How to get the index of specific element (value) of an array?

List::MoreUtils | first_index | grep

Recently I received two very different questions, where the answer is quite similar:

- How can we get index of particular value(element) of array quickly?
- Given a list of input values in `@user_array` how can I find their reference in the `@all_values` array?

Index of a single element

Let's see an example for the first one:

Given an array

```
my @planets = qw(
    Mercury
    Venus
    Earth
    Mars
    Ceres
    Jupiter
    Saturn
    Uranus
    Neptune
    Pluto
    Charon
);
```

Given the value `Mars` we should return the number 3 because `$planets[3]` is `Mars`.

The quick answer is to use the `first_index` method of List::MoreUtils

```
use strict;
use warnings;
use 5.010;
use List::MoreUtils qw(first_index);
```

```
    my @planets = qw(
       Mercury
       Venus
       Earth
       Mars
       Ceres
       Jupiter
       Saturn
       Uranus
       Neptune
       Pluto
       Charon
    );

say first_index { $_ eq 'Mars' } @planets;
```

Get list of indexes

Given the above array and another array listing elements we would like to return the list of the indexes of these elements:

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

my @planets = qw(
   Mercury
   Venus
   Earth
   Mars
   Ceres
   Jupiter
   Saturn
   Uranus
   Neptune
   Pluto
   Charon
);

use List::MoreUtils qw(first_index);

my @input = qw(Mars Pluto Alderaan Venus);
```

```
    my @indexes;
    foreach my $place (@input) {
25.     push @indexes, first_index { $_ eq $place } @planets;
    }

    use Data::Dumper qw(Dumper);
    print Dumper \@indexes;
```

In this code we loop over all the planet names we would like to locate in the array of planets, and for each one of them we call the `first_index` function as we did in the first example. We then use the Data::Dumpermodule to print out the results.

```
$VAR1 = [
          3,
          9,
          -1,
          1
        ];
```

As you can see the 3rd element of the resulting array is -1. That's because we had Alderaan, in the list, a planet that is was not in the original list of planets.

Implementing manually - index of one element

Though you are probably better off using the `first_index` function from List::MoreUtils, it might be interesting to see how we could solve the above questions if we did not have that module available.

The code with List::MoreUtils:

```
1. use List::MoreUtils qw(first_index);
   say first_index { $_ eq 'Mars' } @planets;
```

The code with core perl functions:

```
1. my ($index) = grep { $planets[$_] eq 'Mars' } (0 .. @planets-1);
   say defined $index ? $index : -1;
```

The built-in grep function can filter the values of list or array based on some condition. As we are looking for the **index** of the specific value we need to filter the potential indexes of all the elements. The `0 .. @planet-1` expression generates the list of whole numbers between 0 and one less than the number of elements in the @planet array. As the indexing of an array starts by 0, this will be the largest index available in the array.

As `grep` goes over all the elements of the `0 .. @planets-1` list, in each turn the current element will be placed in the `$_` variable. The condition then checks if the element of the `@planet` array in that

position is equal to the planet we are looking for. In list-context grep will return the list of all the values (all the indexes in our case) that passed the test. In scalar-context it would return the number of elements passed, which is not very interesting for us. Thus we put the parentheses around the variable on the left-hand-side `my ($index) =` to create the list-context. By putting only one scalar variable in the parentheses, we will only capture the first value returned by grep. This distinction is only interesting if there could be more values in the original array matching the searched value and if we were interested all the indexes and not just the first one.

With this the grep expression will work, but there is another difference. `first_index` will return -1 in case it did not find any matching value, while `grep` will return an empty list and thus `$index` will be undefined. This can be useful if we later want to check if there was any value by writing:

```
1. if (defined $index) {
       ...
   }
```

but if we would like to imitate the same behavior as we had with `first_index` we can use the ternary operator:

```
1. say defined $index ? $index : -1;
```

Implementing manually - list of indexes

```
1. my @idxs;
   foreach my $place (@input) {
       my ($index) = grep { $planets[$_] eq $place } (0 .. @planets-1);
       push @idxs, defined $index ? $index : -1;
5. }
```

In this code we used the two lines created for the one-element version together with the ternary operator code to have -1 where the element was not found in the `@planets` array

Performance issues?

One of the big differences between the `first_index` solutions and the manual solutions is that the `first_index` function will return the index immediately when it was found, while `grep` will go over the list of all the planets before returning the result. Even if the first element already matched. This is not a problem if the `@planets` array is very short or if we execute this code rarely, but in other cases this might have a performance penalty.

`grep` is basically `foreach` loop, so if we look at the most recent solution, we can see that we have two loops (a foreach loop and a grep inside it) which means the complexity of the code is O(n*m) where n is the number of elements in `@planets` and m is the number of elements in `@input`. The `first_index` solution is better, but in the worst-case situation (when all the values in the @input array are towards the end of the @planets array) the complexity is similar.

There is another solution we can use. In this solution first we create a look-up table mapping planet names to indexes. This is the `%planet_index` hash. Then in the second step we create the list of the indexes corresponding to the values in the `@input` array.

We use the `map` function of Perl to create pairs of "planet-name" => "index". We need to call `reverse` on the indexes before call `map` in order to ensure that if the same value appears twice in the @planets array we take the one with the smaller index. (The second assignment to the `%planet_index` array will overwrite the first one.)

In the second row we call `map` again. We could use a simple look-up like in this code: `my @idxs = map { $planet_index{$_} } @input;`, but that would put `undef` for the values where the planet does not exists. Instead of that we used the defined-or operator `//` introduced in perl 5.010 to put -1 instead of any undef values.

```
1. my %planet_index = map { $planets[$_] => $_ } reverse 0 .. @planets-1;
   my @idxs = map { $planet_index{$_} // -1 } @input;
```

The complexity of this solution is O(n+m), which, for large m and n values is much smaller than O(m*n).

Profiling and Benchmarking

Of course computing the complexity is one thing, but actually comparing the performance of two solutions is a totally different issue. We won't discuss the details in this article, but before any "optimization" we should first use a profiler, probably the Devel::NYTProf profiler to see if this part of the application has any measurable impact on the overall performance of the application. If yes, then we can use the Benchmarkmodule to compare the performance of two or more solutions.