-- -\*- coding: utf-8 -\*-

--

-- Simple JSON encoding and decoding in pure Lua.

--

-- Copyright 2010-2014 Jeffrey Friedl

-- http://regex.info/blog/

--

-- Latest version: http://regex.info/blog/lua/json

--

-- This code is released under a Creative Commons CC-BY "Attribution" License:

-- http://creativecommons.org/licenses/by/3.0/deed.en\_US

--

-- It can be used for any purpose so long as the copyright notice above,

-- the web-page links above, and the 'AUTHOR\_NOTE' string below are

-- maintained. Enjoy.

--

local VERSION = 20141223.14 -- version history at end of file

local AUTHOR\_NOTE = "-[ JSON.lua package by Jeffrey Friedl (http://regex.info/blog/lua/json) version 20141223.14 ]-"

--

-- The 'AUTHOR\_NOTE' variable exists so that information about the source

-- of the package is maintained even in compiled versions. It's also

-- included in OBJDEF below mostly to quiet warnings about unused variables.

--

local OBJDEF = {

VERSION = VERSION,

AUTHOR\_NOTE = AUTHOR\_NOTE,

}

--

-- Simple JSON encoding and decoding in pure Lua.

-- http://www.json.org/

--

--

-- JSON = assert(loadfile "JSON.lua")() -- one-time load of the routines

--

-- local lua\_value = JSON:decode(raw\_json\_text)

--

-- local raw\_json\_text = JSON:encode(lua\_table\_or\_value)

-- local pretty\_json\_text = JSON:encode\_pretty(lua\_table\_or\_value) -- "pretty printed" version for human readability

--

--

--

-- DECODING (from a JSON string to a Lua table)

--

--

-- JSON = assert(loadfile "JSON.lua")() -- one-time load of the routines

--

-- local lua\_value = JSON:decode(raw\_json\_text)

--

-- If the JSON text is for an object or an array, e.g.

-- { "what": "books", "count": 3 }

-- or

-- [ "Larry", "Curly", "Moe" ]

--

-- the result is a Lua table, e.g.

-- { what = "books", count = 3 }

-- or

-- { "Larry", "Curly", "Moe" }

--

--

-- The encode and decode routines accept an optional second argument,

-- "etc", which is not used during encoding or decoding, but upon error

-- is passed along to error handlers. It can be of any type (including nil).

--

--

--

-- ERROR HANDLING

--

-- With most errors during decoding, this code calls

--

-- JSON:onDecodeError(message, text, location, etc)

--

-- with a message about the error, and if known, the JSON text being

-- parsed and the byte count where the problem was discovered. You can

-- replace the default JSON:onDecodeError() with your own function.

--

-- The default onDecodeError() merely augments the message with data

-- about the text and the location if known (and if a second 'etc'

-- argument had been provided to decode(), its value is tacked onto the

-- message as well), and then calls JSON.assert(), which itself defaults

-- to Lua's built-in assert(), and can also be overridden.

--

-- For example, in an Adobe Lightroom plugin, you might use something like

--

-- function JSON:onDecodeError(message, text, location, etc)

-- LrErrors.throwUserError("Internal Error: invalid JSON data")

-- end

--

-- or even just

--

-- function JSON.assert(message)

-- LrErrors.throwUserError("Internal Error: " .. message)

-- end

--

-- If JSON:decode() is passed a nil, this is called instead:

--

-- JSON:onDecodeOfNilError(message, nil, nil, etc)

--

-- and if JSON:decode() is passed HTML instead of JSON, this is called:

--

-- JSON:onDecodeOfHTMLError(message, text, nil, etc)

--

-- The use of the fourth 'etc' argument allows stronger coordination

-- between decoding and error reporting, especially when you provide your

-- own error-handling routines. Continuing with the the Adobe Lightroom

-- plugin example:

--

-- function JSON:onDecodeError(message, text, location, etc)

-- local note = "Internal Error: invalid JSON data"

-- if type(etc) = 'table' and etc.photo then

-- note = note .. " while processing for " .. etc.photo:getFormattedMetadata('fileName')

-- end

-- LrErrors.throwUserError(note)

-- end

--

-- :

-- :

--

-- for i, photo in ipairs(photosToProcess) do

-- :

-- :

-- local data = JSON:decode(someJsonText, { photo = photo })

-- :

-- :

-- end

--

--

--

--

--

-- DECODING AND STRICT TYPES

--

-- Because both JSON objects and JSON arrays are converted to Lua tables,

-- it's not normally possible to tell which original JSON type a

-- particular Lua table was derived from, or guarantee decode-encode

-- round-trip equivalency.

--

-- However, if you enable strictTypes, e.g.

--

-- JSON = assert(loadfile "JSON.lua")() --load the routines

-- JSON.strictTypes = true

--

-- then the Lua table resulting from the decoding of a JSON object or

-- JSON array is marked via Lua metatable, so that when re-encoded with

-- JSON:encode() it ends up as the appropriate JSON type.

--

-- (This is not the default because other routines may not work well with

-- tables that have a metatable set, for example, Lightroom API calls.)

--

--

-- ENCODING (from a lua table to a JSON string)

--

-- JSON = assert(loadfile "JSON.lua")() -- one-time load of the routines

--

-- local raw\_json\_text = JSON:encode(lua\_table\_or\_value)

-- local pretty\_json\_text = JSON:encode\_pretty(lua\_table\_or\_value) -- "pretty printed" version for human readability

-- local custom\_pretty = JSON:encode(lua\_table\_or\_value, etc, { pretty = true, indent = "| ", align\_keys = false })

--

-- On error during encoding, this code calls:

--

-- JSON:onEncodeError(message, etc)

--

-- which you can override in your local JSON object.

--

-- The 'etc' in the error call is the second argument to encode()

-- and encode\_pretty(), or nil if it wasn't provided.

--

--

-- PRETTY-PRINTING

--

-- An optional third argument, a table of options, allows a bit of

-- configuration about how the encoding takes place:

--

-- pretty = JSON:encode(val, etc, {

-- pretty = true, -- if false, no other options matter

-- indent = " ", -- this provides for a three-space indent per nesting level

-- align\_keys = false, -- see below

-- })

--

-- encode() and encode\_pretty() are identical except that encode\_pretty()

-- provides a default options table if none given in the call:

--

-- { pretty = true, align\_keys = false, indent = " " }

--

-- For example, if

--

-- JSON:encode(data)

--

-- produces:

--

-- {"city":"Kyoto","climate":{"avg\_temp":16,"humidity":"high","snowfall":"minimal"},"country":"Japan","wards":11}

--

-- then

--

-- JSON:encode\_pretty(data)

--

-- produces:

--

-- {

-- "city": "Kyoto",

-- "climate": {

-- "avg\_temp": 16,

-- "humidity": "high",

-- "snowfall": "minimal"

-- },

-- "country": "Japan",

-- "wards": 11

-- }

--

-- The following three lines return identical results:

-- JSON:encode\_pretty(data)

-- JSON:encode\_pretty(data, nil, { pretty = true, align\_keys = false, indent = " " })

-- JSON:encode (data, nil, { pretty = true, align\_keys = false, indent = " " })

--

-- An example of setting your own indent string:

--

-- JSON:encode\_pretty(data, nil, { pretty = true, indent = "| " })

--

-- produces:

--

-- {

-- | "city": "Kyoto",

-- | "climate": {

-- | | "avg\_temp": 16,

-- | | "humidity": "high",

-- | | "snowfall": "minimal"

-- | },

-- | "country": "Japan",

-- | "wards": 11

-- }

--

-- An example of setting align\_keys to true:

--

-- JSON:encode\_pretty(data, nil, { pretty = true, indent = " ", align\_keys = true })

--

-- produces:

--

-- {

-- "city": "Kyoto",

-- "climate": {

-- "avg\_temp": 16,

-- "humidity": "high",

-- "snowfall": "minimal"

-- },

-- "country": "Japan",

-- "wards": 11

-- }

--

-- which I must admit is kinda ugly, sorry. This was the default for

-- encode\_pretty() prior to version 20141223.14.

--

--

-- AMBIGUOUS SITUATIONS DURING THE ENCODING

--

-- During the encode, if a Lua table being encoded contains both string

-- and numeric keys, it fits neither JSON's idea of an object, nor its

-- idea of an array. To get around this, when any string key exists (or

-- when non-positive numeric keys exist), numeric keys are converted to

-- strings.

--

-- For example,

-- JSON:encode({ "one", "two", "three", SOMESTRING = "some string" }))

-- produces the JSON object

-- {"1":"one","2":"two","3":"three","SOMESTRING":"some string"}

--

-- To prohibit this conversion and instead make it an error condition, set

-- JSON.noKeyConversion = true

--

--

-- SUMMARY OF METHODS YOU CAN OVERRIDE IN YOUR LOCAL LUA JSON OBJECT

--

-- assert

-- onDecodeError

-- onDecodeOfNilError

-- onDecodeOfHTMLError

-- onEncodeError

--

-- If you want to create a separate Lua JSON object with its own error handlers,

-- you can reload JSON.lua or use the :new() method.

--

---------------------------------------------------------------------------

local default\_pretty\_indent = " "

local default\_pretty\_options = { pretty = true, align\_keys = false, indent = default\_pretty\_indent }

local isArray = { \_\_tostring = function() return "JSON array" end } isArray.\_\_index = isArray

local isObject = { \_\_tostring = function() return "JSON object" end } isObject.\_\_index = isObject

function OBJDEF:newArray(tbl)

return setmetatable(tbl or {}, isArray)

end

function OBJDEF:newObject(tbl)

return setmetatable(tbl or {}, isObject)

end

local function unicode\_codepoint\_as\_utf8(codepoint)

--

-- codepoint is a number

--

if codepoint <= 127 then

return string.char(codepoint)

elseif codepoint <= 2047 then

--

-- 110yyyxx 10xxxxxx <-- useful notation from http://en.wikipedia.org/wiki/Utf8

--

local highpart = math.floor(codepoint / 0x40)

local lowpart = codepoint - (0x40 \* highpart)

return string.char(0xC0 + highpart,

0x80 + lowpart)

elseif codepoint <= 65535 then

--

-- 1110yyyy 10yyyyxx 10xxxxxx

--

local highpart = math.floor(codepoint / 0x1000)

local remainder = codepoint - 0x1000 \* highpart

local midpart = math.floor(remainder / 0x40)

local lowpart = remainder - 0x40 \* midpart

highpart = 0xE0 + highpart

midpart = 0x80 + midpart

lowpart = 0x80 + lowpart

--

-- Check for an invalid character (thanks Andy R. at Adobe).

-- See table 3.7, page 93, in http://www.unicode.org/versions/Unicode5.2.0/ch03.pdf#G28070

--

if ( highpart == 0xE0 and midpart < 0xA0 ) or

( highpart == 0xED and midpart > 0x9F ) or

( highpart == 0xF0 and midpart < 0x90 ) or

( highpart == 0xF4 and midpart > 0x8F )

then

return "?"

else

return string.char(highpart,

midpart,

lowpart)

end

else

--

-- 11110zzz 10zzyyyy 10yyyyxx 10xxxxxx

--

local highpart = math.floor(codepoint / 0x40000)

local remainder = codepoint - 0x40000 \* highpart

local midA = math.floor(remainder / 0x1000)

remainder = remainder - 0x1000 \* midA

local midB = math.floor(remainder / 0x40)

local lowpart = remainder - 0x40 \* midB

return string.char(0xF0 + highpart,

0x80 + midA,

0x80 + midB,

0x80 + lowpart)

end

end

function OBJDEF:onDecodeError(message, text, location, etc)

if text then

if location then

message = string.format("%s at char %d of: %s", message, location, text)

else

message = string.format("%s: %s", message, text)

end

end

if etc ~= nil then

message = message .. " (" .. OBJDEF:encode(etc) .. ")"

end

if self.assert then

self.assert(false, message)

else

assert(false, message)

end

end

OBJDEF.onDecodeOfNilError = OBJDEF.onDecodeError

OBJDEF.onDecodeOfHTMLError = OBJDEF.onDecodeError

function OBJDEF:onEncodeError(message, etc)

if etc ~= nil then

message = message .. " (" .. OBJDEF:encode(etc) .. ")"

end

if self.assert then

self.assert(false, message)

else

assert(false, message)

end

end

local function grok\_number(self, text, start, etc)

--

-- Grab the integer part

--

local integer\_part = text:match('^-?[1-9]%d\*', start)

or text:match("^-?0", start)

if not integer\_part then

self:onDecodeError("expected number", text, start, etc)

end

local i = start + integer\_part:len()

--

-- Grab an optional decimal part

--

local decimal\_part = text:match('^%.%d+', i) or ""

i = i + decimal\_part:len()

--

-- Grab an optional exponential part

--

local exponent\_part = text:match('^[eE][-+]?%d+', i) or ""

i = i + exponent\_part:len()

local full\_number\_text = integer\_part .. decimal\_part .. exponent\_part

local as\_number = tonumber(full\_number\_text)

if not as\_number then

self:onDecodeError("bad number", text, start, etc)

end

return as\_number, i

end

local function grok\_string(self, text, start, etc)

if text:sub(start,start) ~= '"' then

self:onDecodeError("expected string's opening quote", text, start, etc)

end

local i = start + 1 -- +1 to bypass the initial quote

local text\_len = text:len()

local VALUE = ""

while i <= text\_len do

local c = text:sub(i,i)

if c == '"' then

return VALUE, i + 1

end

if c ~= '\\' then

VALUE = VALUE .. c

i = i + 1

elseif text:match('^\\b', i) then

VALUE = VALUE .. "\b"

i = i + 2

elseif text:match('^\\f', i) then

VALUE = VALUE .. "\f"

i = i + 2

elseif text:match('^\\n', i) then

VALUE = VALUE .. "\n"

i = i + 2

elseif text:match('^\\r', i) then

VALUE = VALUE .. "\r"

i = i + 2

elseif text:match('^\\t', i) then

VALUE = VALUE .. "\t"

i = i + 2

else

local hex = text:match('^\\u([0123456789aAbBcCdDeEfF][0123456789aAbBcCdDeEfF][0123456789aAbBcCdDeEfF][0123456789aAbBcCdDeEfF])', i)

if hex then

i = i + 6 -- bypass what we just read

-- We have a Unicode codepoint. It could be standalone, or if in the proper range and

-- followed by another in a specific range, it'll be a two-code surrogate pair.

local codepoint = tonumber(hex, 16)

if codepoint >= 0xD800 and codepoint <= 0xDBFF then

-- it's a hi surrogate... see whether we have a following low

local lo\_surrogate = text:match('^\\u([dD][cdefCDEF][0123456789aAbBcCdDeEfF][0123456789aAbBcCdDeEfF])', i)

if lo\_surrogate then

i = i + 6 -- bypass the low surrogate we just read

codepoint = 0x2400 + (codepoint - 0xD800) \* 0x400 + tonumber(lo\_surrogate, 16)

else

-- not a proper low, so we'll just leave the first codepoint as is and spit it out.

end

end

VALUE = VALUE .. unicode\_codepoint\_as\_utf8(codepoint)

else

-- just pass through what's escaped

VALUE = VALUE .. text:match('^\\(.)', i)

i = i + 2

end

end

end

self:onDecodeError("unclosed string", text, start, etc)

end

local function skip\_whitespace(text, start)

local \_, match\_end = text:find("^[ \n\r\t]+", start) -- [http://www.ietf.org/rfc/rfc4627.txt] Section 2

if match\_end then

return match\_end + 1

else

return start

end

end

local grok\_one -- assigned later

local function grok\_object(self, text, start, etc)

if text:sub(start,start) ~= '{' then

self:onDecodeError("expected '{'", text, start, etc)

end

local i = skip\_whitespace(text, start + 1) -- +1 to skip the '{'

local VALUE = self.strictTypes and self:newObject { } or { }

if text:sub(i,i) == '}' then

return VALUE, i + 1

end

local text\_len = text:len()

while i <= text\_len do

local key, new\_i = grok\_string(self, text, i, etc)

i = skip\_whitespace(text, new\_i)

if text:sub(i, i) ~= ':' then

self:onDecodeError("expected colon", text, i, etc)

end

i = skip\_whitespace(text, i + 1)

local new\_val, new\_i = grok\_one(self, text, i)

VALUE[key] = new\_val

--

-- Expect now either '}' to end things, or a ',' to allow us to continue.

--

i = skip\_whitespace(text, new\_i)

local c = text:sub(i,i)

if c == '}' then

return VALUE, i + 1

end

if text:sub(i, i) ~= ',' then

self:onDecodeError("expected comma or '}'", text, i, etc)

end

i = skip\_whitespace(text, i + 1)

end

self:onDecodeError("unclosed '{'", text, start, etc)

end

local function grok\_array(self, text, start, etc)

if text:sub(start,start) ~= '[' then

self:onDecodeError("expected '['", text, start, etc)

end

local i = skip\_whitespace(text, start + 1) -- +1 to skip the '['

local VALUE = self.strictTypes and self:newArray { } or { }

if text:sub(i,i) == ']' then

return VALUE, i + 1

end

local VALUE\_INDEX = 1

local text\_len = text:len()

while i <= text\_len do

local val, new\_i = grok\_one(self, text, i)

-- can't table.insert(VALUE, val) here because it's a no-op if val is nil

VALUE[VALUE\_INDEX] = val

VALUE\_INDEX = VALUE\_INDEX + 1

i = skip\_whitespace(text, new\_i)

--

-- Expect now either ']' to end things, or a ',' to allow us to continue.

--

local c = text:sub(i,i)

if c == ']' then

return VALUE, i + 1

end

if text:sub(i, i) ~= ',' then

self:onDecodeError("expected comma or '['", text, i, etc)

end

i = skip\_whitespace(text, i + 1)

end

self:onDecodeError("unclosed '['", text, start, etc)

end

grok\_one = function(self, text, start, etc)

-- Skip any whitespace

start = skip\_whitespace(text, start)

if start > text:len() then

self:onDecodeError("unexpected end of string", text, nil, etc)

end

if text:find('^"', start) then

return grok\_string(self, text, start, etc)

elseif text:find('^[-0123456789 ]', start) then

return grok\_number(self, text, start, etc)

elseif text:find('^%{', start) then

return grok\_object(self, text, start, etc)

elseif text:find('^%[', start) then

return grok\_array(self, text, start, etc)

elseif text:find('^true', start) then

return true, start + 4

elseif text:find('^false', start) then

return false, start + 5

elseif text:find('^null', start) then

return nil, start + 4

else

self:onDecodeError("can't parse JSON", text, start, etc)

end

end

function OBJDEF:decode(text, etc)

if type(self) ~= 'table' or self.\_\_index ~= OBJDEF then

OBJDEF:onDecodeError("JSON:decode must be called in method format", nil, nil, etc)

end

if text == nil then

self:onDecodeOfNilError(string.format("nil passed to JSON:decode()"), nil, nil, etc)

elseif type(text) ~= 'string' then

self:onDecodeError(string.format("expected string argument to JSON:decode(), got %s", type(text)), nil, nil, etc)

end

if text:match('^%s\*$') then

return nil

end

if text:match('^%s\*<') then

-- Can't be JSON... we'll assume it's HTML

self:onDecodeOfHTMLError(string.format("html passed to JSON:decode()"), text, nil, etc)

end

--

-- Ensure that it's not UTF-32 or UTF-16.

-- Those are perfectly valid encodings for JSON (as per RFC 4627 section 3),

-- but this package can't handle them.

--

if text:sub(1,1):byte() == 0 or (text:len() >= 2 and text:sub(2,2):byte() == 0) then

self:onDecodeError("JSON package groks only UTF-8, sorry", text, nil, etc)

end

local success, value = pcall(grok\_one, self, text, 1, etc)

if success then

return value

else

-- if JSON:onDecodeError() didn't abort out of the pcall, we'll have received the error message here as "value", so pass it along as an assert.

if self.assert then

self.assert(false, value)

else

assert(false, value)

end

-- and if we're still here, return a nil and throw the error message on as a second arg

return nil, value

end

end

local function backslash\_replacement\_function(c)

if c == "\n" then

return "\\n"

elseif c == "\r" then

return "\\r"

elseif c == "\t" then

return "\\t"

elseif c == "\b" then

return "\\b"

elseif c == "\f" then

return "\\f"

elseif c == '"' then

return '\\"'

elseif c == '\\' then

return '\\\\'

else

return string.format("\\u%04x", c:byte())

end

end

local chars\_to\_be\_escaped\_in\_JSON\_string

= '['

.. '"' -- class sub-pattern to match a double quote

.. '%\\' -- class sub-pattern to match a backslash

.. '%z' -- class sub-pattern to match a null

.. '\001' .. '-' .. '\031' -- class sub-pattern to match control characters

.. ']'

local function json\_string\_literal(value)

local newval = value:gsub(chars\_to\_be\_escaped\_in\_JSON\_string, backslash\_replacement\_function)

return '"' .. newval .. '"'

end

local function object\_or\_array(self, T, etc)

--

-- We need to inspect all the keys... if there are any strings, we'll convert to a JSON

-- object. If there are only numbers, it's a JSON array.

--

-- If we'll be converting to a JSON object, we'll want to sort the keys so that the

-- end result is deterministic.

--

local string\_keys = { }

local number\_keys = { }

local number\_keys\_must\_be\_strings = false

local maximum\_number\_key

for key in pairs(T) do

if type(key) == 'string' then

table.insert(string\_keys, key)

elseif type(key) == 'number' then

table.insert(number\_keys, key)

if key <= 0 or key >= math.huge then

number\_keys\_must\_be\_strings = true

elseif not maximum\_number\_key or key > maximum\_number\_key then

maximum\_number\_key = key

end

else

self:onEncodeError("can't encode table with a key of type " .. type(key), etc)

end

end

if #string\_keys == 0 and not number\_keys\_must\_be\_strings then

--

-- An empty table, or a numeric-only array

--

if #number\_keys > 0 then

return nil, maximum\_number\_key -- an array

elseif tostring(T) == "JSON array" then

return nil

elseif tostring(T) == "JSON object" then

return { }

else

-- have to guess, so we'll pick array, since empty arrays are likely more common than empty objects

return nil

end

end

table.sort(string\_keys)

local map

if #number\_keys > 0 then

--

-- If we're here then we have either mixed string/number keys, or numbers inappropriate for a JSON array

-- It's not ideal, but we'll turn the numbers into strings so that we can at least create a JSON object.

--

if self.noKeyConversion then

self:onEncodeError("a table with both numeric and string keys could be an object or array; aborting", etc)

end

--

-- Have to make a shallow copy of the source table so we can remap the numeric keys to be strings

--

map = { }

for key, val in pairs(T) do

map[key] = val

end

table.sort(number\_keys)

--

-- Throw numeric keys in there as strings

--

for \_, number\_key in ipairs(number\_keys) do

local string\_key = tostring(number\_key)

if map[string\_key] == nil then

table.insert(string\_keys , string\_key)

map[string\_key] = T[number\_key]

else

self:onEncodeError("conflict converting table with mixed-type keys into a JSON object: key " .. number\_key .. " exists both as a string and a number.", etc)

end

end

end

return string\_keys, nil, map

end

--

-- Encode

--

-- 'options' is nil, or a table with possible keys:

-- pretty -- if true, return a pretty-printed version

-- indent -- a string (usually of spaces) used to indent each nested level

-- align\_keys -- if true, align all the keys when formatting a table

--

local encode\_value -- must predeclare because it calls itself

function encode\_value(self, value, parents, etc, options, indent)

if value == nil then

return 'null'

elseif type(value) == 'string' then

return json\_string\_literal(value)

elseif type(value) == 'number' then

if value ~= value then

--

-- NaN (Not a Number).

-- JSON has no NaN, so we have to fudge the best we can. This should really be a package option.

--

return "null"

elseif value >= math.huge then

--

-- Positive infinity. JSON has no INF, so we have to fudge the best we can. This should

-- really be a package option. Note: at least with some implementations, positive infinity

-- is both ">= math.huge" and "<= -math.huge", which makes no sense but that's how it is.

-- Negative infinity is properly "<= -math.huge". So, we must be sure to check the ">="

-- case first.

--

return "1e+9999"

elseif value <= -math.huge then

--

-- Negative infinity.

-- JSON has no INF, so we have to fudge the best we can. This should really be a package option.

--

return "-1e+9999"

else

return tostring(value)

end

elseif type(value) == 'boolean' then

return tostring(value)

elseif type(value) ~= 'table' then

self:onEncodeError("can't convert " .. type(value) .. " to JSON", etc)

else

--

-- A table to be converted to either a JSON object or array.

--

local T = value

if type(options) ~= 'table' then

options = {}

end

if type(indent) ~= 'string' then

indent = ""

end

if parents[T] then

self:onEncodeError("table " .. tostring(T) .. " is a child of itself", etc)

else

parents[T] = true

end

local result\_value

local object\_keys, maximum\_number\_key, map = object\_or\_array(self, T, etc)

if maximum\_number\_key then

--

-- An array...

--

local ITEMS = { }

for i = 1, maximum\_number\_key do

table.insert(ITEMS, encode\_value(self, T[i], parents, etc, options, indent))

end

if options.pretty then

result\_value = "[ " .. table.concat(ITEMS, ", ") .. " ]"

else

result\_value = "[" .. table.concat(ITEMS, ",") .. "]"

end

elseif object\_keys then

--

-- An object

--

local TT = map or T

if options.pretty then

local KEYS = { }

local max\_key\_length = 0

for \_, key in ipairs(object\_keys) do

local encoded = encode\_value(self, tostring(key), parents, etc, options, indent)

if options.align\_keys then

max\_key\_length = math.max(max\_key\_length, #encoded)

end

table.insert(KEYS, encoded)

end

local key\_indent = indent .. tostring(options.indent or "")

local subtable\_indent = key\_indent .. string.rep(" ", max\_key\_length) .. (options.align\_keys and " " or "")

local FORMAT = "%s%" .. string.format("%d", max\_key\_length) .. "s: %s"

local COMBINED\_PARTS = { }

for i, key in ipairs(object\_keys) do

local encoded\_val = encode\_value(self, TT[key], parents, etc, options, subtable\_indent)

table.insert(COMBINED\_PARTS, string.format(FORMAT, key\_indent, KEYS[i], encoded\_val))

end

result\_value = "{\n" .. table.concat(COMBINED\_PARTS, ",\n") .. "\n" .. indent .. "}"

else

local PARTS = { }

for \_, key in ipairs(object\_keys) do

local encoded\_val = encode\_value(self, TT[key], parents, etc, options, indent)

local encoded\_key = encode\_value(self, tostring(key), parents, etc, options, indent)

table.insert(PARTS, string.format("%s:%s", encoded\_key, encoded\_val))

end

result\_value = "{" .. table.concat(PARTS, ",") .. "}"

end

else

--

-- An empty array/object... we'll treat it as an array, though it should really be an option

--

result\_value = "[]"

end

parents[T] = false

return result\_value

end

end

function OBJDEF:encode(value, etc, options)

if type(self) ~= 'table' or self.\_\_index ~= OBJDEF then

OBJDEF:onEncodeError("JSON:encode must be called in method format", etc)

end

return encode\_value(self, value, {}, etc, options or nil)

end

function OBJDEF:encode\_pretty(value, etc, options)

if type(self) ~= 'table' or self.\_\_index ~= OBJDEF then

OBJDEF:onEncodeError("JSON:encode\_pretty must be called in method format", etc)

end

return encode\_value(self, value, {}, etc, options or default\_pretty\_options)

end

function OBJDEF.\_\_tostring()

return "JSON encode/decode package"

end

OBJDEF.\_\_index = OBJDEF

function OBJDEF:new(args)

local new = { }

if args then

for key, val in pairs(args) do

new[key] = val

end

end

return setmetatable(new, OBJDEF)

end

return OBJDEF:new()

--

-- Version history:

--

-- 20141223.14 The encode\_pretty() routine produced fine results for small datasets, but isn't really

-- appropriate for anything large, so with help from Alex Aulbach I've made the encode routines

-- more flexible, and changed the default encode\_pretty() to be more generally useful.

--

-- Added a third 'options' argument to the encode() and encode\_pretty() routines, to control

-- how the encoding takes place.

--

-- Updated docs to add assert() call to the loadfile() line, just as good practice so that

-- if there is a problem loading JSON.lua, the appropriate error message will percolate up.

--

-- 20140920.13 Put back (in a way that doesn't cause warnings about unused variables) the author string,

-- so that the source of the package, and its version number, are visible in compiled copies.

--

-- 20140911.12 Minor lua cleanup.

-- Fixed internal reference to 'JSON.noKeyConversion' to reference 'self' instead of 'JSON'.

-- (Thanks to SmugMug's David Parry for these.)

--

-- 20140418.11 JSON nulls embedded within an array were being ignored, such that

-- ["1",null,null,null,null,null,"seven"],

-- would return

-- {1,"seven"}

-- It's now fixed to properly return

-- {1, nil, nil, nil, nil, nil, "seven"}

-- Thanks to "haddock" for catching the error.

--

-- 20140116.10 The user's JSON.assert() wasn't always being used. Thanks to "blue" for the heads up.

--

-- 20131118.9 Update for Lua 5.3... it seems that tostring(2/1) produces "2.0" instead of "2",

-- and this caused some problems.

--

-- 20131031.8 Unified the code for encode() and encode\_pretty(); they had been stupidly separate,

-- and had of course diverged (encode\_pretty didn't get the fixes that encode got, so

-- sometimes produced incorrect results; thanks to Mattie for the heads up).

--

-- Handle encoding tables with non-positive numeric keys (unlikely, but possible).

--

-- If a table has both numeric and string keys, or its numeric keys are inappropriate

-- (such as being non-positive or infinite), the numeric keys are turned into

-- string keys appropriate for a JSON object. So, as before,

-- JSON:encode({ "one", "two", "three" })

-- produces the array

-- ["one","two","three"]

-- but now something with mixed key types like

-- JSON:encode({ "one", "two", "three", SOMESTRING = "some string" }))

-- instead of throwing an error produces an object:

-- {"1":"one","2":"two","3":"three","SOMESTRING":"some string"}

--

-- To maintain the prior throw-an-error semantics, set

-- JSON.noKeyConversion = true

--

-- 20131004.7 Release under a Creative Commons CC-BY license, which I should have done from day one, sorry.

--

-- 20130120.6 Comment update: added a link to the specific page on my blog where this code can

-- be found, so that folks who come across the code outside of my blog can find updates

-- more easily.

--

-- 20111207.5 Added support for the 'etc' arguments, for better error reporting.

--

-- 20110731.4 More feedback from David Kolf on how to make the tests for Nan/Infinity system independent.

--

-- 20110730.3 Incorporated feedback from David Kolf at http://lua-users.org/wiki/JsonModules:

--

-- \* When encoding lua for JSON, Sparse numeric arrays are now handled by

-- spitting out full arrays, such that

-- JSON:encode({"one", "two", [10] = "ten"})

-- returns

-- ["one","two",null,null,null,null,null,null,null,"ten"]

--

-- In 20100810.2 and earlier, only up to the first non-null value would have been retained.

--

-- \* When encoding lua for JSON, numeric value NaN gets spit out as null, and infinity as "1+e9999".

-- Version 20100810.2 and earlier created invalid JSON in both cases.

--

-- \* Unicode surrogate pairs are now detected when decoding JSON.

--

-- 20100810.2 added some checking to ensure that an invalid Unicode character couldn't leak in to the UTF-8 encoding

--

-- 20100731.1 initial public release

--