<area>[\d]{3})[-][\d]{2}[-][\d]{4})$

## **Alpha-numeric, literals, digits, lowercase, uppercase chars only**

\w //alpha-numeric only

[a-zA-Z] //literals only

\d //digits only

[a-z] //lowercase literal only

[A-Z] //uppercase literal only

Have fun and do not forget to recommend the article if you liked it 💚

Do not forget to check out my previous article about [**regex tutorial**](https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285)!

<https://medium.com/factory-mind/regex-cookbook-most-wanted-regex-aa721558c3c1>

# Regex tutorial — A quick cheatsheet by examples

UPDATE 1/2021: “Bracket expressions” and “escape rule” -> Thanks to [MoveUpHealth](https://medium.com/@jessica_49196)

Check out my [REGEX COOKBOOK](https://medium.com/@fox.jonny/regex-cookbook-most-wanted-regex-aa721558c3c1) article about the most commonly used (and most wanted) regex 🎉

Regular expressions (regex or regexp) are extremely useful in extracting information from any text by searching for one or more matches of a specific search pattern (i.e. a specific sequence of ASCII or unicode characters).

Fields of application range from validation to parsing/replacing strings, passing through translating data to other formats and web scraping.

One of the most interesting features is that once you’ve learned the syntax, you can actually use this tool in (almost) all programming languages ​​(JavaScript, Java, VB, C #, C / C++, Python, Perl, Ruby, Delphi, R, Tcl, d many others) with the slightest distinctions about the support of the most advanced features and syntax versions supported by the engines).

Let’s start by looking at some examples and explanations.

# Basic topics

## **Anchors — ^ and $**

^The matches any string that starts with The -> [**Try it!**](https://regex101.com/r/cO8lqs/2)end$ matches a string that ends with end^The end$ exact string match (starts and ends with The end)roar matches any string that has the text roar in it

## **Quantifiers — \* + ? and {}**

abc\* matches a string that has ab followed by zero or more c -> [**Try it!**](https://regex101.com/r/cO8lqs/1)abc+ matches a string that has ab followed by one or more cabc? matches a string that has ab followed by zero or one cabc{2} matches a string that has ab followed by 2 cabc{2,} matches a string that has ab followed by 2 or more cabc{2,5} matches a string that has ab followed by 2 up to 5 ca(bc)\* matches a string that has a followed by zero or more copies of the sequence bca(bc){2,5} matches a string that has a followed by 2 up to 5 copies of the sequence bc

## OR operator — | or []

a(b|c) matches a string that has a followed by b or c (and captures b or c) -> [**Try it!**](https://regex101.com/r/cO8lqs/3)a[bc] same as previous, but without capturing b or c

## Character classes — \d \w \s and .

\d matches a single character that is a digit -> [**Try it!**](https://regex101.com/r/cO8lqs/4)\w matches a word character (alphanumeric character plus underscore) -> [**Try it!**](https://regex101.com/r/cO8lqs/4)\s matches a whitespace character (includes tabs and line breaks). matches any character -> [**Try it!**](https://regex101.com/r/cO8lqs/5)

Use the . operator carefully since often class or negated character class (which we’ll cover next) are faster and more precise.

\d, \w and \s also present their negations with \D, \W and \S respectively.

For example, \D will perform the inverse match with respect to that obtained with \d.

\D matches a single non-digit character -> [**Try it!**](https://regex101.com/r/cO8lqs/6)

In order to be taken literally, you must escape the characters ^.[$()|\*+?{\with a backslash \ as they have special meaning.

\$\d matches a string that has a $ before one digit -> [**Try it!**](https://regex101.com/r/cO8lqs/9)

Notice that you can match also non-printable characters like tabs \t, new-lines \n, carriage returns \r.

## Flags

We are learning how to construct a regex but forgetting a fundamental concept: flags.

A regex usually comes within this form /abc/, where the search pattern is delimited by two slash characters /. At the end we can specify a flag with these values (we can also combine them each other):

* g (global) does not return after the first match, restarting the subsequent searches from the end of the previous match
* m (multi-line) when enabled ^ and $ will match the start and end of a line, instead of the whole string
* i (insensitive) makes the whole expression case-insensitive (for instance /aBc/i would match AbC)

# Intermediate topics

## Grouping and capturing — ()

a(bc) parentheses create a capturing group with value bc -> [**Try it!**](https://regex101.com/r/cO8lqs/11)a(?:bc)\* using ?: we disable the capturing group -> [**Try it!**](https://regex101.com/r/cO8lqs/12)a(?<foo>bc) using ?<foo> we put a name to the group -> [**Try it!**](https://regex101.com/r/cO8lqs/17)

This operator is very useful when we need to extract information from strings or data using your preferred programming language. Any multiple occurrences captured by several groups will be exposed in the form of a classical array: we will access their values specifying using an index on the result of the match.

If we choose to put a name to the groups (using (?<foo>...)) we will be able to retrieve the group values using the match result like a dictionary where the keys will be the name of each group.

## Bracket expressions — []

[abc] matches a string that has either an a or a b or a c -> is the same as a|b|c -> [**Try it!**](https://regex101.com/r/cO8lqs/7)[a-c] same as previous[a-fA-F0-9] a string that represents a single hexadecimal digit, case insensitively -> [**Try it!**](https://regex101.com/r/cO8lqs/22)[0-9]% a string that has a character from 0 to 9 before a % sign[^a-zA-Z] a string that has not a letter from a to z or from A to Z. In this case the ^ is used as negation of the expression -> [**Try it!**](https://regex101.com/r/cO8lqs/10)

## Greedy and Lazy match

The quantifiers ( \* + {}) are greedy operators, so they expand the match as far as they can through the provided text.

For example, <.+> matches <div>simple div</div> in This is a <div> simple div</div> test. In order to catch only the div tag we can use a ? to make it lazy:

<.+?> matches any character one or more times included inside < and >, expanding as needed -> [**Try it!**](https://regex101.com/r/cO8lqs/24)

Notice that a better solution should avoid the usage of . in favor of a more strict regex:

<[^<>]+> matches any character except < or > one or more times included inside < and > -> [**Try it!**](https://regex101.com/r/cO8lqs/23)

# Advanced topics

## **Boundaries — \b and \B**

\babc\b performs a "whole words only" search -> [**Try it!**](https://regex101.com/r/cO8lqs/25)

\b represents an anchor like caret (it is similar to $ and ^) matching positions where one side is a word character (like \w) and the other side is not a word character (for instance it may be the beginning of the string or a space character).

It comes with its negation, \B. This matches all positions where \b doesn’t match and could be if we want to find a search pattern fully surrounded by word characters.

\Babc\B matches only if the pattern is fully surrounded by word characters -> [**Try it!**](https://regex101.com/r/cO8lqs/26)

## Back-references — \1

([abc])\1 using \1 it matches the same text that was matched by the first capturing group -> [**Try it!**](https://regex101.com/r/cO8lqs/14)([abc])([de])\2\1 we can use \2 (\3, \4, etc.) to identify the same text that was matched by the second (third, fourth, etc.) capturing group -> [**Try it!**](https://regex101.com/r/cO8lqs/15)(?<foo>[abc])\k<foo> we put the name foo to the group and we reference it later (\k<foo>). The result is the same of the first regex -> [**Try it!**](https://regex101.com/r/cO8lqs/16)

## Look-ahead and Look-behind — **(?=)** and **(?<=)**

d(?=r) matches a d only if is followed by r, but r will not be part of the overall regex match -> [**Try it!**](https://regex101.com/r/cO8lqs/18)(?<=r)d matches a d only if is preceded by an r, but r will not be part of the overall regex match -> [**Try it!**](https://regex101.com/r/cO8lqs/19)

You can use also the negation operator!

d(?!r) matches a d only if is not followed by r, but r will not be part of the overall regex match -> [**Try it!**](https://regex101.com/r/cO8lqs/20)(?<!r)d matches a d only if is not preceded by an r, but r will not be part of the overall regex match -> [**Try it!**](https://regex101.com/r/cO8lqs/21)

# Summary

As you’ve seen, the application fields of regex can be multiple and I’m sure that you’ve recognized at least one of these tasks among those seen in your developer career, here a quick list:

* data validation (for example check if a time string i well-formed)
* data scraping (especially web scraping, find all pages that contain a certain set of words eventually in a specific order)
* data wrangling (transform data from “raw” to another format)
* string parsing (for example catch all URL GET parameters, capture text inside a set of parenthesis)
* string replacement (for example, even during a code session using a common IDE to translate a Java or C# class in the respective JSON object — replace “;” with “,” make it lowercase, avoid type declaration, etc.)
* syntax highlightning, file renaming, packet sniffing and many other applications involving strings (where data need not be textual)

Have fun and do not forget to recommend the article if you liked it 💚

UPDATE! Check out my new [REGEX COOKBOOK](https://medium.com/@fox.jonny/regex-cookbook-most-wanted-regex-aa721558c3c1) about the most commonly used (and most wanted) regex 🎉

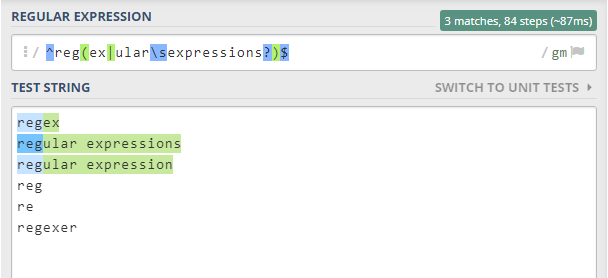
<https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285>

# Regex Tutorial - A Cheatsheet with Examples!

Regular expressions or commonly called as Regex or Regexp is technically a string (a combination of alphabets, numbers and special characters) of text which helps in extracting information from text by matching, searching and sorting.

It can also be used to replace text, regex define a search pattern which is used for find and find and replace in string operations.

If you want to learn Regex with Simple & Practical Examples, I will suggest you to see this simple and to the point [Complete Regex Course](https://www.udemy.com/course/learn-regular-expressions-in-online-regex-course/?referralCode=4C7CF6703664BE3944C0) with step by step approach & exercises. This video course teaches you the Logic and Philosophy of Regular Expressions from scratch to advanced level.

This is an online Regex tutorial for learning Regular expressions effectively and efficiently with examples and exercises. Regular expressions are very useful tool and they are not that much difficult to learn however on internet their are not that much ample resources to learn regex online. This is an attempt to explain regex and make them simple. So if you want to learn regex QUICK and EFFICIENT you are at the right place. Our methodology in this regex course is easy and step by step to understand the philosophy and logic. You can learn regex in just one day or a weekend. Just give it a try.  
  
  
Regex are that much important that most of the programming languages like Python, Java, Javascript, PERL, PHP, Golang, C and C++ etc have regex engines to process regex. Regular expressions is a skill that is must for all programmers, network engineers, network administrators and all those who deal with data, who manage process store search and sort data. Regular expressions will make your work Fast, tidy and save you hundreds and thousands of hours of work. So if you are an IT related individual, not knowing Regex means you are missing a lot. Make a decision and get a skill for better job prospects and career advancement.   
  
  
A simple regex can be /tree/ or even /a/ i.e the word tree or the letter a. Complex regex can take many forms. You don't have to worry as you will study from simple alphabets and numbers and move up the ladder step by step easy way learning complex forms of regular expressions.   
  
A regex for matching the words    regular expression    regular expressions    and    regex    will be   
  
reg(ex|ular\sexpressions?)  
  
Lets look here in regex engine the full code with global multiline mode represented by gm  
  
  
  
  
  
I will again say, don't worry you will learn all this in just a day or two. You can see it is matching the specified three words only and leaving similar words. Beautiful, isn't it?

### Why Learn Regular Expressions?

* For parsing and extracting data from text. Like checking if user inputs a valid email id or card no.
* Regex are fast and they also save your precious time.
* They help you write effective code.
* If you deal with any data, they help you to manage data efficiently making searching and sorting easier and picking up the right information from huge volumes of data.
* They are every where.
* They make you feel if you are a wizard or magician, doing work of hours and hours in minutes.
* Career Advancement and better job prospects
* Regular Expressions are Fun! Learn REGEX and you will love it

### Regex Engines:

A Regular expression engine is infact a computer program, which might run as indenpendent standalone application on your PC or Mac or run inside a web browser from a server or as offline browser extension and processes regular expressions for you. It will match data as suggested in regex. Mostly an engine work for one flavor of regex or a couple of flavors. These engines are not fully compatible and hence you need to know the subtle differences in your specific regex engine.   
  
Here is a quick tutorial for you to learn regular expressions, you can see the details about each individual topic also.

## Regex Tools and Environment:

You are going to use an online Regex tool which is available for free and is quite easy to use and very helpful in sorting out how the regex is working. Open your internet browser and type  regex101.com or simply write regex101 in google or any other search engine, the results of search will have this site. Simply click it and now you are there, ready to work. The working area will look like this



The first textfield is for writing regular expression and after that there is an area to write the Test string from which you are going to search for a match. For the time being that introduction is good enough. Later, I will tell you more about the regex101 engine and its characteristics.

## Regular Expressions Syntax:

The first important thing is regular expression syntax. How to write a regex? what are the syntactical rules and how to follow them. Well there aren't many, in most regex engines the regex starts with a forward slash and ends with a forward slash, like javascript, Php Regex engine.

/regex/

In some regex engines like Python Re module the regex is encapsulated with inverted commas.

r"regex"

However in most cases forward slash at start and end are used and this pattern is followed here.

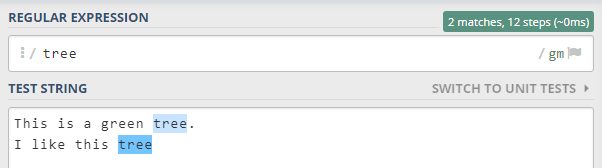
Usually there are one or more modes applied to the regex. The syntax of mode is that after the second forward slash mode is written. like

/regex/ mode

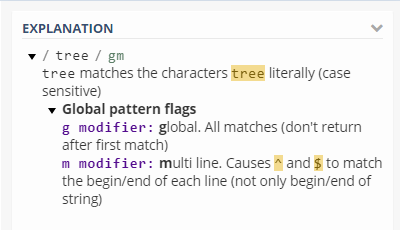
A more practical application is

/tree/ gm

where g is global mode and m is multiline mode and tree is the text to match.



And here is the explanation as shown in  regex engine. We will discuss modes later.



## Literal Characters

The simplest match is a literal character match. Literal means what it appears to be.

a means alphabet  a

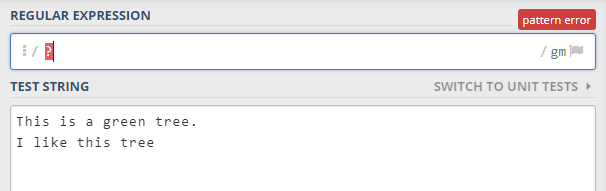
c means single character  c

tree means the word tree  
  
A single literal character matches the first occurence of that character in the string in standard mode.   
  
If the test string is I like driving car fast. The regex is going to match first occurence of a from left to right and that is a in car. This a is in the center of word car but it doesn't matter to regex engine. It will simply match any first occurence of a in the test string. However, if you don't want to match a in the middle of a word you can tell regex by setting boundaries. Then instead of      / a /   the regex will be   / \ba\b /    at this point this is just an illustration that you can also match single characters. Anchoring and boundaries are discussed in detail later and also as a seperate topic on this site.  
  
The regex engine can find all the occurences of a instead of the first a. You can do so by changing the mode to global i.e adding g after the second forward slash like     / a / g

## Metacharacters

Mostly English alphabets upper and lower case, numbers, underscore show literal behavior when used alone or as a combination of words meaningful or meaningless.  
  
ABCDEFGHIJKLMNOPQRSTUVWXYZ  
  
abcdefghijklmnopqrstuvwxyz  
  
0123456789

But not all characters behave like a or b or tree etc. If you try to match a question mark **?** or backslash **\** or square bracket **[** or plus sign **+** etc you will not be able to match them literally. Writing these symbols in regex field is going to raise a pattern error like this



These symbols which don't have a literal match are called metacharacters as they have special meaning in regular expressions. Most of them raise a pattern error when used alone. Metacharacter means special characters. These special characters have special meanings in regex and without knowing their specific behavior you can not use them in your regex as it will generate error or will not give the expected results. Like period or dot is metacharacter, but if you use it as a single character in regex with global mode, it will not raise a pattern error but it will match everything, yes literally everything in your test string instead of matching a dot or period, though it will also match the dot or period but it will not leave anything matching all characters in your test string. This is an example of unexpected results.

## Regular Expressions Modes:

Modes in Regex play an important role. There are many modes however the important ones are discussed below.

1. First is standard mode and it means no mode. It has no symbol the syntax is

/regex/

and it will match the first occurence of regex pattern.

2. If you want to match all occurences of a single character or word you may use global mode represented by g. The syntax of global mode is

/ a / g

using g for global mode after the second or closing forward slash results in matching all occurences of a instead of first occurence match.

3. Another important mode is case insensitive mode represented by i. Its syntax is

/ Tree / i

As expected by its name, it results in a match which is case insensitive, i.e. it will match Tree as well as tree

4. Next are single line mode represented by s and multiline mode represented by m. Single line mode treats your string as one line instead of multiple lines and matches for your search pattern. In single line mode dot or period matches all including newline. Its syntax is

/ tree / s

5. While a multiline mode will treat a string consisting of multiline and hence if you want to have matches at individual lines you may use multiline mode. A multiline mode anchors regex with ^ and $. Its syntax is

/ tree / m

## Escaping Metacharacters:

As discussed earlier metacharacters are special characters with special meanings and behavior. Some of the metacharacters are  ?    [    /    \    +    \*. But question is in case you want to match a metacharacter then what to do, as writing a single metacharacter is usually going to raise a pattern error. Well the answer is very simple. We match a metacharacter by simply placing a backslash just before it. Using a backslash before these metacharacters make them literal characters and then you can simply match them in test string. Also by using a backslash before metacharacter makes the special behavior of that single character metacharacter go away.

At this point, I should tell another group of metacharacters which are called complex metacharacters or metacharacters of more than one character. Its kind of funny that these metacharacters are metacharacters when we use backslash before them. Some of the examples are \d    \w    \s    \D    \W    \S

Now these are metacharacters with special meanings like \d is used to match digits from 0 to 9. Now if we drop backslash before these metacharacters, they become literal characters. Like \d is a metacharacter dropping backslash it will be d a literal character.

As discussed above \d is used to match numbers 0 to 9. \D is used to match anything but numbers 0 to 9.

\w matches word characters which include a to z upper and lower case, 0 to 9 and underscore.

\W matches anything but \w

\s matches whitespace like space, tab etc.

\S matches all but \s

## Period or dot

First see how dot or period works in regex.



As you can see dot matches all characters, therefore it may be called as wildcard character as it matches all.

Dot is the most commonly used metacharacter in regular expressions and it is also the most commonly misused metacharacter. Dot matches a single character at one time and if standard mode is used it will match only one character without looking what that character is as dot will match the first occurence from left to right and it will match the first character whatever it is.

In global mode, it is going to match every character leaving line break. Yes, dot does not match line break by default. It is the only thing it does not match. If you want dot or period to match line breaks change mode to single line or s. In this mode dot is going to match line break also. Javascript and vbscript do not have this facility of dot matching all. However, in most of regex flavors line break can also be matched with dot in single line mode.

## Non Printable characters:

There are certain characters which are non printable. Like tab, new line or line break, carriage return etc. For matching these non printable characters, you have to specify in regex about these characters.

For matching a tab usually \t metacharacter is used.

\f is used to match form feed

\n is for new line character

\r is for carriage return

\v is used in many flavors for a vertical tab.

\a is used for bell

\e is to escape.

Moreover, unicode can also be used for non printable characters. In case the regex engine you are using supports unicode you may use \uFFFF for a unicode character. If your engine doesn't support unicode, you may use \xFF to match a specific character by using its base sixteen hexadecimal index in the character set. Every character which is non printable can be matched in regex or in or part of a set.

## Character Class

With the help of a character class or character set you may direct regex to match only one character out of two or more characters. The syntax of character class is to put two or more characters inside square brackets

[class]

You may put any number of characters in a character class however your regex engine is going to match only one out of a given set of characters.

Like you want to match     affect and effect

/ [ae]ffect / g

The regex will match both affect as well as effect.

Similarly words with two different acceptable spellings can be matched by this

if you want to match  gray     or      grey

/ gr[ae]y / g

Some words are misspelled for example the word separate four possible forms are

separate

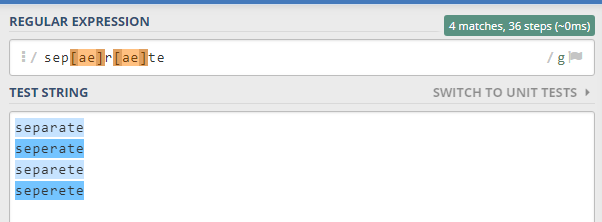
seperete

separete

seperate

to match any variant of separate the regex should be

/ sep[ae]r[ae]te / g



Character ranges:

Instead of writing a large number of characters in a character class you may define ranges. Like instead of writing

[abcdefghij]ar

you can simply write

[a-j]ar

and instead of writing

[ABCDEFGHIJKLMNOPQRSTUVWXYZ] you may use  [A-Z] similarly with lower case letters from a to z you may use the character class [a-z].

For numbers you may use [0-9] instead of [0123456789]

Writing a series makes work a bit easy and code a bit tidy and enhances clarity.

Negated Character Class:

After character class series you may use negated character class. A negated character class means don't match any character given in the set, or in other words match any character that is not in the character class.

The syntax of a negated character class is like

[^class]

using a caret just after the opening square bracket and then writing the character class.

[^abc]ip

will not match aip bip cip but it will match any character followed by ip

hence dip eip fip gip etc etc all will be matches

a[^b] does not mean a not followed by b, it means a followed by a character that is not b.

Metacharacters inside a character class:

Metacharacters have special meaning in regex and most raise a pattern error if used alone. In a character class there are certain metacharacters ] \ ^ - these four are metacharacters inside a character class with some special meanings, all other metacharacters are literal characters inside a character class.

## Quantifiers:

Quantifiers are used for repetition. If you want to repeat a certain character there are quantifiers available. For in detail topic of quantifiers and repetition please see detailed topic.

If you want to match    flavor and flavour you may use

/ flavou?r /

? as a metacharacter here means zero or 1 repetition. It simply looks either that particular character is present or not. It makes the character as optional, the regex will select if the character is there, and it will also match if the character is not in the test string.

For zero or more repetition \* is used. This metacharacter will match in case a character has no existence to any number of occurence.

.\* is going to select any number of characters.

For one or more repetition + is used. + metacharacter will match only if the character or group before + occurs atleast once, however it will match any additional occurences.

\d+ will match one digit up to any digit number. \d is the shorthand character class for 0-9 and + means one or more repetition. \d+ is going to match all numbers in the text string.

Another form of repetition is used with braces. There are three cases.

1. {fixed}

2. {min,max}

3. {min,}

The first case with fixed number of repetition. In this scenario use braces and with in braces write in digits the total number of repetitions required to match. For example if you want to match all four digit numbers simply write

/ \d{4} /

This regex will match 1234    4567    2589    5478

but it will not match 12    258    357    8 etc

In the second case if you want to match numbers with minimum of two digits and maximum of five digits then the regex will be

/ \d{2,5} /

This regex will match all numbers which are two digit, three digit, four digit and five digit and it will not match one digit or six digit numbers.

The last case is when you have minimum number of digits required in a number but you don't have information about the maximum number of digits. In that case {min,} is used.

This is a brief overview of repetition.

## Anchors:

If you want to match a position instead of any character, use anchors. Some of the commonly used anchors are \b ^ and $. ^ matches the start of a string while $ matches the end of a string. If you have multiple lines then you may opt for multiline mode i.e m after the second forward slash.  / regex / m

Some of the examples

/ ^regex$ / mg

/ ^\d{4}$ / mg

If you want to match a word bounday, use \b. A word boundary is a character other than \w. \w is [a-zA-Z0-9\_] now either one or more occurence of word character in a series will be matched until a non word character comes which is usually a space and that will be the boundary. \B matches every position where \b does not match. The word boundary is not always a space it can be any other character not included in \w shorthand character class.

To match with word boundary

/ \bregex\b /mg

## Alternation:

Alternation is used for or operation. It results in selection one option out of a number of available options. These options are seperated by | the pipe symbol. Any number of options can be written with in the alternation group. However an alternation group should contain atleast two options otherwise there is no used of alternation.

/ I\slike\s(java|python) /g

It will match two strings

I like java

I like python

if I like is a complete match then after that if it is java or python in both cases it will be a match.

some other examples can be

/ (regex|regular expression|regular expressions) / g

/ (car|truck|bus|airplane|rocket) / g

The first expression will match either regex or regular expression or regular expressions while second example will match any word out of car truck bus airplane rocket.

## Groups:

Groups help in applying different operations on a collection instead of a single character. Groups are created by putting parenthesis around a collection of tokens or symbols and later you may apply conditions on these groups. For example you may apply a quantifier to a whole group like + or \* or even ?

The whole match is referred to as group 0, if you create a single group it is called group 1. In case of more than one group in regex the regex engine names them as 1,2,3 starting from left to right.

/ ^(\d{3}) - (\d{2}) - (\d{5})$ / g

Here,

group 0    (\d{3}) - (\d{2}) - (\d{5})

group 1    (\d{3})

group 2    (\d{2})

group 3      (\d{5})

If you have to repeat a group instead of writing the regex for that simply write group number and a backslash before it.

/ ^(\d{3}) - (\d{2}) - (\d{5}) \2 \1$ / g

This is also known as backreferencing. Backreferencing is quite helpful in many situations, it also makes your regex look less clumsy and enhances clarity.

If you don't want to capture any group simply place ?: at the start of group regex engine will not number that group i.e. it will not capture that group and it is a non capturing group. An example is like

(?:non-captured group)

## Assertions & Lookaround:

There are certain instances when a certain word before or after match is required. This is called lookaround assertions. They are extremely helpful. Lookaround is of two types lookahead and lookbehind. Lookahead can be positive or negative. Similarly lookbehind can also be positive or negative. Lookaround is technically a group. The regex inside the parenthesis is matched as usuall, however after matching the regex engine looks if a certain word follows it or precedes it or follows and precedes it. In case of a positive reply the match is declared as successful.

First the syntax:

Positive lookahead           ?=

Negative lookahead          ?!

Positive lookbehind          ?<=

Negative lookbehind         ?<!

Positive lookahead is to check the presence of an element or character after the given character or group.

a(?=b) will match a in abc but will not match a in acb or bac

Negative look ahead is to see if a certain element does not follow the match

y(?!z) will match y in xyz but will not match y in zyx

Similarly for Positive and negative lookbehind

(?<=y)z will match z in xyz but will not match z in zyx

(?<!y)z will match z in yxz but will not match z in xyz

<https://www.regextutorial.org/>

# Regex Tutorial with Examples

[Deepanshu Bhalla](https://www.linkedin.com/in/deepanshubhalla) [1 Comment](https://www.listendata.com/2018/07/regular-expression-tutorial-with-examples.html" \l "comment-form) [nlp](https://www.listendata.com/search/label/nlp), [regex](https://www.listendata.com/search/label/regex), [Text Mining](https://www.listendata.com/search/label/Text Mining)

This tutorial covers various concepts of regular expression (regex) with hands-on examples. It also includes usage of regex using various tools such as R and Python.

## Introduction

regex is an acronym for 'Regular Expression'. It is mainly used in extracting sub-string from string by searching a specific search pattern. The search pattern is defined by regular expression.  
  
The search pattern can be finding a single letter, a fixed string or a complex pattern which consists of numeric, punctuation and character values.

Regular expressions can be used to search and replace text.

|  |
| --- |
|  |
| Regex Made Easy |

## Uses of Regular expression

There are several use-cases of regular expression in real-world. Some of them are as follows -

1. Fetch email addresses mentioned in the long paragraph
2. Validate 10-digit phone number, Social Security Number and email address
3. Extract text from HTML or XML code
4. Rename multiple files at a single run
5. Remove punctuation specified in the text
6. Web scraping : Searching specific content from all the web pages that contain a specific string
7. Replace complex pattern with blank or specific character

***Lets start with the basics***

### 1. Anchor and Word Boundaries

|  |  |
| --- | --- |
| Symbol | Description |
| ^ | Beginning of line |
| $ | End of line |
| \b | Whole word |

**Examples**  
  
1. ^abc matches the string that begins with **abc** in text **'abc**d**'**  
[**Test it yourself!**](https://regex101.com/r/n6YLNC/1)  
  
2. ^the matches the string that starts with **the** in text **'the** beginning**'**  
[**Test it yourself!**](https://regex101.com/r/n6YLNC/2)  
  
3. done$ matches the string that ends with **done** in text **'**I am **done'**  
[**Test it yourself!**](https://regex101.com/r/i0RMfk/1)  
  
4. \ban\b matches the whole word **an** in text **'**Elephant **an** animal**'**  
\ban\b does not match **an** from Eleph**an**t and **an**imal as it only perform the whole word searching.  
 [**Test it yourself!**](https://regex101.com/r/1LEHqF/1/)

### 2. OR Condition

OR condition can be defined by symbols | or [ ]. See the examples below.  
  
1. the[m|n] matches strings **them or then** in text **'them then** there **them**e**'**  
[**Test it yourself!**](https://regex101.com/r/RkTv6D/1)  
  
2. the[mn] is equivalent to **the[m|n]**  
[**Test it yourself!**](https://regex101.com/r/RkTv6D/2)  
  
3. \bthe[mn]\b matches the complete **them** or **then** in text **'them then** there theme'  
[**Test it yourself!**](https://regex101.com/r/RkTv6D/3)

### 3. Case Insensitive

Search patterns mentioned in all of the above examples are **case-sensitive**. To make it case insensitive, we have to use the expression (?i)  
  
1. (?i)abc matches both **abc** and **ABC** in text **'abc ABC'**  
[**Test it yourself!**](https://regex101.com/r/RkTv6D/4)  
  
2. (?i)a[bd]a performs insensitive match 'a' followed by either b or d and then a in text 'abc **ABA Ada'**  
[**Test it yourself!**](https://regex101.com/r/RkTv6D/5)

### 4. Quantifiers

It talks about quantity of element(s). In simple words, it means how often a particular regex element can occur.

|  |  |
| --- | --- |
| Expression | Description |
| \* | Item occurs zero or more times |
| + | Item occurs one or more times |
| ? | Item occurs zero or one time |
| {A} | Item occurs A number of times |
| {A,B} | Item occurs between A and B times |
| . | Any character |
| .\* | Matches zero or more of any character |

1. def\* matches strings that contains **de** then followed by **f** zero or more times. Example - **de** **def** **deff defff**  
[**Test it yourself!**](https://regex101.com/r/Cs1Twq/2)  
  
2. def+ matches strings having de then followed by **f** at least 1 time. Example - **def deff defff**  
[**Test it yourself!**](https://regex101.com/r/Cs1Twq/1)  
  
3. \bdef?\b matches strings having exact match of whole **de** then followed by **f** zero or one time. Example - **de def**  
[**Test it yourself!**](https://regex101.com/r/Cs1Twq/3)  
  
4. \bdef{2}\b matches strings having exact match of **de**then followed by **f**exactly two times. Example - **deff**  
[**Test it yourself!**](https://regex101.com/r/Cs1Twq/3)  
  
5. \bdef{2,}\b matches strings having exact match of **de**then followed by **f**two or more times. Example - **deff defff**  
[**Test it yourself!**](https://regex101.com/r/Cs1Twq/4)  
  
6. \bdef{3,4}\b matches strings having exact match of **de**then followed by **f**either 3 or 4 times. Example - **deff defff**  
[**Test it yourself!**](https://regex101.com/r/Cs1Twq/5)  
7. a.\* matches all characters after **a**  
[**Test it yourself!**](https://regex101.com/r/Cs1Twq/6)

### 5. Create Grouping

By using regular expression inside **( )**, you can create a group which would let you apply OR condition to portion of regex or you can put in quantifier to the entire group.  
  
It also helps to extract a portion of information from strings.  
  
ab(cd|de)\* matches strings having **ab**then followed by either **cd** or **de** zero or more times.  
[**Test it yourself!**](https://regex101.com/r/Cs1Twq/7)

### 6. Back Reference

(name)\1 matches text '**name'**that is matched first.  
[**Test it yourself!**](https://regex101.com/r/ZCAybT/2)

**Replace (Substitution) using Back-reference**

(ab|cd)e(fg|hi) matches either **ab** or **cd** then followed by **e** then either fg or hi  
Enter \1\2 in substitution, it will return values of first and second group.  
[**Test it yourself!**](https://regex101.com/r/ZCAybT/3)

### 7. Lazy Quantifier

Lazy (or non-greedy) quantifier matches a regex element as few times as possible. However greedy quantifier matches a regex element as many as possible.

You can covert a greedy quantifier into a lazy quantifier by simply adding a **?**

<.\*?> matches strings having <character(s) >.

|  |
| --- |
|  |
| Regex lazy quantifier |

[**Test it yourself!**](https://regex101.com/r/ZCAybT/2)

### 8. How to program literal meaning of dot, asterisk

By using **backslash \**  you can avoid asterisk and dot. In other words, it makes regex understand the literal meaning of character.

**abc\\*** means abc\* not abcc

[**Test it yourself!**](https://regex101.com/r/o4I57u/1)  
In R programming language, you need to add one more backslash abc\\\* to make R understand the true meaning of asterik here.

### 9. POSIX Regular Expressions

POSIX expressions use square brackets. Like regular expressions, it matches characters, digits, punctuations and many more

|  |  |  |
| --- | --- | --- |
| POSIX | Description | ASCII |
| [:digit:] | Digits | [0-9] |
| [:lower:] | Lowercase letters | [a-z] |
| [:upper:] | Uppercase letters | [A-Z] |
| [:alpha:] | Lower and uppercase letters | [a-zA-Z] |
| [:alnum:] | Lower and uppercase letters and digits | [a-zA-Z0-9] |
| [:blank:] | Space and tab | [ \t] |
| [:space:] | All whitespace characters, including line breaks | [ \t\r\n\v\f] |
| [:punct:] | Punctuations | "[!\#$%()\*+,\-./:;?@\\\\]^\_'{|}~]" |

**Select string having first letter character followed by numeric**

[[:alpha:]][[:digit:]]+

1. [[:alpha:]] means any letter character
2. [[:digit:]] means any digit
3. +  means previous one or more time

[**Test it yourself!**](https://regex101.com/r/hdgHcQ/1)

Find first match of character

Suppose you have text x = "Hello How are you doing? are you okay? I am fine" You need to extract till first question mark ?.

1. str\_extract(x, "Hello.\*\\?")

returns "Hello How are you doing? are you okay?" Here \\? to find literal question mark (not regex question mark).

1. str\_extract(x, "Hello.\*?\\?")

returns "Hello How are you doing?" Here **?** in .\*? makes it non-greedy so it stops after first question mark.

How to use regex with R and Python

**R**  
 **1. grep(pattern, x)**  
Search for a particular pattern in each element of a vector x  
  
**2. gsub(pattern, replacement, x)**  
Replace a particular pattern in each element of a vector x

x = "sample text B2 testing B52"  
gsub('[[:alpha:]][[:digit:]]+', '',x)

**Python**  
  
The package **re** can be used for regular expressions in Python.  
  
**1. re.search(pattern, x)**  
Search for a particular pattern in each element of a vector x  
  
**2. re.sub(pattern, replacement, x)**  
Replace a particular pattern in each element of a vector x

import re  
x = 'Welcome to Python3.6'  
re.sub( '[a-zA-Z]+[0-9|.]+','', x)

**Exercises : Regular Expression**

### 1. Replace abbreviation of thousand (K) with 000?

x = "K 25K 2K"  
**Desired Output :** K 25000 2000

Show Solution

gsub('([0-9])K', '\\1000',x)  
  
Using two backward slash as a single backward slash not allowed in R

### 2. Remove extra characters

x = "var1\_avg\_a1 var1\_a\_avg\_7"  
**Desired Output :**var1 var1\_a

Show Solution

gsub('\_avg\_.\*?[0-9]', '',x)  
  
? making the regular expression non-greedy (lazy) quantifier

<https://www.listendata.com/2018/07/regular-expression-tutorial-with-examples.html>

# PHP Regex for Web Developers

Regular expressions are a very useful tool for developers. They allow to find, identify or replace a word, character or any kind of string. This tutorial will teach you how to master PHP regexp and show you extremely useful, ready-to-use PHP regular expressions that any web developer should have in his toolkit.

**Table of Contents** [show](https://catswhocode.com/php-regex/)

## Getting Started With Regular Expressions

For many beginners, regular expressions seem to be hard to learn and use. In fact, they’re far less hard than you may think. Before we dive deep inside regexp with useful and reusable codes, let’s quickly see the basics of PCRE regex patterns:

### Regular Expressions Syntax

A regular expression (regex or regexp for short) is a special text string for describing a search pattern. A regex pattern matches a target string. The following table describes most common regex:

|  |  |
| --- | --- |
| Regular Expression | Will match… |
| foo | The string “foo” |
| ^foo | “foo” at the start of a string |
| foo$ | “foo” at the end of a string |
| ^foo$ | “foo” when it is alone on a string |
| [abc] | a, b, or c |
| [a-z] | Any lowercase letter |
| [^A-Z] | Any character that is not a uppercase letter |
| (gif|jpg) | Matches either “gif” or “jpg” |
| [a-z]+ | One or more lowercase letters |
| [0-9.-] | Any number, dot, or minus sign |
| ^[a-zA-Z0-9\_]{1,}$ | Any word of at least one letter, number or \_ |
| ([wx])([yz]) | wy, wz, xy, or xz |
| [^A-Za-z0-9] | Any symbol (not a number or a letter) |
| ([A-Z]{3}|[0-9]{4}) | Matches three letters or four numbers |

### PHP Regular Expression Functions

PHP has many useful functions to work with regular expressions. Here is a quick cheat sheet of the main PHP regex functions. Remember that all of them are case sensitive.

For more information about the native functions for PHP regular expressions, have a look at the [manual](https://www.php.net/manual/en/reference.pcre.pattern.syntax.php).

|  |  |
| --- | --- |
| Function | Description |
| preg\_match() | The preg\_match() function searches string for pattern, returning true if pattern exists, and false otherwise. |
| preg\_match\_all() | The preg\_match\_all() function matches all occurrences of pattern in string. Useful for search and replace. |
| preg\_replace() | The preg\_replace() function operates just like ereg\_replace(), except that regular expressions can be used in the pattern and replacement input parameters. |
| preg\_split() | Preg Split (preg\_split()) operates exactly like the split() function, except that regular expressions are accepted as input parameters. |
| preg\_grep() | The preg\_grep() function searches all elements of input\_array, returning all elements matching the regex pattern within a string. |
| preg\_ quote() | Quote regular expression characters |

## Validate a Domain Name

Case sensitive regex to verify if a string is a valid domain name. This is very useful when validating [web forms](https://catswhocode.com/bootstrap-forms/).

$url = "http://komunitasweb.com/";

if (preg\_match('/^(http|https|ftp)://([A-Z0-9][A-Z0-9\_-]\*(?:.[A-Z0-9][A-Z0-9\_-]\*)+):?(d+)?/?/i', $url)) {

echo "Your url is ok.";

} else {

echo "Wrong url.";

}

» [**Source**](https://snipplr.com/view/14198/useful-regex-functions)

## Enlight a Word From a Text

This very useful regular expression will find a specific word in a string and enlight it. Extremely useful for search results. Remember that it’s case sensitive.

$text = "Sample sentence... regex has become popular in web programming. Now we learn regex. According to wikipedia, Regular expressions (abbreviated as regex or regexp, with plural forms regexes, regexps, or regexen) are written in a formal language that can be interpreted by a regular expression processor";

echo preg\_replace("/b(regex)b/i", '<span style="background:#5fc9f6">1</span>', $text);

» [**Source**](https://snipplr.com/view/14198/useful-regex-functions)

## Enlight Search Results in Your WordPress Blog

The previous code snippet can be very handy when it comes to displaying search results. If your website is powered by WordPress, here is a more specific snippet that will search and replace a text by the same text within an HTML tag that you can style later, using CSS.

Open your search.php file and find the the\_title() function. Replace it with the following:

echo $title;

Now, just before the modified line, add this code:

<?php

$title = get\_the\_title();

$keys= explode(" ",$s);

$title = preg\_replace('/('.implode('|', $keys) .')/iu',

'<strong class="search-excerpt">\0</strong>',

$title);

?>

Save the search.php file and open style.css. Append the following line to it:

strong.search-excerpt { background: yellow; }

» [**Source**](https://yoast.com/wordpress-search/)

## Get All Images From a HTML Document

If you ever wanted to be able to get all images form a webpage, this code is a must have for you. You should easily create an image downloader using the power of [cURL](https://catswhocode.com/php-curl-example/).

$images = array();

preg\_match\_all('/(img|src)=("|')[^"'>]+/i', $data, $media);

unset($data);

$data=preg\_replace('/(img|src)("|'|="|=')(.\*)/i',"$3",$media[0]);

foreach($data as $url)

{

$info = pathinfo($url);

if (isset($info['extension']))

{

if (($info['extension'] == 'jpg') ||

($info['extension'] == 'jpeg') ||

($info['extension'] == 'gif') ||

($info['extension'] == 'png'))

array\_push($images, $url);

}

}

» [**Source**](http://davebrooks.wordpress.com/2009/04/22/php-preg_replace-some-useful-regular-expressions/)

## Remove Repeated Words (Case Insensitive)

Often repeating words while typing? This handy case insensitive PCRE regex will be very helpful.

$text = preg\_replace("/s(w+s)1/i", "$1", $text);

» [**Source**](http://davebrooks.wordpress.com/2009/04/22/php-preg_replace-some-useful-regular-expressions/)

## Remove Repeated Punctuation

Same php regex as above, but this one will look for repeated punctuation within a string. Goodbye multiple commas!

$text = preg\_replace("/.+/i", ".", $text);

» [**Source**](http://davebrooks.wordpress.com/2009/04/22/php-preg_replace-some-useful-regular-expressions/)

## Match a XML/HTML Tag

This simple function takes two arguments: The first is the tag you’d like to match, and the second is the variable containing the XML or HTML. Once again, this can be very powerful used along with [cURL](https://catswhocode.com/php-curl-example/).

function get\_tag( $tag, $xml ) {

$tag = preg\_quote($tag);

preg\_match\_all('{<'.$tag.'[^>]\*>(.\*?)</'.$tag.'>.'}',

$xml,

$matches,

PREG\_PATTERN\_ORDER);

return $matches[1];

}

## Match an HTML/XML Tag With a Specific Attribute Value

This function is very similar to the previous one, but it allow you to match a tag having a specific attribute. For example, you could easily match <div id="header">.

function get\_tag( $attr, $value, $xml, $tag=null ) {

if( is\_null($tag) )

$tag = '\w+';

else

$tag = preg\_quote($tag);

$attr = preg\_quote($attr);

$value = preg\_quote($value);

$tag\_regex = "/<(".$tag.")[^>]\*$attr\s\*=\s\*".

"(['\"])$value\\2[^>]\*>(.\*?)<\/\\1>/"

preg\_match\_all($tag\_regex,

$xml,

$matches,

PREG\_PATTERN\_ORDER);

return $matches[3];

}

## Match Hexadecimal Color Values

Another interesting tool for web developers! It allows you to match/validate a hexadecimal color value.

$string = "#555555";

if (preg\_match('/^#(?:(?:[a-fd]{3}){1,2})$/i', $string)) {

echo "example 6 successful.";

}

## Find Page Title

This handy code snippet will find and print the text within the <title> and </title> tags of a HTML page.

$fp = fopen("https://catswhocode.com/blog","r");

while (!feof($fp) ){

$page .= fgets($fp, 4096);

}

$titre = eregi("<title>(.\*)</title>",$page,$regs);

echo $regs[1];

fclose($fp);

## Parse Apache Logs

Most websites are running on the Apache webserver. If your website does, you can easily use PHP and regular expressions to parse Apache logs.

//Logs: Apache web server

//Successful hits to HTML files only. Useful for counting the number of page views.

'^((?#client IP or domain name)S+)s+((?#basic authentication)S+s+S+)s+[((?#date and time)[^]]+)]s+"(?:GET|POST|HEAD) ((?#file)/[^ ?"]+?.html?)??((?#parameters)[^ ?"]+)? HTTP/[0-9.]+"s+(?#status code)200s+((?#bytes transferred)[-0-9]+)s+"((?#referrer)[^"]\*)"s+"((?#user agent)[^"]\*)"$'

//Logs: Apache web server

//404 errors only

'^((?#client IP or domain name)S+)s+((?#basic authentication)S+s+S+)s+[((?#date and time)[^]]+)]s+"(?:GET|POST|HEAD) ((?#file)[^ ?"]+)??((?#parameters)[^ ?"]+)? HTTP/[0-9.]+"s+(?#status code)404s+((?#bytes transferred)[-0-9]+)s+"((?#referrer)[^"]\*)"s+"((?#user agent)[^"]\*)"$'

» [**Source**](https://www.roscripts.com/php_regular_expressions_examples-136/)

## Replace Double Quotes by Smart Quotes

If you’re a typography lover, you’ll probably love this regex pattern which allow you to replace double quotes by smart quotes. A similar regular expression is used by WordPress to make the content more beautiful.

preg\_replace('B"b([^"x84x93x94rn]+)b"B', '?1?', $text);

» [**Source**](https://www.roscripts.com/php_regular_expressions_examples-136/)

## Check Password Complexity

This regular expression will tests if the input consists of 6 or more letters, digits, underscores, and hyphens.  
The input must contain at least one uppercase letter, one lowercase letter and one digit.

'A(?=[-\_a-zA-Z0-9]\*?[A-Z])(?=[-\_a-zA-Z0-9]\*?[a-z])(?=[-\_a-zA-Z0-9]\*?[0-9])[-\_a-zA-Z0-9]{6,}z'

» [**Source**](https://www.roscripts.com/php_regular_expressions_examples-136/)

## WordPress: Using Regexp to Retrieve Images From a Post

As I know many of you are WordPress users, you’ll probably enjoy that code which allows you to retrieve all images from post content and display it.

To use this code on your blog, simply paste the following code on one of your theme files.

<?php if (have\_posts()) : ?>

<?php while (have\_posts()) : the\_post(); ?>

<?php

$szPostContent = $post->post\_content;

$szSearchPattern = '~<img [^>]\* />~';

// Run preg\_match\_all to grab all the images and save the results in $aPics

preg\_match\_all( $szSearchPattern, $szPostContent, $aPics );

// Check to see if we have at least 1 image

$iNumberOfPics = count($aPics[0]);

if ( $iNumberOfPics > 0 ) {

// Now here you would do whatever you need to do with the images

// For this example the images are just displayed

for ( $i=0; $i < $iNumberOfPics ; $i++ ) {

echo $aPics[0][$i];

};

};

endwhile;

endif;

?>

## Generate Emoticons Automatically

Another function used by WordPress. This one allow you to automatically replace an emoticon symbol by an image.

$texte='A text with a smiley :-)';

echo str\_replace(':-)','<img src="smileys/souriant.png">',$texte);

<https://catswhocode.com/php-regex/>

# PHP regular expressions

last modified July 12, 2020

In this part of the PHP tutorial, we cover regular expressions in PHP.

Regular expressions are used for text searching and more advanced text manipulation. Regular expressions are built-in tools like grep, sed, text editors like vi, emacs, programming languages like Tcl, Perl, and Python. PHP has a built-in support for regular expressions too.

In PHP, there are two modules for regular expressions: the POSIX Regex and the PCRE. The POSIX Regex is depreciated. In this chapter, we will use the PCRE examples. PCRE stands for Perl compatible regular expressions.

Two things are needed when we work with regular expressions: Regex functions and the pattern.

A pattern is a regular expression that defines the text we are searching for or manipulating. It consists of text literals and metacharacters. The pattern is placed inside two delimiters. These are usually //, ##, or @@ characters. They inform the regex function where the pattern starts and ends.

Here is a partial list of metacharacters used in PCRE.

|  |  |
| --- | --- |
| . | Matches any single character. |
| \* | Matches the preceding element zero or more times. |
| [ ] | Bracket expression. Matches a character within the brackets. |
| [^ ] | Matches a single character that is not contained within the brackets. |
| ^ | Matches the starting position within the string. |
| $ | Matches the ending position within the string. |
| | | Alternation operator. |

## PHP PRCE functions

We define some PCRE regex functions. They all have a preg prefix.

* preg\_split() - splits a string by regex pattern
* preg\_match() - performs a regex match
* preg\_replace() - search and replace string by regex pattern
* preg\_grep() - returns array entries that match the regex pattern

Next we will have an example for each function.

php> print\_r(preg\_split("@\s@", "Jane\tKate\nLucy Marion"));

Array

(

[0] => Jane

[1] => Kate

[2] => Lucy

[3] => Marion

)

We have four names divided by spaces. The \s is a character class which stands for spaces. The preg\_split() function returns the split strings in an array.

php> echo preg\_match("#[a-z]#", "s");

1

The preg\_match() function looks if the 's' character is in the character class [a-z]. The class stands for all characters from a to z. It returns 1 for success.

php> echo preg\_replace("/Jane/","Beky","I saw Jane. Jane was beautiful.");

I saw Beky. Beky was beautiful.

The preg\_replace() function replaces all occurrences of the word 'Jane' for the word 'Beky'.

php> print\_r(preg\_grep("#Jane#", ["Jane", "jane", "Joan", "JANE"]));

Array

(

[0] => Jane

)

The preg\_grep() function returns an array of words that match the given pattern. In this example, only one word is returned in the array. This is because by default, the search is case sensitive.

php> print\_r(preg\_grep("#Jane#i", ["Jane", "jane", "Joan", "JANE"]));

Array

(

[0] => Jane

[1] => jane

[3] => JANE

)

In this example, we perform a case insensitive grep. We put the i modifier after the right delimiter. The returned array has now three words.

## PHP regex dot metacharacter

The . (dot) metacharacter stands for any single character in the text.

single.php

<?php

$words = [ "Seven", "even", "Maven", "Amen", "Leven" ];

$pattern = "/.even/";

foreach ($words as $word) {

if (preg\_match($pattern, $word)) {

echo "$word matches the pattern\n";

} else {

echo "$word does not match the pattern\n";

}

}

In the $words array, we have five words.

$pattern = "/.even/";

Here we define the search pattern. The pattern is a string. The regular expression is placed within delimiters. The delimiters are mandatory. In our case, we use forward slashes / / as delimiters. Note that we can use different delimiters if we want. The dot character stands for any single character.

if (preg\_match($pattern, $word)) {

echo "$word matches the pattern\n";

} else {

echo "$word does not match the pattern\n";

}

We test all five words if they match with the pattern.

$ php single.php

Seven matches the pattern

even does not match the pattern

Maven does not match the pattern

Amen does not match the pattern

Leven matches the pattern

The Seven and Leven words match our search pattern.

## PHP regex anchors

Anchors match positions of characters inside a given text.

In the next example, we look if a string is located at the beginning of a sentence.

anchors.php

<?php

$sentence1 = "Everywhere I look I see Jane";

$sentence2 = "Jane is the best thing that happened to me";

if (preg\_match("/^Jane/", $sentence1)) {

echo "Jane is at the beginning of the \$sentence1\n";

} else {

echo "Jane is not at the beginning of the \$sentence1\n";

}

if (preg\_match("/^Jane/", $sentence2)) {

echo "Jane is at the beginning of the \$sentence2\n";

} else {

echo "Jane is not at the beginning of the \$sentence2\n";

}

We have two sentences. The pattern is ^Jane. The pattern checks if the 'Jane' string located at the beginning of the text.

$ php anchors.php

Jane is not at the beginning of the $sentence1

Jane is at the beginning of the $sentence2

php> echo preg\_match("#Jane$#", "I love Jane");

1

php> echo preg\_match("#Jane$#", "Jane does not love me");

0

The Jane$ pattern matches a string in which the word Jane is at the end.

## PHP regex exact word match

In the following examples we show how to look for exact word matches.

php> echo preg\_match("/mother/", "mother");

1

php> echo preg\_match("/mother/", "motherboard");

1

php> echo preg\_match("/mother/", "motherland");

1

The mother pattern fits the words mother, motherboard and motherland. Say, we want to look just for exact word matches. We will use the aforementioned anchor ^ and $ characters.

php> echo preg\_match("/^mother$/", "motherland");

0

php> echo preg\_match("/^mother$/", "Who is your mother?");

0

php> echo preg\_match("/^mother$/", "mother");

1

Using the anchor characters, we get an exact word match for a pattern.

## PHP regex quantifiers

A quantifier after a token or a group specifies how often that preceding element is allowed to occur.

? - 0 or 1 match

\* - 0 or more

+ - 1 or more

{n} - exactly n

{n,} - n or more

{,n} - n or less (??)

{n,m} - range n to m

The above is a list of common quantifiers.

The question mark ? indicates there is zero or one of the preceding element.

zeroorone.php

<?php

$words = [ "color", "colour", "comic", "colourful", "colored",

"cosmos", "coloseum", "coloured", "colourful" ];

$pattern = "/colou?r/";

foreach ($words as $word) {

if (preg\_match($pattern, $word)) {

echo "$word matches the pattern\n";

} else {

echo "$word does not match the pattern\n";

}

}

We have four nine in the $words array.

$pattern = "/colou?r/";

Color is used in American English, colour in British English. This pattern matches both cases.

$ php zeroorone.php

color matches the pattern

colour matches the pattern

comic does not match the pattern

colourful matches the pattern

colored matches the pattern

cosmos does not match the pattern

coloseum does not match the pattern

coloured matches the pattern

colourful matches the pattern

This is the output of the zeroorone.php script.

The \* metacharacter matches the preceding element zero or more times.

zeroormore.php

<?php

$words = [ "Seven", "even", "Maven", "Amen", "Leven" ];

$pattern = "/.\*even/";

foreach ($words as $word) {

if (preg\_match($pattern, $word)) {

echo "$word matches the pattern\n";

} else {

echo "$word does not match the pattern\n";

}

}

In the above script, we have added the \* metacharacter. The .\* combination means, zero, one or more single characters.

$ php zeroormore.php

Seven matches the pattern

even matches the pattern

Maven does not match the pattern

Amen does not match the pattern

Leven matches the pattern

Now the pattern matches three words: Seven, even and Leven.

php> print\_r(preg\_grep("#o{2}#", ["gool", "root", "foot", "dog"]));

Array

(

[0] => gool

[1] => root

[2] => foot

)

The o{2} pattern matches strings that contain exactly two 'o' characters.

php> print\_r(preg\_grep("#^\d{2,4}$#", ["1", "12", "123", "1234", "12345"]));

Array

(

[1] => 12

[2] => 123

[3] => 1234

)

We have this ^\d{2,4}$ pattern. The \d is a character set; it stands for digits. The pattern matches numbers that have 2, 3, or 4 digits.

## PHP regex alternation

The next example explains the alternation operator |. This operator enables to create a regular expression with several choices.

alternation.php

<?php

$names = [ "Jane", "Thomas", "Robert", "Lucy", "Beky",

"John", "Peter", "Andy" ];

$pattern = "/Jane|Beky|Robert/";

foreach ($names as $name) {

if (preg\_match($pattern, $name)) {

echo "$name is my friend\n";

} else {

echo "$name is not my friend\n";

}

}

We have eight names in the $names array.

$pattern = "/Jane|Beky|Robert/";

This is the search pattern. The pattern looks for 'Jane', 'Beky', or 'Robert' strings.

$ php alternation.php

Jane is my friend

Thomas is not my friend

Robert is my friend

Lucy is not my friend

Beky is my friend

John is not my friend

Peter is not my friend

Andy is not my friend

This is the output of the script.

## PHP regex subpatterns

We can use square brackets () to create subpatterns inside patterns.

php> echo preg\_match("/book(worm)?$/", "bookworm");

1

php> echo preg\_match("/book(worm)?$/", "book");

1

php> echo preg\_match("/book(worm)?$/", "worm");

0

We have the following regex pattern: book(worm)?$. The (worm) is a subpattern. The ? character follows the subpattern, which means that the subpattern might appear 0, 1 times in the final pattern. The $ character is here for the exact end match of the string. Without it, words like bookstore, bookmania would match too.

php> echo preg\_match("/book(shelf|worm)?$/", "book");

1

php> echo preg\_match("/book(shelf|worm)?$/", "bookshelf");

1

php> echo preg\_match("/book(shelf|worm)?$/", "bookworm");

1

php> echo preg\_match("/book(shelf|worm)?$/", "bookstore");

0

Subpatterns are often used with alternation. The (shelf|worm) subpattern enables to create several word combinations.

## PHP regex character classes

We can combine characters into character classes with the square brackets. A character class matches any character that is specified in the brackets.

characterclass.php

<?php

$words = [ "sit", "MIT", "fit", "fat", "lot" ];

$pattern = "/[fs]it/";

foreach ($words as $word) {

if (preg\_match($pattern, $word)) {

echo "$word matches the pattern\n";

} else {

echo "$word does not match the pattern\n";

}

}

We define a character set with two characters.

$pattern = "/[fs]it/";

This is our pattern. The [fs] is the character class. Note that we work only with one character at a time. We either consider f, or s, but not both.

$ php characterclass.php

sit matches the pattern

MIT does not match the pattern

fit matches the pattern

fat does not match the pattern

lot does not match the pattern

This is the outcome of the script.

We can also use shorthand metacharacters for character classes. The \w stands for alphanumeric characters, \d for digit, and \s whitespace characters.

shorthand.php

<?php

$words = [ "Prague", "111978", "terry2", "mitt##" ];

$pattern = "/\w{6}/";

foreach ($words as $word) {

if (preg\_match($pattern, $word)) {

echo "$word matches the pattern\n";

} else {

echo "$word does not match the pattern\n";

}

}

In the above script, we test for words consisting of alphanumeric characters. The \w{6} stands for six alphanumeric characters. Only the word mitt## does not match, because it contains non-alphanumeric characters.

php> echo preg\_match("#[^a-z]{3}#", "ABC");

1

The #[^a-z]{3}# pattern stands for three characters that are not in the class a-z. The "ABC" characters match the condition.

php> print\_r(preg\_grep("#\d{2,4}#", [ "32", "234", "2345", "3d3", "2"]));

Array

(

[0] => 32

[1] => 234

[2] => 2345

)

In the above example, we have a pattern that matches 2, 3, and 4 digits.

## PHP regex extracting matches

The preg\_match() takes an optional third parameter. If it is provided, it is filled with the results of the search. The variable is an array whose first element contains the text that matched the full pattern, the second element contains the first captured parenthesized subpattern, and so on.

extract\_matches.php

<?php

$times = [ "10:10:22", "23:23:11", "09:06:56" ];

$pattern = "/(\d\d):(\d\d):(\d\d)/";

foreach ($times as $time) {

$r = preg\_match($pattern, $time, $match);

if ($r) {

echo "The $match[0] is split into:\n";

echo "Hour: $match[1]\n";

echo "Minute: $match[2]\n";

echo "Second: $match[3]\n";

}

}

In the example, we extract parts of a time string.

$times = [ "10:10:22", "23:23:11", "09:06:56" ];

We have three time strings in English locale.

$pattern = "/(\d\d):(\d\d):(\d\d)/";

The pattern is divided into three subpatterns using square brackets. We want to refer specifically to exactly to each of these parts.

$r = preg\_match($pattern, $time, $match);

We pass a third parameter to the preg\_match() function. In case of a match, it contains text parts of the matched string.

if ($r) {

echo "The $match[0] is split into:\n";

echo "Hour: $match[1]\n";

echo "Minute: $match[2]\n";

echo "Second: $match[3]\n";

}

The $match[0] contains the text that matched the full pattern, $match[1] contains text that matched the first subpattern, $match[2] the second, and $match[3] the third.

$ php extract\_matches.php

The 10:10:22 is split into:

Hour: 10

Minute: 10

Second: 22

The 23:23:11 is split into:

Hour: 23

Minute: 23

Second: 11

The 09:06:56 is split into:

Hour: 09

Minute: 06

Second: 56

This is the output of the example.

## PHP regex email example

Next have a practical example. We create a regex pattern for checking email addresses.

emails.php

<?php

$emails = [ "luke@gmail.com", "andy@yahoocom", "34234sdfa#2345",

"f344@gmail.com"];

# regular expression for emails

$pattern = "/^[a-zA-Z0-9.\_-]+@[a-zA-Z0-9-]+\.[a-zA-Z.]{2,18}$/";

foreach ($emails as $email) {

if (preg\_match($pattern, $email)) {

echo "$email matches \n";

} else {

echo "$email does not match\n";

}

}

>?

Note that this example provides only one solution. It does not have to be the best one.

$pattern = "/^[a-zA-Z0-9.\_-]+@[a-zA-Z0-9-]+\.[a-zA-Z.]{2,18}$/";

This is the pattern. The first ^ and the last $ characters are here to get an exact pattern match. No characters before and after the pattern are allowed. The email is divided into five parts. The first part is the local part. This is usually a name of a company, individual, or a nickname. The [a-zA-Z0-9.\_-]+ lists all possible characters, we can use in the local part. They can be used one or more times. The second part is the literal @ character. The third part is the domain part. It is usually the domain name of the email provider, like yahoo, or gmail. The [a-zA-Z0-9-]+ is a character set providing all characters that can be used in the domain name. The + quantifier makes use of one or more of these characters. The fourth part is the dot character. It is preceded by the escape character (\). This is because the dot character is a metacharacter and has a special meaning. By escaping it, we get a literal dot. The final part is the top level domain. The pattern is as follows: [a-zA-Z.]{2,18} Top level domains can have from 2 to 18 characters, like sk, net, info, travel, cleaning, travelinsurance. The maximum lenght can be 63 characters, but most domain are shorter than 18 characters today. There is also a dot character. This is because some top level domains have two parts; for example co.uk.

$ php emails.php

luke@gmail.com matches

andy@yahoocom does not match

34234sdfa#2345 does not match

f344@gmail.com matches

This is the output of the emails.php example.

## Recap

Finally, we provide a quick recap of the regex patterns.

Jane the 'Jane' string

^Jane 'Jane' at the start of a string

Jane$ 'Jane' at the end of a string

^Jane$ exact match of the string 'Jane'

[abc] a, b, or c

[a-z] any lowercase letter

[^A-Z] any character that is not a uppercase letter

(Jane|Becky) Matches either 'Jane' or 'Becky'

[a-z]+ one or more lowercase letters

^[98]?$ digits 9, 8 or empty string

([wx])([yz]) wy, wz, xy, or xz

[0-9] any digit

[^A-Za-z0-9] any symbol (not a number or a letter)

In this chapter, we have covered regular expressions in PHP.

<https://zetcode.com/php/regex/>

# PHP Regular Expressions

In this tutorial you will learn how regular expressions work, as well as how to use them to perform pattern matching in an efficient way in PHP.

## What is Regular Expression

Regular Expressions, commonly known as "regex" or "RegExp", are a specially formatted text strings used to find patterns in text. Regular expressions are one of the most powerful tools available today for effective and efficient text processing and manipulations. For example, it can be used to verify whether the format of data i.e. name, email, phone number, etc. entered by the user was correct or not, find or replace matching string within text content, and so on.

PHP (version 5.3 and above) supports Perl style regular expressions via its preg\_ family of functions. Why Perl style regular expressions? Because Perl (Practical Extraction and Report Language) was the first mainstream programming language that provided integrated support for regular expressions and it is well known for its strong support of regular expressions and its extraordinary text processing and manipulation capabilities.

Let's begin with a brief overview of the commonly used PHP's built-in pattern-matching functions before delving deep into the world of regular expressions.

| Function | What it Does |
| --- | --- |
| preg\_match() | Perform a regular expression match. |
| preg\_match\_all() | Perform a global regular expression match. |
| preg\_replace() | Perform a regular expression search and replace. |
| preg\_grep() | Returns the elements of the input array that matched the pattern. |
| preg\_split() | Splits up a string into substrings using a regular expression. |
| preg\_quote() | Quote regular expression characters found within a string. |

Note: The PHP preg\_match() function stops searching after it finds the first match, whereas the preg\_match\_all() function continues searching until the end of the string and find all possible matches instead of stopping at the first match.

## Regular Expression Syntax

Regular expression syntax includes the use of special characters (do not confuse with the [HTML special characters](https://www.tutorialrepublic.com/html-tutorial/html-entities.php)). The characters that are given special meaning within a regular expression, are: . \* ? + [ ] ( ) { } ^ $ | \. You will need to backslash these characters whenever you want to use them literally. For example, if you want to match ".", you'd have to write \.. All other characters automatically assume their literal meanings.

The following sections describe the various options available for formulating patterns:

## Character Classes

Square brackets surrounding a pattern of characters are called a character class e.g. [abc]. A character class always matches a single character out of a list of specified characters that means the expression [abc] matches only a, b or c character.

Negated character classes can also be defined that match any character except those contained within the brackets. A negated character class is defined by placing a caret (^) symbol immediately after the opening bracket, like this [^abc].

You can also define a range of characters by using the hyphen (-) character inside a character class, like [0-9]. Let's look at some examples of character classes:

| RegExp | What it Does |
| --- | --- |
| [abc] | Matches any one of the characters a, b, or c. |
| [^abc] | Matches any one character other than a, b, or c. |
| [a-z] | Matches any one character from lowercase a to lowercase z. |
| [A-Z] | Matches any one character from uppercase a to uppercase z. |
| [a-Z] | Matches any one character from lowercase a to uppercase Z. |
| [0-9] | Matches a single digit between 0 and 9. |
| [a-z0-9] | Matches a single character between a and z or between 0 and 9. |

The following example will show you how to find whether a pattern exists in a string or not using the regular expression and PHP preg\_match() function:

#### Example

[Run this code »](https://www.tutorialrepublic.com/codelab.php?topic=php&file=match-a-pattern-against-a-string-using-regular-expression)

<?php

$pattern = "/ca[kf]e/";

$text = "He was eating cake in the cafe.";

if(preg\_match($pattern, $text)){

echo "Match found!";

} else{

echo "Match not found.";

}

?>

Similarly, you can use the preg\_match\_all() function to find all matches within a string:

#### Example

[Run this code »](https://www.tutorialrepublic.com/codelab.php?topic=php&file=find-all-occurrences-of-a-pattern-in-a-string-using-regular-expression)

<?php

$pattern = "/ca[kf]e/";

$text = "He was eating cake in the cafe.";

$matches = preg\_match\_all($pattern, $text, $array);

echo $matches . " matches were found.";

?>

Tip: Regular expressions aren't exclusive to PHP. Languages such as Java, Perl, Python, etc. use the same notation for finding patterns in text.

## Predefined Character Classes

Some character classes such as digits, letters, and whitespaces are used so frequently that there are shortcut names for them. The following table lists those predefined character classes:

| Shortcut | What it Does |
| --- | --- |
| . | Matches any single character except newline \n. |
| \d | matches any digit character. Same as [0-9] |
| \D | Matches any non-digit character. Same as [^0-9] |
| \s | Matches any whitespace character (space, tab, newline or carriage return character). Same as [ \t\n\r] |
| \S | Matches any non-whitespace character. Same as [^ \t\n\r] |
| \w | Matches any word character (definned as a to z, A to Z,0 to 9, and the underscore). Same as [a-zA-Z\_0-9] |
| \W | Matches any non-word character. Same as [^a-zA-Z\_0-9] |

The following example will show you how to find and replace space with a hyphen character in a string using regular expression and PHP preg\_replace() function:

#### Example

[Run this code »](https://www.tutorialrepublic.com/codelab.php?topic=php&file=find-and-replace-characters-in-a-string-using-regular-expression)

<?php

$pattern = "/\s/";

$replacement = "-";

$text = "Earth revolves around\nthe\tSun";

// Replace spaces, newlines and tabs

echo preg\_replace($pattern, $replacement, $text);

echo "<br>";

// Replace only spaces

echo str\_replace(" ", "-", $text);

?>

## Repetition Quantifiers

In the previous section we've learnt how to match a single character in a variety of fashions. But what if you want to match on more than one character? For example, let's say you want to find out words containing one or more instances of the letter p, or words containing at least two p's, and so on. This is where quantifiers come into play. With quantifiers you can specify how many times a character in a regular expression should match.

The following table lists the various ways to quantify a particular pattern:

| RegExp | What it Does |
| --- | --- |
| p+ | Matches one or more occurrences of the letter p. |
| p\* | Matches zero or more occurrences of the letter p. |
| p? | Matches zero or one occurrences of the letter p. |
| p{2} | Matches exactly two occurrences of the letter p. |
| p{2,3} | Matches at least two occurrences of the letter p, but not more than three occurrences of the letter p. |
| p{2,} | Matches two or more occurrences of the letter p. |
| p{,3} | Matches at most three occurrences of the letter p |

The regular expression in the following example will splits the string at comma, sequence of commas, whitespace, or combination thereof using the PHP preg\_split() function:

#### Example

[Run this code »](https://www.tutorialrepublic.com/codelab.php?topic=php&file=split-a-string-using-regular-expression)

<?php

$pattern = "/[\s,]+/";

$text = "My favourite colors are red, green and blue";

$parts = preg\_split($pattern, $text);

// Loop through parts array and display substrings

foreach($parts as $part){

echo $part . "<br>";

}

?>

## Position Anchors

There are certain situations where you want to match at the beginning or end of a line, word, or string. To do this you can use anchors. Two common anchors are caret (^) which represent the start of the string, and the dollar ($) sign which represent the end of the string.

| RegExp | What it Does |
| --- | --- |
| ^p | Matches the letter p at the beginning of a line. |
| p$ | Matches the letter p at the end of a line. |

The regular expression in the following example will display only those names from the names array which start with the letter "J" using the PHP preg\_grep() function:

#### Example

[Run this code »](https://www.tutorialrepublic.com/codelab.php?topic=php&file=match-strings-beginning-with-specific-characters-using-regular-expression)

<?php

$pattern = "/^J/";

$names = array("Jhon Carter", "Clark Kent", "John Rambo");

$matches = preg\_grep($pattern, $names);

// Loop through matches array and display matched names

foreach($matches as $match){

echo $match . "<br>";

}

?>

## Pattern Modifiers

A pattern modifier allows you to control the way a pattern match is handled. Pattern modifiers are placed directly after the regular expression, for example, if you want to search for a pattern in a case-insensitive manner, you can use the i modifier, like this: /pattern/i. The following table lists some of the most commonly used pattern modifiers.

| Modifier | What it Does |
| --- | --- |
| i | Makes the match case-insensitive manner. |
| m | Changes the behavior of ^ and $ to match against a newline boundary (i.e. start or end of each line within a multiline string), instead of a string boundary. |
| g | Perform a global match i.e. finds all occurrences. |
| o | Evaluates the expression only once. |
| s | Changes the behavior of . (dot) to match all characters, including newlines. |
| x | Allows you to use whitespace and comments within a regular expression for clarity. |

The following example will show you how to perform a global case-insensitive search using the i modifier and the PHP preg\_match\_all() function.

#### Example

[Run this code »](https://www.tutorialrepublic.com/codelab.php?topic=php&file=match-the-pattern-in-a-case-insensitive-manner-using-regular-expression)

<?php

$pattern = "/color/i";

$text = "Color red is more visible than color blue in daylight.";

$matches = preg\_match\_all($pattern, $text, $array);

echo $matches . " matches were found.";

?>

Similarly, the following example shows how to match at the beginning of every line in a multi-line string using ^ anchor and m modifier with PHP preg\_match\_all() function.

#### Example

[Run this code »](https://www.tutorialrepublic.com/codelab.php?topic=php&file=match-a-pattern-in-a-multiline-string-using-regular-expression)

<?php

$pattern = "/^color/im";

$text = "Color red is more visible than \ncolor blue in daylight.";

$matches = preg\_match\_all($pattern, $text, $array);

echo $matches . " matches were found.";

?>

## Word Boundaries

A word boundary character ( \b) helps you search for the words that begins and/or ends with a pattern. For example, the regexp /\bcar/ matches the words beginning with the pattern car, and would match cart, carrot, or cartoon, but would not match oscar.

Similarly, the regexp /car\b/ matches the words ending with the pattern car, and would match scar, oscar, or supercar, but would not match cart. Likewise, the /\bcar\b/ matches the words beginning and ending with the pattern car, and would match only the word car.

The following example will highlight the words beginning with car in bold:

#### Example

[Run this code »](https://www.tutorialrepublic.com/codelab.php?topic=php&file=match-words-starts-or-ends-with-a-pattern-using-regular-expression)

<?php

$pattern = '/\bcar\w\*/';

$replacement = '<b>$0</b>';

$text = 'Words begining with car: cart, carrot, cartoon. Words ending with car: scar, oscar, supercar.';

echo preg\_replace($pattern, $replacement, $text);

?>

We hope you have understood the basics of regular expression. To learn how to validate form data using regular expression, please check out the tutorial on [PHP Form Validation](https://www.tutorialrepublic.com/php-tutorial/php-form-validation.php).

<https://www.tutorialrepublic.com/php-tutorial/php-regular-expressions.php>

# Use regular expressions in Visual Studio

* 6 minutes to read
  + [](https://github.com/TerryGLee)
  + [](https://github.com/xh286286)
  + [](https://github.com/ghogen)
  + [](https://github.com/john-par)
  + [](https://github.com/mijacobs)

Visual Studio uses [.NET regular expressions](https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions) to find and replace text.

## Regular expression examples

The following table contains some regular expression characters, operators, constructs, and pattern examples. For a more complete reference, see [Regular expression language](https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference).

Regular expression examples

| Purpose | Expression | Example |
| --- | --- | --- |
| Match any single character (except a line break). For more information, see [Any character](https://docs.microsoft.com/en-us/dotnet/standard/base-types/character-classes-in-regular-expressions" \l "any-character-). | . | a.o matches "aro" in "around" and "abo" in "about" but not "acro" in "across" |
| Match zero or more occurrences of the preceding expression (match as many characters as possible). For more information, see [Match zero or more times](https://docs.microsoft.com/en-us/dotnet/standard/base-types/quantifiers-in-regular-expressions" \l "match-zero-or-more-times-). | \* | a\*r matches "r" in "rack", "ar" in "ark", and "aar" in "aardvark" |
| Match any character zero or more times. | .\* | c.\*e matches "cke" in "racket", "comme" in "comment", and "code" in "code" |
| Match one or more occurrences of the preceding expression (match as many characters as possible). For more information, see [Match one or more times](https://docs.microsoft.com/en-us/dotnet/standard/base-types/quantifiers-in-regular-expressions" \l "match-one-or-more-times-). | + | e+d matches "eed" in "feeder" and "ed" in "faded" |
| Match any character one or more times. | .+ | e.+e matches "eede" in "feeder" but finds no matches in "feed" |
| Match zero or more occurrences of the preceding expression (match as few characters as possible). For more information, see [Match zero or more times (lazy match)](https://docs.microsoft.com/en-us/dotnet/standard/base-types/quantifiers-in-regular-expressions" \l "match-zero-or-more-times-lazy-match-). | \*? | \w\*?d matches "fad" and "ed" in "faded" but not the entire word "faded" due to the lazy match |
| Match one or more occurrences of the preceding expression (match as few characters as possible). For more information, see [Match one or more times (lazy match)](https://docs.microsoft.com/en-us/dotnet/standard/base-types/quantifiers-in-regular-expressions" \l "match-one-or-more-times-lazy-match-). | +? | e\w+? matches "ee" in "asleep" and "ed" in "faded" but finds no matches in "fade" |
| Anchor the match string to the [beginning of a line or string](https://docs.microsoft.com/en-us/dotnet/standard/base-types/anchors-in-regular-expressions" \l "start-of-string-or-line-) | ^ | ^car matches the word "car" only when it appears at the beginning of a line |
| Anchor the match string to the [end of a line](https://docs.microsoft.com/en-us/dotnet/standard/base-types/anchors-in-regular-expressions" \l "end-of-string-or-line-) | \r?$ | car\r?$ matches "car" only when it appears at the end of a line |
| Anchor the match string to the end of the file | $ | car$ matches "car" only when it appears at the end of the file |
| Match any single character in a set | [abc] | b[abc] matches "ba", "bb", and "bc" |
| Match any character in a range of characters | [a-f] | be[n-t] matches "bet" in "between", "ben" in "beneath", and "bes" in "beside", but finds no matches in "below" |
| Capture and implicitly number the expression contained within parenthesis | () | ([a-z])X\1 matches "aXa"and "bXb", but not "aXb". "\1" refers to the first expression group "[a-z]". For more information, see [Capture groups and replacement patterns](https://docs.microsoft.com/en-us/visualstudio/ide/using-regular-expressions-in-visual-studio?view=vs-2019" \l "capture-groups-and-replacement-patterns). |
| Invalidate a match | (?!abc) | real(?!ity) matches "real" in "realty" and "really" but not in "reality." It also finds the second "real" (but not the first "real") in "realityreal". |
| Match any character that is not in a given set of characters. For more information, see [Negative character group](https://docs.microsoft.com/en-us/dotnet/standard/base-types/character-classes-in-regular-expressions" \l "negative-character-group-). | [^abc] | be[^n-t] matches "bef" in "before", "beh" in "behind", and "bel" in "below", but finds no matches in "beneath" |
| Match either the expression before or the one after the symbol | | | (sponge|mud) bath matches "sponge bath" and "mud bath" |
| [Escape the character](https://docs.microsoft.com/en-us/dotnet/standard/base-types/character-escapes-in-regular-expressions) following the backslash | \ | \^ matches the character ^ |
| Specify the number of occurrences of the preceding character or group. For more information, see [Match exactly n times](https://docs.microsoft.com/en-us/dotnet/standard/base-types/quantifiers-in-regular-expressions" \l "match-exactly-n-times-n). | {n}, where 'n' is the number of occurrences | x(ab){2}x matches "xababx" x(ab){2,3}x matches "xababx" and "xabababx" but not "xababababx" |
| [Match text in a Unicode category](https://docs.microsoft.com/en-us/dotnet/standard/base-types/character-classes-in-regular-expressions" \l "unicode-category-or-unicode-block-p). For more information about Unicode character classes, see [Unicode Standard 5.2 Character Properties](http://www.unicode.org/versions/Unicode5.2.0/ch04.pdf). | \p{X}, where "X" is the Unicode number. | \p{Lu} matches "T" and "D" in "Thomas Doe" |
| [Match a word boundary](https://docs.microsoft.com/en-us/dotnet/standard/base-types/anchors-in-regular-expressions" \l "word-boundary-b) | \b (Outside a character class \b specifies a word boundary, and inside a character class \b specifies a backspace.) | \bin matches "in" in "inside" but finds no matches in "pinto" |
| Match a line break (that is, a carriage return followed by a new line) | \r?\n | End\r?\nBegin matches "End" and "Begin" only when "End" is the last string in a line and "Begin" is the first string in the next line |
| Match any [word character](https://docs.microsoft.com/en-us/dotnet/standard/base-types/character-classes-in-regular-expressions" \l "word-character-w) | \w | a\wd matches "add" and "a1d" but not "a d" |
| Match any [whitespace character](https://docs.microsoft.com/en-us/dotnet/standard/base-types/character-classes-in-regular-expressions" \l "whitespace-character-s) | \s | Public\sInterface matches the phrase "Public Interface" |
| Match any [decimal digit character](https://docs.microsoft.com/en-us/dotnet/standard/base-types/character-classes-in-regular-expressions" \l "decimal-digit-character-d) | \d | \d matches "4" and "0" in "wd40" |

An example regular expression that combines some of the operators and constructs to match a hexadecimal number is \b0[xX]([0-9a-fA-F]+)\b. This expression matches "0xc67f" but not "0xc67g".

Tip

In Windows operating systems, most lines end in "\r\n" (a carriage return followed by a new line). These characters aren't visible but are present in the editor and passed to the .NET regular expression service.

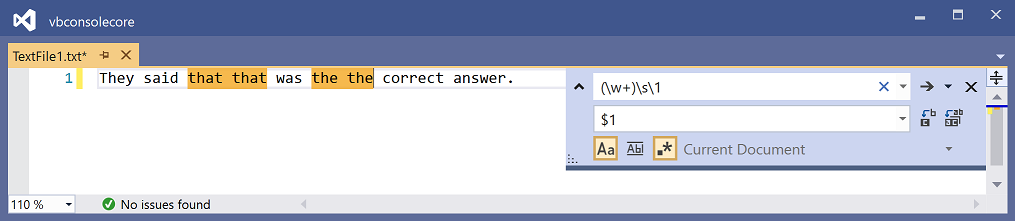
## Capture groups and replacement patterns

A capture group delineates a subexpression of a regular expression and captures a substring of an input string. You can use captured groups within the regular expression itself (for example, to look for a repeated word), or in a replacement pattern. For detailed information, see [Grouping constructs in regular expressions](https://docs.microsoft.com/en-us/dotnet/standard/base-types/grouping-constructs-in-regular-expressions).

To create a numbered capture group, surround the subexpression with parentheses in the regular expression pattern. Captures are numbered automatically from left to right based on the position of the opening parenthesis in the regular expression. To access the captured group:

* within the regular expression: Use \number. For example, \1 in the regular expression (\w+)\s\1 references the first capture group (\w+).
* in a replacement pattern: Use $number. For example, the grouped regular expression (\d)([a-z]) defines two groups: the first group contains a single decimal digit, and the second group contains a single character between a and z. The expression finds four matches in the following string: 1a 2b 3c 4d. The replacement string z$1 references the first group only ($1), and converts the string to z1 z2 z3 z4.

The following image shows a regular expression (\w+)\s\1 and a replacement string $1. Both the regular expression and the replacement pattern reference the first capture group that's automatically numbered 1. When you choose Replace all in the Quick Replace dialog box in Visual Studio, repeated words are removed from the text.



Tip

Make sure the Use Regular Expressions button is selected in the Quick Replace dialog box.

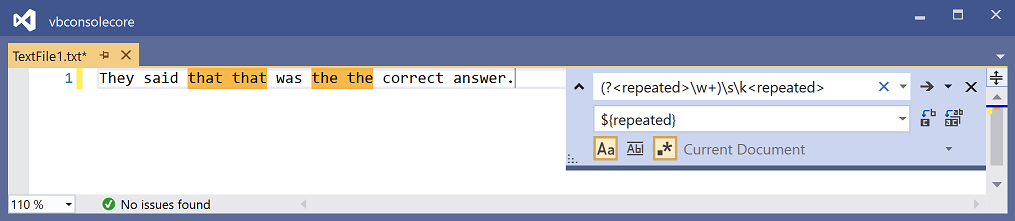
### Named capture groups

Instead of relying on the automatic numbering of a capture group, you can give it a name. The syntax for a named capture group is (?<name>subexpression).

Named capture groups, like numbered capture groups, can be used within the regular expression itself or in a replacement pattern. To access the named capture group:

* within the regular expression: Use \k<name>. For example, \k<repeated> in the regular expression (?<repeated>\w+)\s\k<repeated> references the capture group that's named repeated and whose subexpression is \w+.
* in a replacement pattern: Use ${name}. For example, ${repeated}.

As an example, the following image shows a regular expression (?<repeated>\w+)\s\k<repeated> and a replacement string ${repeated}. Both the regular expression and the replacement pattern reference the capture group named repeated. When you choose Replace all in the Quick Replace dialog box in Visual Studio, repeated words are removed from the text.



Tip

Make sure the Use Regular Expressions button is selected in the Quick Replace dialog box.

For more information about named capture groups, see [Named matched subexpressions](https://docs.microsoft.com/en-us/dotnet/standard/base-types/grouping-constructs-in-regular-expressions" \l "named-matched-subexpressions). For more information about regular expressions that are used in replacement patterns, see [Substitutions in regular expressions](https://docs.microsoft.com/en-us/dotnet/standard/base-types/substitutions-in-regular-expressions).

## See also

* [Regular expression language](https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference)
* [Find and replace text](https://docs.microsoft.com/en-us/visualstudio/ide/finding-and-replacing-text?view=vs-2019)

<https://docs.microsoft.com/en-us/visualstudio/ide/using-regular-expressions-in-visual-studio?view=vs-2019>

1

I don't know the answer to (1), but the answer to (2) is as follows:

Firstly, if you've only used grep and not other flavours of regex, you should know that there are some syntax differences. In most flavours, for example, \+ is a literal + and + is the quantifier; in grep + is literal and \+ is the quantifier. And there are other characters where the meaning of \ is reversed in this way.

Secondly, the string literal isn't the same as the string itself, because of backslash-escaping. The string literal looks like this:

"(%)(?!(\\[=\*\\[|\\]=\*\\])).\*$\n?"

while the string itself looks like this:

(%)(?!(\[=\*\[|\]=\*\])).\*$

?

(with a newline character near the end).

Let's look at the following subexpression:

\[=\*\[|\]=\*\]

At first I thought this was a character class, delimited by \[ and \]. But (a) I don't know of any flavour of regex where backslash-escaped square brackets are character class delimiters and unescaped ones are literal square brackets, rather than vice versa; (b) why would someone write a character class with repeated characters?; (c) there's no obvious reason why the first \] would be a literal ] and the second one would end the character class. So it looks like \[ and \] are literal square brackets.

| means "or" in regexes. It is a low-precedence operator. So this subexpression means either \[=\*\[ or \]=\*\]. In other words, it matches strings such as [[, [=[, [======[, etc, as well as ]], ]=], etc.

(?!...) is a zero-width assertion. It is a negative lookahead: it matches at any point in the string where the positive lookahead (?=...) would not match. In general, if the regex A matches the string a and C matches string c then the regex A(?!B)C matches the string ac, unless the regex B matches c (or some substring of c). In other words, the match fails if the string is something like %]==].

.\* matches any number of characters. (0 is a number). (I assume this doesn't match newlines.) $ is another zero-width assertion: it can only match at the end of the line. Actually, it's not needed in this case - the .\* subexpression is greedy and will match all non-newline characters, so the end of the .\* match is guaranteed to be the end of the line. That is, unless there's some edge case I'm not aware of involving carriage returns or some even more exotic line terminating character.

Finally, \n? will match the newline character itself, if it exists (? is a quantifier). If this is the last line of the string then there may not be a newline; in that case the regex match would fail without the ?.

Putting it all together: The regex will match from a % until the end of the line, including the newline character if it exists, unless the string it's trying to match starts with %[[ or %]==] or something similar.

# Rex Eats Regular Expressions for Breakfast

[](http://www.copyscape.com/)

Rex eats regular expressions for breakfast. And so can you! This regex tutorial, one of the most detailed on the web, takes you all the way to mastery.  
  
This page explains what makes this site special among all other regex sites, but first let's answer a burning question:   
  
What is the meaning of life?  
That's easy. As per the [regex humor page](http://www.rexegg.com/regex-humor.html" \l "meaning-of-life), it's simply  
^(?=(?!(.)\1)([^\DO:105-93+30])(?-1)(?<!\d(?<=(?![5-90-3])\d))).[^\WHY?]$   
Now for the other burning question…

## What is a Regex?

First, a regex is a text string. For instance, foo is a regex. So is [A-Z]+:\d+.  
  
Those text strings describe patterns to find text or *positions* within a body of text. For instance, the regex foo matches the string *foo*, the regex [A-Z]+:\d+ matches string fragments like *F:1* and *GO:30*, and the regex (?<=[a-z])(?=[A-Z]) matches the position in the string *CamelCase* where we shift from a lower-case letter to an upper-case letter.  
  
Typically, these patterns (which can be beautifully intricate and precise) are used for four main tasks: to **find** text within a larger body of text; to **validate** that a string conforms to a desired format; to **replace** text (or **insert** text at matched positions, which is the same process); and to **split** strings.   
  
For instance, the CamelCase pattern from the last paragraph can be used to split *MyLovelyValentine* into its three component words. And you could use the regex \_\d+\_ to find digits within underscores (as in *\_12\_*) and to replace the underscores with double dashes, yielding *--12--*, something you could not do with a conventional search-and-replace (details for that technique are in the recipe about [replacing one delimiter with another](http://www.rexegg.com/regex-cookbook.html" \l "tagdelimiter)).   
  
Who does this work of finding, replacing, splitting? A *regex engine*. For instance, you can find regex engines in text editors such as Notepad++ and EditPad Pro. You also find regex engines ready to roar in most programming languages—such as C#, Python, Perl, PHP, Java, JavaScript and Ruby.   
  
Let's compress the definition from the earlier paragraphs:

A regex is a text string that describes a pattern that a regex engine uses in order to find text (or positions) in a body of text, typically for the purposes of validating, finding, replacing or splitting.

**Is a Regex the same as a Regular Expression?**  
Mostly yes, with a little bit of no. At this stage, this is a semantic question—it depends on what one means by *regular expression*. That topic and other juicy details are discussed on the page about [Regex vs. Regular Expressions](http://www.rexegg.com/regex-vs-regular-expression.html).

## About this Site

Before we dive in—and only if you have time—I'd like to introduce this site and what makes it special.  
  
I love regular expressions. They are a small computer language of their own.  
When I was a young dinosaur, I didn't take the time to properly learn the syntax, largely because I really didn't feel like learning another language. Who needs regex, I thought, when your programming language has functions that let you dig into strings from the left, the middle and the right?  
  
What's more, the raw syntax you usually see in code that contains regexes used to intimidate me. Who wants to deal with a language that looks like this?  
  
(?s)/\\*(?:(?!\\*/)[\*$ \_/+\\-])\*(.\*?)[\*$ \_/+\\-]\*?\\*/

It is well worth investing a bit of time in Regular Expressions. You won't look back!

As it turns out, you really don't have to write your regular expressions like this. In many regex flavors, you can aerate your regex just like code, indenting and inserting comments as you go. If you walk with me through this site, you will be able to understand the expression above. Just as a preview, here is how the very same regex might look once "aerated" and commented, on multiple lines:

(?xs) # Turn on free-spacing and DOTALL modes

/\\* # Match a forward slash and a star

(?: # Some comment goes here

(?!\\*/) # Blah

[-+\*\_/ \\] # Blah blah

)\* # Blah blah blah

(.\*?) # More blah

[-+\*\_/$ \\]\*? # Yadda yadda blah

\\*/ # Match a star and a forward slash

No doubt about it, even with comments and breathing room, there is something raw and experimental about writing a regex pattern.   
  
Besides, how well your pattern performs doesn't only depend on applying correct syntax. There are several ways of doing things, and various regex engines may optimize some of these ways behind your back.   
  
With regex, you are stepping down to a fairly low level, within earshot of the machine room. I like that. And I've been liking it all the more since learning about tools and safeguards to keep me from falling into the boiler.

## A (hopefully) Different Presentation of Regex

To really learn, you need to see the same information in different ways.

There are excellent web pages about regex. Not many, but there are some, and I reference my favorite ones throughout the site. Then there are many pages that repeat the same old syntax reference. The problem is that for unfamiliar technical information to anchor itself in your mind—or at least in mine—you need to see it presented from various angles. When I started learning regex, as I was hopping from page to page and book to book, the content was much alike so the "information tree" wasn't yielding all its fruits. As a result, several questions that cut diagonally through the field of regex were staying unresolved.  
  
*RexEgg* tries to present regular expressions a bit differently, in the hope that these different angles help many people become more grounded in their knowledge of regex. If you are looking for a drawn-out primer, this is not the place, as I don't see the need to pollute our beautiful world wide web with another explanation of how to match "foo" in "foo bar". But if you take your time to read the carefully-built tables on the [quick-launch page](http://www.rexegg.com/regex-quickstart.html) then perhaps the [page about (? … ) syntax](http://www.rexegg.com/regex-disambiguation.html), you will experience what may be the most accelerated regex introduction around.

## What Will you Find on this Site

Oh, yes, and forget about practice, that's completely overrated. Just kidding.

Get ready, because as far as I know, this site is one of the two most comprehensive regex sources on the net—along with Jan Goyvaerts excellent [regex tutorial site](http://www.regular-expressions.info/). It aims to fill gaps in how regex information is presented elsewhere, including the major regex books. Here are some of the things you will find here.   
  
✽ A step-by-step explanation of simple and advanced regular expressions crafted for various contexts (such as text matching, file renaming, search-and-replace).  
  
✽ A presentation of the many contexts where you may run into regular expressions (from Apache to your html editor and file manager), complete with examples.  
  
✽ A reference about **(? … )**—to reduce confusion by bringing all the pieces of syntax that start with an opening parenthesis and a question mark into a single place.  
  
✽ A discussion of Conditional Regexes, a topic about which there is little information.   
  
✽ A discussion of Recursive Regexes, a topic about which there is very little information.   
  
✽ Pages dedicated to regex in C#, Python, PHP and other languages.  
  
✽ Plenty of tips & tricks.  
  
✽ Sections about regex tools and regex books.   
  
✽ And much more!  
  
I wish you lots of fun on your journey with regular expressions.   
  
Smiles,   
  
Rex

http://www.rexegg.com/

# Quick-Start: Regex Cheat Sheet

[](http://www.copyscape.com/)

The tables below are a reference to basic regex. While reading the rest of the site, when in doubt, you can always come back and look here. (It you want a bookmark, here's a direct link to the [regex reference tables](_blank)). I encourage you to print the tables so you have a cheat sheet on your desk for quick reference.  
  
The tables are not exhaustive, for two reasons. First, every regex flavor is different, and I didn't want to crowd the page with overly exotic syntax. For a full reference to the particular regex flavors you'll be using, it's always best to go straight to the source. In fact, for some regex engines (such as Perl, PCRE, Java and .NET) you may want to check once a year, as their creators often introduce new features.   
  
The other reason the tables are not exhaustive is that I wanted them to serve as a quick introduction to regex. If you are a complete beginner, you should get a firm grasp of basic regex syntax just by reading the examples in the tables. I tried to introduce features in a logical order and to keep out oddities that I've never seen in actual use, such as the "bell character". With these tables as a jumping board, you will be able to advance to mastery by exploring the other pages on the site.

## How to use the tables

The tables are meant to serve as an accelerated regex course, and they are meant to be read slowly, one line at a time. On each line, in the leftmost column, you will find a new element of regex syntax. The next column, "Legend", explains what the element means (or encodes) in the regex syntax. The next two columns work hand in hand: the "Example" column gives a valid regular expression that uses the element, and the "Sample Match" column presents a text string that could be matched by the regular expression.  
  
You can read the tables online, of course, but if you suffer from even the mildest case of online-ADD (attention deficit disorder), like most of us… Well then, I highly recommend you print them out. You'll be able to study them slowly, and to use them as a cheat sheet later, when you are reading the rest of the site or experimenting with your own regular expressions.   
  
Enjoy!   
  
If you overdose, make sure not to miss the next page, which comes back down to Earth and talks about some really cool stuff: [**The 1001 ways to use Regex**](http://www.rexegg.com/regex-uses.html).

## Regex Accelerated Course and Cheat Sheet

For easy navigation, here are some jumping points to various sections of the page:  
  
✽ [Characters](http://www.rexegg.com/regex-quickstart.html" \l "chars)  
✽ [Quantifiers](http://www.rexegg.com/regex-quickstart.html" \l "quantifiers)  
✽ [More Characters](http://www.rexegg.com/regex-quickstart.html" \l "morechars)  
✽ [Logic](http://www.rexegg.com/regex-quickstart.html" \l "logic)  
✽ [More White-Space](http://www.rexegg.com/regex-quickstart.html" \l "whitespace)  
✽ [More Quantifiers](http://www.rexegg.com/regex-quickstart.html" \l "morequants)  
✽ [Character Classes](http://www.rexegg.com/regex-quickstart.html" \l "classes)  
✽ [Anchors and Boundaries](http://www.rexegg.com/regex-quickstart.html" \l "anchors)  
✽ [POSIX Classes](http://www.rexegg.com/regex-quickstart.html" \l "posix)  
✽ [Inline Modifiers](http://www.rexegg.com/regex-quickstart.html" \l "modifiers)  
✽ [Lookarounds](http://www.rexegg.com/regex-quickstart.html" \l "lookarounds)  
✽ [Character Class Operations](http://www.rexegg.com/regex-quickstart.html" \l "classoperations)  
✽ [Other Syntax](http://www.rexegg.com/regex-quickstart.html" \l "other)  
  
  
[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "chars)

## Characters

|  |  |  |  |
| --- | --- | --- | --- |
| Character | Legend | Example | Sample Match |
| \d | Most engines: one digit from 0 to 9 | file\_\d\d | file\_25 |
| \d | .NET, Python 3: one Unicode digit in any script | file\_\d\d | file\_9੩ |
| \w | Most engines: "word character": ASCII letter, digit or underscore | \w-\w\w\w | A-b\_1 |
| \w | .Python 3: "word character": Unicode letter, ideogram, digit, or underscore | \w-\w\w\w | 字-ま\_۳ |
| \w | .NET: "word character": Unicode letter, ideogram, digit, or connector | \w-\w\w\w | 字-ま‿۳ |
| \s | Most engines: "whitespace character": space, tab, newline, carriage return, vertical tab | a\sb\sc | a b c |
| \s | .NET, Python 3, JavaScript: "whitespace character": any Unicode separator | a\sb\sc | a b c |
| \D | One character that is not a *digit* as defined by your engine's *\d* | \D\D\D | ABC |
| \W | One character that is not a *word character* as defined by your engine's *\w* | \W\W\W\W\W | \*-+=) |
| \S | One character that is not a *whitespace character* as defined by your engine's *\s* | \S\S\S\S | Yoyo |

[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "quantifiers)

## Quantifiers

|  |  |  |  |
| --- | --- | --- | --- |
| Quantifier | Legend | Example | Sample Match |
| + | One or more | Version \w-\w+ | Version A-b1\_1 |
| {3} | Exactly three times | \D{3} | ABC |
| {2,4} | Two to four times | \d{2,4} | 156 |
| {3,} | Three or more times | \w{3,} | regex\_tutorial |
| \* | Zero or more times | A\*B\*C\* | AAACC |
| ? | Once or none | plurals? | plural |

[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "morechars)

## More Characters

|  |  |  |  |
| --- | --- | --- | --- |
| Character | Legend | Example | Sample Match |
| **.** | Any character except line break | a.c | abc |
| **.** | Any character except line break | .\* | whatever, man. |
| \**.** | A period (special character: needs to be escaped by a \) | a\.c | a.c |
| \ | Escapes a special character | \.\\*\+\?    \$\^\/\\ | .\*+?    $^/\ |
| \ | Escapes a special character | \[\{\(\)\}\] | [{()}] |

[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "logic)

## Logic

|  |  |  |  |
| --- | --- | --- | --- |
| Logic | Legend | Example | Sample Match |
| | | Alternation / OR operand | 22|33 | 33 |
| ( … ) | Capturing group | A(nt|pple) | Apple (captures "pple") |
| \1 | Contents of Group 1 | r(\w)g\1x | regex |
| \2 | Contents of Group 2 | (\d\d)\+(\d\d)=\2\+\1 | 12+65=65+12 |
| (?: … ) | Non-capturing group | A(?:nt|pple) | Apple |

[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "whitespace)

## More White-Space

|  |  |  |  |
| --- | --- | --- | --- |
| Character | Legend | Example | Sample Match |
| \t | Tab | T\t\w{2} | T     ab |
| \r | Carriage return character | see below |  |
| \n | Line feed character | see below |  |
| \r\n | Line separator on Windows | AB\r\nCD | AB CD |
| \N | Perl, PCRE (C, PHP, R…): one character that is not a line break | \N+ | ABC |
| \h | Perl, PCRE (C, PHP, R…), Java: one horizontal whitespace character: tab or Unicode space separator |  |  |
| \H | One character that is not a horizontal whitespace |  |  |
| \v | .NET, JavaScript, Python, Ruby: vertical tab |  |  |
| \v | Perl, PCRE (C, PHP, R…), Java: one vertical whitespace character: line feed, carriage return, vertical tab, form feed, paragraph or line separator |  |  |
| \V | Perl, PCRE (C, PHP, R…), Java: any character that is not a vertical whitespace |  |  |
| \R | Perl, PCRE (C, PHP, R…), Java: one line break (carriage return + line feed pair, and all the characters matched by \v) |  |  |

[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "morequants)

## More Quantifiers

|  |  |  |  |
| --- | --- | --- | --- |
| Quantifier | Legend | Example | Sample Match |
| + | The + (one or more) is "greedy" | \d+ | 12345 |
| ? | Makes quantifiers "lazy" | \d+? | 1 in **1**2345 |
| \* | The \* (zero or more) is "greedy" | A\* | AAA |
| ? | Makes quantifiers "lazy" | A\*? | empty in AAA |
| {2,4} | Two to four times, "greedy" | \w{2,4} | abcd |
| ? | Makes quantifiers "lazy" | \w{2,4}? | ab in **ab**cd |

[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "classes)

## Character Classes

|  |  |  |  |
| --- | --- | --- | --- |
| Character | Legend | Example | Sample Match |
| [ … ] | One of the characters in the brackets | [AEIOU] | One uppercase vowel |
| [ … ] | One of the characters in the brackets | T[ao]p | *Tap* or *Top* |
| - | Range indicator | [a-z] | One lowercase letter |
| [x-y] | One of the characters in the range from x to y | [A-Z]+ | GREAT |
| [ … ] | One of the characters in the brackets | [AB1-5w-z] | One of either: A,B,1,2,3,4,5,w,x,y,z |
| [x-y] | One of the characters in the range from x to y | [ -~]+ | Characters in the printable section of the [ASCII table](http://www.asciitable.com/). |
| [^x] | One character that is not x | [^a-z]{3} | A1! |
| [^x-y] | One of the characters **not** in the range from x to y | [^ -~]+ | Characters that are **not** in the printable section of the [ASCII table](http://www.asciitable.com/). |
| [\d\D] | One character that is a digit or a non-digit | [\d\D]+ | Any characters, inc- luding new lines, which the regular dot doesn't match |
| [\x41] | Matches the character at hexadecimal position 41 in the ASCII table, i.e. A | [\x41-\x45]{3} | ABE |

[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "anchors)

## [Anchors](http://www.rexegg.com/regex-anchors.html) and [Boundaries](http://www.rexegg.com/regex-boundaries.html)

|  |  |  |  |
| --- | --- | --- | --- |
| Anchor | Legend | Example | Sample Match |
| ^ | [Start of string](http://www.rexegg.com/regex-anchors.html" \l "caret) or [start of line](http://www.rexegg.com/regex-anchors.html" \l "carmulti) depending on multiline mode. (But when [^inside brackets], it means "not") | ^abc .\* | abc (line start) |
| $ | [End of string](http://www.rexegg.com/regex-anchors.html" \l "dollar) or [end of line](http://www.rexegg.com/regex-anchors.html" \l "eol) depending on multiline mode. Many engine-dependent subtleties. | .\*? the end$ | this is the end |
| \A | [Beginning of string](http://www.rexegg.com/regex-anchors.html" \l "A) (all major engines except JS) | \Aabc[\d\D]\* | abc (string... ...start) |
| \z | [Very end of the string](http://www.rexegg.com/regex-anchors.html" \l "z) Not available in Python and JS | the end\z | this is...\n...**the end** |
| \Z | [End of string](http://www.rexegg.com/regex-anchors.html" \l "Z) or (except Python) before final line break Not available in JS | the end\Z | this is...\n...**the end**\n |
| \G | [Beginning of String or End of Previous Match](http://www.rexegg.com/regex-anchors.html" \l "G) .NET, Java, PCRE (C, PHP, R…), Perl, Ruby |  |  |
| \b | [Word boundary](http://www.rexegg.com/regex-boundaries.html" \l "wordboundary) Most engines: position where one side only is an ASCII letter, digit or underscore | Bob.\*\bcat\b | Bob ate the cat |
| \b | [Word boundary](http://www.rexegg.com/regex-boundaries.html" \l "wordboundary) .NET, Java, Python 3, Ruby: position where one side only is a Unicode letter, digit or underscore | Bob.\*\b\кошка\b | Bob ate the кошка |
| \B | [Not a word boundary](http://www.rexegg.com/regex-boundaries.html" \l "notb) | c.\*\Bcat\B.\* | copycats |

[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "posix)

## POSIX Classes

|  |  |  |  |
| --- | --- | --- | --- |
| Character | Legend | Example | Sample Match |
| [:alpha:] | PCRE (C, PHP, R…): ASCII letters A-Z and a-z | [8[:alpha:]]+ | WellDone88 |
| [:alpha:] | Ruby 2: Unicode letter or ideogram | [[:alpha:]\d]+ | кошка99 |
| [:alnum:] | PCRE (C, PHP, R…): ASCII digits and letters A-Z and a-z | [[:alnum:]]{10} | ABCDE12345 |
| [:alnum:] | Ruby 2: Unicode digit, letter or ideogram | [[:alnum:]]{10} | кошка90210 |
| [:punct:] | PCRE (C, PHP, R…): ASCII punctuation mark | [[:punct:]]+ | ?!.,:; |
| [:punct:] | Ruby: Unicode punctuation mark | [[:punct:]]+ | ‽,:〽⁆ |

[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "modifiers)

## [Inline Modifiers](http://www.rexegg.com/regex-modifiers.html)

None of these are supported in JavaScript. In Ruby, beware of (?s) and (?m).

|  |  |  |  |
| --- | --- | --- | --- |
| Modifier | Legend | Example | Sample Match |
| (?i) | [Case-insensitive mode](http://www.rexegg.com/regex-modifiers.html" \l "i) (except JavaScript) | (?i)Monday | monDAY |
| (?s) | [DOTALL mode](http://www.rexegg.com/regex-modifiers.html" \l "dotall) (except JS and Ruby). The dot (.) matches new line characters (\r\n). Also known as "single-line mode" because the dot treats the entire input as a single line | (?s)From A.\*to Z | From A to Z |
| (?m) | [Multiline mode](http://www.rexegg.com/regex-modifiers.html" \l "multiline) (except Ruby and JS) ^ and $ match at the beginning and end of every line | (?m)1\r\n^2$\r\n^3$ | 1 2 3 |
| (?m) | [In Ruby](http://www.rexegg.com/regex-modifiers.html" \l "rubym): the same as (?s) in other engines, i.e. DOTALL mode, i.e. dot matches line breaks | (?m)From A.\*to Z | From A to Z |
| (?x) | [Free-Spacing Mode mode](http://www.rexegg.com/regex-modifiers.html" \l "freespacing) (except JavaScript). Also known as comment mode or whitespace mode | (?x) # this is a # comment abc # write on multiple # lines [ ]d # spaces must be # in brackets | abc d |
| (?n) | [.NET, PCRE 10.30+: named capture only](http://www.rexegg.com/regex-modifiers.html" \l "n) | Turns all (parentheses) into non-capture groups. To capture, use [named groups](http://www.rexegg.com/regex-capture.html" \l "namedgroups). |  |
| (?d) | [Java: Unix linebreaks only](http://www.rexegg.com/regex-modifiers.html" \l "d) | The dot and the ^ and $ anchors are only affected by \n |  |
| (?^) | [PCRE 10.32+: unset modifiers](http://www.rexegg.com/regex-disambiguation.html" \l "unset-all) | Unsets ismnx modifiers |  |

[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "lookarounds)

## [Lookarounds](http://www.rexegg.com/regex-lookarounds.html)

|  |  |  |  |
| --- | --- | --- | --- |
| Lookaround | Legend | Example | Sample Match |
| (?=…) | [Positive lookahead](http://www.rexegg.com/regex-disambiguation.html" \l "lookahead) | (?=\d{10})\d{5} | 01234 in **01234**56789 |
| (?<=…) | [Positive lookbehind](http://www.rexegg.com/regex-disambiguation.html" \l "lookbehind) | (?<=\d)cat | cat in 1**cat** |
| (?!…) | [Negative lookahead](http://www.rexegg.com/regex-disambiguation.html" \l "negative-lookahead) | (?!theatre)the\w+ | theme |
| (?<!…) | [Negative lookbehind](http://www.rexegg.com/regex-disambiguation.html" \l "negative-lookbehind) | \w{3}(?<!mon)ster | Munster |

[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "classoperations)

## [Character Class Operations](http://www.rexegg.com/regex-class-operations.html)

|  |  |  |  |
| --- | --- | --- | --- |
| Class Operation | Legend | Example | Sample Match |
| […-[…]] | .NET: character class subtraction. One character that is in those on the left, but not in the subtracted class. | [a-z-[aeiou]] | Any lowercase consonant |
| […-[…]] | .NET: character class subtraction. | [\p{IsArabic}-[\D]] | An Arabic character that is not a non-digit, i.e., an Arabic digit |
| […&&[…]] | Java, Ruby 2+: character class intersection. One character that is both in those on the left and in the && class. | [\S&&[\D]] | An non-whitespace character that is a non-digit. |
| […&&[…]] | Java, Ruby 2+: character class intersection. | [\S&&[\D]&&[^a-zA-Z]] | An non-whitespace character that a non-digit and not a letter. |
| […&&[^…]] | Java, Ruby 2+: character class subtraction is obtained by intersecting a class with a negated class | [a-z&&[^aeiou]] | An English lowercase letter that is not a vowel. |
| […&&[^…]] | Java, Ruby 2+: character class subtraction | [\p{InArabic}&&[^\p{L}\p{N}]] | An Arabic character that is not a letter or a number |

[(direct link)](http://www.rexegg.com/regex-quickstart.html" \l "other)

## Other Syntax

|  |  |  |  |
| --- | --- | --- | --- |
| Syntax | Legend | Example | Sample Match |
| \K | [Keep Out](http://www.rexegg.com/regex-best-trick.html" \l "bsk) Perl, PCRE (C, PHP, R…), Python's alternate [*regex*](https://pypi.python.org/pypi/regex) engine, Ruby 2+: drop everything that was matched so far from the overall match to be returned | prefix\K\d+ | 12 |
| \Q…\E | Perl, PCRE (C, PHP, R…), Java: treat anything between the delimiters as a literal string. Useful to escape metacharacters. | \Q(C++ ?)\E | (C++ ?) |

http://www.rexegg.com/regex-quickstart.html

# How to Regex: A Practical Guide to Regular Expressions (Regex) for Hackers

Regular Expressions (a.k.a regex, or regexp) is one of those things that has a fairly steep learning curve, but once you dedicate an hour or so to learning the basics, you will find that you will be far more efficient with everyday tasks. By the time you finish reading this blog, hopefully you will have a practical understanding of:

* Regex fundamentals
* How to use regex in a practical sense
* How to bypass regex-based security controls

*Let’s go!*

### What is a Regex?

A regex is a string of characters that defines a search pattern. The most basic example would be a straight string match, for example “abc”.

|  |  |  |
| --- | --- | --- |
| **Regex String** | **Matches** | **Doesn’t Match** |
| abc | abc | bbc |

Of course, if regex was only capable of doing straight string matches, it wouldn’t be very useful! Here’s another example:

|  |  |  |
| --- | --- | --- |
| **Regex String** | **Matches** | **Doesn’t Match** |
| .bc | abc bbc | abb acc |

You may have suspected this already, but the “.” dot character is a wildcard, it can be replaced with any single character and the search string will still match! Now regex is looking *slightly* more useful, but we haven’t even scratched the surface yet!

### When is Regex Useful?

Oftentimes when hacking, you will find yourself in a situation where you need to parse or edit some text in an automated fashion.

As an example, let’s use a list of URLs as our input.

https://example.com/test

https://www.bugcrowd.com/?param=value

http://hakluke.com

ftp://EXAMPLE.org?test=success

We are using a tool that requires domain names, not URLs. We need to somehow extract the domain names from these URLs, but how? As you can see, the URLs in this list are quite varied, there are three different schemas (http, https and ftp), some have directories, some do not, some have parameters, others do not.

We could open it up in a text editor and manually remove them, but what if there are 1 million URLs? Do we waste a day doing this? Train monkeys? No, we don’t train monkeys. We are civilized humans. Civilized humans use regex!

There are many different ways to do this, but here’s the regex string I came up with:

(?<=:\/\/)(?i)[a-z,.]\*

See below for the regex in action:



If you’re new to regex, the example above will probably provide more questions than answers – but hopefully it will give you an idea of the power of regular expressions. Read on to understand what it all means!

### The Tools

Before we dive into regular expressions further, it’s important to mention a few of the basic tools that utilise regular expressions, and their common use cases.

* sed allows you to find and replace text. You can provide it with a regular expression to find, any match will be replaced with the text you provide.
* grep allows you to filter by regular expressions. The input could be a large chunk of text, and the output will only contain the filtered results. It’s also good to know that pcregrep exists, which uses the feature-rich Perl variant of regex to parse the input. If you’re using GNU grep, you should be able to use grep -P instead, which should have the same result.
* Regex101.com is a website that allows you to test out your regular expressions.

### The Basics

|  |  |
| --- | --- |
| **Regular Expression** | **Meaning** |
| a | The literal character “a” |
| A | The literal character “A” |
| . | Any single character |
| [a-z] | Any lowercase alphabet character |
| [A-Z] | Any uppercase alphabet character |
| [A-F] | One of the following: A, B, C, D, E or F |
| [0-9] | Any single digit number |
| [a-fA-F0-9] | Any hexadecimal character |
| ^ | The start of a line |
| $ | The end of a line |
| \* | Match *zero* or more of the preceding expression |
| + | Match *one* or more of the preceding expression |

Even just knowing the expressions above, we can construct some very useful queries. For example, let’s say we are parsing a file with the following contents:

Monday

Green

Tuesday

Weather

Wednesday

Thursday

Friday

Saturday

Sunday

Pinecone

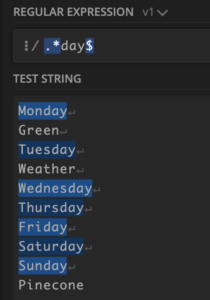
If we wanted to extract the words ending with “day”, we could use the following regex:

.\*day$

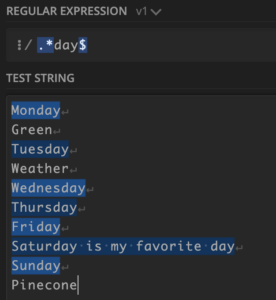
Let’s go through what this actually does:

|  |  |
| --- | --- |
| **Expression** | **Meaning** |
| . | Any single character |
| \* | Zero or more |
| day | Literally “day” |
| $ | The end of a line |

When we put it all together we get “Zero or more characters followed by ‘day’ followed by the end of a line”. Below is an image of the results in regex101.com. The blue highlights indicate a matched string.



Looking good! But if we are ingesting more dynamic data, there’s a good chance this will break. One of the issues with this method is that it matches any line ending with “day”, not just single words. For example:



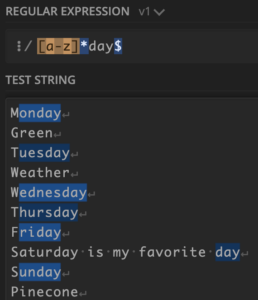
Of course, this is not ideal, because we are wanting to extract only weekdays.

### Case Sensitivity

If we wanted to be more specific and not include the rest of the line, we could try something like this:

[a-z]\*day$

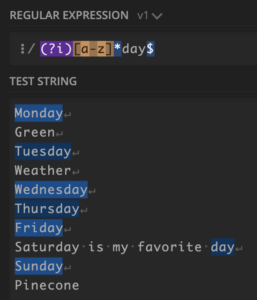
Except something annoying will happen:



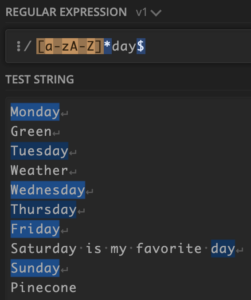
There are two problems here:

* The first letter of each line is not selected (because it is uppercase, and therefore does not satisfy [a-z])
* The “day” at the end of the line is selected over the “Saturday” at the start of the line

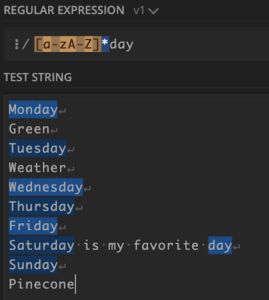
We can solve the first issue by switching off case sensitivity by preceding the expression with (?i). For example:



We could also have explicitly included uppercase letters in the search, which would have the same result:



We still have the second problem though, where “day” is chosen over “Saturday”. You might think that removing the $ at the end of our regex would fix this problem, but it doesn’t quite work because now both “Saturday” *and* “day” are selected:

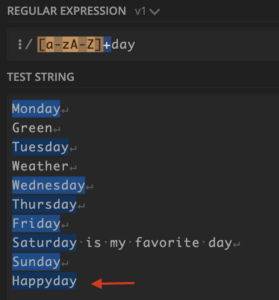


### Quantifiers

One way to solve this problem would be to use a different quantifier. Currently the quantifier we are using is \*. Here is a list of quantifiers:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Quantifier** | **Meaning** | **Example** | **Matches** | **Doesn’t match** |
| \* | Zero or more | a\*b | b,ab,aab | bb,abb |
| + | One or more | a+b | ab,aab | b,abb |
| ? | Once or not at all | a?b | b,ab | aab,bb |
| {5} | Exactly 5 times | a{5}b | aaaaab | ab,aaaaaab |
| {3,6} | 3 to 6 times | a{3,6}b | aaab,aaaaab | aab,aaaaaaab |
| {3,} | 3 or more times | a{3,}b | aaab,aaaaaaab | aab,bb |

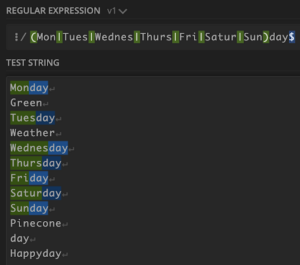
So we could solve this issue by using the “+” quantifier instead of the “\*” quantifier. This would mean that “day” would not be matched, because it does not have an alphabetical character preceding it. Remember that + is the same as \* except that + requires *one* or more instance of the preceding expression, while \* requires *zero* or more. Let’s try out the +:



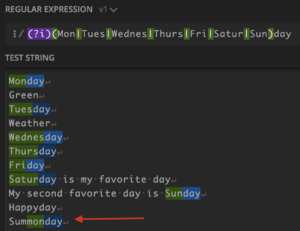
Great! The word “day” is no longer matched by itself. Now we have another issue though. Words like “Happyday” or “Holiday” would still be matched.

### The OR operator

We can match one expression OR another expression by using the OR operator, for example (this|that) would match “this” or “that”. To match all of the weekdays explicitly, we could do something like this:



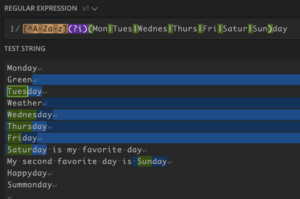
Yes! Finally it is working! Now we can just add case insensitivity and we’re all done! Wait.. what is this?:



We’re getting warmer, but we’re still not there.

### The NOT operator

To fix this, we want to make sure that none of the characters directly before the weekday name are letters or numbers. We can do this by using the NOT operator, which is the ^ character. For example:



Now “Summonday” is not selected, but we have another two issues to solve:

* The whitespace character before the weekday name is selected (space or new line)
* Monday is not selected, because it is at the start of the file

So how can we check if the preceding character is not an alphabet character without actually selecting it?

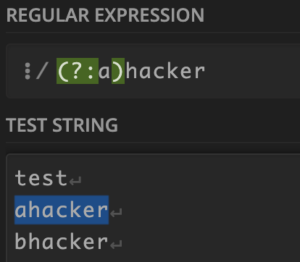
### Lookaheads and Lookbehinds

To achieve this, we need to use what is known as a “negative lookbehind”. It is called a “lookbehind” because it checks the expression immediately before the match, but without actually including it as part of the match. Before we dive into negative lookbehinds, let’s give an example of a positive lookbehind.

The syntax is:

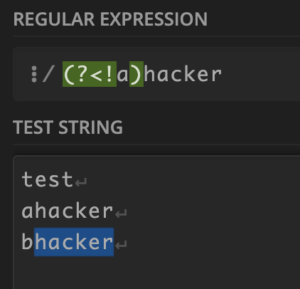
(?:<thing you want to check for>)<rest of expression>

For example:



As you can see, “hacker” is only matched if it is preceded with an “a”. A negative lookbehind does exactly the opposite, and the syntax is:

(?<!thing you want to check for>)<rest of expression>



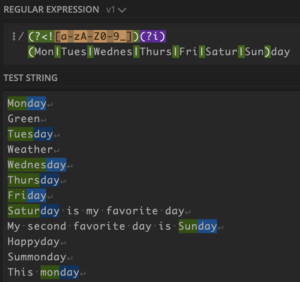
The example above is a “*negative* lookbehind”, so the word hacker will match as long as it is NOT preceded by an “a”.

So! To solve the issues with our original problem, we can use the following:

(?<![a-zA-Z0-9\_])(?i)(Mon|Tues|Wednes|Thurs|Fri|Satur|Sun)day

This will match any weekday that is not preceded by a-z, A-Z 0-9 or an underscore \_.

And here is the outcome:



### Bypassing Regex-Based Security Controls

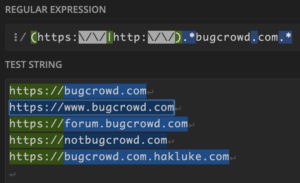
At a glance, writing custom regex may appear to be a great solution for implementing various types of input validation. Above, while learning some basic regex, we have demonstrated that there are many edge cases that need to be accounted for. As such, it is generally recommended that using regex for input validation is *avoided* in favour of allowlists where possible.

To demonstrate why this is generally a bad idea, let’s say that we need to validate that a URL belongs to either bugcrowd.com, or a subdomain of bugcrowd.com. If the regex matches the URL, we will allow it, otherwise we will deny it. What regex should we use?

Many developers make the mistake of using something like this:

([https://|http://](about:blank)).\*bugcrowd.com.\*

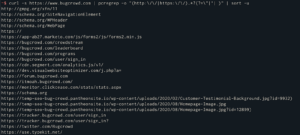
Of course, this can be easily bypassed by prepending or appending text that results in a completely different domain, such as “https://notbugcrowd.com” or “https://bugcrowd.com.hakluke.com”.



If you do ever run into a situation where your input is being checked or filtered in some way, through trial and error, see if you can figure out how it is being filtered. There’s a chance that regex has been used, and that means there’s a chance that it can be bypassed. Get creative!

### Practical Regex Example

As a hacker, I most often use regex for filtering large chunks of text to parse out useful information. For example, below we curl [https://www.bugcrowd.com](https://www.bugcrowd.com/) and then use some regex magic to extract all of the URLs out of the response:



The regular expression I used here is:

(http:\/\/|https:\/\/).\*?(?="|'| )

The tool that I used is “pcregrep”. It is basically the same as “grep” except it uses the PCRE regular expression library, which has more features. If you’re using GNU grep, you should be able to use grep -P instead.

### Stay in Touch

At Bugcrowd, we post these kinds of how-to articles fairly frequently! If you’d like to learn more, you can [join our Discord](https://discord.com/invite/TWr3Brs), follow us on [Twitter](https://twitter.com/bugcrowd), or check out our [video content on YouTube](https://youtube.com/bugcrowd).

If you’d like to see more from the author personally, follow hakluke on [Twitter](https://twitter.com/hakluke), [YouTube](https://youtube.com/hakluke), [Instagram](https://instagram.com/hakluke_) or check out his [website](https://hakluke.com/).

https://www.bugcrowd.com/blog/how-to-regex-a-practical-guide-to-regular-expressions-regex-for-hackers/