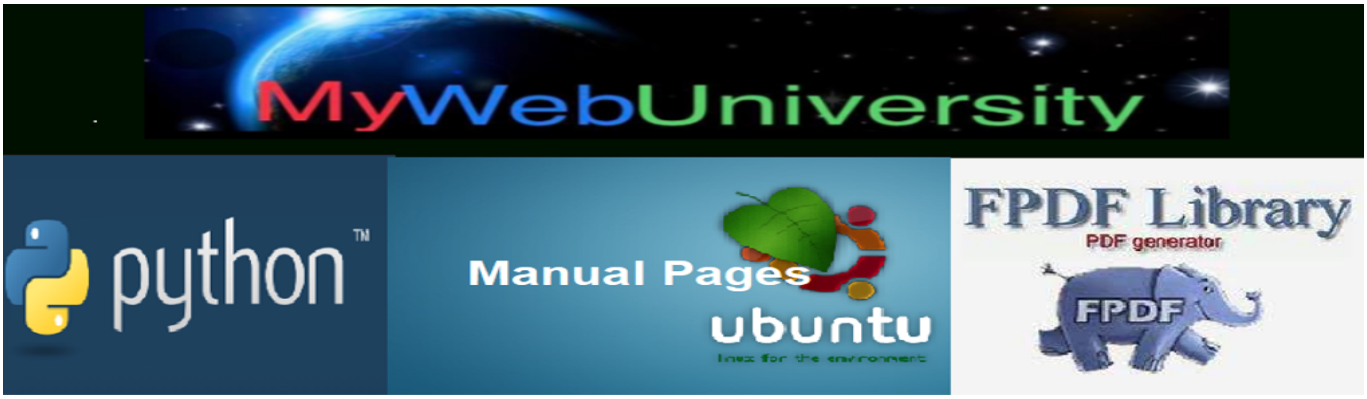

Rocky Enterprise Linux 9.2 Manual Pages on command 'Scalar::Util.3perl'

$ man Scalar::Util.3perl

NAME

Scalar::Util - A selection of general-utility scalar subroutines

SYNOPSIS

```
use Scalar::Util qw(blessed dualvar isdual readonly refaddr reftype
                    tainted weaken isweak isvstring looks_like_number
                    set_prototype);
                    # and other useful utils appearing below
```

DESCRIPTION

"Scalar::Util" contains a selection of subroutines that people have expressed would be nice to have in the perl core, but the usage would not really be high enough to warrant the use of a keyword, and the size would be so small that being individual extensions would be wasteful.

By default "Scalar::Util" does not export any subroutines.

FUNCTIONS FOR REFERENCES

The following functions all perform some useful activity on reference values.

```
my $pkg = blessed( $ref );
```

If $ref is a blessed reference, the name of the package that it is blessed into is returned. Otherwise "undef" is returned.

```
$scalar = "foo";
$class  = blessed $scalar;          # undef

$ref    = [];
$class  = blessed $ref;             # undef

$obj    = bless [], "Foo";
$class  = blessed $obj;             # "Foo"
```

Take care when using this function simply as a truth test (such as in "if(blessed $ref)...") because the package name "0" is defined yet false.

refaddr

```
my $addr = refaddr( $ref );
```

If $ref is reference, the internal memory address of the referenced value is returned as a plain integer. Otherwise "undef" is returned.

```
$addr = refaddr "string";          # undef
$addr = refaddr \$var;             # eg 12345678
$addr = refaddr [];               # eg 23456784

$obj  = bless {}, "Foo";
$addr = refaddr $obj;              # eg 88123488
```

reftype

```
my $type = reftype( $ref );
```

If $ref is a reference, the basic Perl type of the variable referenced is returned as a plain string (such as "ARRAY" or "HASH"). Otherwise "undef" is returned.

```
$type = reftype "string";          # undef
$type = reftype \$var;             # SCALAR
$type = reftype [];               # ARRAY

$obj  = bless {}, "Foo";
$type = reftype $obj;              # HASH
```

Note that for internal reasons, all precompiled regexps ("qr/.../") are blessed references; thus "ref()" returns the package name string "Regexp" on these but "reftype()" will return the underlying C structure type of "REGEXP" in all capitals.

weaken

```
weaken( $ref );
```

The lvalue $ref will be turned into a weak reference. This means that it will not hold a reference count on the object it references. Also, when the reference count on that object reaches zero, the reference will be set to undef. This function mutates the lvalue passed as its argument and returns no value.

This is useful for keeping copies of references, but you don't want to prevent the object being DESTROY-ed at its usual time.

```
{
  my $var;
  $ref = \$var;
  weaken($ref);                    # Make $ref a weak reference
}
# $ref is now undef
```

Note that if you take a copy of a scalar with a weakened reference, the copy will be a

strong reference.

```
my $var;
my $foo = \$var;
weaken($foo);                   # Make $foo a weak reference
my $bar = $foo;                 # $bar is now a strong reference
```

This may be less obvious in other situations, such as "grep()", for instance when grepping through a list of weakened references to objects that may have been destroyed already:

```
@object = grep { defined } @object;
```

This will indeed remove all references to destroyed objects, but the remaining references to objects will be strong, causing the remaining objects to never be destroyed because there is now always a strong reference to them in the @object array.

unweaken

```
unweaken( $ref );
```

Since version 1.36.

The lvalue "REF" will be turned from a weak reference back into a normal (strong) reference again. This function mutates the lvalue passed as its argument and returns no value. This undoes the action performed by "weaken".

This function is slightly neater and more convenient than the otherwise-equivalent code

```
my $tmp = $REF;
undef $REF;
$REF = $tmp;
```

(because in particular, simply assigning a weak reference back to itself does not work to unweaken it; "$REF = $REF" does not work).

isweak

```
my $weak = isweak( $ref );
```

Returns true if $ref is a weak reference.

```
$ref  = \$foo;
$weak = isweak($ref);             # false
weaken($ref);
$weak = isweak($ref);             # true
```

NOTE: Copying a weak reference creates a normal, strong, reference.

```
$copy = $ref;
$weak = isweak($copy);            # false
```

OTHER FUNCTIONS

dualvar

```
my $var = dualvar( $num, $string );
```

Returns a scalar that has the value $num in a numeric context and the value $string in a string context.

```
$foo = dualvar 10, "Hello";
$num = $foo + 2;                  # 12
$str = $foo . " world";           # Hello world
```

isdual

```
my $dual = isdual( $var );
```

Since version 1.26.

If $var is a scalar that has both numeric and string values, the result is true.

```
$foo = dualvar 86, "Nix";
$dual = isdual($foo);           # true
```

Note that a scalar can be made to have both string and numeric content through numeric operations:

```
$foo = "10";
$dual = isdual($foo);           # false
$bar = $foo + 0;
$dual = isdual($foo);           # true
```

Note that although $! appears to be a dual-valued variable, it is actually implemented as a magical variable inside the interpreter:

```
$! = 1;
print("$!\n");                  # "Operation not permitted"
$dual = isdual($!);             # false
```

You can capture its numeric and string content using:

```
$err = dualvar $!, $!;
$dual = isdual($err);           # true
```

isvstring

```
my $vstring = isvstring( $var );
```

If $var is a scalar which was coded as a vstring, the result is true.

```
$vs   = v49.46.48;
$fmt  = isvstring($vs) ? "%vd" : "%s"; #true
printf($fmt,$vs);
```

looks_like_number

```
my $isnum = looks_like_number( $var );
```

Returns true if perl thinks $var is a number. See "looks_like_number" in perlapi.

openhandle

```
my $fh = openhandle( $fh );
```

Returns $fh itself, if $fh may be used as a filehandle and is open, or if it is a tied handle. Otherwise "undef" is returned.

```
$fh = openhandle(*STDIN);          # \*STDIN
$fh = openhandle(\*STDIN);         # \*STDIN
$fh = openhandle(*NOTOPEN);        # undef
$fh = openhandle("scalar");        # undef
```

readonly

```
my $ro = readonly( $var );
```

Returns true if $var is readonly.

```
sub foo { readonly($_[0]) }

$readonly = foo($bar);             # false
$readonly = foo(0);                # true
```

set_prototype

```
my $code = set_prototype( $code, $prototype );
```

Sets the prototype of the function given by the $code reference, or deletes it if $prototype is "undef". Returns the $code reference itself.

```
set_prototype \&foo, '$$';
```

tainted

```
my $t = tainted( $var );
```

Return true if $var is tainted.

```
$taint = tainted("constant");       # false
$taint = tainted($ENV{PWD});        # true if running under -T
```

DIAGNOSTICS

Module use may give one of the following errors during import.

Weak references are not implemented in the version of perl

The version of perl that you are using does not implement weak references, to use "isweak" or "weaken" you will need to use a newer release of perl.

Vstrings are not implemented in the version of perl

The version of perl that you are using does not implement Vstrings, to use "isvstring" you will need to use a newer release of perl.

KNOWN BUGS

There is a bug in perl5.6.0 with UV's that are >= 1<<31. This will show up as tests 8 and 9 of dualvar.t failing

SEE ALSO

List::Util

COPYRIGHT

Additionally "weaken" and "isweak" which are