NAME

Params::Classify – argument type classification

SYNOPSIS

```
    use Params::Classify qw(
        scalar_class
        is_undef check_undef
        is_string check_string
        is_number check_number
        is_glob check_glob
        is_regexp check_regexp
        is_ref check_ref ref_type
        is_blessed check_blessed blessed_class
        is_strictly_blessed check_strictly_blessed
        is_able check_able);

    $c = scalar_class($arg);

    if(is_undef($arg)) {
    check_undef($arg);

    if(is_string($arg)) {
    check_string($arg);
    if(is_number($arg)) {
    check_number($arg);

    if(is_glob($arg)) {
    check_glob($arg);
    if(is_regexp($arg)) {
    check_regexp($arg);

    if(is_ref($arg)) {
    check_ref($arg);
    $t = ref_type($arg);
    if(is_ref($arg, "HASH")) {
    check_ref($arg, "HASH");

    if(is_blessed($arg)) {
    check_blessed($arg);
    if(is_blessed($arg, "IO::Handle")) {
    check_blessed($arg, "IO::Handle");
    $c = blessed_class($arg);
    if(is_strictly_blessed($arg, "IO::Pipe::End")) {
    check_strictly_blessed($arg, "IO::Pipe::End");
    if(is_able($arg, ["print", "flush"])) {
    check_able($arg, ["print", "flush"]);
```

DESCRIPTION

This module provides various type-testing functions. These are intended for functions that, unlike most Perl code, care what type of data they are operating on. For example, some functions wish to behave differently depending on the type of their arguments (like overloaded functions in C++).

There are two flavours of function in this module. Functions of the first flavour only provide type classification, to allow code to discriminate between argument types. Functions of the second flavour package up the most common type of type discrimination: checking that an argument is of an expected type. The functions come in matched pairs, of the two flavours, and so the type enforcement functions handle only the simplest requirements for arguments of the types handled by the classification functions. Enforcement of more complex types may, of course, be built using the classification functions, or it may be more convenient to use a module designed for the more complex job, such as Params::Validate.

This module is implemented in XS, with a pure Perl backup version for systems that can't handle XS.

TYPE CLASSIFICATION

This module divides up scalar values into the following classes:

- undef
- string (defined ordinary scalar)
- typeglob (yes, typeglobs fit into scalar variables)
- regexp (first-class regular expression objects in Perl 5.11 onwards)
- reference to unblessed object (further classified by physical data type of the referenced object)
- reference to blessed object (further classified by class blessed into)

These classes are mutually exclusive and should be exhaustive. This classification has been chosen as the most useful when one wishes to discriminate between types of scalar. Other classifications are possible. (For example, the two reference classes are distinguished by a feature of the referenced object; Perl does not internally treat this as a feature of the reference.)

FUNCTIONS

Each of these functions takes one scalar argument (*ARG*) to be tested, possibly with other arguments specifying details of the test. Any scalar value is acceptable for the argument to be tested. Each `is_` function returns a simple truth value result, which is true iff *ARG* is of the type being checked for. Each `check_` function will return normally if the argument is of the type being checked for, or will `die` if it is not.

Classification

scalar_class(ARG)

Determines which of the five classes described above *ARG* falls into. Returns "**UNDEF**“, ”**STRING**“, ”**GLOB**“, ”**REGEXP**“, ”**REF**“, or ”**BLESSED**" accordingly.

The Undefined Value

is_undef(ARG)
check_undef(ARG)

Check whether *ARG* is `undef`. `is_undef(ARG)` is precisely equivalent to `!defined(ARG)`, and is included for completeness.

Strings

is_string(ARG)
check_string(ARG)

Check whether *ARG* is defined and is an ordinary scalar value (not a reference, typeglob, or regexp). This is what one usually thinks of as a string in Perl. In fact, any scalar (including `undef` and references) can be coerced to a string, but if you're trying to classify a scalar then you don't want to do that.

is_number(ARG)
check_number(ARG)

Check whether *ARG* is defined and an ordinary scalar (i.e., satisfies "is_string" above) and is an acceptable number to Perl. This is what one usually thinks of as a number.

Note that simple ("is_string"−satisfying) scalars may have independent numeric and string values, despite the usual pretence that they have only one value. Such a scalar is deemed to be a number if *either* it already has a numeric value (e.g., was generated by a numeric literal or an arithmetic computation) *or* its string value has acceptable syntax for a number (so it can be converted). Where a scalar has separate numeric and string values (see "dualvar" in Scalar::Util), it is possible for it to have an acceptable numeric value while its string value does *not* have acceptable numeric syntax. Be careful to use such a value only in a numeric context, if you are using it as a number. "scalar_num_part" in Scalar::Number extracts the numeric part of a scalar as an ordinary number. (`0+ARG` suffices for that unless you need to preserve floating point signed zeroes.)

A number may be either a native integer or a native floating point value, and there are several subtypes of floating point value. For classification, and other handling of numbers in scalars, see Scalar::Number. For details of the two numeric data types, see Data::Integer and Data::Float.

This function differs from `looks_like_number` (see "looks_like_number" in Scalar::Util; also "looks_like_number" in perlapi for a lower-level description) in excluding `undef`, typeglobs, and references. Why `looks_like_number` returns true for `undef` or typeglobs is anybody's guess. References, if treated as numbers, evaluate to the address in memory that they reference; this is useful for comparing references for equality, but it is not otherwise useful to treat references as numbers. Blessed references may have overloaded numeric operators, but if so then they don't necessarily behave like ordinary numbers. `looks_like_number` is also confused by dualvars: it looks at the string portion of the scalar.

Typeglobs

is_glob(ARG)
check_glob(ARG)
: Check whether *ARG* is a typeglob.

Regexps

is_regexp(ARG)
check_regexp(ARG)
: Check whether *ARG* is a regexp object.

References to Unblessed Objects

is_ref(ARG)
check_ref(ARG)
: Check whether *ARG* is a reference to an unblessed object. If it is, then the referenced data type can be determined using `ref_type` (see below), which will return a string such as "HASH" or "SCALAR".

ref_type(ARG)
: Returns `undef` if *ARG* is not a reference to an unblessed object. Otherwise, determines what type of object is referenced. Returns "**SCALAR**", "**ARRAY**", "**HASH**", "**CODE**", "**FORMAT**", or "**IO**" accordingly.

Note that, unlike `ref`, this does not distinguish between different types of referenced scalar. A reference to a string and a reference to a reference will both return "**SCALAR**". Consequently, what `ref_type` returns for a particular reference will not change due to changes in the value of the referent, except for the referent being blessed.

is_ref(ARG, TYPE)
check_ref(ARG, TYPE)
: Check whether *ARG* is a reference to an unblessed object of type *TYPE*, as determined by "ref_type". *TYPE* must be a string. Possible *TYPE*s are "**SCALAR**", "**ARRAY**", "**HASH**", "**CODE**", "**FORMAT**", and "**IO**".

References to Blessed Objects

is_blessed(ARG)
check_blessed(ARG)
: Check whether *ARG* is a reference to a blessed object. If it is, then the class into which the object was blessed can be determined using "blessed_class".

is_blessed(ARG, CLASS)
check_blessed(ARG, CLASS)
: Check whether *ARG* is a reference to a blessed object that claims to be an instance of *CLASS* (via its `isa` method; see "isa" in perlobj). *CLASS* must be a string, naming a Perl class.

blessed_class(ARG)
: Returns `undef` if *ARG* is not a reference to a blessed object. Otherwise, returns the class into which the object is blessed.

  `ref` (see "ref" in perlfunc) gives the same result on references to blessed objects, but different results on other types of value. `blessed_class` is actually identical to "blessed" in Scalar::Util.

is_strictly_blessed(ARG)
check_strictly_blessed(ARG)
: Check whether *ARG* is a reference to a blessed object, identically to "is_blessed". This exists only for symmetry; the useful form of `is_strictly_blessed` appears below.

is_strictly_blessed(ARG, CLASS)
check_strictly_blessed(ARG, CLASS)
: Check whether *ARG* is a reference to an object blessed into *CLASS* exactly. *CLASS* must be a string, naming a Perl class. Because this excludes subclasses, this is rarely what one wants, but there are some specialised occasions where it is useful.

is_able(ARG)
check_able(ARG)
: Check whether *ARG* is a reference to a blessed object, identically to "is_blessed". This exists only for symmetry; the useful form of `is_able` appears below.

is_able(ARG, METHODS)
check_able(ARG, METHODS)
: Check whether *ARG* is a reference to a blessed object that claims to implement the methods specified by *METHODS* (via its `can` method; see "can" in perlobj). *METHODS* must be either a single method name or a reference to an array of method names. Each method name is a string. This interface check is often more appropriate than a direct ancestry check (such as "is_blessed" performs).

BUGS

Probably ought to handle something like Params::Validate's scalar type specification system, which makes much the same distinctions.

SEE ALSO

Data::Float, Data::Integer, Params::Validate, Scalar::Number, Scalar::Util

AUTHOR

Andrew Main (Zefram) <zefram@fysh.org>

COPYRIGHT

LICENSE