Scalar and List context in Perl, the size of an array

scalar | list | array | size | length | context | Perl

Prev Next

In this episode of the Perl tutorial we are going to look at **context sensitivity** in Perl.

In English, as in most of the other spoken languages, words can have multiple meanings. For example the word "left" has several meanings:

I left the building.

I turned left at the building.

We understand which is the correct meaning by the sentences around the word. This is called the context.

Perl 5 is similar. Words, function calls, and other expressions can have different meaning depending on context. It makes learning harder, but provides more expressiveness.

There are two major contexts in Perl: SCALAR and LIST context.

Array in LIST context

Let's see an example:

```
1. my @words = ('Foo', 'Bar', 'Baz');
   my @names = @words;
```

After the above assignment `@names` contains a copy of the values that were in `@words`;

Assignment of an array to another array copies the content of the array.

Array in SCALAR context

```
1. my @words = ('Foo', 'Bar', 'Baz');
   my $people =  @words;
```

This time we assigned the `@words` array to `$people`, a scalar variable.

Other languages would behave differently, but in Perl this assignment places **the number of elements of the array** in the scalar variable.

That's arbitrary, and in the above case not very useful either, but there are a number of other cases when this behavior can be very useful.

SCALAR and LIST context

The above two are called SCALAR and LIST context. They mean if the expectation is to get a single value (in SCALAR context), or if multiple values are expected (LIST context). In LIST context the number of values can be 0, 1, 2, or any other number.

The context of the if statement

Look at this example:

```
1. my @words = ('Foo', 'Bar', 'Baz');

   if (@words) {
      say "There are some words in the array";
5. }
```

Inside the the condition part of the `if` statement we are expecting exactly one value. That must be SCALAR context then.

By now we know that the value of an array in SCALAR context is the number of elements. We also know that this is 0 (that is FALSE) when the array is empty, and some other positive number (that is TRUE), when the array has 1 or more elements.

So because of that arbitrary decision above, the code `if (@words)` checks if there is any content in the array and fails if the array is empty.

Turning the if-statement around `if (! @words)` will be true if the array is empty.

SCALAR and LIST context

In the previous episode we saw how `localtime()` behaves in SCALAR and LIST context, and now we saw how an array behaves in SCALAR and LIST context.

There is no general rule about context, and you will have to learn the specific cases, but usually they are quite obvious. In any case, when you look up a function using perldoc, you will see an explanation of this for each function. At least in the cases where the SCALAR and LIST contexts yield different results.

We should now look at a few more examples for expressions in Perl, and what kind of context they create.

Creating SCALAR context

We already saw that no matter what you assign to a scalar variable that thing will be in SCALAR context. Let's describe it this way:

```
$x = SCALAR;
```

Because individual elements of an array are also scalars, assignment to them also creates SCALAR context:

```
$word[3] = SCALAR;
```

Concatenation expects two strings on either side so it creates SCALAR context on both sides:

```
"string" . SCALAR;
```

but also

```
SCALAR . "string"
```

So

```
1. my @words = ('Foo', 'Bar', 'Baz');
   say "Number of elements: " . @words;
   say "It is now " . localtime();
```

Will print

```
Number of elements: 3
It is now Thu Feb 30 14:15:53 1998
```

Numerical operators usually expect two numbers - two scalars - on either side. So numerical operators create SCALAR context on both sides:

```
5 + SCALAR;

SCALAR + 5;
```

Creating LIST context

There are constructs that create LIST context:

Assignment to an array is one of them:

```
@x = LIST;
```

Assignment to a list is another place:

```
($x, $y) = LIST;
```

Even if that list only has one element:

```
($x) =  LIST;
```

That brings us to an important issue that can easily trick people:

When are the parentheses significant?

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

 5. my @words = ('Foo', 'Bar', 'Baz');

    my ($x) = @words;
    my $y   = @words;

10. say $x;
11. say $y;
```

the output is:

```
Foo
3
```

This is one of the few places where the parentheses are very important.

In the first assignment `my ($x) = @words;` we assigned to a **list** of scalar variable(s). That created LIST context on the right hand side. That means the **values** of the array were copied to the list on the left hand side. Because there was only one scalar, the first element of the array got copied and the rest not.

In the second assignment `my $y = @words;` we assigned **directly** to a scalar variable. That created SCALAR context on the right hand side. An array in SCALAR context returns the number of elements in it.

This will be very important when you look at passing parameters to functions.

Forcing SCALAR context

Both `print()` and `say()` create LIST context for their parameters. So what if you would like to print the number of elements in an array? What if you'd like to print the nicely formatted date that is returned by `localtime()`?

Let's try this:

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

my @words = ('Foo', 'Bar', 'Baz');

say @words;
say localtime();
```

And the output is

```
FooBarBaz
3542071011113100
```

The former is somehow understandable. These are the values in the array smashed together.

The second one is confusing. It is NOT the same as the result of the `time()` function one might think. It is actually the 9 numbers returned by the `localtime()` function in LIST context. If you don't remember, check it out in the episode about the year of 19100.

The solution is to use the `scalar()` function that will create SCALAR context for its parameter. Actually that's the whole job of the `scalar()` function. Some people might think about it as **casting** from plural to singular, even though I think this word is not used often in Perl-land.

```
say scalar @words;
say scalar localtime();
```

And the output will be:

```
3
Mon Nov  7 21:02:41 2011
```

Length or size of an array in Perl

In a nutshell, if you would like to get the size of an array in Perl you can use the `scalar()` function to force it in SCALAR context and return the size.

The tricky way

Sometimes you might see code like this:

```
1. 0 + @words;
```

This is basically a tricky way to get the size of the array. The `+` operator creates SCALAR context on both sides. An array will return its size in SCALAR context. Adding 0 to it does not change the number, so the above expression returns the size of the array.

I'd recommend writing the slightly longer but clearer way using the `scalar` function.