JSON-delta Documentation

Release 2.0

Philip J. Roberts

Dec 10, 2017

Contents

JSON-delta is a multi-language software suite for computing deltas between JSON-serialized data structures, and applying those deltas as patches. It enables separate programs at either end of a communications channel (e.g. client and server over HTTP, or two processes using IPC) to manipulate the same data structure while minimizing communications overhead.

If you're not clear what this means, or you simply can't see the point, see *JSON-delta by example* for a (somewhat whimsical) exposition of the basic idea.

If, on the other hand, you're fully sold and want to know how to *use* one of the available implementations of JSON-delta, see *The JSON-delta API*, which documents the functions every implementation makes available. This is something like a "standard" for the JSON-delta "API" (I'll consider that the software has outgrown these quotes when it gets more users than just me...) Since the Python implementation is the most developed, if you're thinking in terms of standards, you can consider it the reference implementation.

Further documentation of individual implementations is also available, along with manpages for the CLI programs *json_diff(1)* and *json_patch(1)*

Donations to support the continuing development of JSON-delta will be gratefully received via gratipay, PayPal (himself@phil-roberts.name) or Bitcoin: 1HPJHRpVSm1Y4zrgppd2c6LysjxeabbQN4

CHAPTER 1

News

- Bugfix release 1.1.3 is out. Serious bugs are addressed in this release, so **Javascript users and anyone who relies on the upatch functionality should upgrade**.
- Bugfix release 1.1.2 is out, featuring a sharper distinction between minimal and non-minimal diffs. Non-minimal diffs can now be called for on the command line by running `json_diff --fast`.
- Bugfix release 1.1.1 is out. **Javascript users and anyone who uses udiffs should upgrade**.
- The Heisenbug referred to below has been fixed, along with a more serious bug: in v1.0 of both the Python and Javascript implementations, more than one addition to a non-top-level array—that is, an array nested within one or more other arrays or objects—is not encoded properly in diffs where the *minimal* flag is set to *True*. There was no official release of v1.0 for Python, but for ease of reference I'm calling both the fixed versions 1.1. **It is recommended that all users upgrade**.
- There is a Heisenbug in the udiff part of what I'm retroactively calling v1.0 of the python implementation: it shows up some of the time for one test case, and then only when running python3. Due to this and other bugfixes, I recommed upgrading to v1.1.
- v1.0 of *the Javascript implementation* has now been released. For JSON-format diffs it is now every bit as capable as the python version, and it has been extensively tested, not only against the JSON-delta test suite, but also with JSlint and JShint.

CHAPTER 2

Downloads

The Python implementation (compatible with version 2.7 and later, including 3) is available from PyPI.

You can download the Javascript versions here (full, minified)

(Please, for the sake of my bandwidth bills, serve your own copy rather than hot-linking mine!)

The racket and perl implementations are alpha-quality at best. If you want to check them out, I recommend looking at the source repo (no pun intended): development of JSON-delta takes place against a master repository containing all implementations.

CHAPTER 3

Implementations

The following table summarizes the feature-completeness of the available implementations of JSON-delta, with links to implementation-specific notes for each:

Language	Patch	Diff	Compact	U-patch	U-diff
Python (2.7 and newer)	✓	✓	✓	✓	✓
Javascript	✓	✓	✓		
Racket	✓	✓			

The features described are as follows:

Patch The implementation can manipulate data structures according to a diff in the format specified above.

Diff The implementation can calculate deltas between two data structures in the format specified above.

Compact Diffs produced by the implementation are as small as I can possibly make them, using a variant of Needleman-Wunsch sequence alignment to optimize stanzas modifying JSON arrays.

U-diff The implementation is capable of emitting diffs in a format reminiscent of the output of `diff -u`, which is designed to be more human-readable than the JSON format, to facilitate debugging.

U-patch The implementation can apply U-format patches.

The beginnings of a patch implementation in Perl can also be found in the source repo, but, because Perl doesn't have as ramified a type ontology as Javascript, numeric values do not round-trip cleanly, so work on the Perl implementation is stalled for now.

CHAPTER 4

Further Reading

4.1 JSON-delta by example

Consider the example JSON-LD entry for John Lennon from http://json-ld.org/:

```
{
 "@context": "http://json-ld.org/contexts/person.jsonld",
 "@id": "http://dbpedia.org/resource/John_Lennon",
 "name": "John Lennon",
 "born": "1940-10-09",
 "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
}
```

Suppose we have a piece of software that updates this record to show his date of death, like so:

```
{
 "@context": "http://json-ld.org/contexts/person.jsonld",
 "@id": "http://dbpedia.org/resource/John_Lennon",
 "name": "John Lennon",
 "born": "1940-10-09",

 "died": "1980-12-07",

 "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
 }
```

Further suppose that we wish to communicate this update to another piece of software whose only job is to store information about John Lennon in JSON-LD format. (Yes, I know this is getting unlikely, but stay with me.) If this Lennon-record-keeper accepts updates in json-delta format, all you have to do is send the following over the wire:

```
[["died"],"1980-12-07"]]
```

This is a complete diff in json-delta format. It is itself a JSON-serializable data structure: specifically, it is a sequence of what I refer to as **diff stanzas** for some reason. The format for a diff stanza is `[<key path>, (<update>)]`

(The parentheses mean that the <update> part is optional. I'll get to that in a minute). A key path is a sequence of keys specifying where in the data structure the node you want to alter is found, much like those emitted by JSON.sh. The stanza may be thought of as an instruction to update the node found at that path so that its content is equal to <update>.

Now, let's do some more supposing. Suppose the software we're communicating with is dedicated to storing information about the Beatles in general. Also, suppose we've remembered that it was actually on the 8th of December 1980 that John Lennon died, not the 7th. Finally, suppose we live in an Orwellian dystopia, and Cynthia Lennon has been declared a non-person who must be expunged from all records. Unfortunately, json-delta is incapable of overthrowing corrupt and despotic governments, so let's make one last supposition, that what we're interested in is updating the record kept by the software on the other end of the wire, which looks like this:

```
[
 {
  "@context": "http://json-ld.org/contexts/person.jsonld",
  "@id": "http://dbpedia.org/resource/John_Lennon",
  "name": "John Lennon",
  "born": "1940-10-09",

  "died": "1980-12-07",

  "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
 },
 {"name": "Paul McCartney"},
 {"name": "George Harrison"},
 {"name": "Ringo Starr"}
]
```

(Allegations of bias in favor of specific Beatles on the part of the maintainer of this record are punished by the aforementioned despotic government. All glory to Arstotzka!)

To make the changes we've decided on (correcting John's date of death, and expunging Cynthia Lennon from the record), we need to send the following sequence:

```
[
 [[0, "died"], "1980-12-08"],
 [[0, "spouse"]]
]
```

Now, of course, you see what I meant when I said I'd tell you why <update> is optional later. If a stanza includes no update material, it is interpreted as an instruction to delete the node the key-path points to.

Note also that there is no difference between a stanza that adds a node, and one that changes one.

The intention is to save as much communications bandwidth as possible without sacrificing the ability to communicate arbitrary modifications to the data structure (this format can be used to describe a change from any JSON-serialized object into any other). The worst-case scenario, where there is no commonality between the two structures, is that the protocol adds seven octets of overhead, because a diff can always be expressed as [[[],<target>]], meaning "substitute <target> for the data structure that is to be modified".

4.2 The JSON-delta API

This document is intended to describe the behaviour of the main entry points for every implementation of JSON-delta. For now, it effectively documents the top-level namespace of *the Python implementation*, as that is the most fully-developed implementation in the suite.

4.2.1 Core functions

json_delta.**diff**(*left_struc*, *right_struc*, *minimal=None*, *verbose=True*, *key=None*, *array_align=True*, *compare_lengths=True*, *common_key_threshold=0.0*)

Compose a sequence of diff stanzas sufficient to convert the structure `left_struc` into the structure `right_struc`. (The goal is to add 'necessary and' to 'sufficient' above!).

Optional parameters: `verbose`: Print compression statistics to stderr, and warn if the setting of `minimal` contradicts the other parms.

`array_align`: Use `_diff.needle_diff()` to compute deltas between arrays. Relatively computationally expensive, but likely to produce shorter diffs. Defaults to `True`.

`compare_lengths`: If `[[key, right_struc]]` can be encoded as a shorter JSON-string, return it instead of examining the internal structure of `left_struc` and `right_struc`. It involves calling `json.dumps()` twice for every node in the structure, but may result in smaller diffs. Defaults to `True`.

`common_key_threshold`: Skip recursion into `left_struc` and `right_struc` if the fraction of keys they have in common (with the same value) is less than this parm (which should be a float between `0.0` and `1.0`). Defaults to 0.0.

`minimal`: Included for backwards compatibility. `True` is equivalent to `(array_align=True, compare_lengths=True, common_key_threshold=0.0)`; `False` is equivalent to `(array_align=False, compare_lengths=False, common_key_threshold=0.5)`. Specific settings of `array_align`, `compare_lengths` or `common_key_threshold` will supersede this parm, warning on stderr if `verbose` and `minimal` are both set.

`key`: Also included for backwards compatibility. If set, will be prepended to the key in each stanza of the output.

The parameter `key` is present because this function is mutually recursive with `_diff.needle_diff()` and `_diff.keyset_diff()`. If set to a list, it will be prefixed to every keypath in the output.

json_delta.**patch**(*struc*, *diff*, *in_place=True*)

Apply the sequence of diff stanzas `diff` to the structure `struc`.

By default, this function modifies `struc` in place; set `in_place` to `False` to return a patched copy of struc instead:

```
>>> will_change = [16]
>>> wont_change = [16]
>>> patch(will_change, [[[0]]])
[]
>>> will_change
[]
>>> patch(wont_change, [[[0]]], False)
[]
>>> wont_change
[16]
```

json_delta.**udiff**(*left*, *right*, *patch=None*, *indent=0*, *use_ellipses=True*, *entry=True*)

Render the difference between the structures `left` and `right` as a string in a fashion inspired by **`diff -u`**.

Generating a udiff is strictly slower than generating a normal diff with the same option parameters, since the udiff is computed on the basis of a normal diff between `left` and `right`. If such a diff has already been computed (e.g. by calling *`diff()`*), pass it as the `patch` parameter:

```
>>> (next(udiff({"foo": None}, {"foo": None}, patch=[])) ==
...  ' {...}')
True
```

As you can see above, structures that are identical in `left` and `right` are abbreviated using `'...'` by default. To disable this behavior, set `use_ellipses` to `False`.

```
>>> ('\n'.join(udiff({"foo": None}, {"foo": None},
...             patch=[], use_ellipses=False)) ==
... """ {
...   "foo":
...     null
...   }""")
True
```

```
>>> ('\n'.join(udiff([None, None, None], [None, None, None],
...              patch=[], use_ellipses=False)) ==
... """ [
...    null,
...    null,
...    null
...   ]""")
True
```

`json_delta.`**`upatch`**(*struc*, *udiff*, *reverse=False*, *in_place=True*)

Apply a patch as output by *`json_delta.udiff()`* to `struc`.

As with *`json_delta.patch()`*, `struc` is modified in place by default. Set the parm `in_place` to `False` if this is not the desired behaviour.

The udiff format has enough information in it that this transformation can be applied in reverse: i.e. if `udiff` is the output of `udiff(left, right)`, you can reconstruct `right` given `left` and `udiff` (by running `upatch(left, udiff)`), or you can also reconstruct `left` given `right` and udiff (by running `upatch(right, udiff, reverse=True)`). This is not possible for JSON-format diffs, since a `[keypath]` stanza (meaning "delete the structure at `keypath`") does not record what the deleted structure was.

4.2.2 load_and_*

For convenience when handling input that is already JSON-serialized, implementations should offer entry points named *load_and_{FUNC}*, which deserialize their input and then apply *{FUNC}* to it.

`json_delta.`**`load_and_diff`**(*left=None*, *right=None*, *both=None*, *array_align=None*, *compare_lengths=None*, *common_key_threshold=None*, *minimal=None*, *verbose=True*)

Apply *`diff()`* to strings or files representing JSON-serialized structures.

Specify either `left` and `right`, or `both`, like so:

```
>>> (load_and_diff('{"foo":"bar"}', '{"foo":"baz"}', verbose=False)
...  == [["foo"],"baz"]])
True
>>> (load_and_diff(both='[{"foo":"bar"},{"foo":"baz"}]', verbose=False)
...  == [["foo"],"baz"]])
True
```

`left`, `right` and `both` may be either strings (instances of *basestring* in 2.7) or file-like objects.

`minimal` and `verbose` are passed through to *`diff()`*, which see.

A call to this function with string arguments is strictly equivalent to calling `diff(json.loads(left),`

`json.loads(right), minimal=minimal, verbose=verbose)` or `diff(*json.loads(both), minimal=minimal, verbose=verbose)`, as appropriate.

`json_delta.`**`load_and_patch`**(*struc=None*, *stanzas=None*, *both=None*)

Apply *patch()* to strings or files representing JSON-serialized structures.

Specify either `struc` and `stanzas`, or `both`, like so:

```
>>> (load_and_patch('{"foo":"bar"}', '[[["foo"],"baz"]]') ==
...  {"foo": "baz"})
True
>>> (load_and_patch(both='[{"foo":"bar"},[["foo"],"baz"]]]') ==
...  {"foo": "baz"})
True
```

`struc`, `stanzas` and `both` may be either strings (instances of *basestring* in 2.7) or file-like objects.

A call to this function with string arguments is strictly equivalent to calling `patch(json.loads(struc), json.loads(stanzas), in_place=in_place)` or `patch(*json.loads(both), in_place=in_place)`, as appropriate.

`json_delta.`**`load_and_udiff`**(*left=None*, *right=None*, *both=None*, *stanzas=None*, *indent=0*)

Apply *udiff()* to strings representing JSON-serialized structures.

Specify either `left` and `right`, or `both`, like so:

```
>>> udiff = """ {
...  "foo":
... -  "bar"
... +  "baz"
...  }"""
>>> test = load_and_udiff('{"foo":"bar"}', '{"foo":"baz"}')
>>> '\n'.join(test) == udiff
True
>>> test = load_and_udiff(both='[{"foo":"bar"},{"foo":"baz"}]')
>>> '\n'.join(test) == udiff
True
```

`left`, `right` and `both` may be either strings (instances of *basestring* in 2.7) or file-like objects.

`stanzas` and `indent` are passed through to *udiff()*, which see.

A call to this function with string arguments is strictly equivalent to calling `udiff(json.loads(left), json.loads(right), stanzas=stanzas, indent=indent)` or `udiff(*json.loads(both), stanzas=stanzas, indent=indent)`, as appropriate.

`json_delta.`**`load_and_upatch`**(*struc=None*, *json_udiff=None*, *both=None*, *reverse=False*)

Apply *upatch()* to strings representing JSON-serialized structures.

Specify either `struc` and `json_udiff`, or `both`, like so:

```
>>> struc = '{"foo":"bar"}'
>>> json_udiff = r'" {\n  \"foo\":\n-  \"bar\"\n+  \"baz\"\n }"'
>>> both = r'[{"foo":"baz"}," '\
... r'{\n  \"foo\":\n-  \"bar\"\n+  \"baz\"\n }"]'
>>> load_and_upatch(struc, json_udiff) == {"foo": "baz"}
True
>>> load_and_upatch(both=both, reverse=True) == {"foo": "bar"}
True
```

`struc`, `json_udiff` and `both` may be either strings (instances of *basestring* in 2.7) or file-like objects. Note that `json_udiff` is so named because it must be a JSON-serialized representation of the udiff string, not the udiff string itself.

`reverse` is passed through to *`upatch()`*, which see.

A call to this function with string arguments is strictly equivalent to calling `upatch(json.loads(struc), json.loads(json_udiff), reverse=reverse, in_place=in_place)` or `upatch(*json.loads(both), reverse=reverse, in_place=in_place)`, as appropriate.

4.3 json_diff

4.3.1 Synopsis

```
json_diff [--output FILE] [--verbose] [--unified] [left] [right]
json_diff [--version]
json_diff [--help]
```

4.3.2 Description

json_diff produces deltas between JSON-serialized data structures. If no arguments are specified, stdin will be expected to be a JSON array `[left, right]`, and the output will be written to stdout.

The default output is itself a JSON data structure, specifically an array of arrays of the form `[<keypath>]` or `[<keypath>, <replacement>]`. The companion program *`json_patch(1)`* can be used to apply such a diff.

A keypath is an array of string or integer tokens specifying a path to a terminal node in the data structure. For example, in the structure `[{}, {"foo": "bar"}]`, the string `"bar"` appears at the node addressed by the key sequence `[1, 'foo']`, and the empty object `{}` appears at key sequence `[0]`.

If a diff stanza is an array of length 1, consisting only of a key sequence, *`json_patch(1)`* interprets it as an instruction to delete the node the key sequence points to. If a stanza is of length 2, the node is replaced by the last element of the stanza.

An alternative output format for *json_diff* is accessed using the `--unified` / `-u` option. This is designed to be more legible to the human eye, inspired by unified diffs as output by *`diff(1)`*. *`json_patch(1)`* can read either format, and, since there is enough information in the format, can apply `--unified` patches in reverse.

json_diff will accept input in any of the encodings specified in RFC 7159, namely UTF-8, -16 or -32, with or without byte-order marks. The default encoding for output is UTF-8 with no BOM, but this can be changed using the `--encoding` option.

4.3.3 Options

--output FILE, -o FILE Write output to FILE instead of stdout.

--unified, -u Write diffs in a more legible format, inspired by the output of `diff -u`

--encoding ENCODING Select the encoding for the output.

--verbose Print compression statistics on stderr.

--version Show the program's version number and exit.

--help, -h Show a brief help message and exit.

4.3.4 Examples

```
$ json_diff << 'EOF'
> [{"foo": "bar"},
>  {"foo": "bar",
>   "baz": ["quux"]}]
> EOF
[[["baz"],["quux"]]]

$ cat > foofile << 'EOF'
> {"foods": ["spam", "spam", "spam", "spam"],
>  "weaponry": "Mainly battleaxes.",
>  "spanish inquisition expected": false,
>  "drinks": "Delicious mead!",
>  "other supplies": null}
> EOF
$ cat > barfile << 'EOF'
> {"foods": ["spam", "spam", "spam", "pickled eggs", "spam"],
>  "weaponry": "Mainly battleaxes.",
>  "spanish inquisition expected": false,
>  "drinks": "Soda water."}
> EOF
$ json_diff -u foofile barfile
--- foofile   2014-04-14 21:32:00 BST
+++ barfile   2014-04-14 21:32:17 BST
 {
  "foods":
  ...
  "weaponry": "Mainly battleaxes.",
   ["spam",
    ...(2),
+   "pickled eggs",
    "spam"]

  "drinks":
-  "Delicious mead!",
+  "Soda water.",

- "other supplies": null
 }
```

4.3.5 Implementation Notes

The value of the `--encoding` option in the Python implementation of `json_diff` is fed straight to the `encode()` function, so it is possible to get output in any encoding supported by the Python implementation used to run the script. This makes various mildly interesting things possible, like getting compressed output using `--encoding bz2` or `--encoding zlib`, or even `--encoding rot-13` (Furrfu!)

4.4 json_patch

4.4.1 Synopsis

```
json_patch [--output FILE] [--unified | --normal]
           [--strip [NUM]] [--reverse] [originalfile] [patchfile]
json_patch [--version]
json_patch [--help]
```

4.4.2 Description

json_patch applies diffs in the format produced by *json_diff(1)* to JSON-serialized data structures.

The program attempts to mimic the interface of the *patch(1)* utility as far as possible, while also remaining compatible with the script functionality of the *json_delta.py* library on which it relies. There are, therefore, at least four different ways its input can be specified.

1. The simplest, of course, is if the filenames are both specified as positional arguments.
2. Closely following in terms of simplicity, the inputs can be fed as a JSON array `[<structure>,<patch>]` to standard input.
3. If only one positional argument is specified, it is read as the filename of the original data structure, and the patch is expected to appear on stdin.
4. Finally, if there are no positional arguments, and stdin cannot be parsed as JSON, it can alternatively be a udiff, as output by `json_diff -u`. In this case, json_patch will read the name of the file containing the structure to modify out of the first header line of the udiff (the one beginning with `---`).

The most salient departure from the behavior of *patch(1)* is that, by default, json_patch will **not** modify files in place. Instead, the patched structure is written as JSON to stdout. Frankly, this is to save having to implement backup filename options, getting it wrong, and having angry hackers blame me for their lost data.

However, the input structure is read into memory before the output file handle is opened, so an in-place modification can be accomplished by setting the option `--output` to point to `<originalfile>`.

Also, note that json_diff and json_patch can only manipulate a single file at a time: even the output of `json_diff -u` is not a "unified" diff *sensu stricto*.

json_patch will accept input in any of the encodings specified in RFC 7159, namely UTF-8, -16 or -32, with or without byte-order marks. The default encoding for output is UTF-8 with no BOM, but this can be changed using the `--encoding` option.

4.4.3 Options

--output FILE, -o FILE Write output to FILE instead of stdout.

--unified, -u Force the patch to be interpreted as a udiff.

--normal, -n Force the patch to be interpreted as a normal (i.e. JSON-format) patch

--reverse, -R Assume the patch was created with old and new files swapped.

--strip NUM, -p NUM Strip NUM leading components from file names read out of udiff headers.

--encoding ENCODING Select the encoding for the output.

--version Show the program's version number and exit.

--help, -h Show a brief help message and exit.

4.4.4 Udiff Format

The program has strict requirements of the format of "unified" diffs. It works by discarding header lines, then creating two strings: one by discarding every line beginning with −, then discarding the first character of every remaining line, and one following the same procedure, but with lines beginning with + discarded. For json_patch to function, these strings must be interpretable according to the following superset of the JSON spec:

- Within objects, the string `...` may appear in any context where a `"property": <object>` construction would be valid JSON. This indicates that one or more properties have been omitted from the representation of the object.
- Within arrays, the string `...` may appear as an array element. It may optionally be followed by an integer in parentheses, e.g. `(1)`, `(15)`. This indicates that that number of elements have been omitted from the array, or that one element has, if no parenthesized number is present.

The program reconstructs the JSON-format diff on the basis of these strings, and then applies it to the input structure.

4.4.5 Implementation Notes

The value of the `--encoding` option in the Python implementation of `json_diff` is fed straight to the `encode()` function, so it is possible to get output in any encoding supported by the Python implementation used to run the script. This makes various mildly interesting things possible, like getting compressed output using `--encoding bz2` or `--encoding zlib`, or even `--encoding rot-13` (Furrfu!)

4.5 json_cat

4.5.1 Synopsis

```
json_cat [FILE]...
```

4.5.2 Description

Concatenate FILE(s), or standard input together and write them to standard output as a JSON array.

Input streams are parsed as JSON if possible, otherwise they are added to the array as strings.

Output is always UTF-8 encoded.

4.5.3 Examples

```
$ echo '{"foo": true, "bar": false,
>        "baz": null}' > foofile
$ json_cat foofile - << 'EOF'
> This text cannot be parsed as JSON.
> EOF
[{"foo": true, "bar": false, "baz": null}, "This text cannot be parsed as JSON."]
$ echo 'You can use json_cat to create 1-element JSON arrays of text,
> if that'\''s something you like to do...' | json_cat
["You can use json_cat to create 1-element JSON arrays of text, if that's something␣
↪you like to do..."]
```

4.6 Python implementation notes

The Python implementation of JSON-delta consists of a package *json_delta*, whose top-level namespace is documented in *The JSON-delta API*. The implementation is divided into five sub-modules of the package, whose names all begin with an underscore to highlight the fact that they are not part of the API: the way the functions documented in *The JSON-delta API* are implemented is subject to refactoring at any time. Nevertheless, the sub-modules are documented here.

4.6.1 json_delta._diff

Functions for computing JSON-format diffs.

json_delta._diff.**diff**(*left_struc*, *right_struc*, *array_align=True*, *compare_lengths=True*, *common_key_threshold=0.0*, *verbose=True*, *key=None*)

Compose a sequence of diff stanzas sufficient to convert the structure `left_struc` into the structure `right_struc`. (Whether you can add 'necessary and' to 'sufficient to' depends on the setting of the other parms, and how many cycles you want to burn; see below).

Optional parameters: `array_align`: Use `needle_diff()` to compute deltas between arrays. Computationally expensive, but likely to produce shorter diffs. If this parm is set to the string `'udiff'`, `needle_diff()` will optimize for the shortest udiff, instead of the shortest JSON-format diff. Otherwise, set to any value that is true in a Boolean context to enable.

`compare_lengths`: If at any level `[[key, right_struc]]` can be encoded as a shorter JSON-string, return it instead of examining the internal structure of `left_struc` and `right_struc`. May result in smaller diffs.

`common_key_threshold`: Skip recursion into `left_struc` and `right_struc` if the fraction of keys they have in common (as computed by *commonality()*, which see) is less than this parm (which should be a float between `0.0` and `1.0`).

`verbose`: Print compression statistics to stderr.

The parameter `key` is present because this function is mutually recursive with `needle_diff()` and `keyset_diff()`. If set to a list, it will be prefixed to every keypath in the output.

json_delta._diff.**append_key**(*stanzas*, *left_struc*, *keypath=()*)

Get the appropriate key for appending to the sequence `left_struc`.

`stanzas` should be a diff, some of whose stanzas may modify a sequence `left_struc` that appears at path `keypath`. If any of the stanzas append to `left_struc`, the return value is the largest index in `left_struc` they address, plus one. Otherwise, the return value is `len(left_struc)` (i.e. the index that a value would have if it was appended to `left_struc`).

```
>>> append_key([], [])
0
>>> append_key([[[2], 'Baz']], ['Foo', 'Bar'])
3
>>> append_key([[[2], 'Baz'], [['Quux', 0], 'Foo']], [], ['Quux'])
1
```

json_delta._diff.**commonality**(*left_struc*, *right_struc*)

Return a float between `0.0` and `1.0` representing the amount that the structures `left_struc` and `right_struc` have in common.

Return value is computed as the fraction (elements in common) / (total elements).

json_delta._diff.**compute_diff_stats**(*target*, *diff*, *percent=True*)

Calculate the size of a minimal JSON dump of `target` and `diff`, and the ratio of the two sizes.

The ratio is expressed as a percentage if `percent` is `True` in a Boolean context, or as a float otherwise.

Return value is a tuple of the form `({ratio}, {size of target}, {size of diff})`

```
>>> compute_diff_stats([{}, 'foo', 'bar'], [], False)
(0.125, 16, 2)
>>> compute_diff_stats([{}, 'foo', 'bar'], [[0], {}])
(50.0, 16, 8)
```

json_delta._diff.**compute_keysets**(*left_seq*, *right_seq*)

Compare the keys of `left_seq` vs. `right_seq`.

Determines which keys `left_seq` and `right_seq` have in common, and which are unique to each of the structures. Arguments should be instances of the same basic type, which must be a non-terminal: i.e. `list` or `dict`. If they are lists, the keys compared will be integer indices.

Returns: Return value is a 3-tuple of sets `({overlap}, {left_only}, {right_only})`. As their names suggest, `overlap` is a set of keys `left_seq` have in common, `left_only` represents keys only found in `left_seq`, and `right_only` holds keys only found in `right_seq`.

Raises: AssertionError if `left_seq` is not an instance of `type(right_seq)`, or if they are not of a non-terminal type.

```
>>> (compute_keysets({'foo': None}, {'bar': None})
...  == (set([]), {'foo'}, {'bar'}))
True
>>> (compute_keysets({'foo': None, 'baz': None},
...                  {'bar': None, 'baz': None})
...  == ({'baz'}, {'foo'}, {'bar'}))
True
>>> (compute_keysets(['foo', 'baz'], ['bar', 'baz'])
...  == ({0, 1}, set([]), set([])))
True
>>> compute_keysets(['foo'], ['bar', 'baz']) == ({0}, set([]), {1})
True
>>> compute_keysets([], ['bar', 'baz']) == (set([]), set([]), {0, 1})
True
```

json_delta._diff.**diff**(*left_struc*, *right_struc*, *array_align=True*, *compare_lengths=True*, *common_key_threshold=0.0*, *verbose=True*, *key=None*)

Compose a sequence of diff stanzas sufficient to convert the structure `left_struc` into the structure `right_struc`. (Whether you can add 'necessary and' to 'sufficient to' depends on the setting of the other parms, and how many cycles you want to burn; see below).

Optional parameters: `array_align`: Use `needle_diff()` to compute deltas between arrays. Computationally expensive, but likely to produce shorter diffs. If this parm is set to the string `'udiff'`, `needle_diff()` will optimize for the shortest udiff, instead of the shortest JSON-format diff. Otherwise, set to any value that is true in a Boolean context to enable.

`compare_lengths`: If at any level `[[key, right_struc]]` can be encoded as a shorter JSON-string, return it instead of examining the internal structure of `left_struc` and `right_struc`. May result in smaller diffs.

`common_key_threshold`: Skip recursion into `left_struc` and `right_struc` if the fraction of keys they have in common (as computed by *`commonality()`*, which see) is less than this parm (which should be a float between `0.0` and `1.0`).

`verbose`: Print compression statistics to stderr.

The parameter `key` is present because this function is mutually recursive with `needle_diff()` and `keyset_diff()`. If set to a list, it will be prefixed to every keypath in the output.

`json_delta._diff.`**`sort_stanzas`**(*stanzas*)
Sort the stanzas in a diff.

Object changes can occur in any order, but deletions from arrays have to happen last node first: `['foo', 'bar', 'baz']` → `['foo', 'bar']` → `['foo']` → `[]`; additions to arrays have to happen leftmost-node-first: `[]` → `['foo']` → `['foo', 'bar']` → `['foo', 'bar', 'baz']`, and insert-and-shift alterations to arrays must happen last: `['foo', 'quux']` → `['foo', 'bar', 'quux']` → `['foo', 'bar', 'baz', 'quux']`.

Finally, stanzas are sorted in descending order of *length* of keypath, so that the most deeply-nested structures are altered before alterations which might change their keypaths take place.

Note that this will also sort changes to objects (dicts) so that they occur first of all.

`json_delta._diff.`**`split_diff`**(*stanzas*)
Split a diff into modifications, deletions and insertions.

Return value is a 4-tuple of lists: the first is a list of stanzas from `stanzas` that modify JSON objects, the second is a list of stanzas that add or change elements in JSON arrays, the third is a list of stanzas which delete elements from arrays, and the fourth is a list of stanzas which insert elements into arrays (stanzas ending in `"i"`).

`json_delta._diff.`**`structure_comparable`**(*left_struc*, *right_struc*)
Test if `left_struc` and `right_struc` can be efficiently diffed.

`json_delta._diff.`**`this_level_diff`**(*left_struc*, *right_struc*, *key=None*, *common=None*)
Return a sequence of diff stanzas between the structures `left_struc` and `right_struc`, assuming that they are each at the key-path `key` within the overall structure.

```
>>> (this_level_diff({'foo': 'bar', 'baz': 'quux'},
...                  {'foo': 'bar'})
...  == [[['baz']]])
True
>>> (this_level_diff({'foo': 'bar', 'baz': 'quux'},
...                  {'foo': 'bar'}, ['quordle'])
...  == [[['quordle', 'baz']]])
True
```

4.6.2 json_delta._patch

Functions for applying JSON-format patches.

`json_delta._patch.`**`patch`**(*struc*, *diff*, *in_place=True*)
Apply the sequence of diff stanzas `diff` to the structure `struc`.

By default, this function modifies `struc` in place; set `in_place` to `False` to return a patched copy of struc instead:

```
>>> will_change = [16]
>>> wont_change = [16]
>>> patch(will_change, [[[0]]])
[]
>>> will_change
[]
```

```
>>> patch(wont_change, [[[0]]], False)
[]
>>> wont_change
[16]
```

json_delta._patch.**patch**(*struc*, *diff*, *in_place=True*)

Apply the sequence of diff stanzas `diff` to the structure `struc`.

By default, this function modifies `struc` in place; set `in_place` to `False` to return a patched copy of struc instead:

```
>>> will_change = [16]
>>> wont_change = [16]
>>> patch(will_change, [[[0]]])
[]
>>> will_change
[]
>>> patch(wont_change, [[[0]]], False)
[]
>>> wont_change
[16]
```

json_delta._patch.**patch_stanza**(*struc*, *stanza*)

Applies the stanza `stanza` to the structure `struc` as a patch.

Note that this function modifies `struc` in-place into the target of `stanza`. If `struc` is a `tuple()`, you get a new tuple with the appropriate modification made:

```
>>> patch_stanza((17, 3.141593, None), [[1], 3.14159265])
(17, 3.14159265, None)
```

4.6.3 json_delta._udiff

Functions for computing udiffs. Main entry point: *`udiff()`*.

The data structure representing a udiff that these functions all manipulate is a pair of lists of iterators `(left_lines, right_lines)`. These lists are expected (principally by *`generate_udiff_lines()`*, which processes them), to be of the same length. A pair of iterators `(left_lines[i], right_lines[i])` may yield exactly the same sequence of output lines, each with `' '` as the first character (representing parts of the structure the input and output have in common). Alternatively, they may each yield zero or more lines (referring to parts of the structure that are unique to the inputs they represent). In this case, all lines yielded by `left_lines[i]` should begin with `'-'`, and all lines yielded by `right_lines[i]` should begin with `'+'`.

json_delta._udiff.**udiff**(*left*, *right*, *patch=None*, *indent=0*, *use_ellipses=True*, *entry=True*)

Render the difference between the structures `left` and `right` as a string in a fashion inspired by **`diff -u`**.

Generating a udiff is strictly slower than generating a normal diff with the same option parameters, since the udiff is computed on the basis of a normal diff between `left` and `right`. If such a diff has already been computed (e.g. by calling `diff()`), pass it as the `patch` parameter:

```
>>> (next(udiff({"foo": None}, {"foo": None}, patch=[])) ==
...  ' {...}')
True
```

As you can see above, structures that are identical in `left` and `right` are abbreviated using `'...'` by default. To disable this behavior, set `use_ellipses` to `False`.

```
>>> ('\n'.join(udiff({"foo": None}, {"foo": None},
...             patch=[], use_ellipses=False)) ==
... """ {
...  "foo":
...    null
...  }""")
True
```

```
>>> ('\n'.join(udiff([None, None, None], [None, None, None],
...              patch=[], use_ellipses=False)) ==
... """ [
...   null,
...   null,
...   null
...  ]""")
True
```

class json_delta._udiff.**Gap**
Class to represent gaps introduced by sequence alignment.

json_delta._udiff.**add_matter**(*seq*, *matter*, *indent*)
Add material to seq, treating it appropriately for its type.

matter may be an iterator, in which case it is appended to seq. If it is a sequence, it is assumed to be a sequence of iterators, the sequence is concatenated onto seq. If matter is a string, it is turned into a patch band using *single_patch_band()*, which is appended. Finally, if matter is None, an empty iterable is appended to seq.

This function is a udiff-forming primitive, called by more specific functions defined within *udiff_dict()* and *udiff_list()*.

json_delta._udiff.**commafy**(*gen*, *comma=True*)
Yield from gen, ensuring that the final result ends with a comma iff comma is True.

```
>>> gen = ['Example line']
>>> next(commafy(iter(gen))) == 'Example line,'
True
>>> next(commafy(iter(gen), False)) == 'Example line'
True
>>> gen = ['Line with a comma at the end,']
>>> (next(commafy(iter(gen), comma=True))
...  == next(commafy(iter(gen), comma=False))
...  == 'Line with a comma at the end,')
True
```

json_delta._udiff.**curry_functions**(*local_ns*)
Create partials of _add_common_matter(), _add_differing_matter() and _commafy_last(), with values for left_lines, right_lines and (where appropriate) indent taken from the dictionary local_ns.

Appropriate defaults are also included in the partials, namely left=None and right=None for _add_differing_matter() and left_comma=True and right_comma=None for _commafy_last().

json_delta._udiff.**generate_udiff_lines**(*left*, *right*)
Combine the diff lines from left and right, and generate the lines of the resulting udiff.

json_delta._udiff.**patch_bands**(*indent*, *material*, *sigil=u' '*)
Generate appropriately indented patch bands, with sigil as the first character.

json_delta._udiff.**reconstruct_alignment**(*left*, *right*, *stanzas*)
Reconstruct the sequence alignment between the lists `left` and `right` implied by `stanzas`.

json_delta._udiff.**single_patch_band**(*indent*, *line*, *sigil=u' '*)
Convenience function returning an iterable that generates a single patch band.

json_delta._udiff.**udiff**(*left*, *right*, *patch=None*, *indent=0*, *use_ellipses=True*, *entry=True*)
Render the difference between the structures `left` and `right` as a string in a fashion inspired by **`diff -u`**.

Generating a udiff is strictly slower than generating a normal diff with the same option parameters, since the udiff is computed on the basis of a normal diff between `left` and `right`. If such a diff has already been computed (e.g. by calling `diff()`), pass it as the `patch` parameter:

```
>>> (next(udiff({"foo": None}, {"foo": None}, patch=[])) ==
...  ' {...}')
True
```

As you can see above, structures that are identical in `left` and `right` are abbreviated using `'...'` by default. To disable this behavior, set `use_ellipses` to `False`.

```
>>> ('\n'.join(udiff({"foo": None}, {"foo": None},
...             patch=[], use_ellipses=False)) ==
... """ {
...  "foo":
...    null
...  }""")
True
```

```
>>> ('\n'.join(udiff([None, None, None], [None, None, None],
...              patch=[], use_ellipses=False)) ==
... """ [
...   null,
...   null,
...   null
...  ]""")
True
```

json_delta._udiff.**udiff_dict**(*left*, *right*, *stanzas*, *indent=0*, *use_ellipses=True*)
Construct a human-readable delta between `left` and `right`.

This function probably shouldn't be called directly. Instead, use *`udiff()`* with the same arguments. *`udiff()`* and *`udiff_dict()`* are mutually recursive, anyway.

json_delta._udiff.**udiff_list**(*left*, *right*, *stanzas*, *indent=0*, *use_ellipses=True*)
Construct a human-readable delta between `left` and `right`.

This function probably shouldn't be called directly. Instead, use *`udiff()`* with the same arguments. *`udiff()`* and *`udiff_list()`* are mutually recursive, anyway.

4.6.4 json_delta._upatch

json_delta._upatch.**upatch**(*struc*, *udiff*, *reverse=False*, *in_place=True*)
Apply a patch as output by *`json_delta.udiff()`* to `struc`.

As with *`json_delta.patch()`*, `struc` is modified in place by default. Set the parm `in_place` to `False` if this is not the desired behaviour.

The udiff format has enough information in it that this transformation can be applied in reverse: i.e. if `udiff` is the output of `udiff(left, right)`, you can reconstruct `right` given `left` and `udiff` (by

running `upatch(left, udiff)`), or you can also reconstruct `left` given `right` and udiff (by running `upatch(right, udiff, reverse=True)`). This is not possible for JSON-format diffs, since a `[keypath]` stanza (meaning "delete the structure at `keypath`") does not record what the deleted structure was.

`json_delta._upatch.`**`ellipsis_handler`**(*jstring*, *point*, *key*)
Extends `key_tracker()` to handle the `...` construction.

`json_delta._upatch.`**`is_none_key`**(*key*)
Is the last element of `key` `None`?

`json_delta._upatch.`**`reconstruct_diff`**(*udiff*, *reverse=False*)
Turn a udiff back into a JSON-format diff.

Set `reverse` to `True` to generate a reverse diff (i.e. swap the significance of line-initial + and −).

Header lines (if present) are ignored:

```
>>> udiff = """--- <stdin>
... +++ <stdin>
... -false
... +true"""
>>> reconstruct_diff(udiff)
[[[], True]]
>>> reconstruct_diff(udiff, reverse=True)
[[[], False]]
```

`json_delta._upatch.`**`skip_key`**(*point*, *key*, *origin*, *keys*, *predicate*)
Find the next result in `keys` for which `predicate(key)` is `False`.

If none is found, or if `key` is already such a result, the return value is `(point, key)`.

`json_delta._upatch.`**`sort_stanzas`**(*stanzas*)
Sorts the stanzas in a diff.

`reconstruct_diff()` works on different assumptions from `json_delta._diff.needle_diff()` when it comes to stanzas altering arrays: keys in such stanzas relate to the element's position within the array's longest intermediate representation during the transformation (that is after all insert-and-shifts, after all appends, but *before* any deletions). This function sorts `stanzas` to reflect that order of operations.

As with *`json_delta._diff.sort_stanzas()`* (which see), stanzas are sorted for length so the most deeply-nested structures get their modifications first.

`json_delta._upatch.`**`udiff_key_tracker`**(*udiff*, *point=0*, *start_key=None*)
Find points within the udiff where the active keypath changes.

`json_delta._upatch.`**`upatch`**(*struc*, *udiff*, *reverse=False*, *in_place=True*)
Apply a patch as output by *`json_delta.udiff()`* to `struc`.

As with *`json_delta.patch()`*, `struc` is modified in place by default. Set the parm `in_place` to `False` if this is not the desired behaviour.

The udiff format has enough information in it that this transformation can be applied in reverse: i.e. if `udiff` is the output of `udiff(left, right)`, you can reconstruct `right` given `left` and `udiff` (by running `upatch(left, udiff)`), or you can also reconstruct `left` given `right` and udiff (by running `upatch(right, udiff, reverse=True)`). This is not possible for JSON-format diffs, since a `[keypath]` stanza (meaning "delete the structure at `keypath`") does not record what the deleted structure was.

4.6.5 json_delta._util

Utility functions and constants used by more than one submodule.

The majority of python 2/3 compatibility shims also appear in this module.

`json_delta._util.`**`predicate_count`**(*iterable*, *predicate=lambda x: True*)
Count items `x` in `iterable` such that `predicate(x)`.

The default `predicate` is `lambda x: True`, so `predicate_count(iterable)` will count the values generated by `iterable`. Note that if the iterable is a generator, this function will exhaust it, and if it is an infinite generator, this function will never return!

```
>>> predicate_count([True] * 16)
16
>>> predicate_count([True, True, False, True, True], lambda x: x)
4
```

`json_delta._util.`**`uniquify`**(*bytestring*, *key=lambda x: x*)
Remove duplicate elements from a list while preserving order.

`key` works as for `min()`, `max()`, etc. in the standard library.

`json_delta._util.`**`sniff_encoding`**(*bytestring*, *starts=JSON_STARTS*, *complete=True*)
Determine the encoding of a UTF-x encoded string.

The argument `starts` must be a mapping of bytestrings the input can begin with onto the encoding that such a beginning would represent (see *`licit_starts()`* for a function that can build such a mapping).

The `complete` flag signifies whether the input represents the entire string: if it is set `False`, the function will attempt to determine the encoding, but will raise a `UnicodeError` if it is ambiguous. For example, an input of `b'\xff\xfe'` could be the UTF-16 little-endian byte-order mark, or, if the input is incomplete, it could be the first two characters of the UTF-32-LE BOM:

```
>>> sniff_encoding(b'\xff\xfe') == 'utf_16'
True
>>> sniff_encoding(b'\xff\xfe', complete=False)
Traceback (most recent call last):
    ...
UnicodeError: String encoding is ambiguous.
```

`json_delta._util.`**`_load_and_func`**(*func*, *parm1=None*, *parm2=None*, *both=None*, ***flags*)
Decode JSON-serialized parameters and apply func to them.

`json_delta._util.`**`all_paths`**(*struc*)
Generate key-paths to every node in `struc`.

Both terminal and non-terminal nodes are visited, like so:

```
>>> paths = [x for x in all_paths({'foo': None, 'bar': ['baz', 'quux']})]
>>> [] in paths # ([] is the path to ``struc`` itself.)
True
>>> ['foo'] in paths
True
>>> ['bar'] in paths
True
>>> ['bar', 0] in paths
True
>>> ['bar', 1] in paths
True
```

```
>>> len(paths)
5
```

json_delta._util.**check_diff_structure**(*diff*)

Return `diff` (or `True`) if it is structured as a sequence of `diff` stanzas. Otherwise return `False`.

`[]` is a valid diff, so if it is passed to this function, the return value is `True`, so that the return value is always true in a Boolean context if `diff` is valid.

```
>>> check_diff_structure('This is certainly not a diff!')
False
>>> check_diff_structure([])
True
>>> check_diff_structure([None])
False
>>> example_valid_diff = [[["foo", 6, 12815316313, "bar"], None]]
>>> check_diff_structure(example_valid_diff) == example_valid_diff
True
>>> check_diff_structure([[["foo", 6, 12815316313, "bar"], None],
...                       [["foo", False], True]])
False
```

json_delta._util.**compact_json_dumps**(*obj*)

Compute the most compact possible JSON representation of `obj`.

```
>>> test = {
...              'foo': 'bar',
...              'baz':
...                 ['quux', 'spam',
...        'eggs']
... }
>>> compact_json_dumps(test) in (
...     '{"foo":"bar","baz":["quux","spam","eggs"]}',
...     '{"baz":["quux","spam","eggs"],"foo":"bar"}'
... )
True
>>>
```

json_delta._util.**decode_json**(*file_or_str*)

Decode a JSON file-like object or string.

The following doctest is probably pointless as documentation. It is here so json-delta can claim 100% code coverage for its test suite!

```
>>> try:
...     from StringIO import StringIO
... except ImportError:
...     from io import StringIO
>>> foo = '[]'
>>> decode_json(foo)
[]
>>> decode_json(StringIO(foo))
[]
```

json_delta._util.**decode_udiff**(*file_or_str*)

Decode a file-like object or bytestring udiff into a unicode string.

The udiff may be encoded in UTF-8, -16 or -32 (with or without BOM):

```
>>> udiff = u'- true\n+ false'
>>> decode_udiff(udiff.encode('utf_32_be')) == udiff
True
>>> try:
...     from StringIO import StringIO
... except ImportError:
...     from io import BytesIO as StringIO
>>> decode_udiff(StringIO(udiff.encode('utf-8-sig'))) == udiff
True
```

An empty string is a valid udiff; this function will convert it to a unicode string:

```
>>> decode_udiff(b'') == u''
True
```

The function is idempotent: if you pass it a unicode string, it will be returned unmodified:

```
>>> decode_udiff(udiff) is udiff
True
```

If you pass it a non-empty bytestring that cannot be interpreted as beginning with `' '`, `'+'`, `'-'` or a BOM in any encoding, a `ValueError` is raised:

```
>>> decode_udiff(b':-)')
Traceback (most recent call last):
    ...
ValueError: String does not begin with any of the specified start chars.
```

`json_delta._util.`**`follow_path`**(*struc*, *path*)

Retrieve the value found at the key-path `path` within `struc`.

`json_delta._util.`**`in_array`**(*key*, *accept_None=False*)

Should the keypath `key` point at a JSON array (`[]`)?

Works by testing whether `key[-1]` is an `int` or (where appropriate) `long`:

```
>>> in_array([u'bar', 16])
True
>>> import sys
>>> sys.version >= '3' or eval("in_array([u'foo', 94L])")
True
```

Returns `False` if `key` addresses a non-array object…

```
>>> in_array(["foo"])
False
>>> in_array([u'bar'])
False
```

…or if `key == []` (as in that case there's no way of knowing whether `key` addresses an object or an array).

```
>>> in_array([])
False
```

If the `accept_None` flag is set, this function will not raise a `ValueError` if `key[-1] is None` (keypaths of this form are used by *`key_tracker()`*, to signal points within a JSON string where a new object key is expected, but not yet found).

```
>>> in_array([None])
Traceback (most recent call last):
    ...
ValueError: keypath elements must be instances of str, unicode, int or long,
    not NoneType (key[0] == None)
```

```
>>> in_array([None], True)
False
>>> in_array([None], accept_None=True)
False
```

Otherwise, a `ValueError` is raised if `key` is not a valid keypath:

```
>>> keypath = [{str("spam"): str("spam")}, "pickled eggs and spam", 7]
>>> in_array(keypath)
Traceback (most recent call last):
    ...
ValueError: keypath elements must be instances of str, unicode, int or long,
    not dict (key[0] == {'spam': 'spam'})
```

`json_delta._util.`**`in_object`**(*key*, *accept_None=False*)

Should the keypath `key` point at a JSON object (`{}`)?

Works by testing whether `key[-1]` is a string or (where appropriate) `unicode()`:

```
>>> in_object(["foo"])
True
>>> in_object([u'bar'])
True
```

Returns `False` if `key` addresses an array…

```
>>> in_object([u'bar', 16])
False
>>> import sys
>>> False if sys.version >= '3' else eval("in_object([u'bar', 16L])")
False
```

…if `key == []`…

```
>>> in_object([])
False
```

If the `accept_None` flag is set, this function will also return `True` if `key[-1] is None` (this functionality is used by *`key_tracker()`*, to signal points within a JSON string where a new object key is expected, but not yet found).

```
>>> in_object([None])
Traceback (most recent call last):
    ...
ValueError: keypath elements must be instances of str, unicode, int or long,
    not NoneType (key[0] == None)
```

```
>>> in_object([None], True)
True
>>> in_object([None], accept_None=True)
True
```

Raises a `ValueError` if `key` is not a valid keypath:

```
>>> in_object(['foo', {}])
Traceback (most recent call last):
    ...
ValueError: keypath elements must be instances of str, unicode, int or long,
    not dict (key[1] == {})
```

```
>>> in_object([False, u'foo'])
Traceback (most recent call last):
    ...
ValueError: keypath elements must be instances of str, unicode, int or long,
    not bool (key[0] == False)
```

`json_delta._util.`**`in_x_error`**(*key*, *offender*)
Build the instance of `ValueError` *`in_object()`* and *`in_array()`* raise if `keypath` is invalid.

`json_delta._util.`**`json_bytestring_length`**(*string*)
Find the length of the JSON for a string without actually encoding it.

Attempts to give the shortest possible version: encoding as UTF-8 and using escape sequences only where necessary.

`json_delta._util.`**`json_length`**(*obj*)
Find the length of the JSON for `obj` without actually encoding it.

`json_delta._util.`**`key_tracker`**(*jstring*, *point=0*, *start_key=None*, *special_handler=None*)
Generate points within `jstring` where the keypath changes.

This function also identifies points within objects where a new `key: value` pair is expected, by yielding a pseudo-keypath with `None` as the final element.

Parameters:

- `jstring`: The JSON string to search.
- `point`: The point to start at.
- `start_key`: The starting keypath.
- `special_handler`: A function for handling extensions to JSON syntax (e.g. `_upatch.ellipsis_handler()`, used to handle the `...` construction in udiffs).

```
>>> next(key_tracker('{}'))
(1, (None,))
```

`json_delta._util.`**`keypath_lengths`**(*keypaths*)
Build a dict of lengths of (hashable!) keypaths from a structure.

`keypaths` must be a list of all keypaths within a single structure, e.g. as returned by *`all_paths()`*.

`json_delta._util.`**`licit_starts`**(*start_chars=u'{}[]"-0123456789tfn \t\n\r'*)
Compute the bytestrings a UTF-x encoded string can begin with.

This function is intended for encoding detection when the beginning of the encoded string must be one of a limited set of characters, as for JSON or the udiff format. The argument `start_chars` must be an iterable of valid beginnings.

`json_delta._util.`**`nearest_of`**(*string*, **subs*)
Find the index of the substring in `subs` that occurs earliest in `string`, or `len(string)` if none of them do.

json_delta._util.**predicate_count**(*iterable*, *predicate=<function <lambda>>*)

Count items `x` in `iterable` such that `predicate(x)`.

The default `predicate` is `lambda x: True`, so `predicate_count(iterable)` will count the values generated by `iterable`. Note that if the iterable is a generator, this function will exhaust it, and if it is an infinite generator, this function will never return!

```
>>> predicate_count([True] * 16)
16
>>> predicate_count([True, True, False, True, True], lambda x: x)
4
```

json_delta._util.**read_bytestring**(*file*)

Read the contents of `file` as a `bytes` object.

json_delta._util.**skip_string**(*jstring*, *point*)

Assuming `jstring` is a string, and `jstring[point]` is a `"` that starts a JSON string, return `x` such that `jstring[x-1]` is the `"` that terminates the string.

When a `"` is found, it is necessary to check that it is not escaped by a preceding backslash. As a backslash may itself be escaped, this amounts to checking that the number of backslashes immediately preceding the `"` is even (counting 0 as an even number):

```
>>> test_string = r'"Fred \"Foonly\" McQuux"'
>>> skip_string(test_string, 0) == len(test_string)
True
>>> backslash = chr(0x5c)
>>> dbl_quote = chr(0x22)
>>> even_slashes = ((r'"\\\\\\"', json.dumps(backslash * 3)),
...                 (r'"\\\\"',   json.dumps(backslash * 2)),
...                 (r'"\\"',     json.dumps(backslash)))
>>> all((json.loads(L) == json.loads(R) for (L, R) in even_slashes))
True
>>> all((skip_string(L, 0) == len(L) for (L, R) in even_slashes))
True
>>> def cat_dump(*args): return json.dumps(''.join(args))
>>> odd_slashes = (
...     (r'"\\\\\\\"  "', cat_dump(backslash * 3, dbl_quote, ' ' * 2)),
...     (r'"\\\\\"    "', cat_dump(backslash * 2, dbl_quote, ' ' * 4)),
...     (r'"\\\"      "', cat_dump(backslash * 1, dbl_quote, ' ' * 6)),
...     (r'"\"        "', cat_dump(dbl_quote, ' ' * 8)),
... )
>>> all((json.loads(L) == json.loads(R) for (L, R) in odd_slashes))
True
>>> all((skip_string(L, 0) == 12 for (L, R) in odd_slashes))
True
```

json_delta._util.**sniff_encoding**(*bytestring*, *starts={'\x00\x00\x007': u'utf_32_be', '\x00\n': u'utf_16_be', '\x00\x00\x00\r': u'utf_32_be', '\x00\t': u'utf_16_be', '\x00\x00\x00\t': u'utf_32_be', '\x00\x00\x00\n': u'utf_32_be', '\x00\r': u'utf_16_be', '"\x00\x00\x00': u'utf_32_le', '2\x00': u'utf_16_le', '\x00\x00\x00]': u'utf_32_be', '\xef\xbb\xbf': u'utf_8_sig', '\x00"': u'utf_16_be', ' ': u'utf_8', '\x00 ': u'utf_16_be', '\x00\x00\x00 ': u'utf_32_be', '\x00\x00\x00"': u'utf_32_be', '\x00\x00\x00-': u'utf_32_be', '\x00-': u'utf_16_be', '\x002': u'utf_16_be', '0': u'utf_8', '\x000': u'utf_16_be', '\x001': u'utf_16_be', '\x006': u'utf_16_be', '4': u'utf_8', '\x004': u'utf_16_be', '\x005': u'utf_16_be', '8': u'utf_8', '\x008': u'utf_16_be', '\xff\xfe\x00\x00': u'utf_32', '\x00\x00\x008': u'utf_32_be', '\x00\x00\x001': u'utf_32_be', ']\x00\x00\x00': u'utf_32_le', '-\x00': u'utf_16_le', 'f\x00\x00\x00': u'utf_32_le', '\x00\x00\x00f': u'utf_32_be', '\x00[': u'utf_16_be', '5\x00': u'utf_16_le', 't\x00': u'utf_16_le', '\x00]': u'utf_16_be', ' \x00': u'utf_16_le', '\x00f': u'utf_16_be', '\x00\x00\x00n': u'utf_32_be', '\x00n': u'utf_16_be', '1\x00\x00\x00': u'utf_32_le', '\x00\x00\x00t': u'utf_32_be', 't': u'utf_8', '\x00t': u'utf_16_be', '4\x00\x00\x00': u'utf_32_le', '\x00{': u'utf_16_be', '\x00}': u'utf_16_be', '\x00\x00\xfe\xff': u'utf_32', '7\x00\x00\x00': u'utf_32_le', '0\x00': u'utf_16_le', '8\x00': u'utf_16_le', 'f\x00': u'utf_16_le', '3': u'utf_8', '7': u'utf_8', '{\x00\x00\x00': u'utf_32_le', '[\x00': u'utf_16_le', '\x00\x00\x00}': u'utf_32_be', '\t\x00': u'utf_16_le', '[': u'utf_8', '3\x00': u'utf_16_le', '\x00\x00\x00{': u'utf_32_be', '{': u'utf_8', '-\x00\x00\x00': u'utf_32_le', '\n': u'utf_8', '0\x00\x00\x00': u'utf_32_le', 'n\x00\x00\x00': u'utf_32_le', '6\x00': u'utf_16_le', '\x00\x00\x004': u'utf_32_be', '"': u'utf_8', '3\x00\x00\x00': u'utf_32_le', '\x003': u'utf_16_be', '\x00\x00\x00[': u'utf_32_be', '\x00\x00\x006': u'utf_32_be', '2': u'utf_8', '}\x00': u'utf_16_le', '6\x00\x00\x00': u'utf_32_le', '6': u'utf_8', 't\x00\x00\x00': u'utf_32_le', '\x00\x00\x000': u'utf_32_be', '\x007': u'utf_16_be', '\x00\x00\x002': u'utf_32_be', '9\x00\x00\x00': u'utf_32_le', '\t\x00\x00\x00': u'utf_32_le', '1\x00': u'utf_16_le', '[\x00': u'utf_16_le', '[\x00\x00\x00': u'utf_32_le', '\x009': u'utf_16_be', ' \x00\x00\x00': u'utf_32_le', 'f': u'utf_8', '9\x00': u'utf_16_le', '}\x00\x00\x00': u'utf_32_le', 'n': u'utf_8', '\xfe\xff': u'utf_16', '\t': u'utf_8', '\n\x00\x00\x00': u'utf_32_le', '\r': u'utf_8', '\r\x00\x00\x00': u'utf_32_le', '\n\x00': u'utf_16_le', '4\x00': u'utf_16_le', '-': u'utf_8', '1': u'utf_8', '{\x00': u'utf_16_le', '5': u'utf_8', '9': u'utf_8', '\xff\xfe': u'utf_16', '2\x00\x00\x00': u'utf_32_le', '\x00\x00\x005': u'utf_32_be', 'n\x00': u'utf_16_le', '5\x00\x00\x00': u'utf_32_le', '\x00\x00\x003': u'utf_32_be', ']': u'utf_8', '\x00\x00\x009': u'utf_32_be', '"\x00': u'utf_16_le', '\r\x00': u'utf_16_le', '7\x00': u'utf_16_le', '8\x00\x00\x00': u'utf_32_le', '}': u'utf_8'}*, *complete=True*)

Determine the encoding of a UTF-x encoded string.

The argument `starts` must be a mapping of bytestrings the input can begin with onto the encoding that such a beginning would represent (see *`licit_starts()`* for a function that can build such a mapping).

The `complete` flag signifies whether the input represents the entire string: if it is set `False`, the function will attempt to determine the encoding, but will raise a `UnicodeError` if it is ambiguous. For example, an input of `b'\xff\xfe'` could be the UTF-16 little-endian byte-order mark, or, if the input is incomplete, it could be the first two characters of the UTF-32-LE BOM:

```
>>> sniff_encoding(b'\xff\xfe') == 'utf_16'
True
>>> sniff_encoding(b'\xff\xfe', complete=False)
Traceback (most recent call last):
    ...
UnicodeError: String encoding is ambiguous.
```

json_delta._util.**stanzas_addressing**(*stanzas*, *keypath*)

Find diff stanzas modifying the structure at `keypath`.

The purpose of this function is to keep track of changes made to the overall structure by stanzas earlier in the sequence, e.g.:

```
>>> struc = [
...     'foo',
...     'bar', [
...         'baz'
...     ]
... ]
>>> stanzas = [
...     [ [2, 1], 'quux'],
...     [ [0] ],
...     [ [1, 2], 'quordle']
... ]
>>> (stanzas_addressing(stanzas, [2])
...  == [
...   [ [1], 'quux' ],
...   [ [2], 'quordle' ]
...  ])
True
```

`stanzas[0]` and `stanzas[2]` both address the same element of `struc` — the list that starts off as `['baz']`, even though their keypaths are completely different, because the diff stanza `[[0]]` moves the list `['baz']` from index 2 of `struc` to index 1.

The return value is a sub-diff: a list of stanzas fit to modify the element at `keypath` within the overall structure.

json_delta._util.**struc_lengths**(*struc*)

Build dicts for lengths of nodes in a JSON-serializable structure.

Return value is a 2-tuple `(terminals, nonterminals)`. The `terminals` dict is keyed by the values of the terminal nodes themselves, as these are all hashable types.

WARNING: The `nonterminals` dict is keyed by the `id()` value of the list or dict, so if the object is modified after this function is called, the lengths recorded may no longer be valid.

json_delta._util.**uniquify**(*obj*, *key=<function <lambda>>*)

Remove duplicate elements from a list while preserving order.

`key` works as for `min()`, `max()`, etc. in the standard library.

json_delta._util.**whitespace_count**(*obj*, *indent=1*, *margin=1*, *nest_level=0*)

Count whitespace chars that `json` will use encoding `obj`

4.7 Javascript implementation notes

The Javascript implementation provides an object `JSON_delta` that encapsulates the principal functions (use `JSON_delta.patch` and `JSON_delta.diff`). JSON-format diffs and patches are supported, and the diffs can be made compact (set the `minimal` parm to `JSON_delta.diff` to `true`). Udiffs and upatching are not yet supported: email me if you'd like to see them!

4.8 Racket implementation notes

The Racket implementation passes the test suite, but is about as slow as a wet weekend. Refactoring for speed will be gotten around to Real Soon Now...

4.9 Perl implementation notes

Development of the Perl implementation stalled when it was discovered that, because Perl doesn't have as ramified a type ontology as Javascript, numeric values do not round-trip cleanly. This makes it impossible to produce an implementation that passes the test suite using available JSON libraries.

4.10 Licenses

4.10.1 Source

4.10.2 This documentation

CHAPTER 5

Indices and tables

- genindex
- search

Python Module Index

j

Index

Symbols

A

C

D

E

F

G

I

J

K

L

N

P

R

S

T

U

W