perlintro

perlintro

NAME

perlintro - eine kurze Einführung und Überblick über Perl

BESCHREIBUNG

Dieses Dokument soll Ihnen einen schnellen Überblick über die Programmiersprache Perl zu geben, zusammen mit Verweisen auf weitere Dokumentation. Es wird als "Bootstrap" Leitfaden für diejenigen, die auf die Sprache neu gedacht und bietet gerade genug Informationen für Sie in der Lage zu sein, andere Völker Perl zu lesen und zu verstehen, grob, was es tut, oder Ihre eigene einfache Skripte schreiben.

Dieses einleitende Dokument zielt nicht auf Vollständigkeit. Es ist nicht einmal zielen ganz genau zu sein. In einigen Fällen über im Ziel des Erhaltens der allgemeinen Idee geopfert Perfektion ist. Sie sind *stark* diese Einführung empfohlen , von der vollständigen Perl Handbuch mit weiteren Informationen zu folgen, um die Tabelle der Inhalte , die in gefunden werden kann perltoc .

In diesem Dokument finden Sie Verweise auf andere Teile der Perl - Dokumentation. Sie können diese Dokumentation zu lesen , die unter Verwendung von `perldoc` Befehl oder was auch immer Methode , die Sie dieses Dokument zu lesen verwenden.

Während der Perl-Dokumentation finden Sie zahlreiche Beispiele sollen helfen, erklären die diskutierten Merkmale finden. Bitte beachten Sie, dass viele von ihnen Codefragmente sind anstatt komplette Programme.

Diese Beispiele spiegeln oft den Stil und die Vorlieben des Autors dieses Stück der Dokumentation und kann kürzer als eine entsprechende Codezeile in einem realen Programm. Sofern nicht anders vermerkt, sollten Sie davon ausgehen , dass `verwenden strenge` und `verwenden Warnungen` Aussagen früher im "Programm" angezeigt wird , und dass alle Variablen verwendet haben bereits erklärt worden, auch wenn diese Erklärungen weggelassen wurden das Beispiel leichter zu lesen.

Beachten Sie, dass die Beispiele wurden von vielen verschiedenen Autoren über einen Zeitraum von mehreren Jahrzehnten geschrieben worden. Stile und Techniken unterscheiden sich daher, auch wenn einige Anstrengungen unternommen wurden, um Stile nicht zu weit in den gleichen Abschnitten variieren. Achten Sie nicht auf einen Stil, besser als andere zu sein - "Es gibt mehr als einen Weg, es zu tun" eine von Perl Mottos ist. Immerhin als Programmierer in Ihrer Reise, werden Sie wahrscheinlich verschiedene Stile zu begegnen.

Was ist Perl?

Perl ist eine Programmiersprache für allgemeine Zwecke ursprünglich für Textmanipulation entwickelt wurde und nun für eine Vielzahl von Aufgaben, einschließlich der Systemverwaltung, Web-Entwicklung, Netzwerkprogrammierung, GUI-Entwicklung und vieles mehr verwendet.

Die Sprache soll praktisch sein (leicht zu bedienen, effizient, komplett) eher als schön (winzige, elegant, minimal). Seine wichtigsten Merkmale sind, dass es einfach zu bedienen, unterstützt sowohl prozedurale und objektorientierte (OO) Programmierung, verfügt über leistungsfähige integrierte Unterstützung für die Textverarbeitung, und verfügt über eine der weltweit beeindruckendsten Sammlungen von Modulen von Drittanbietern.

Unterschiedliche Definitionen von Perl finden sich in Perl , perlfaq1 und ohne Zweifel an anderen Orten. Daraus können wir feststellen , dass Perl verschiedene Dinge für verschiedene Menschen, aber , dass viele Leute denken , dass es zumindest wert ist das Schreiben über.

Ausführen von Perl-Programmen

Um ein Perl-Programm aus der Unix-Kommandozeile ausführen:

```
1.  perl  Progname . pl
```

Alternativ dazu können Sie diese als erste Zeile des Skripts:

```
1.  #! / Usr / bin / env perl
```

... Und das Skript als laufen */path/to/script.pl* . Natürlich müssen sie ausführbar erste, so zu sein `chmod 755 script.pl` (unter Unix).

(Dieser Startlinie vorausgesetzt , dass Sie das haben **env** - Programm. Sie können auch direkt den Pfad zu Ihrem Perl ausführbar setzen, wie in `#! / Usr / bin / perl`).

Für weitere Informationen, Anweisungen für andere Plattformen wie Windows und Mac OS, lesen perlrun .

Sicherheitsnetz

Perl standardmäßig ist sehr nachsichtig. Um ist es robuster zu machen es empfehlenswert, jedes Programm mit den folgenden Zeilen zu starten:

```
1.  #! / Usr / bin / perl
2.  verwenden  strenge ;
3.  verwenden  Warnungen ;
```

Die beiden zusätzlichen Linien fordern von Perl verschiedene häufige Probleme in Ihrem Code zu fangen. Sie prüfen verschiedene Dinge , so dass Sie beide brauchen. Ein mögliches Problem durch gefangen `Verwendung streng ,` wird Ihr Code sofort zu stoppen , wenn es auftritt, während `Verwendung Warnungen ,` wird lediglich eine Warnung (wie die Befehlszeilenoption **-w**) und der Code laufen lassen. Um mehr zu lesen über sie ihre jeweiligen Handbuchseiten zu überprüfen streng und Warnungen .

Basic-Syntax Überblick

Ein Perl - Skript oder Programm besteht aus einer oder mehreren Anweisungen. Diese Aussagen werden einfach in das Skript auf einfache Art und Weise geschrieben. Es gibt keine Notwendigkeit , einen zu haben , `Haupt ( )` Funktion oder irgendetwas dieser Art.

Perl-Anweisungen enden in einem Semikolon:

```
1.  drucken  "Hallo, Welt" ;
```

Kommentare mit einem Hash-Symbol beginnen und bis zum Ende der Leitung zu verlegen

```
1.  # Dies ist ein Kommentar
```

Whitespace ist irrelevant:

```
1.  drucken
2.      "Hallo Welt"
3.      ;
```

... Außer in Strings in Anführungszeichen:

```
1.  # Würde dies mit einem Zeilenumbruch in der Mitte drucken
2.  drucken  "Hallo
3.  Welt " ;
```

Doppelte Anführungszeichen oder einfache Anführungszeichen können um Literalzeichenfolgen verwendet werden:

```
1.  drucken  "Hallo, Welt" ;
2.  Print  "Hallo, Welt ' ;
```

Allerdings nur doppelte Anführungszeichen "interpolieren" Variablen und Sonderzeichen wie Zeilenumbrüche (`\ n`):

```
1.  Druck :  "Hallo, $ name \ n" ;      # funktioniert gut
2.  Druck  'Hallo, $ name \ n' ;      # druckt $ name \ n wahrsten Sinne des Wortes
```

Zahlen müssen keine Anführungszeichen um sie herum:

```
1.  drucken  42 ;
```

Sie können sie Klammern Argumente 'Funktionen verwenden oder weglassen Ihrem persönlichen Geschmack entsprechend. Sie werden nur gelegentlich zu klären Fragen der Vorrang erforderlich.

```
1.  Druck ( "Hallo, Welt \ n" ) ;
```

```
2.  drucken  "Hallo, Welt \ n" ;
```

Detailliertere Informationen über Perl - Syntax finden Sie in perlsyn .

Perl-Variablen-Typen

Perl hat drei Hauptvariablentypen: Skalare, Arrays und Hashes.

- **Skalare**

  Ein Skalar repräsentiert einen einzelnen Wert:

```
1.  mein  $ Tier = "Kamel" ;
2.  meine  Antwort $ = 42 ;
```

  Scalar Werte können Strings, Integer oder Fließkommazahlen sein, und Perl wird zwischen ihnen automatisch konvertieren je nach Bedarf. Es gibt keine Notwendigkeit, Ihre Variablentypen im Voraus zu erklären, aber man muss sie zu erklären , die mit meinem Keyword das erste Mal , wenn Sie sie verwenden. (Dies ist eine der Voraussetzungen der Verwendung streng ; .)

  Skalare Werte können auf verschiedene Arten verwendet werden:

```
1.  Druck  $ Tier ;
2.  drucken  "Das Tier ist $ Tier \ n" ;
3.  Druck  "Der Platz von $ Antwort ist" ,  $ Antwort * $ Antwort ,  "\ n" ;
```

  Es gibt eine Reihe von "Magie" Skalare mit Namen , die wie Zeichensetzung oder Leitungsrauschen aussehen. Diese speziellen Variablen werden für alle Arten von Zwecken verwendet werden, und werden in dokumentiert perlvar . Der einzige , der du jetzt wissen müssen für ist _ $, die der "Standardgröße" ist. Es ist , als das Standardargument für eine Reihe von Funktionen in Perl, und es wird implizit gesetzt durch bestimmte looping Konstrukte verwendet.

```
1.  Druck ;           # druckt Inhalt von $ _ standardmäßig
```

- **Arrays**

  Ein Array stellt eine Liste von Werten:

```
1.  mein  @animals = ( "Kamel" ,  "Lama" ,  "Eule" ) ;
2.  my  @zahlen = ( 23 ,  42 ,  69 ) ;
3.  mein  @mixed    = ( "Kamel" ,  42 ,  1,23 ) ;
```

  Arrays sind nullindizierte. Hier ist, wie Sie auf Elemente in einem Array erhalten:

```
1.  Druck  $ Tiere [ 0 ] ;              # prints "Kamel"
2.  Druck  $ Tiere [ 1 ] ;              # prints "Lama"
```

  Die spezielle Variable $ # Array zeigt Ihnen den Index des letzten Elements eines Arrays:

```
1.  drucken  $ gemischt [ $ # gemischt ] ;        # letzte Element, Drucke 1,23
```

  Sie könnten versucht sein , zu verwenden $ # array + 1 , Ihnen zu sagen , wie viele Artikel gibt es in einer Reihe sind. Kümmern Sie sich nicht. Wie es passiert, mit @array wo Perl einen skalaren Wert ("in Skalarkontext") , die Sie in dem Array die Anzahl der Elemente geben zu finden erwartet:

```
1.  wenn  ( @animals < 5 )  { ... }
```

  Die Elemente wir aus dem Array immer mit einem Start $, weil wir nur einen einzigen Wert aus dem Array , wenn es darum; Sie bitten um einen Skalar, ein Skalar bekommen.

  Um mehrere Werte aus einem Array erhalten:

```
1.  @animals [ 0 , 1 ] ;                 # gibt ( "Kamel", "Lama");
2.  @animals [ 0 .. 2 ] ;                # gibt ( "Kamel", "Lama", "Eule");
3.  @animals [ 1 .. $ # Tiere ] ;        # gibt alle außer dem ersten Element
```

  Dies ist ein "Array-Slice" genannt.

Sie können verschiedene nützliche Dinge zu tun Listen:

```
1.  mein  @sorted     = sort  @animals ;
2.  mein  @backwards = Reverse  @zahlen ;
```

Es gibt ein paar spezielle Arrays zu, wie `@ARGV` (die Befehlszeilenargumente an das Skript) und `@_` (die an ein Unterprogramm übergebenen Argumente). Diese werden in dokumentiert perlvar .

- **Hashes**

  Ein Hash stellt einen Satz von Schlüssel / Wert-Paare:

```
1.  meine  % fruit_color = ( "Apfel" ,  "rot" ,  "Banane" ,  "gelb" ) ;
```

  Sie können Leerzeichen verwenden und die `=>` Operator sie mehr schön zu legen:

```
1.  meine  % fruit_color = (
2.      Apfel   =>  "rot" ,
3.      Banane  =>  "gelb" ,
4.  ) ;
```

  Um bei Hash-Elemente erhalten:

```
1.  $ fruit_color { "apple" } ;            # gibt "rot"
```

  Sie können Listen von Schlüssel und Werte mit bekommen `Tasten ()` und `Werte ()` .

```
1.  meine  @fruits = Tasten  % fruit_colors ;
2.  meine  @colors = Werte  % fruit_colors ;
```

  Hashes haben keine besondere innere Ordnung, wenn Sie die Schlüssel und Schleife durch sie zu sortieren.

  Genau wie spezielle Skalare und Arrays, gibt es auch spezielle Hashes. Der bekannteste von ihnen ist `% ENV` die Umgebungsvariablen enthält. Lesen Sie alles über sie (und andere spezielle Variablen) in perlvar .

Skalare, Arrays und Hashes sind ausführlicher in dokumentiert perldata .

Komplexere Datentypen Referenzen werden können, konstruiert, das können Sie Listen und Hashes innerhalb von Listen und Hashes zu bauen.

Eine Referenz ist ein skalarer Wert und für alle anderen Perl-Datentyp beziehen. So durch einen Verweis als Wert eines Arrays oder Hash-Element zu speichern, können Sie einfach Listen und Hashes innerhalb von Listen und Hashes erstellen. Das folgende Beispiel zeigt einen 2-Ebene Hash-Hash-Struktur mit einem anonymen Hashreferenzen.

```
1.  meine  $ Variablen = {
2.      Skalar   =>   {
3.                    Beschreibung  =>  "Einzelposten" ,
4.                    Sigil  =>  '$' ,
5.                   } ,
6.      Array    =>   {
7.                    Beschreibung  =>  "geordnete Liste der Elemente" ,
8.                    Sigil  =>  '@' ,
9.                   } ,
10.     Hash     =>   {
11.                   Beschreibung  =>  "Schlüssel / Wert - Paare" ,
12.                   Sigil  =>  '%' ,
13.                  } ,
14. } ;
15.
16.  Druck  "Skalare beginnen mit $ Variablen -> { 'Skalar'} -> { 'Sigil'} \ n" ;
```

Ausgiebige Informationen zum Thema Referenzen finden Sie in perlreftut , perllol , perlref und perldsc .

Variable Scoping

Im Laufe des vorigen Abschnitt haben alle die Beispiele, die die Syntax verwendet:

```
1.  mein  $ var = "value" ;
```

Der `meine` ist eigentlich nicht erforderlich; verwenden Sie könnten nur:

```
1.  $ var = "value" ;
```

Allerdings wird die obige Verwendung von globalen Variablen in Ihrem Programm zu erstellen, die Praxis schlechte Programmierung ist. Meine schafft lexikalisch statt scoped Variablen. Die Variablen werden auf den Block scoped (dh ein Bündel von Erklärungen von geschweiften Klammern umgeben) , in dem sie definiert sind.

```
1.  mein  $ x = "foo" ;
2.  mein  $ some_condition = 1 ;
3.  wenn  ( $ some_condition )  {
4.      mein  $ y = "bar" ;
5.      Druck  $ x ;            # prints "foo"
6.      Druck  $ y ;            # prints "bar"
7.  }
8.  Druck  $ x ;                # prints "foo"
9.  Druck  $ y ;                # druckt nichts; $ hat y out of scope gefallen
```

Mit meiner in Kombination mit einem Einsatz streng , an der Spitze Ihrer Perl - Skripten bedeutet , dass der Interpreter bestimmte Fehler gemeinsame Programmierung abholen. Zum Beispiel, in dem obigen Beispiel, die endgültige Druck $ y verursachen würde einen Compiler-Fehler und verhindern , dass Sie das Programm ausgeführt wird . Mit strengen wird dringend empfohlen.

Bedingte und Schleifenkonstrukte

Perl hat die meisten der üblichen bedingten und Schleifenkonstrukte. Ab Perl 5.10, hat es auch einen Fall / switch - Anweisung (Dinkel gegeben / wenn). Siehe Switch - Anweisungen in perlsyn für weitere Details.

Die Bedingungen können beliebige Perl Ausdruck sein. Siehe die Liste der Operatoren im nächsten Abschnitt für Informationen über den Vergleich und die Booleschen logischen Operatoren, die in bedingten Anweisungen häufig verwendet werden.

- **ob**

  ```
  1.  wenn  (  Bedingung  )  {
  2.      ...
  3.  }  Elsif  (  andere  Bedingung  )  {
  4.      ...
  5.  }  Else  {
  6.      ...
  7.  }
  ```

  Es gibt auch eine negierte Version davon:

  ```
  1.  es sei denn ,  (  Bedingung  )  {
  2.      ...
  3.  }
  ```

  Dies wird als lesbarer Version bereitgestellt if (! *Zustand*) .

  Beachten Sie, dass die Klammern in Perl erforderlich sind, auch wenn Sie nur eine Zeile in dem Block haben. Allerdings gibt es ein kluger Weg, um Ihre einzeiligen bedingte Blöcke mehr Englisch zu machen, wie:

  ```
  1.  # Die traditionelle Art und Weise
  2.  wenn  ( $ zippy )  {
  3.      drucken  "Yow!" ;
  4.  }
  5.
  6.  # Die Perlish post-Zustand Weg
  7.  drucken  "Yow!" ,  wenn  $ flink ;
  8.  drucken  "Wir haben keine Bananen"  es sei denn ,  $ Bananen ;
  ```

- **während**

  ```
  1.  während  (  Bedingung  )  {
  2.      ...
  3.  }
  ```

  Es gibt auch eine negierte Version, aus dem gleichen Grund haben wir , es sei denn :

  ```
  1.  bis  (  Bedingung  )  {
  2.      ...
  3.  }
  ```

Sie können auch verwendet werden, `während` in einer post-Zustand:

```
drucken "LA LA LA \ n" während 1 ;           # immer Schleifen
```

- **für**

  Genau wie C:

  ```
  für ( $ i = 0 ;  $ i <= $ max ;  $ i ++ )  {
      ...
  }
  ```

  Die C - Stil - for - Schleife ist selten in Perl benötigt , da Perl die freundlichere Liste Scanning bietet `foreach` - Schleife.

- **für jede**

  ```
  foreach ( @array )  {
      drucken "Dieses Element $ _ \ n" ;
  }

  Druck $ list [ $ _ ] foreach 0 .. $ max ;

  # Sie müssen nicht den Standard verwenden $ _ entweder ...
  foreach my $ key  ( Schlüssel % hash )  {
      Druck "Der Wert von $ key ist $ hash {$ key} \ n" ;
  }
  ```

  Die `foreach` Schlüsselwort ist eigentlich ein Synonym für die `für` Schlüsselwort. Siehe `foreach - Schleifen in perlsyn` .

Für weitere Einzelheiten über Schleifenkonstrukte (und einige , die in dieser Übersicht nicht erwähnt wurden) sehen perlsyn .

Builtin Operatoren und Funktionen

Perl kommt mit einer breiten Auswahl an eingebauten Funktionen. Einige von denen , haben wir bereits gesehen , sind `Druck` , `Art` und `umgekehrt` . Eine Liste von ihnen wird zu Beginn des gegebenen perlfunc und Sie können mit einem beliebigen gegebenen Funktion leicht zu lesen über `perldoc -f` *`functionName-`* .

Perl Operatoren werden in voller Höhe in dokumentiert perlop , aber hier sind ein paar der häufigsten sind:

- **Arithmetik**

  ```
  + zusätzlich
  - Subtraktion
  * Multiplikation
  / Division
  ```

- **Numerische Vergleich**

  ```
  == Gleichheit
  ! = Ungleichheit
  < Weniger als
  > Größer als
  <= Kleiner als oder gleich
  > = Größer als oder gleich
  ```

- **Stringvergleich**

  ```
  eq Gleichheit
  ne Ungleichheit
  lt weniger als
  gt größer als
  le weniger als oder gleich
  ge größer als oder gleich
  ```

  (Warum müssen wir getrennte numerische und String-Vergleiche? Weil wir Variablen-Typen haben keine speziellen und Perl muss wissen, ob numerisch zu sortieren (wo 99 weniger als 100) oder alphabetisch (wo 100 vor 99 kommt).

- **Boolesche Logik**

```
1. &&   und
2. ||   oder
3. !    nicht
```

(Und , oder und nicht nicht nur in der obigen Tabelle als Beschreibungen der Operatoren. Sie sind auch als Betreiber in ihrem eigenen Recht unterstützt. Sie sind besser lesbar als die C-Stil Operatoren, haben aber unterschiedliche Vorrang && und Freunde. Überprüfen Sie perlop für weitere Details.)

- **Sonstiges**

```
1. =    Zuordnung
2. .    String -  Verkettung
3.  x    String  Multiplikation
4. ..   Bereich  Operator  ( erstellt  eine  Liste  von  Zahlen  oder  Strings )
```

Viele Betreiber kann mit einem kombinierbar = wie folgt:

```
1.  $ a + = 1 ;         # gleiche wie $ a = $ a + 1
2.  $ a - = 1 ;         # gleiche wie $ a = $ - 1
3.  $ a =. "\ n" ;      # gleiche wie $ a = $ a. "\ n";
```

Dateien und E / A

Sie können eine Datei als Eingang oder Ausgang öffnen Sie die Verwendung von open () Funktion. Es ist in extravaganten Detail in dokumentiert perlfunc und perlopentut , aber kurz:

```
1.  offen ( mein $ in ,   "<" ,   "input.txt" )   oder sterben : "Can not open input.txt $!" ;
2.  offen ( meine $ aus ,  ">" ,   "output.txt" )  oder sterben : "nicht output.txt Kann öffnen $!" ;
3.  offen ( mein $ log ,  ">>" ,  "My.Log" )      oder sterben "kann nicht geöffnet werden My.Log: $!" ;
```

Sie können aus einem offenen Dateihandle mit dem Lesen <> Operator. Im skalaren Kontext liest es nur eine einzige Zeile aus dem Dateihandle, und im Listenkontext liest er die ganze Datei in jede Zeile zu einem Element der Liste zuweisen:

```
1.  meine $ line   = <$ in> ;
2.  mein @lines = <$ in> ;
```

Lesen in der gesamten Datei auf einmal heißt schlürfen. Es kann sinnvoll sein, aber es kann ein Speicher Schwein sein. Die meisten Textdatei Verarbeitung kann eine Zeile zu einem Zeitpunkt mit Perl Schleifenkonstrukte erfolgen.

Der <> Operator wird am häufigsten in einem gesehen , während Schleife:

```
1.  während ( <$ in> )  {      # ordnet jede Zeile wiederum zu $ _
2.      Druck "Just in dieser Zeile zu lesen: $ _" ;
3.  }
```

Wir haben bereits gesehen , wie die Standardausgabe zu drucken mit print () . Allerdings print () kann auch ein optionales erstes Argument Spezifizierungs nehmen, die zu drucken Dateikennung:

```
1.  drucken STDERR  "Dies ist Ihre letzte Warnung . \ n" ;
2.  drucken $ aus  $ record ;
3.  Druck $ log  $ LogMessage ;
```

Wenn Sie mit Ihrem Dateihandies fertig sind, sollten Sie schließen () sie (wenn auch ehrlich zu sein, wird Perl , nachdem Sie reinigen , wenn Sie vergessen):

```
1.  schließen $ in  oder sterben "$ in: $!" ;
```

Reguläre Ausdrücke

Perl-Unterstützung für reguläre Ausdrücke ist sowohl breit und tief, und ist Gegenstand von umfangreichen Dokumenten in perlrequick , perlretut und anderswo. Jedoch, kurz:

- **Einfache Matching**

```
1.  wenn ( / foo / )        { ... }   # true , wenn $ _ enthält "foo"
2.  wenn ( $ a = ~ / foo / )  { ... }   # true , wenn $ a enthält "foo"
```

Der // Anpassungsoperator ist in dokumentiert perlop . Es arbeitet auf $ _ standardmäßig oder auf eine andere Variable gebunden werden , um die Verwendung von = ~ Bindung Operator (auch in dokumentiert perlop).

- **Einfache Substitution**

```
s / foo / bar / ;                # ersetzt foo mit Bar in $ _
$ a = ~ s / foo / bar / ;          # ersetzt foo mit Bar in $ a
$ a = ~ s / foo / bar / g ;         # alle Instanzen von foo mit Bar ersetzt
                           # In $ a
```

Der s /// Substitution Operator wird dokumentiert perlop .

- **Komplexere reguläre Ausdrücke**

Sie müssen nicht nur auf festen Strings entsprechen. In der Tat können Sie auf einfach alles passen Sie mithilfe komplexer regulären Ausdrücken träumen konnte. Diese sind sehr ausführlich in dokumentiert perlre , aber für die Zwischenzeit, hier ist eine schnelle Spickzettel:

```
.                    Ein  einzelnes  Zeichen
\ Sa Leerzeichen  ( Leerzeichen ,  Tab ,  Newline ,
                     ... )
\ S                   nicht - Leerzeichen  Zeichen
\ D                   eine  Ziffer  ( 0 - 9 )
\ D                   eine  nicht - Ziffer
\ W                   ein  Wort  Zeichen  ( a - z ,  A - Z ,  0 - 9 ,  _ )
\ W                   ein  nicht - Wort  Zeichen
 [ Aeiou ]              entspricht  einem  einzelnen  Zeichen  in  der  gegebenen  Satz
 [ ^ Aeiou ]             entspricht  einem  einzelnen  Zeichen  außerhalb  der  gegebenen
                     Set
 ( Foo | bar | baz )        entspricht  jeder  von  den  Alternativen  angegeben

^                    Anfang  der  Zeichenfolge
 $ Ende  der  Zeichenfolge
```

Quantifiers kann verwendet werden, um festzulegen, wie viele der vorherigen, was Sie auf zu passen wollen, wo "Ding" bedeutet entweder eine wörtliche Charakter, einer der Meta-Zeichen oben aufgeführten oder eine Gruppe von Zeichen oder Metazeichen in Klammern.

```
 * Null  oder  mehr  von  den  früheren  Sache
+                    Ein  oder  mehr  von  den  früheren  Sache
?                    Null  oder  eins  von  der  vorherigen  Sache
 { 3 }                  entspricht  genau  3  von  der  vorherigen  Sache
 { 3 , 6 }                passt  zwischen  3  und  6  von  der  vorherigen  Sache
 { 3 , }                 entspricht  3  oder  mehr  von  der  vorherigen  Sache
```

Einige kurze Beispiele:

```
/ ^ \ d + /               String  beginnt  mit  einem  oder  mehreren  Ziffern
/ ^ $ /                  Nichts  in  der  Zeichenfolge  ( Start  und  Ende  sind
                     benachbarte )
/ ( \ D \ s) {3} / drei Ziffern, die jeweils durch ein Leerzeichen folgt
                                          Zeichen (zB "3 4 5")
/ (a.) + / entspricht einem String , in dem jeder ungeraden
                                          Schreiben ist ein (zB "abacadaf")

 # Diese Schleife liest aus STDIN und druckt nicht leere Zeilen:
 while (<>) {
         neben if / ^ $ / ;
    Druck ;
 }
```

- **Klammern für die Erfassung**

Neben der Gruppierung dienen Klammern einen zweiten Zweck. Sie können die Ergebnisse der Teile der regexp Spiel zu erfassen , für den späteren Gebrauch verwendet werden. Die Ergebnisse am Ende in $ 1 , $ 2 und so weiter.

```
# A billige und schlechte Möglichkeit, eine E-Mail zu brechen Adresse in Teile

wenn  ( $ email = ~ /([^@]+)@(.+)/ )  {
    Druck  "Benutzername ist $ 1 \ n" ;
    Druck  "Hostname ist $ 2 \ n" ;
}
```

- **Andere regexp Funktionen**

  Perl regexps auch Rückreferenzierungen, Lookaheads, und alle Arten von anderen komplexen Details zu unterstützen. Lesen Sie alles über sie in perlrequick , perlretut und perlre .

Schreiben Subroutinen

Schreiben Subroutinen ist einfach:

```
1. Unter Logger  {
2.     mein  $ LogMessage = Verschiebung ;
3.     öffnen  meine  $ Logfile ,  ">>" ,  "My.Log"  oder  sterben  : "Could not open My.Log $!" ;
4.     Druck  $ Logfile  $ LogMessage ;
5.  }
```

Nun können wir das Unterprogramm wie jede andere integrierte Funktion verwenden:

```
1.  Logger ( "Wir haben einen Logger Unterprogramm!" ) ;
```

Was ist das `Shift` ? Nun, die Argumente zu einem Unterprogramm zur Verfügung stehen , um uns als ein besonderes Array namens `@_` (siehe perlvar für mehr dazu). Die Standard - Argument für die `Verschiebung` Funktion nun ausgerechnet `@_` . Also `meine $ LogMessage = Verschiebung ;` verschiebt sich das erste Element aus der Liste der Argumente und weist sie `$ LogMessage` .

Wir können manipulieren `@_` auch auf andere Weise:

```
1.  mein  ( $ LogMessage ,  $ Priorität ) = @_ ;        # gemeinsame
2.  mein  $ LogMessage = $ _ [ 0 ] ;                   # ungewöhnlich, und hässlich
```

Unterprogramme können auch Werte zurückgeben:

```
1. Unter Platz  {
2.      meine  $ num = Verschiebung ;
3.      mein  $ result = $ num * $ num ;
4.      Rückkehr  $ result ;
5.  }
```

Benutzen Sie diese dann wie:

```
1.  $ sq = Quadrat ( 8 ) ;
```

Weitere Informationen zum Schreiben von Subroutinen finden perlsub .

OO Perl

OO Perl ist relativ einfach und wird unter Verwendung von Referenzen implementiert , die wissen , welche Art von Objekt sind sie auf der Perl-Konzept der Pakete basieren. Jedoch ist OO Perl weitgehend über den Rahmen dieses Dokuments. Lesen Sie perlootut und perlobj .

Als Anfang Perl-Programmierer, Ihre häufigste Verwendung von OO Perl wird mit Modulen von Drittanbietern, der im Folgenden dokumentiert sind.

Mit Perl-Module

Perl - Module bieten eine Reihe von Funktionen , die Sie vermeiden zu helfen , das Rad neu zu erfinden, und kann von CPAN (heruntergeladen werden http://www.cpan.org/). Eine Reihe von beliebten Module sind mit der Perl - Distribution selbst enthalten.

Kategorien von Modulen reichen von Textmanipulation zu Netzwerk-Protokolle zu Datenbank-Integration zu Grafiken. Eine kategorisierte Liste der Module ist auch von CPAN verfügbar.

Um zu erfahren , wie Module zu installieren , die Sie von CPAN herunterladen, lesen perlmodinstall .

Um zu erfahren , wie ein bestimmtes Modul zu verwenden, benutzen Sie `perldoc` *`Module :: Name`* des . Typischerweise werden Sie wollen `verwenden` *`Module :: Name`* des , die dann an exportierten Funktionen oder eine OO - Schnittstelle

geben Sie auf das Modul zugreifen.

perlfaq enthält Fragen und auf viele gemeinsame Aufgaben im Zusammenhang mit Antworten und liefert oft Anregungen für eine gute CPAN - Module zu verwenden.

perlmod beschreibt Perl - Module im Allgemeinen. perlmodlib die Module auflistet , die mit Ihrer Perl - Installation kam.

Wenn Sie den Drang verspüren , Perl - Module zu schreiben, perlnewmod geben Ihnen gute Ratschläge.

AUTOR

Kirrily "Skud" Robert <skud@cpan.org>

perldoc.perl.org - Offizielle Dokumentation für die Programmiersprache Perl

Kontaktdetails

Website verwaltet von Jon Allen (JJ)

Dokumentation durch die gepflegten 5 Porters Perl