Function Anti-Patterns

Alberto Manuel Simões
ambs@di.uminho.pt

In the last couple of issues of The Perl Review, I've written about how *not* to do things. This time, I would like to talk about methods, functions, subroutines or whatever you like to call them. There are a few tips to make functions easier to use, less buggy, and more readable.

What's the difference?

First I should explain the basic difference from a method and a function. A method is the object-oriented version of a function: it is a function called on an object. In Perl there isn't a syntatic difference between the two. When I call a function, Perl passes just the arguments I name put in the parentheses.

```
copy_to_dir( $file, $dir );
```

On the other hand, if I use the arrow notation with an object, I'm calling a method. I am using the method named finish(), which is just a Perl function that gets the object as its first argument automatically.

```
$database->finish;
```

Perl converts this to this function call (along with some other magic for inheritance):

DBI::finish($database);

Naming Conventions

As any variable, functions should be named so other programmers can easily figure out what they do. A good name can, in many cases, substitute for documentation. Chose a simple name that reads correctly in English.

For instance, if I are writing a method to check for a property that returns a boolean value, I give it a name that reads well when I use it in an `if` statement. A bad example doesn't read like a sentence. What does the return value of getActiveState() mean? What decision am I trying to make based on it?

```
if( $obj->getActiveState() ) {
  #...
  }
```

Instead, I try to make a sentence out of it. In this example, I'm telling other programmers that the block should only run if the object is active (whatever that means for that hypothetical object).

```
if( $obj->isActive() ) {
  #...
  }
```

Also, I choose a coherent and consistent naming style for all of my code. Perl allows me to make quite long names (how much memory do you have?), so I don't have to remove all of the vowels:

smfnctn(@rgs)

I program so that the next person after me can figure out what I'm doing. It's easier to say it so the next guy doesn't have to guess. This can be especially hard if the cryptic name comes from a language the reader doesn't know, or doesn't know well. It's not a puzzle after all (unless you are in an obfuscated Perl contest, where anti-patterns turn into good patterns).

some_function(@args);

Several naming schemes exist, and I should be consistent. I can use the CamelCase, where all of the words cram together without anything but capitalization to separate them, or the more perly some_function naming schemes for functions where I separate words with underscores. If I'm working with someone else's code, I should follow their naming scheme (at least until I can convince them to do it right!).

Argument Management

Perl doesn't handle function signatures with argument names, so I have to do that myself in the code. My code needs to be clear, and just as function names help to document the code, so do good variable names. In this example, what are **$p** and **$v**? Maybe the reader can figure out what they mean when they are right next to the function name, but how about 10 lines later?

```
sub setProperty {
  my $self = shift;
  my $p    = shift;
  my $v    = shift;
  # ...
  }
```

This code has another problem. I am using a lot of **shift** commands while I could have done it all in one line. A better approach assigns the elements from **@_** all at once.

```
sub setProperty {
  my ($self, $property, $value) = @_;
  # ...
  }
```

One of the common function bugs comes from the bad habit of using shift too much. How many people have had this bug elude them for several minutes (or hours)? If I'm using **shift** all the time, I don't think it's odd to see it there, and I still wonder why nothing shows up in **$property** or **$value**.

```
sub setProperty {
  my ($self, $property, $value) = shift;
  # ...
  }
```

If I'm creating a method, I use the same name for the object in every method. Normally I use the conventional **$self**, but I can use anything I prefer, such as **$obj** or the type name for the object. Don't be too creative because your goal is to make things readable for the next person.

Expected Arguments

Remember that Perl does not check for the right number of arguments for my function, so I have to do that. Another programmer can pass more than I

expected or less than I need. In the beginning of my function code, I check if the user supplied all of the arguments, and assign a default value to the optional ones that don't show up in the argument list.

```
sub some_function {
  my( $value, $time, $debug ) = @_;
  $time  ||= time;
  $debug ||= 0;
  }
```

I have to be careful with false but valid values, though. Sometimes I might have to check for definedness instead of truth. In my previous example, what if I really wanted to start at the beginning of time? I don't want to reset **$time** just because the value is **0**.

```
sub some_function {
  my( $value, $time, $debug ) = @_;
  $time = time unless defined $time;
  }
```

Optional arguments

Optional arguments are at the end of the argument list. The problem is when I want to have multiple optional arguments. I don't want to make the programmer put in a bunch of placeholders.

```
some_function( $required1, $required2,
  $optional1, undef, $optional3 );
```

A function that takes several arguments can be pretty confusing. How many times do you catch yourself pointing at the screen and counting over to see which variable is in the right position?

Instead of positional arguments, I can to use named arguments, which are nothing more than passing an hash as argument list. The hash key is the name and the value is the argument itself. For instance, when calling **do_something** with named parameters, I create the hash in the argument list and build the hash in the subroutine. Notice I might have to quote key names that conflict with Perl reserved words.

```
do_something(  value => 'b', 'time' => 0,
debug => 1 );

sub do_something {
  my %args = @_;
  # ...
  }
```

This way I can check for arguments existence in the hash, then decide what to do based on it being there or not. If it's there, I use its value, and if it's not there, I use the default value. This is also a good approach since it lets users change the order of the arguments, and they can also right out long argument lists to make them easier to read, just like I can write out lists.

```
do_something(
  'time' => 0,
  debug  => 1,
  );
```

With this code structure, I can easily add additional arguments or comment out ones that are already there. Other programmers can also easily see the key names since they show up in one column.

Use references

If I have large strings to pass to functions, I don't want to make Perl do all of the work to copy that string as an argument then pass it to the function just to copy it again into a variable that I can use. If I pass a reference to the large string, Perl only has to pass a small scalar (the reference). Perl isn't going to give back any memory it uses, so I shouldn't waste it.

```
do_something( \$large_string );

sub do_something {
  my( $string_ref ) = @_;
  # ..
  }
```

I also don't have to pass a hash as a list. In that case, Perl has to undo the hash, pass it as one long list, then I have to recreate the hash in the function.

```
do_something( %some_hash );

sub do_something {
  my( %hash ) = @_;
  # ..
  }
```

Instead, I pass a reference to the hash.

```
do_something( \%some_hash );

sub do_something {
  my( $hash_ref ) = @_;
  # ..
  }
```

I can do the same thing to pass other data structures, and even pass multiple hashes or arrays at the same time.

Calling syntax

There are several ways that I can write a call to a function. When calling functions with arguments, I surround the arguments with parentheses. That's enough to tell the perl compiler that I'm using a function name even if I haven't defined the function yet. If I declare them beforehand, parenthesis are not needed. This makes them barewords because I haven't used a sigil or given Perl anyway to figure out what it is yet. This is also known as Perl poetry mode. The use subs declaration tells Perl which names I plan to use, but I can also forward declare the function like I do for do_other_thing. I don't give it the definition yet, and I can do that later. For a short subroutine, I can also just put the whole definition before the other code.

```
use subs qw(do_something);
sub do_other_thing;
sub return_true { 1 };

do_something;
do_other_thing;
return_true;
```

I use the parentheses even if I don't have an argument list though. I don't need them, but using them makes it easier to distinguish between built-in functions and my own, and it's enough of a hint that I don't need to declare my function names ahead of time.

```
do_something();
do_other_thing();
return_true();
```

Crypto-context

On one project, I lost a lot of time debugging code from a friend who always used the **&** everytime he called a function. Do not do that unless you really know what it does! The & by itself is a little bit of magic, hence the reason Tom Christainsen called it "crypto-context". Without anything after the function name, like

```
&function;
```

what I am really doing is passing the current value of the argument list, which in most cases is not what I want to do.

```
&func( @_ );
```

Even if I want to do this, I should be kind to the next guy who has to read my code and who probably doesn't know about this (just like I didn't realize it).

Return values

In Perl, I can return anything I want from a function, including simple scalars, references, arrays and so on. I normally use a scalar, and sometimes that's a reference when I need to return huge lists, or huge data structures (just like I use references to pass large data as arguments).

If I really need to return a list I try to return something useful in scalar context too. To detect if my function was called in scalar context I use the **wantarray** Perl function. If wantarray returns true, the caller used my function in a list context, and it should be safe for me to return a list. If wantarray returns undef, the caller used my function in a void context and will just throw away anything I return, so I don't have to return anything. Beyond that special case, if wantarray returns false, I'm in scalar context.

The context make a difference. Consider this small function that simply returns nothing (or does it?)

```
sub return_nothing { return undef; }
```

In scalar context, I get back the undef value, which is the same thing as not getting back any value. My scalar will not be any different.

```
# $string is not defined
my $string = return_nothing();
```

In list context the situation is much different, Now return_nothing gives back a list of one item, and that one item happens to be the undef item. Instead of an empty list, I get back a list with something in it.

```
# list of one item
my @should_be_empty = return_nothing();
```

Finally, do not forget to use the **return** keyword. Although Perl always returns the last evaluated expression on the code, it makes the code harder to read if I return values implicitly.

About the Author

Alberto Simões is doing his Ph.D. in Machine Translation. He has been working in natural language for four years which explains the large amount of Perl contributions under the Lingua::PT namespace. He was one of the YAPC::EU::2005 organizers.