NAME

JSON::PP - JSON::XS compatible pure-Perl module.

SYNOPSIS

```
use JSON::PP;

# exported functions, they croak on error
# and expect/generate UTF-8

$utf8_encoded_json_text = encode_json $perl_hash_or_arrayref;
$perl_hash_or_arrayref = decode_json $utf8_encoded_json_text;

# OO-interface

$coder = JSON::PP->new->ascii->pretty->allow_nonref;

$json_text = $json->encode( $perl_scalar );
$perl_scalar = $json->decode( $json_text );

$pretty_printed = $json->pretty->encode( $perl_scalar ); # pretty-printing

# Note that JSON version 2.0 and above will automatically use
# JSON::XS or JSON::PP, so you should be able to just:

use JSON;
```

VERSION

2.27202

JSON::XS 2.27 (~2.30) compatible.

NOTE

JSON::PP had been inculded in JSON distribution (CPAN module). It was a perl core module in Perl 5.14.

DESCRIPTION

This module is JSON::XS compatible pure Perl module. (Perl 5.8 or later is recommended)

JSON::XS is the fastest and most proper JSON module on CPAN. It is written by Marc Lehmann in C, so must be compiled and installed in the used environment.

JSON::PP is a pure-Perl module and has compatibility to JSON::XS.

FEATURES

- correct unicode handling

  This module knows how to handle Unicode (depending on Perl version).

  See to "A FEW NOTES ON UNICODE AND PERL" in JSON::XS and "UNICODE HANDLING ON PERLS".

- round-trip integrity

  When you serialise a perl data structure using only data types supported by JSON and Perl, the deserialised data structure is identical on the Perl level. (e.g. the string "2.0" doesn't suddenly become "2" just because it looks like a number). There *are* minor exceptions to this, read the MAPPING section below to learn about those.

- strict checking of JSON correctness

  There is no guessing, no generating of illegal JSON texts by default, and only JSON is accepted as input by default (the latter is a security feature). But when some options are

set, loose chcking features are available.

FUNCTIONAL INTERFACE

Some documents are copied and modified from "FUNCTIONAL INTERFACE" in JSON::XS

encode_json

```
$json_text = encode_json $perl_scalar
```

Converts the given Perl data structure to a UTF-8 encoded, binary string.

This function call is functionally identical to:

```
$json_text = JSON::PP->new->utf8->encode($perl_scalar)
```

decode_json

```
$perl_scalar = decode_json $json_text
```

The opposite of `encode_json`: expects an UTF-8 (binary) string and tries to parse that as an UTF-8 encoded JSON text, returning the resulting reference.

This function call is functionally identical to:

```
$perl_scalar = JSON::PP->new->utf8->decode($json_text)
```

JSON::PP::is_bool

```
$is_boolean = JSON::PP::is_bool($scalar)
```

Returns true if the passed scalar represents either JSON::PP::true or JSON::PP::false, two constants that act like `1` and `0` respectively and are also used to represent JSON `true` and `false` in Perl strings.

JSON::PP::true

Returns JSON true value which is blessed object. It `isa` JSON::PP::Boolean object.

JSON::PP::false

Returns JSON false value which is blessed object. It `isa` JSON::PP::Boolean object.

JSON::PP::null

Returns `undef`.

See MAPPING, below, for more information on how JSON values are mapped to Perl.

HOW DO I DECODE A DATA FROM OUTER AND ENCODE TO OUTER

This section supposes that your perl vresion is 5.8 or later.

If you know a JSON text from an outer world - a network, a file content, and so on, is encoded in UTF-8, you should use `decode_json` or `JSON` module object with `utf8` enable. And the decoded result will contain UNICODE characters.

```
# from network
my $json = JSON::PP->new->utf8;
my $json_text = CGI->new->param( 'json_data' );
my $perl_scalar = $json->decode( $json_text );

# from file content
local $/;
open( my $fh, '<', 'json.data' );
$json_text = <$fh>;
$perl_scalar = decode_json( $json_text );
```

If an outer data is not encoded in UTF-8, firstly you should `decode` it.

```
use Encode;
local $/;
open( my $fh, '<', 'json.data' );
my $encoding = 'cp932';
my $unicode_json_text = decode( $encoding, <$fh> ); # UNICODE

# or you can write the below code.
#
# open( my $fh, "<:encoding($encoding)", 'json.data' );
# $unicode_json_text = <$fh>;
```

In this case, `$unicode_json_text` is of course UNICODE string. So you **cannot** use `decode_json` nor `JSON` module object with `utf8` enable. Instead of them, you use `JSON` module object with `utf8` disable.

```
$perl_scalar = $json->utf8(0)->decode( $unicode_json_text );
```

Or `encode 'utf8'` and `decode_json`:

```
$perl_scalar = decode_json( encode( 'utf8', $unicode_json_text ) );
# this way is not efficient.
```

And now, you want to convert your `$perl_scalar` into JSON data and send it to an outer world - a network or a file content, and so on.

Your data usually contains UNICODE strings and you want the converted data to be encoded in UTF-8, you should use `encode_json` or `JSON` module object with `utf8` enable.

```
print encode_json( $perl_scalar ); # to a network? file? or display?
# or
print $json->utf8->encode( $perl_scalar );
```

If `$perl_scalar` does not contain UNICODE but `$encoding`-encoded strings for some reason, then its characters are regarded as **latin1** for perl (because it does not concern with your `$encoding`). You **cannot** use `encode_json` nor `JSON` module object with `utf8` enable. Instead of them, you use `JSON` module object with `utf8` disable. Note that the resulted text is a UNICODE string but no problem to print it.

```
# $perl_scalar contains $encoding encoded string values
$unicode_json_text = $json->utf8(0)->encode( $perl_scalar );
# $unicode_json_text consists of characters less than 0x100
print $unicode_json_text;
```

Or `decode $encoding` all string values and `encode_json`:

```
$perl_scalar->{ foo } = decode( $encoding, $perl_scalar->{ foo } );
# ... do it to each string values, then encode_json
$json_text = encode_json( $perl_scalar );
```

This method is a proper way but probably not efficient.

See to Encode, perluniintro.

METHODS

Basically, check to JSON or JSON::XS

new

```
$json = JSON::PP->new
```

Rturns a new JSON::PP object that can be used to de/encode JSON strings.

All boolean flags described below are by default *disabled*.

The mutators for flags all return the JSON object again and thus calls can be chained:

```
my $json = JSON::PP->new->utf8->space_after->encode({a => [1,2]})
=> {"a": [1, 2]}
```

ascii

```
$json = $json->ascii([$enable])

$enabled = $json->get_ascii
```

If `$enable` is true (or missing), then the encode method will not generate characters outside the code range 0..127. Any Unicode characters outside that range will be escaped using either a single uXXXX or a double uHHHHuLLLLL escape sequence, as per RFC4627. (See to ''OBJECT-ORIENTED INTERFACE'' in JSON::XS

In Perl 5.005, there is no character having high value (more than 255). See to ''UNICODE HANDLING ON PERLS''.

If `$enable` is false, then the encode method will not escape Unicode characters unless required by the JSON syntax or other flags. This results in a faster and more compact format.

```
JSON::PP->new->ascii(1)->encode([chr 0x10401])
=> ["\ud801\udc01"]
```

latin1

```
$json = $json->latin1([$enable])

$enabled = $json->get_latin1
```

If `$enable` is true (or missing), then the encode method will encode the resulting JSON text as latin1 (or iso-8859-1), escaping any characters outside the code range 0..255.

If `$enable` is false, then the encode method will not escape Unicode characters unless required by the JSON syntax or other flags.

```
JSON::XS->new->latin1->encode (["\x{89}\x{abc}"]
=> ["\x{89}\\u0abc"] # (perl syntax, U+abc escaped, U+89 not)
```

See to ''UNICODE HANDLING ON PERLS''.

utf8

```
$json = $json->utf8([$enable])

$enabled = $json->get_utf8
```

If `$enable` is true (or missing), then the encode method will encode the JSON result into UTF-8, as required by many protocols, while the decode method expects to be handled an UTF-8-encoded string. Please note that UTF-8-encoded strings do not contain any characters outside the range 0..255, they are thus useful for bytewise/binary I/O.

(In Perl 5.005, any character outside the range 0..255 does not exist. See to ''UNICODE HANDLING ON PERLS''.)

In future versions, enabling this option might enable autodetection of the UTF-16 and UTF-32 encoding families, as described in RFC4627.

If `$enable` is false, then the encode method will return the JSON string as a (non-encoded) Unicode string, while decode expects thus a Unicode string. Any decoding or encoding (e.g. to UTF-8 or UTF-16) needs to be done yourself, e.g. using the Encode module.

Example, output UTF-16BE-encoded JSON:

```
use Encode;
$jsontext = encode "UTF-16BE", JSON::PP->new->encode ($object);
```

Example, decode UTF-32LE-encoded JSON:

```
use Encode;
$object = JSON::PP->new->decode (decode "UTF-32LE", $jsontext);
```

pretty

```
$json = $json->pretty([$enable])
```

This enables (or disables) all of the `indent`, `space_before` and `space_after` flags in one call to generate the most readable (or most compact) form possible.

Equivalent to:

```
$json->indent->space_before->space_after
```

indent

```
$json = $json->indent([$enable])

$enabled = $json->get_indent
```

The default indent space length is three. You can use `indent_length` to change the length.

space_before

```
$json = $json->space_before([$enable])

$enabled = $json->get_space_before
```

If `$enable` is true (or missing), then the `encode` method will add an extra optional space before the `:` separating keys from values in JSON objects.

If `$enable` is false, then the `encode` method will not add any extra space at those places.

This setting has no effect when decoding JSON texts.

Example, space_before enabled, space_after and indent disabled:

```
{"key" :"value"}
```

space_after

```
$json = $json->space_after([$enable])

$enabled = $json->get_space_after
```

If `$enable` is true (or missing), then the `encode` method will add an extra optional space after the `:` separating keys from values in JSON objects and extra whitespace after the `,` separating key-value pairs and array members.

If `$enable` is false, then the `encode` method will not add any extra space at those places.

This setting has no effect when decoding JSON texts.

Example, space_before and indent disabled, space_after enabled:

```
{"key": "value"}
```

relaxed

```
$json = $json->relaxed([$enable])

$enabled = $json->get_relaxed
```

If `$enable` is true (or missing), then `decode` will accept some extensions to normal JSON syntax (see below). `encode` will not be affected in anyway. *Be aware that this option makes you accept invalid JSON texts as if they were valid!*. I suggest only to use this option to parse application-specific files written by humans (configuration files, resource files etc.)

If `$enable` is false (the default), then `decode` will only accept valid JSON texts.

Currently accepted extensions are:

- list items can have an end-comma

  JSON *separates* array elements and key-value pairs with commas. This can be annoying if you write JSON texts manually and want to be able to quickly append elements, so this extension accepts comma at the end of such items not just between them:

  ```
  [
  1,
  2, <- this comma not normally allowed
  ]
  {
  "k1": "v1",
  "k2": "v2", <- this comma not normally allowed
  }
  ```

- shell-style '#'-comments

  Whenever JSON allows whitespace, shell-style comments are additionally allowed. They are terminated by the first carriage-return or line-feed character, after which more white-space and comments are allowed.

  ```
  [
  1, # this comment not allowed in JSON
  # neither this one...
  ]
  ```

canonical

```
$json = $json->canonical([$enable])

$enabled = $json->get_canonical
```

If `$enable` is true (or missing), then the `encode` method will output JSON objects by sorting their keys. This is adding a comparatively high overhead.

If `$enable` is false, then the `encode` method will output key-value pairs in the order Perl stores them (which will likely change between runs of the same script).

This option is useful if you want the same data structure to be encoded as the same JSON text (given the same overall settings). If it is disabled, the same hash might be encoded differently even if contains the same data, as key-value pairs have no inherent ordering in Perl.

This setting has no effect when decoding JSON texts.

If you want your own sorting routine, you can give a code referece or a subroutine name to `sort_by`. See to JSON::PP OWN METHODS.

allow_nonref

```
$json = $json->allow_nonref([$enable])

$enabled = $json->get_allow_nonref
```

If `$enable` is true (or missing), then the `encode` method can convert a non-reference into its corresponding string, number or null JSON value, which is an extension to RFC4627. Likewise, `decode` will accept those JSON values instead of croaking.

If `$enable` is false, then the `encode` method will croak if it isn't passed an arrayref or hashref, as JSON texts must either be an object or array. Likewise, `decode` will croak if given something that is not a JSON object or array.

```
JSON::PP->new->allow_nonref->encode ("Hello, World!")
=> "Hello, World!"
```

allow_unknown

```
$json = $json->allow_unknown ([$enable])

$enabled = $json->get_allow_unknown
```

If `$enable` is true (or missing), then ''encode'' will *not* throw an exception when it encounters values it cannot represent in JSON (for example, filehandles) but instead will encode a JSON ''null'' value. Note that blessed objects are not included here and are handled separately by c<allow_nonref>.

If `$enable` is false (the default), then ''encode'' will throw an exception when it encounters anything it cannot encode as JSON.

This option does not affect ''decode'' in any way, and it is recommended to leave it off unless you know your communications partner.

allow_blessed

```
$json = $json->allow_blessed([$enable])

$enabled = $json->get_allow_blessed
```

If `$enable` is true (or missing), then the `encode` method will not barf when it encounters a blessed reference. Instead, the value of the **convert_blessed** option will decide whether `null` (`convert_blessed` disabled or no `TO_JSON` method found) or a representation of the object (`convert_blessed` enabled and `TO_JSON` method found) is being encoded. Has no effect on `decode`.

If `$enable` is false (the default), then `encode` will throw an exception when it encounters a blessed object.

convert_blessed

```
$json = $json->convert_blessed([$enable])

$enabled = $json->get_convert_blessed
```

If `$enable` is true (or missing), then `encode`, upon encountering a blessed object, will check for the availability of the `TO_JSON` method on the object's class. If found, it will be called in scalar context and the resulting scalar will be encoded instead of the object. If no `TO_JSON` method is found, the value of `allow_blessed` will decide what to do.

The `TO_JSON` method may safely call die if it wants. If `TO_JSON` returns other blessed objects, those will be handled in the same way. `TO_JSON` must take care of not causing an endless recursion cycle (== crash) in this case. The name of `TO_JSON` was chosen because other methods called by the Perl core (== not by the user of the object) are usually in upper case letters and to avoid collisions with the `to_json` function or method.

This setting does not yet influence `decode` in any way.

If `$enable` is false, then the `allow_blessed` setting will decide what to do when a blessed object is found.

filter_json_object

```
$json = $json->filter_json_object([$coderef])
```

When `$coderef` is specified, it will be called from `decode` each time it decodes a JSON object. The only argument passed to the coderef is a reference to the newly-created hash. If the code references returns a single scalar (which need not be a reference), this value (i.e. a copy of that scalar to avoid aliasing) is inserted into the deserialised data structure. If it returns an empty list (NOTE: *not* `undef`, which is a valid scalar), the original deserialised hash will be inserted. This setting can slow down decoding considerably.

When `$coderef` is omitted or undefined, any existing callback will be removed and `decode` will not change the deserialised hash in any way.

Example, convert all JSON objects into the integer 5:

```
my $js = JSON::PP->new->filter_json_object (sub { 5 });
# returns [5]
$js->decode ('[{}]'); # the given subroutine takes a hash reference.
# throw an exception because allow_nonref is not enabled
# so a lone 5 is not allowed.
$js->decode ('{"a":1, "b":2}');
```

filter_json_single_key_object

```
$json = $json->filter_json_single_key_object($key [=> $coderef])
```

Works remotely similar to `filter_json_object`, but is only called for JSON objects having a single key named `$key`.

This `$coderef` is called before the one specified via `filter_json_object`, if any. It gets passed the single value in the JSON object. If it returns a single value, it will be inserted into the data structure. If it returns nothing (not even `undef` but the empty list), the callback from `filter_json_object` will be called next, as if no single-key callback were specified.

If `$coderef` is omitted or undefined, the corresponding callback will be disabled. There can only ever be one callback for a given key.

As this callback gets called less often then the `filter_json_object` one, decoding speed will not usually suffer as much. Therefore, single-key objects make excellent targets to serialise Perl objects into, especially as single-key JSON objects are as close to the type-tagged value concept as JSON gets (it's basically an ID/VALUE tuple). Of course, JSON does not support this in any way, so you need to make sure your data never looks like a serialised Perl hash.

Typical names for the single object key are `__class_whatever__`, or `$__dollars_are_rarely_used__$` or `}ugly_brace_placement`, or even things like `__class_md5sum(classname)__`, to reduce the risk of clashing with real hashes.

Example, decode JSON objects of the form `{ "__widget__" => <id> }` into the corresponding `$WIDGET{<id>}` object:

```
# return whatever is in $WIDGET{5}:
JSON::PP
->new
->filter_json_single_key_object (__widget__ => sub {
$WIDGET{ $_[0] }
})
->decode ('{"__widget__": 5')

# this can be used with a TO_JSON method in some "widget" class
# for serialisation to json:
sub WidgetBase::TO_JSON {
my ($self) = @_;

unless ($self->{id}) {
$self->{id} = ..get..some..id..;
$WIDGET{$self->{id}} = $self;
}

{ __widget__ => $self->{id} }
}
```

shrink

```
$json = $json->shrink([$enable])
```

```
$enabled = $json->get_shrink
```

In JSON::XS, this flag resizes strings generated by either `encode` or `decode` to their minimum size possible. It will also try to downgrade any strings to octet-form if possible.

In JSON::PP, it is noop about resizing strings but tries `utf8::downgrade` to the returned string by `encode`. See to utf8.

See to "OBJECT-ORIENTED INTERFACE" in JSON::XS

max_depth

```
$json = $json->max_depth([$maximum_nesting_depth])

$max_depth = $json->get_max_depth
```

Sets the maximum nesting level (default `512`) accepted while encoding or decoding. If a higher nesting level is detected in JSON text or a Perl data structure, then the encoder and decoder will stop and croak at that point.

Nesting level is defined by number of hash- or arrayrefs that the encoder needs to traverse to reach a given point or the number of `{` or `[` characters without their matching closing parenthesis crossed to reach a given character in a string.

If no argument is given, the highest possible setting will be used, which is rarely useful.

See "SSECURITY CONSIDERATIONS" in JSON::XS for more info on why this is useful.

When a large value (100 or more) was set and it de/encodes a deep nested object/text, it may raise a warning 'Deep recursion on subroutin' at the perl runtime phase.

max_size

```
$json = $json->max_size([$maximum_string_size])

$max_size = $json->get_max_size
```

Set the maximum length a JSON text may have (in bytes) where decoding is being attempted. The default is `0`, meaning no limit. When `decode` is called on a string that is longer then this many bytes, it will not attempt to decode the string but throw an exception. This setting has no effect on `encode` (yet).

If no argument is given, the limit check will be deactivated (same as when `0` is specified).

See "SSECURITY CONSIDERATIONS" in JSON::XS for more info on why this is useful.

encode

```
$json_text = $json->encode($perl_scalar)
```

Converts the given Perl data structure (a simple scalar or a reference to a hash or array) to its JSON representation. Simple scalars will be converted into JSON string or number sequences, while references to arrays become JSON arrays and references to hashes become JSON objects. Undefined Perl values (e.g. `undef`) become JSON `null` values. References to the integers `0` and `1` are converted into `true` and `false`.

decode

```
$perl_scalar = $json->decode($json_text)
```

The opposite of `encode`: expects a JSON text and tries to parse it, returning the resulting simple scalar or reference. Croaks on error.

JSON numbers and strings become simple Perl scalars. JSON arrays become Perl arrayrefs and JSON objects become Perl hashrefs. `true` becomes `1` (`JSON::true`), `false` becomes `0` (`JSON::false`) and `null` becomes `undef`.

decode_prefix

```
($perl_scalar, $characters) = $json->decode_prefix($json_text)
```

This works like the `decode` method, but instead of raising an exception when there is trailing garbage after the first JSON object, it will silently stop parsing there and return the number of characters consumed so far.

```
JSON->new->decode_prefix ("[1] the tail")
=> ([], 3)
```

INCREMENTAL PARSING

Most of this section are copied and modified from "INCREMENTAL PARSING" in JSON::XS

In some cases, there is the need for incremental parsing of JSON texts. This module does allow you to parse a JSON stream incrementally. It does so by accumulating text until it has a full JSON object, which it then can decode. This process is similar to using `decode_prefix` to see if a full JSON object is available, but is much more efficient (and can be implemented with a minimum of method calls).

This module will only attempt to parse the JSON text once it is sure it has enough text to get a decisive result, using a very simple but truly incremental parser. This means that it sometimes won't stop as early as the full parser, for example, it doesn't detect parenthese mismatches. The only thing it guarantees is that it starts decoding as soon as a syntactically valid JSON text has been seen. This means you need to set resource limits (e.g. `max_size`) to ensure the parser will stop parsing in the presence if syntax errors.

The following methods implement this incremental parser.

incr_parse

```
$json->incr_parse( [$string] ) # void context

$obj_or_undef = $json->incr_parse( [$string] ) # scalar context

@obj_or_empty = $json->incr_parse( [$string] ) # list context
```

This is the central parsing function. It can both append new text and extract objects from the stream accumulated so far (both of these functions are optional).

If `$string` is given, then this string is appended to the already existing JSON fragment stored in the `$json` object.

After that, if the function is called in void context, it will simply return without doing anything further. This can be used to add more text in as many chunks as you want.

If the method is called in scalar context, then it will try to extract exactly *one* JSON object. If that is successful, it will return this object, otherwise it will return `undef`. If there is a parse error, this method will croak just as `decode` would do (one can then use `incr_skip` to skip the errornous part). This is the most common way of using the method.

And finally, in list context, it will try to extract as many objects from the stream as it can find and return them, or the empty list otherwise. For this to work, there must be no separators between the JSON objects or arrays, instead they must be concatenated back-to-back. If an error occurs, an exception will be raised as in the scalar context case. Note that in this case, any previously-parsed JSON texts will be lost.

Example: Parse some JSON arrays/objects in a given string and return them.

```
my @objs = JSON->new->incr_parse ("[5][7][1,2]");
```

incr_text

```
$lvalue_string = $json->incr_text
```

This method returns the currently stored JSON fragment as an lvalue, that is, you can manipulate it. This *only* works when a preceding call to `incr_parse` in *scalar context* successfully returned an object. Under all other circumstances you must not call this function (I mean it. although in simple tests it might actually work, it *will* fail under real world conditions). As a special

exception, you can also call this method before having parsed anything.

This function is useful in two cases: a) finding the trailing text after a JSON object or b) parsing multiple JSON objects separated by non-JSON text (such as commas).

```
$json->incr_text = s/\s*,\s*//;
```

In Perl 5.005, `lvalue` attribute is not available. You must write codes like the below:

```
$string = $json->incr_text;
$string = s/\s*,\s*//;
$json->incr_text( $string );
```

incr_skip

```
$json->incr_skip
```

This will reset the state of the incremental parser and will remove the parsed text from the input buffer. This is useful after `incr_parse` died, in which case the input buffer and incremental parser state is left unchanged, to skip the text parsed so far and to reset the parse state.

incr_reset

```
$json->incr_reset
```

This completely resets the incremental parser, that is, after this call, it will be as if the parser had never parsed anything.

This is useful if you want ot repeatedly parse JSON objects and want to ignore any trailing data, which means you have to reset the parser after each successful decode.

See to "INCREMENTAL PARSING" in JSON::XS for examples.

JSON::PP OWN METHODS

allow_singlequote

```
$json = $json->allow_singlequote([$enable])
```

If `$enable` is true (or missing), then `decode` will accept JSON strings quoted by single quotations that are invalid JSON format.

```
$json->allow_singlequote->decode({"foo":'bar'});
$json->allow_singlequote->decode({'foo':"bar"});
$json->allow_singlequote->decode({'foo':'bar'});
```

As same as the `relaxed` option, this option may be used to parse application-specific files written by humans.

allow_barekey

```
$json = $json->allow_barekey([$enable])
```

If `$enable` is true (or missing), then `decode` will accept bare keys of JSON object that are invalid JSON format.

As same as the `relaxed` option, this option may be used to parse application-specific files written by humans.

```
$json->allow_barekey->decode('{foo:"bar"}');
```

allow_bignum

```
$json = $json->allow_bignum([$enable])
```

If `$enable` is true (or missing), then `decode` will convert the big integer Perl cannot handle as integer into a Math::BigInt object and convert a floating number (any) into a Math::BigFloat.

On the contary, `encode` converts `Math::BigInt` objects and `Math::BigFloat` objects into JSON numbers with `allow_blessed` enable.

```
$json->allow_nonref->allow_blessed->allow_bignum;
$bigfloat = $json->decode('2.000000000000000000000000001');
print $json->encode($bigfloat);
# => 2.000000000000000000000000001
```

See to "MAPPING" in JSON::XS aboout the normal conversion of JSON number.

loose

```
$json = $json->loose([$enable])
```

The unescaped [x00-x1fx22x2fx5c] strings are invalid in JSON strings and the module doesn't allow to `decode` to these (except for x2f). If `$enable` is true (or missing), then `decode` will accept these unescaped strings.

```
$json->loose->decode(qq|["abc
def"]|);
```

See "SSECURITY CONSIDERATIONS" in JSON::XS

escape_slash

```
$json = $json->escape_slash([$enable])
```

According to JSON Grammar, *slash* (U+002F) is escaped. But default JSON::PP (as same as JSON::XS encodes strings without escaping slash.

If `$enable` is true (or missing), then `encode` will escape slashes.

indent_length

```
$json = $json->indent_length($length)
```

JSON::XS indent space length is 3 and cannot be changed. JSON::PP set the indent space length with the given `$length`. The default is 3. The acceptable range is 0 to 15.

sort_by

```
$json = $json->sort_by($function_name)
$json = $json->sort_by($subroutine_ref)
```

If `$function_name` or `$subroutine_ref` are set, its sort routine are used in encoding JSON objects.

```
$js = $pc->sort_by(sub { $JSON::PP::a cmp $JSON::PP::b })->encode($obj);
# is($js, q|{"a":1,"b":2,"c":3,"d":4,"e":5,"f":6,"g":7,"h":8,"i":9}|);

$js = $pc->sort_by('own_sort')->encode($obj);
# is($js, q|{"a":1,"b":2,"c":3,"d":4,"e":5,"f":6,"g":7,"h":8,"i":9}|);

sub JSON::PP::own_sort { $JSON::PP::a cmp $JSON::PP::b }
```

As the sorting routine runs in the JSON::PP scope, the given subroutine name and the special variables `$a`, `$b` will begin 'JSON::PP::'.

If `$integer` is set, then the effect is same as `canonical` on.

INTERNAL

For developers.

PP_encode_box

Returns

```
{
depth => $depth,
indent_count => $indent_count,
}
```

PP_decode_box
Returns

```
{
text => $text,
at => $at,
ch => $ch,
len => $len,
depth => $depth,
encoding => $encoding,
is_valid_utf8 => $is_valid_utf8,
};
```

MAPPING

This section is copied from JSON::XS and modified to `JSON::PP` JSON::XS and JSON::PP mapping mechanisms are almost equivalent.

See to "MAPPING" in JSON::XS

JSON -> PERL

object
A JSON object becomes a reference to a hash in Perl. No ordering of object keys is preserved (JSON does not preserver object key ordering itself).

array
A JSON array becomes a reference to an array in Perl.

string
A JSON string becomes a string scalar in Perl - Unicode codepoints in JSON are represented by the same codepoints in the Perl string, so no manual decoding is necessary.

number
A JSON number becomes either an integer, numeric (floating point) or string scalar in perl, depending on its range and any fractional parts. On the Perl level, there is no difference between those as Perl handles all the conversion details, but an integer may take slightly less memory and might represent more values exactly than floating point numbers.

If the number consists of digits only, `JSON` will try to represent it as an integer value. If that fails, it will try to represent it as a numeric (floating point) value if that is possible without loss of precision. Otherwise it will preserve the number as a string value (in which case you lose roundtripping ability, as the JSON number will be re-encoded toa JSON string).

Numbers containing a fractional or exponential part will always be represented as numeric (floating point) values, possibly at a loss of precision (in which case you might lose perfect roundtripping ability, but the JSON number will still be re-encoded as a JSON number).

Note that precision is not accuracy - binary floating point values cannot represent most decimal fractions exactly, and when converting from and to floating point, `JSON` only guarantees precision up to but not including the leats significant bit.

When `allow_bignum` is enable, the big integers and the numeric can be optionally converted into Math::BigInt and Math::BigFloat objects.

true, false
These JSON atoms become `JSON::PP::true` and `JSON::PP::false` respectively. They are overloaded to act almost exactly like the numbers `1` and `0`. You can check wether a scalar is a JSON boolean by using the `JSON::is_bool` function.

```
print JSON::PP::true . "\n";
=> true
print JSON::PP::true + 1;
=> 1

ok(JSON::true eq '1');
ok(JSON::true == 1);
```

`JSON` will install these missing overloading features to the backend modules.

null
: A JSON null atom becomes `undef` in Perl.

  `JSON::PP::null` returns `unddef`.

PERL –> JSON

The mapping from Perl to JSON is slightly more difficult, as Perl is a truly typeless language, so we can only guess which JSON type is meant by a Perl value.

hash references
: Perl hash references become JSON objects. As there is no inherent ordering in hash keys (or JSON objects), they will usually be encoded in a pseudo-random order that can change between runs of the same program but stays generally the same within a single run of a program. `JSON` optionally sort the hash keys (determined by the *canonical* flag), so the same datastructure will serialise to the same JSON text (given same settings and version of JSON::XS but this incurs a runtime overhead and is only rarely useful, e.g. when you want to compare some JSON text against another for equality.

array references
: Perl array references become JSON arrays.

other references
: Other unblessed references are generally not allowed and will cause an exception to be thrown, except for references to the integers `0` and `1`, which get turned into `false` and `true` atoms in JSON. You can also use `JSON::false` and `JSON::true` to improve readability.

  ```
  to_json [\0,JSON::PP::true] # yields [false,true]
  ```

JSON::PP::true, JSON::PP::false, JSON::PP::null
: These special values become JSON true and JSON false values, respectively. You can also use `\1` and `\0` directly if you want.

  JSON::PP::null returns `undef`.

blessed objects
: Blessed objects are not directly representable in JSON. See the `allow_blessed` and `convert_blessed` methods on various options on how to deal with this: basically, you can choose between throwing an exception, encoding the reference as if it weren't blessed, or provide your own serialiser method.

  See to convert_blessed.

simple scalars
: Simple Perl scalars (any scalar that is not a reference) are the most difficult objects to encode: JSON::XS and JSON::PP will encode undefined scalars as JSON `null` values, scalars that have last been used in a string context before encoding as JSON strings, and anything else as number value:

```
# dump as number
encode_json [2] # yields [2]
encode_json [-3.0e17] # yields [-3e+17]
my $value = 5; encode_json [$value] # yields [5]

# used as string, so dump as string
print $value;
encode_json [$value] # yields ["5"]

# undef becomes null
encode_json [undef] # yields [null]
```

You can force the type to be a string by stringifying it:

```
my $x = 3.1; # some variable containing a number
"$x"; # stringified
$x .= ""; # another, more awkward way to stringify
print $x; # perl does it for you, too, quite often
```

You can force the type to be a number by numifying it:

```
my $x = "3"; # some variable containing a string
$x += 0; # numify it, ensuring it will be dumped as a number
$x *= 1; # same thing, the choise is yours.
```

You can not currently force the type in other, less obscure, ways.

Note that numerical precision has the same meaning as under Perl (so binary to decimal conversion follows the same rules as in Perl, which can differ to other languages). Also, your perl interpreter might expose extensions to the floating point numbers of your platform, such as infinities or NaN's - these cannot be represented in JSON, and it is an error to pass those in.

Big Number

When `allow_bignum` is enable, `encode` converts `Math::BigInt` objects and `Math::BigFloat` objects into JSON numbers.

UNICODE HANDLING ON PERLS

If you do not know about Unicode on Perl well, please check "A FEW NOTES ON UNICODE AND PERL" in JSON::XS

Perl 5.8 and later

Perl can handle Unicode and the JSON::PP de/encode methods also work properly.

```
$json->allow_nonref->encode(chr hex 3042);
$json->allow_nonref->encode(chr hex 12345);
```

Reuturns `"\u3042"` and `"\ud808\udf45"` respectively.

```
$json->allow_nonref->decode('"\u3042"');
$json->allow_nonref->decode('"\ud808\udf45"');
```

Returns UTF-8 encoded strings with UTF8 flag, regarded as `U+3042` and `U+12345`.

Note that the versions from Perl 5.8.0 to 5.8.2, Perl built-in `join` was broken, so JSON::PP wraps the `join` with a subroutine. Thus JSON::PP works slow in the versions.

Perl 5.6

Perl can handle Unicode and the JSON::PP de/encode methods also work.

Perl 5.005

Perl 5.005 is a byte sementics world — all strings are sequences of bytes. That means the unicode handling is not available.

In encoding,

```
$json->allow_nonref->encode(chr hex 3042); # hex 3042 is 12354.
$json->allow_nonref->encode(chr hex 12345); # hex 12345 is 74565.
```

Returns `B` and `E`, as `chr` takes a value more than 255, it treats as `$value % 256`, so the above codes are equivalent to :

```
$json->allow_nonref->encode(chr 66);
$json->allow_nonref->encode(chr 69);
```

In decoding,

```
$json->decode('"\u00e3\u0081\u0082"');
```

The returned is a byte sequence `0xE3 0x81 0x82` for UTF-8 encoded japanese character (`HIRAGANA LETTER A`). And if it is represented in Unicode code point, `U+3042`.

Next,

```
$json->decode('"\u3042"');
```

We ordinary expect the returned value is a Unicode character `U+3042`. But here is 5.005 world. This is `0xE3 0x81 0x82`.

```
$json->decode('"\ud808\udf45"');
```

This is not a character `U+12345` but bytes - `0xf0 0x92 0x8d 0x85`.

TODO

speed
memory saving

SEE ALSO

Most of the document are copied and modified from JSON::XS doc.

JSON::XS

RFC4627 (<http://www.ietf.org/rfc/rfc4627.txt>)

AUTHOR

Makamaka Hannyaharamitu, <makamaka[at]cpan.org>

COPYRIGHT AND LICENSE