LES EXPRESSIONS RATIONNELLES

[Introduction](#_j5omz82nvu1j)

[Exemple](#_r0kizoc4pmt5)

[Les caractères littéraux](#_u6718jbqq1d4)

[Les symboles](#_22qeo9hyiyc9)

[Le point](#_ue59ragys6x0)

[Les ancres de début et de fin](#_69hh8ejes8uc)

[Les quantificateurs](#_u97l04xca80e)

[Les intervalles](#_npqywoj21v4w)

[Circonscrire la portée d'un masque](#_sow3rm5d9yyx)

[Les classes](#_vxkxianrklcp)

[Les classes de caractères](#_fiut5am8dtt6)

[Les classes prédéfinies](#_voui62cyavkf)

[Les classes exclues](#_656a6fjpzxw5)

[L'alternative](#_z0hc1ql4ggc8)

[Les métacaractères](#_i2otv14b8j4a)

[Les autres ancres](#_p8oxmeljo0qp)

[Les fonctions PCRE](#_wd278kmh4x7k)

[Exemples d'utilisation de quelques fonctions](#_a84kl1howdym)

[preg\_match($motif, $sujet)](#_ylhgjwhurho3)

[preg\_match($motif, $sujet, $res)](#_5d2h11yd0ofq)

[preg\_match\_all($motif, $sujet, $resultats)](#_e99iboa9txyp)

[preg\_split($motif, $sujet)](#_9qfobo5784z4)

[preg\_replace($motif, $rempl, $sujet)](#_gv5d1equh8qh)

[Les options PCRE](#_61g5svoivlz)

[La capture](#_v8nmznntjs8q)

[Les assertions](#_chtw8klyoxid)

[Fin provisoire...](#_w8kw48m4wkyi)

[Ressources](#_3lcrym6s7jxl)

# 

# Introduction

Regex signifie en anglais Regular expression. En français, on traduit par "expression rationnelle" ou "motif" (mais le terme "expression régulière" est encore très utilisé, bien que ce soit un anglicisme). Une regex est une chaîne de caractère, un "motif", qui permet de formaliser une recherche de façon extrêmement efficace. C'est un héritage des théories mathématiques des années 1940.

La bibliothèque PCRE permet de mettre en œuvre les expressions rationnelles. Cette bibliothèque est issue du langage PERL (Perl Compatible Regular Expression), et elle est [prise en charge par PHP](http://php.net/manual/fr/book.pcre.php). Attention néanmoins à ne pas utiliser des regex quand PHP propose des fonctions prédéfinies telles que str\_replace(), strcmp(), ltrim() etc. PHP intègre aussi une bibliothèque plus ancienne et moins performante nommée POSIX que nous ne décrirons pas ici : faites attention à ne pas confondre les documentations des bibliothèques PCRE et POSIX !

En HTML, les balises <input> supportent également les regex, avec l'attribut pattern.

### Exemple

Le motif suivant permet de savoir si mon nom de domaine commence par une chaîne alphanumérique de 2 à 20 caractères, suivie d'un point, et se terminant par "com" ou "fr".

**^\w{2,20}\.(com|fr)$**

On peut décomposer ce motif de la manière suivante :

|  |  |
| --- | --- |
| **^** | commence par |
| **\w** | a-zA-Z0-9\_ |
| **{2-20}** | de 2 à 20 caractères |
| **\.** | "." |
| **(com|fr)** | "com" ou "fr" |
| **$** | finit par |

En PHP, ce code affiche "1" (vrai) :

**$subject = "chatpersan.fr";**

**$pattern = "#^\w{2,20}\.(com|fr)$#";**

**echo preg\_match($pattern\_, $subject); // VRAI**

Notez les **délimiteurs** #, obligatoires en PHP (on aussi en utiliser d'autres : %, /, +...). Nous les omettrons dans les exemples qui suivent pour clairifier la syntaxe, mais ils sont bien obligatoires.

# Les caractères littéraux

Les **littéraux** correspondent aux caractères qui sont interprétés comme ils sont écrits. Le motif suivant ne contient que des littéraux :

**chat persan**

**"chat persan"; // VRAI**

**"le chat persan gris"; // VRAI**

**"chatpersan"; // FAUX**

# Les symboles

## Le point

Le point remplace n'importe quel caractère.

**chat.persan**

**"chat persan"; // VRAI**

**"chat9persan"; // VRAI**

**"chat9\_persan"; // FAUX**

**"chatpersan"; // FAUX**

**chat..persan**

**"chat persan"; // FAUX**

**"chat9\_persan"; // VRAI**

**Lorsqu'on souhaite utiliser un métacaractère en tant que caractère, il faut l'échapper avec "\".** Pour le point, par exemple, écrire \.

**chat\.persan**

**"chat persan"; // FAUX**

**"chat.persan"; // VRAI**

## Les ancres de début et de fin

L'accent circonflexe ^ et le symbole dollar $ sont des sortes d'ancres qui définissent respectivement le début et la fin d'une chaîne.

**^chat persan**

**"chat persan"; // VRAI**

**"le chat persan gris"; // FAUX**

**chat persan$**

**"chat persan"; // VRAI**

**"chat persan gris"; // FAUX**

**^chat persan$**

**"chat persan"; // VRAI**

**"chat persan gris"; // FAUX**

**"le chat persan"; // FAUX**

**"le chat persan gris"; // FAUX**

## Les quantificateurs

L'étoile \*, le plus + et le point d'interrogation ? sont des quantificateurs. Ils servent à quantifier le caractère ou la classe qui les précèdent.

|  |  |
| --- | --- |
| **\*** | Pas de caractère, ou un caractère, ou plusieurs caractères |
| **+** | Au moins un caractère |
| **?** | Pas de caractère ou un caractère |

**chat\*persan**

**"chapersan"; // VRAI**

**"chatpersan"; // VRAI**

**"chattpersan"; // VRAI**

**chat+persan**

**"chapersan"; // FAUX**

**"chatpersan"; // VRAI**

**"chattpersan"; // VRAI**

**chat?persan**

**"chapersan"; // VRAI**

**"chatpersan"; // VRAI**

**"chattpersan"; // FAUX**

**chat.+persan**

**"chatpersan"; // FAUX**

**"chat persan"; // VRAI**

**"chatte persanne"; // VRAI**

## Les intervalles

Les accolades { } permettent de définir des **intervalles**.

**chat{3}**

**"chatpersan"; // FAUX**

**"chatttpersan"; // VRAI**

**"chattttpersan"; // VRAI**

**chat{3}persan**

**"chatttpersan"; // VRAI**

**"chattttpersan"; // FAUX**

**chat{2,4}persan**

**"chatpersan"; // FAUX**

**"chattpersan"; // VRAI**

**"chatttpersan"; // VRAI**

**"chattttpersan"; // VRAI**

**"chatttttpersan"; // FAUX**

**{0,1}** est l'équivalent de **?**

**{1,}** est l'équivalent de **+**

**{0,}**  est l'équivalent de **\***

## Circonscrire la portée d'un masque

Les exemples ci-dessus de masques de quantification ou d'intervalle ne s'appliquent qu'au caractère qui précède le masque. Les parenthèses ( ) permettent de circonscrire la portée d'un masque à un sous-ensemble de caractères. Nous verons plus loin que les parenthèses sont également *capturantes*.

**(chat)? persan**

**"chat persan"; // VRAI**

**"chatchat persan"; // VRAI**

**"chatchatchat persan"; // VRAI**

**^(chat)? persan**

**"chatchatchat persan"; // FAUX**

**^(chat){2,3} persan**

**"chat persan"; // FAUX**

**"chatchat persan"; // VRAI**

**"chatchatchat persan"; // VRAI**

**"chatchatchatchat persan"; // FAUX**

## Les classes

### Les classes de caractères

Les crochets [ ] définissent des **classes de caractères**.

**ch[aei]t**

**"chat"; // VRAI**

**"chet"; // VRAI**

**"chit"; // VRAI**

**"chot"; // FAUX**

Le tiret - indique un intervalle de caractères.

**ch[a-d]t**

**"chat"; // VRAI**

**"chbt"; // VRAI**

**"chdt"; // VRAI**

**"chet"; // FAUX**

**<h[1-6]>**

**"<h1>"; // VRAI**

**"<h6>"; // VRAI**

**"<h0>"; // FAUX**

Le tiret placé en première position dans une classe devient un simple caractère, et indique donc que le caractère "-" est possible.

**ch[-ad]t**

**"chat"; // VRAI**

**"chbt"; // FAUX**

**"ch-t"; // VRAI**

**[A-Z] Lettres majuscules**

**[A-Za-z] Lettres majuscules et minuscules**

**[0-9] Chiffres**

**[A-Za-z0-9] Lettres majuscules, minuscules et chiffres…**

### Les classes prédéfinies

|  |  |
| --- | --- |
| **[[:alpha:]]**  **[[:digit:]] ou \d**  **[[:alnum:]]**  **[[:punct:]]**  **[[:lower:]]**  **[[:upper:]]**  **...** | n'importe quelle lettre  n'importe quel chiffre  n'importe quelle lettre ou chiffre  n'importe quel signe de ponctuation  n'importe quelle lettre en minuscule  n'importe quelle lettre capitale  ... |

|  |  |
| --- | --- |
| **\D**  **\w**  **\W**  **...** | n'est pas un chiffre **[^0-9]**  **[0-9a-zA-Z\_]**  **[^0-9a-zA-Z\_]**  ... |

### 

### Les classes exclues

Le circonflexe ^ placé au début d'une classe permet d'**exclure** des caractères. Ce métacaractère possède donc deux significations : à l'extérieur des classes, c'est une ancre, dans une classe en première position, c'est un métacaractère signifiant l'exclusion. Dans une classe, sur d'autres positions que la première, c'est un simple caractère.

**ch[^[:digit:]]t**

**"chat"; // VRAI**

**"ch\_t"; // VRAI**

**"ch5t"; // FAUX**

## L'alternative

La barre verticale | utilisée à l'intérieur de parenthèses ( ) indique une **alternative**.

**ch(a|abba)t**

**"chat"; // VRAI**

**"chabt"; // FAUX**

**"chabbat"; // VRAI**

**^(from|to):@**

**"from:@tom"; // VRAI**

**"to:@jerry"; // VRAI**

## Les métacaractères

Les **métacaractères** sont des caractères ayant un sens particulier. Ils doivent être échappés avec une barre oblique inverse \

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

Attention, on a vu que le tiret et l'accent circonflexe pouvaient avoir, en fonction de leur emplacement, plusieurs significations.

Dans certaines assertions, les caractère suivants peuvent également devenir des métacaractères :

**! < > = :**

## Les autres ancres

Les ancres \b et \B indiquent respectivement une limite de mot dans une chaîne de caractères et ce qui n'est pas une limite de mot.

**minet**

**"J'ai cru voir un Grosminet"; // VRAI**

**\bminet**

**"J'ai cru voir un Grosminet"; // FAUX**

**\Bminet**

**"J'ai cru voir un Grosminet"; // VRAI**

D'autres ancres existent, pour les débuts et fin de ligne, de chaîne ou de mot.

# Les fonctions PCRE

Comme nous l'avons indiqué dans l'introduction, PHP implémente la syntaxe PCRE. Les différentes fonctions PCRE de PHP sont décrites dans la [documentation](http://php.net/manual/fr/ref.pcre.php) de façon exhaustive. Nous nous limiterons ici à quelques cas.

## Exemples d'utilisation de quelques fonctions

### preg\_match($motif, $sujet)

Les exemples ci-dessus fonctionnent tous avec les paramètres requis de preg\_match().

### preg\_match($motif, $sujet, $res)

Le tableau $res contient le premier résultat de la recherche.

### preg\_match\_all($motif, $sujet, $resultats)

Le tableau $res contient tous les résultats de la recherche :

**$motif = "#pers.n#";**

**$sujet = "chat persan, persin, pers\_n";**

**preg\_match\_all($motif, $sujet, $res);**

**[0] => persan**

**[1] => persin**

**[2] => pers\_n**

Le drapeau (*flag*) PREG\_OFFSET\_CAPTURE ajoute les positions (*offsets*) des résultats dans la chaîne :

**$motif = "#pers.n#";**

**$sujet = "chat persan, persin, pers\_n";**

**preg\_match\_all($motif, $sujet, $res, PREG\_OFFSET\_CAPTURE);**

**[0] => persan**

**[1] => 5**

**[0] => persin**

**[1] => 13**

**[0] => pers\_n**

**[1] => 21**

### preg\_split($motif, $sujet)

Eclate une chaîne en sous-chaînes selon un motif et renvoie celles-ci sous forme de tableau.

**$motif = "#[,;]#";**

**$sujet = "persan;persin,pers\_n";**

**preg\_split($motif, $sujet);**

**[0] => persan**

**[1] => persin**

**[2] => pers\_n**

### preg\_replace($motif, $rempl, $sujet)

Recherche et remplace en fonction d'un motif. Les parenthèses capturent le sous-ensemble de caractères à remplacer.

**$motif = "#(\w+)$#"; // dernier mot du sujet**

**$sujet = "chat persan gris";**

**preg\_replace($motif, "noir", $sujet); // "chat persan noir"**

## Les options PCRE

Certaines options peuvent faciliter l'élaboration des motifs. Il est possible, par exemple, de rendre les masques insensibles à la casse. C'est une des options les plus utilisées.

**#chat persan# // les # sont des délimiteurs (PHP)**

**"chat Persan"; // FAUX**

**#chat persan#i**

**"chat Persan"; // VRAI**

# La capture

Comme nous l'avons déjà indiqué plus haut, les parenthèses servent à circonscrire la portée d'un masque. Elles peuvent également être **capturantes** (sauf lorsqu'on les utilise dans une assertion)

Dans l'exemple qui va suivre, on cherche à remplacer les chaînes de caractères comprises entre les § par des chaînes en gras :

**§persan§ ce §chat§ ⇒ <b>persan</b> ce <b>chat</b>**

Cette regex trouve toutes les occurrences qui semblent correspondre :

**$motif = "#§.+§#";**

**$sujet = "C'est un §persan§ ce chat";**

**preg\_match\_all($motif, $sujet, $res);**

**[0] => §persan§**

Mais en rajoutant des §, ça se complique :

**$motif = "#§.+§#";**

**$sujet = "C'est un §persan§ ce §chat§";**

**preg\_match\_all($motif, $sujet, $res);**

**[0] => §persan§ ce §chat§**

En effet, le point représente n'importe quel caractère, même le §. Excluons donc le § des caractères possibles :

**$motif = "#§[^§]+§#";**

**$sujet = "C'est un §persan§ ce §chat§";**

**preg\_match\_all($motif, $sujet, $res);**

**[0] => §persan§**

**[1] => §chat§**

C'est mieux, mais on souhaite avoir uniquement les caractères contenus entre les §. Les **parenthèses capturantes** vont nous servir à cela :

**$motif = "#§([^§]+)§#";**

**$sujet = "C'est un §persan§ ce §chat§";**

**preg\_match\_all($motif, $sujet, $res);**

**[0]**

**[0] => §persan§**

**[1] => §chat§**

**[1]**

**[0] => persan**

**[1] => chat**

Le tableau contient maintenant deux tableaux de résultats : en effet, l'expression évalue d'abord la chaîne sans les parenthèses, puis avec les parenthèses. Il y a autant de tableaux que de parenthèses imbriquées + 1.

Maintenant que nous sommes sûrs de notre expression, utilisons la fonction preg\_replace pour remplacer les § par des balises <b> :

**$motif = "#§([^§]+)§#";**

**$sujet = "C'est un §persan§ ce §chat§";**

**$rempl = "<b>$1</b>";**

**echo preg\_replace($motif, $rempl, $sujet);**

**// C'est un <b>persan</b> ce <b>chat</b>**

La fonction place chaque sous-ensemble capturé par les parenthèses dans une variable nommée $1 et effectue le remplacement dans la chaîne.

Voici un autre exemple dans lequel on cherche à formater une date placée en fin de chaîne :

**Mon chat est né en novembre 2017 ⇒**

**Mon chat est né en <b>novembre</b> <i>2017</i>**

**$motif = "#([[:alpha:]]+) ([[:digit:]]+)$#";**

**$sujet = "Mon chat est né en novembre 2017";**

**$rempl = "<b>$1</b> <i>$2</i>";**

**echo preg\_replace($motif, $rempl, $sujet);**

**// Mon chat est né en <b>novembre</b> <i>2017</i>**

La fonction place chaque sous-ensemble capturé par les parenthèses dans des variables nommées $1 et $2 (il y a deux groupes de parenthèses) et effectue les remplacements dans la chaîne.

# Les assertions

Les **assertions** sont des pointeurs qui permettent de vérifier une position. En fait, nous avons déjà travaillé avec des assertions, lorsque nous avons abordé les ancres. Les ancres sont en effet des assertions simples. Voici quelles sont les autres assertions :

|  |  |  |
| --- | --- | --- |
| **Types d'assertion** | **Motif** | **Succès si le motif dans l'assertion...** |
| Les assertions arrières positives | **(?<=motif)** | ...trouve une concordance à gauche |
| Les assertions arrières négatives | **(?<!motif)** | ...ne trouve pas de concordance à gauche |
| Les assertions avant positives | **(?=motif)** | ...trouve une concordance à droite |
| Les assertions avant négatives | **(?!motif)** | ...ne trouve pas de concordance à droite |

Dans l'exemple suivant, on cherche à recueillir, dans un texte, tous les nombres qui sont suivis de "an" ou "ans". L'expression ci-dessous, sans assertion, résout le problème : les parenthèses capturent tous les résultats qui correspondent au motif. On obtient également les résultats globaux, "1 an", "2 ans" etc.

**$motif = "#([0-9]+) ans?#";**

**$sujet = "Mes chats ont 1 an, 2 ans et 8 ans.";**

**preg\_match\_all($motif, $sujet, $res);**

**[0]**

**[0] => 1 an**

**[1] => 2 ans**

**[2] => 8 ans**

**[1]**

**[0] => 1**

**[1] => 2**

**[2] => 8**

Observons maintenant cette autre expression, qui met en œuvre une assertion avant positive : l'expression vérifie quelles sont les nombres positionnés avant "an" ou "ans" et renvoie les résultats.

**$motif = "#[0-9]+ (?=ans?)#";**

**$sujet = "Mes chats ont 1 an, 2 ans et 8 ans.";**

**preg\_match\_all($motif, $sujet, $res);**

**[0]**

**[0] => 1**

**[1] => 2**

**[2] => 8**

Quel est l'intérêt d'une telle approche ? **Les assertions sont moins gourmandes en ressources**, car elles ne "consomment" pas de caractères. Les assertions présentent un autre avantage, celui de définir des **conditions**.

Dans l'exemple suivant, on cherche à vérifier si un mot de passe contient au moins 8 caractères dont un chiffre mais pas placé au début :

Ecrivons les motifs qui vérifient les différentes contraintes :

**.{8,} // au moins 8 caractères**

**^[^0-9].\*\d // un chiffre mais pas au début**

Sans assertion, l'écriture du motif risque d'être malaisée… utilisons donc les assertions :

**(?=.{8,})(?=^[^0-9].\*\d)**

Ce motif signifie : VRAI s'il y a au moins 8 caractères, VRAI si la chaîne ne commence pas par un chiffre mais en contient, VRAI ET VRAI = VRAI

**"chat.persan72"; // VRAI**

**"chat.persan"; // FAUX**

**"chat72p"; // FAUX**

**"72chat\_persan"; // FAUX**

# Fin provisoire...

Ce cours est un bref aperçu du potentiel des expressions rationnelles. Pour un traitement exhaustif de cette question, consultez des ouvrages spécialisés comme *Regular Expressions Cookbook*, de Jay Goyvaerts et Steven Levithan (v. ci-dessous).

# Ressources

Un cours en ligne déjà ancien, mais plutôt bien fait : [http://expreg.com](http://expreg.com/)

Un livre PDF en anglais : [Regular Expressions Cookbook](https://karczmarczuk.users.greyc.fr/TEACH/TAL/Doc/Regular%20Expressions.pdf)

Pour les amateurs de mots croisés : [https://regexcrossword.com](https://regexcrossword.com/)

Un pense-bête : <http://www.cbs.dtu.dk/courses/27610/regular-expressions-cheat-sheet-v2.pdf>