Perl: Is It Really That Bad?

Wednesday April 26, 2023

I have a simple Python script I wrote years ago that simplifies using rsync to maintain a copy of important data on a second hard drive. I decided to refactor the script and clean it up a bit. As I got ready to do this, it occurred to me: For a non-complex script like this, I wonder how difficult it would be to rewrite it in Perl?

I hadn't used Perl for anything serious for a *long* time. I knew I'd have to re-learn the basics. How difficult would this be? Especially now that I've grown used to using much friendlier languages?

So, I went for it, and here's some of the weirdness I encountered.

First of all, you don't literally specify the type of a variable in Perl. With most languages, you'd expect to be able to either:

1. Spell out the type, e.g., int my_integer, or
2. Have the compiler/interpreter infer the type from the usage, e.g., my_integer = 10.

Instead, Perl uses a special prefix character to indicate the type:

```
# scalars (numbers, strings, and references) use $
my $string_var = "Hello!";
my $int_var = 10;

# arrays use @
my @array_of_numbers = (1, 2, 3);

# hashes (key/value pairs) use %
my %color_codes = ("blue" => 1, "red" => 2, "green" => 3);
```

Functions ("subroutines") take a parameter list instead of individual arguments:

```
sub display_info {
  my ($name, $age) = @_;

  print("Hello, ${name}. You are ${age} years old.\n");
}

display_info("John", 42);
```

You can send named arguments as a hash, but they aren't terribly friendly:

```
sub display_info {
  my (%params) = @_;

  print("Hello, $params{name}. You are $params{age} years old.\n");
}
```

```
display_info(
  name => "John",
  age => 42
);
```

Compare that to Python:

```
def display_info(name, age):
    print(f"Hello, {name}. You are {age} years old.")

display_info("John", 42)
```

Passing an array and a scalar to a function really tripped me up. If you try to do this:

```
sub favorite_colors {
  my @colors = @{ $_[0] };
  my $name = $ { $_[1] };
}

favorite_colors(("red","blue"), "John");
```

Then the array assignment consumes all of the arguments. In other words, @colors will contain (“red”, “blue”, “John”), and $name will be unassigned. In order for this to work, the array must be passed as a reference:

```
my @color_list = ("red","blue");

favorite_colors(\@color_list, "John");
```

Then, the reference scalar will be deferenced back into an array inside the function.

Despite the quirkiness, I did find some things to be pretty clean. I do like the syntax for iterating through an array:

```
my @color_list = ("red","blue");
foreach my $color (@color_list) {
  print("Color: $color\n");
}
```

Calling external programs is also very straightforward:

```
system("program_name arg1 arg2");
```

File system operations, such as checking for the existence of a directory, are easy as well:

```
if ( -d "/path/to/check") {
  # do stuff
}
```

When all was said and done, my backup script, written in Perl, was actually pretty nice:

```
#!/usr/bin/perl

use strict;
use warnings;

sub exec_backup {
    my ( $source, $target ) = @_;

    my $proc_name = "rsync -lrtv --delete \"${source}\" \"${target}\"";

    unless ( -d $target ) {
        mkdir($target);
    }

    if ( -d $target ) {
        print("Syncing ${source}...\n");
        system($proc_name);
        print("Synced ${source}\n");
    }

    print("----------\n");
}

sub exec_backup_set {
    my @source_paths = @{ $_[0] };
    my $target_path  = $_[1];

    foreach my $source_path (@source_paths) {
        exec_backup( $source_path, $target_path );
    }
}

my @target_paths =
  ( "/target1/", "/target2/" );

# regular file sets
my @regular_files = (
    "/home/jimc/source1",  "/home/jimc/source2",
    "/home/jimc/source3",  "/home/jimc/source4"
);
exec_backup_set( \@regular_files, $target_paths[0] );

# large file sets
my @large_files = ( "/home/jimc/large1", "/home/jimc/large2" );
exec_backup_set( \@large_files, $target_paths[1] );

sleep(2);
```

But, I think the Python version is cleaner and more intuitive:

```python
#!/usr/bin/python3

import os
import subprocess
import time

def exec_backup(source, target):
    proc_name = f'rsync -lrtv --delete "{source}" "{target}"'

    if (not os.path.isdir(target)):
        os.makedirs(target)

    if (os.path.isdir(target)):
        print(f"Syncing {source}...")
        subprocess.call(proc_name, shell=True)
        print(f"Synced {source}")

    print("----------")

def exec_backup_set(source_paths, target_path):
    for source_path in source_paths:
        exec_backup(source_path, target_path)

if (__name__ == "__main__"):
    target_paths = ["/target1/", "/target2/"]

    # regular files
    exec_backup_set(
        ["/home/jimc/source1", "/home/jimc/source2",
         "/home/jimc/source3", "/home/jimc/source4"],
        target_paths[0]
    )

    # large files
    exec_backup_set(
        ["/home/jimc/large1", "/home/jimc/large2"],
        target_paths[1]
    )

    time.sleep(2)
```

So, in summary, I think that if I ever need to work in Perl again, I'm not too worried about it. But, given the choice, I'll stick with Python.