Unique values in an array in Perl

unique uniq distinct filter grep array List::MoreUtils duplicate

Prev Next

In this part of the Perl tutorial we are going to see how to make sure we only have **distinct values in an array**.

Perl 5 does not have a built in function to filter out duplicate values from an array, but there are several solutions to the problem.

A small clarification

I am not a native English speaker, but it seems that at least in the world of computers, the word **unique** is a bit overloaded. As far as I can tell in most programming environments the word **unique** is a synonym of the word **distinct** and in this article we use that meaning. So given a list of values like this `foo, bar, baz, foo, zorg, baz` by **unique values**, we mean `foo, bar, baz, zorg`.

The other meaning would be "those values that appear only once" which would give us `bar, zorg`.

List::MoreUtils

Most of the time, the simplest way is to use the `uniq` function of the List::MoreUtils module from CPAN.

```
1. use List::MoreUtils qw(uniq);

   my @words = qw(foo bar baz foo zorg baz);
   my @unique_words = uniq @words;
```

A full example is this:

```
1. use strict;
   use warnings;
   use 5.010;

5. use List::MoreUtils qw(uniq);
   use Data::Dumper qw(Dumper);
```

```
my @words = qw(foo bar baz foo zorg baz);

my @unique_words = uniq @words;

say Dumper \@unique_words;
```

The result is:

```
$VAR1 = [
        'foo',
        'bar',
        'baz',
        'zorg'
      ];
```

For added fun the same module also provides a function called `distinct`, which is just an alias of the `uniq` function.

In order to use this module you'll have to install it from CPAN.

Home made uniq

If you cannot install the above module for whatever reason, or if you think the overhead of loading it is too big, there is a very short expression that will do the same:

```
my @unique = do { my %seen; grep { !$seen{$_}++ } @data };
```

This, of course can look cryptic to someone who does not know it already, so it is recommended to define your own `uniq` subroutine, and use that in the rest of the code:

```
use strict;
use warnings;
use 5.010;

use Data::Dumper qw(Dumper);

my @words = qw(foo bar baz foo zorg baz);

my @unique = uniq( @words );

say Dumper \@unique_words;

sub uniq {
```

```
  my %seen;
  return grep { !$seen{$_}++ } @_;
}
```

Home made uniq explained

I can't just throw this example here and leave it like that. I'd better explain it. Let's start with an easier version:

```
my @unique;
my %seen;

foreach my $value (@words) {
  if (! $seen{$value}) {
    push @unique, $value;
    $seen{$value} = 1;
  }
}
```

Here we are using a regular `foreach` loop to go over the values in the original array, one by one. We use a helper hash called `%seen`. The nice thing about the hashes is that their keys are **unique**.

We start with an empty hash so when we encounter the first "foo", `$seen{"foo"}` does not exist and thus its value is `undef` which is considered false in Perl. Meaning we have not seen this value yet. We push the value to the end of the new `@uniq` array where we are going to collect the distinct values.

We also set the value of `$seen{"foo"}` to 1. Actually any value would do as long as it is considered "true" by Perl.

The next time we encounter the same string, we already have that key in the `%seen` hash. Since its value is true, the `if` condition will fail, and we won't `push` the duplicate value in the resulting array.

Shortening the home made unique function

First of all we replace the assignment of 1 `$seen{$value} = 1;` by the post-increment operator `$seen{$value}++`. This does not change the behavior of the previous solution - any positive number is going to be evaluated as TRUE, but it will allow us to include the setting of the "seen flag" within the `if` condition. It is important that this is a postfix increment (and not a prefix increment) as this means the increment only takes place after the boolean expression was evaluated. The first time we encounter a value the expression will be TRUE and the rest of the times it will be FALSE.

```
my @unique;
my %seen;
```

```
    foreach my $value (@data) {
5.    if (! $seen{$value}++ ) {
        push @unique, $value;
      }
    }
```

This is shorter, but we can do even better.

Filtering duplicate values using grep

The `grep` function in Perl is a generalized form of the well known grep command of Unix.

It is basically a filter. You provide an array on the right hand side and an expression in the block. The `grep` function will take each value of the array one-by-one, put it in `$_`, the default scalar variable of Perl and then execute the block. If the block evaluates to TRUE, the value can pass. If the block evaluates to FALSE the current value is filtered out.

That's how we got to this expression:

```
1. my %seen;
   my @unique = grep { !$seen{$_}++ } @words;
```

Wrapping it in 'do' or in 'sub'

The last little thing we have to do, is wrapping the above two statements in either a `do` block

```
1. my @unique = do { my %seen; grep { !$seen{$_}++ } @words };
```

or, better yet, in a function with an expressive name:

```
1. sub uniq {
     my %seen;
     return grep { !$seen{$_}++ } @_;
   }
```

Home made uniq - round 2

Prakash Kailasa suggested an even shorted version of implementing uniq, for Perl version 5.14 and above, if there is no requirement to preserve the order of elements.

Inline:

```
1. my @unique = keys { map { $_ => 1 } @data };
```

or within a subroutine:

```
my @unique = uniq(@data);
sub uniq { keys { map { $_ => 1 } @_ } };
```

Let's take this expression apart:

`map` has a similar syntax to `grep`: a block and an array (or a list of values). It goes over all the elements of the array, executes the block and passes the result to the left.

In our case, for every value in the array it will pass the value itself followed by the number 1. Remember `=>`, a.k.a. fat comma, is just a comma. Assuming @data has ('a', 'b', 'a') in it, this expression will return ('a', 1, 'b', 1, 'a', 1).

```
map { $_ => 1 } @data
```

If we assigned that expression to a hash, we would get the original data as keys, each with value of the number 1. Try this:

```
use strict;
use warnings;

use Data::Dumper;

my @data = qw(a b a);
my %h = map { $_ => 1 } @data;
print Dumper \%h;
```

and you will get:

```
$VAR1 = {
          'a' => 1,
          'b' => 1
        };
```

If, instead of assigning it to an array, we wrap the above expression in curly braces, we will get a reference to an anonymous hash.

```
{ map { $_ => 1 } @data }
```

Let's see it in action:

```
use strict;
use warnings;

use Data::Dumper;
my @data = qw(a b a);
```

```
my $hr = { map { $_ => 1 } @data };
print Dumper $hr;
```

Will print the same output as the previous one, barring any change in order in the dumping of the hash.

Finally, starting from perl version 5.14, we can call the `keys` function on hash references as well. Thus we can write:

```
1. my @unique = keys { map { $_ => 1 } @data };
```

and we'll get back the unique values from `@data`

Unique values in an array reference

There is a separate article showing the above solution in case the data is in a reference to an array.

Exercise

Given the following file, print out the unique values:

input.txt:

```
foo Bar bar first second
Foo foo another foo
```

expected output:

```
foo Bar bar first second Foo another
```

Exercise 2

This time filter out duplicates regardless of case.

expected output:

```
foo Bar first second another
```