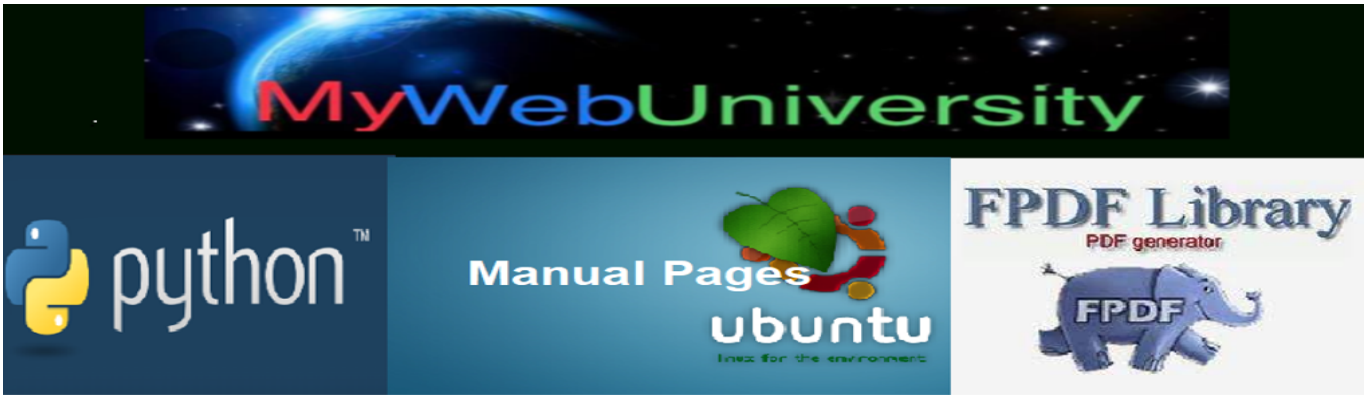

Full credit is given to the above companies including the Operating System (OS) that this PDF file was generated!

Rocky Enterprise Linux 9.2 Manual Pages on command 'Type::Tiny::Manual::NonOO.3pm'

$ man Type::Tiny::Manual::NonOO.3pm

Type::Tiny::Manual::NonOO(3pm) User Contributed Perl Documentation Type::Tiny::Manual::NonOO(3pm)

NAME

Type::Tiny::Manual::NonOO - Type::Tiny in non-object-oriented code

MANUAL

Although Type::Tiny was designed with object-oriented programming in mind, especially Moose-style classes and roles, it can be used in procedural and imperative programming.

If you have read Type::Tiny::Manual::UsingWithMoo, you should understand how Type::Params can be used to validate method parametters. This same technique can be applied to regular subs too; just don't "shift" off $self. More information about checking parameters can be found in Type::Tiny::Manual::Params.

The "is_*" and "assert_*" functions exported by type libraries may be useful in non-OO code too. See Type::Tiny::Manual::UsingWithMoo3.

Type::Tiny and Smart Match

Perl 5.10 introduced the smart match operator "~~", which has since been deprecated because though the general idea is fairly sound, the details were a bit messy.

Nevertheless, Type::Tiny has support for smart match and I'm documenting it here because there's nowhere better to put it.

The following can be used as to check if a value passes a type constraint:

```
$value ~~ SomeType
```

Where it gets weird is if $value is an object and overloads "~~". Which overload of "~~" wins? I don't know.

Better to use:

```
SomeType->check( $value )   # more reliable, probably faster
```

```
  is_SomeType($value)         # more reliable, definitely faster
```

It's also possible to do:

```
  $value ~~ SomeType->coercion
```

This checks to see if $value matches any type that can be coerced to SomeType.

But better to use:

```
  SomeType->coercion->has_coercion_for_value( $value )
```

"given" and "when"

Related to the smart match operator is the "given"/"when" syntax.

This will not do what you want it to do:

```
  use Types::Standard qw( Str Int );
  given ($value) {
    when (Int) { ... }
    when (Str) { ... }
  }
```

This will do what you wanted:

```
  use Types::Standard qw( is_Str is_Int );
  given ($value) {
    when (\&is_Int) { ... }
    when (\&is_Str) { ... }
  }
```

Sorry, that's just how Perl be.

Better though:

```
  use Types::Standard qw( Str Int );
  use Type::Utils qw( match_on_type );
  match_on_type $value => (
    Str, sub { ... },
    Int, sub { ... },
  );
```

If this is part of a loop or other frequently called bit of code, you can compile the checks once and use them many times:

```
  use Types::Standard qw( Str Int );
  use Type::Utils qw( compile_match_on_type );
  my $dispatch_table = compile_match_on_type(
```

```
    Str, sub { ... },
    Int, sub { ... },
  );
  $dispatch_table->($_) for @lots_of_values;
```

As with most things in Type::Tiny, those coderefs can be replaced by strings of Perl code.

NEXT STEPS

Here's your next step:

? Type::Tiny::Manual::Optimization

Squeeze the most out of your CPU.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES