NestedText

Release 1.0.0

unknown

Feb 27, 2021

GETTING STARTED

Authors: Ken & Kale Kundert

Version: 1.0.0

Released: 2020-10-03

Please post all questions, suggestions, and bug reports to NestedText Github.

NestedText is a file format for holding data that is to be entered, edited, or viewed by people. It allows data to be organized into a nested collection of dictionaries, lists, and strings. In this way it is similar to *JSON* and *YAML*, but without the complexity and risk of *YAML* and without the syntatic clutter of *JSON*. *NestedText* is both simple and natural. Only a small number of concepts and rules must be kept in mind when creating it. It is easily created, modified, or viewed with a text editor and easily understood and used by both programmers and non-programmers.

NestedText is convenient for configuration files, address books, account information and the like. Here is an example of a file that contains a few addresses:

```
# Contact information for our officers

president:
    name: Katheryn McDaniel
    address:
        > 138 Almond Street
        > Topika, Kansas 20697
    phone:
        cell: 1-210-555-5297
        home: 1-210-555-8470
            # Katheryn prefers that we always call her on her cell phone.
    email: KateMcD@aol.com
    additional roles:
        - board member

vice president:
    name: Margaret Hodge
    address:
        > 2586 Marigold Lane
        > Topika, Kansas 20682
    phone: 1-470-555-0398
    email: margaret.hodge@uk.edu
    additional roles:
        - new membership task force
        - accounting task force

treasurer:
    name: Fumiko Purvis
        # Fumiko's term is ending at the end of the year.
        # She will be replaced by Merrill Eldridge.
    address:
        > 3636 Buffalo Ave
```

(continues on next page)

(continued from previous page)

```
        > Topika, Kansas 20692
    phone: 1-268-555-0280
    email: fumiko.purvis@hotmail.com
    additional roles:
        - accounting task force
```

The format holds dictionaries (ordered collections of name/value pairs), lists (ordered collections of values) and strings (text) organized hierarchically to any depth. Indentation is used to indicate the hierarchy of the data, and a simple natural syntax is used to distinguish the types of data in such a manner that it is not easily confused. Specifically, lines that begin with a word or words followed by a colon are dictionary items; a dash introduces list items, and a leading greater-than symbol signifies a line in a multiline string. Dictionaries and lists are used for nesting, the leaf values are always strings.

CHAPTER

ONE

ALTERNATIVES

There are no shortage of well established alternatives to *NestedText* for storing data in a human-readable text file. Probably the most obvious are json and YAML. Both have serious short comings.

JSON is a subset of JavaScript suitable for holding data. Like *NestedText*, it consists of a hierarchical collection of dictionaries, lists, and strings, but also allows integers, floats, Booleans and nulls. The problem with *JSON* for this application is that it is awkward. With all those data types it must syntactically distinguish between them. For example, in *JSON* 32 is an integer, 32.0 is the real version of 32, and "32" is the string version. These distinctions are not meaningful and can be confusing to non-programmers. In addition, in most datasets a majority of leaf values are strings and the required quotes adds substantial visual clutter. *NestedText* avoids these issues by treating all leaf values as strings with no need for quoting or escaping. It is up to the application that employs *NestedText* as an input format to sort things out later.

JSON does not provide for multiline strings and any special characters like newlines or unicode are encoded with escape codes, which can make strings quite difficult to interpret. Finally, dictionary and list items must be separated with commas, but a comma must not follow last item. All of this results in *JSON* being a frustrating format for humans to read, enter or edit.

NestedText has the following clear advantages over *JSON* as human readable and writable data file format:

- text does not require quotes
- data type does not change based on seemingly insignificant details (32, 32.0, "32")
- comments
- multiline strings
- special characters without escaping them
- Unicode characters without encoding them
- commas are not used to separate dictionary and list items

YAML was to be the human friendly alternative to *JSON*, but the authors were too ambitious and tried to support too many data types and too many formats. To distinguish between all the various types and formats, a complicated and non-intuitive set of rules developed. For example, 2 is interpreted as an integer, 2.0 as a real number, and both 2.0.0 and "2" are strings. *YAML* at first appears very appealing when used with simple examples, but things can quickly become complicated or provide unexpected results. A reaction to this is the use of *YAML* subsets, such as StrictYAML. However, the subsets still try to maintain compatibility with *YAML* and so inherit much of its complexity. For example, both *YAML* and *StrictYAML* support the nine different ways to write multi-line strings in YAML.

YAML recognized the problems that result from *JSON* needing to unambiguously distinguish between many data types and instead uses implicit typing, which creates its own problems. For example, consider the following *YAML* fragment:

```
Enrolled: NO
Country Code: NO
```

Presumably *Enrolled* is meant to be a Boolean value whereas *Country Code* is meant to be a string (*NO* is the country code for Norway). Reading this fragment with *YAML* results in {'Enrolled': *False*, 'Country Code': *False*}. When read by *NestedText* both values are retained in their original form as strings. With *NestedText* any decisions about how to interpret the leaf values are passed to the end application, which is the only place where they can be made knowledgeably. The assumption is that the end application knows that *Enrolled* should be a Boolean and knows how to convert 'NO' to *False*. The same is not possible with *YAML* because the *Country Code* value has been transformed and because there are many possible strings that map to *False* (*n*, *no*, *false*, *off*; etc.).

This is one example of the many possible problems that stem from implicit typing. In fact, many people make it a habit to add quotes to all values simply to avoid the ambiguities, which makes *YAML* more like *JSON*.

NestedText was inspired by *YAML*, but eschews its complexity. It has the following clear advantages over *YAML* as human readable and writable data file format:

- simple
- unambiguous (no implicit typing)
- data type does not change based on seemingly insignificant details (2, 2.0, 2.0.0, "2")
- syntax is insensitive to special characters within text
- safe, no risk of malicious code execution

CHAPTER TWO

ISSUES

Please ask questions or report problems on Github.

CHAPTER

THREE

CONTRIBUTING

This package contains a Python reference implementation of *NestedText*. Implementation in many languages is required for *NestedText* to catch on widely. If you like the format, please consider contributing additional implementations.

3.1 Installation

```
pip3 install --user nestedtext
```

3.1.1 Releases

Latest development release:

Version: 1.0.0

Released: 2020-10-03

0.6 (2020-09-26):

- Added `load()` and `dump()`.
- Eliminated *NestedTextError.get_extended_codicil*.

0.5 (2020-09-11):

- allow user to manage duplicate keys detected by `loads()`.

0.4 (2020-09-07):

- Change rest-of-line strings to include all characters given, including leading and trailing quotes and spaces.
- The *NestedText* top-level is no longer restricted to only dictionaries and lists. The top-level can now also be a single string.
- `loads()` now returns *None* when given an empty *NestedText* document.
- Change `NestedTextError` attribute names to make them more consistent with those used by JSON package.
- Added *NestedTextError.get_extended_codicil*.

0.3 (2020-09-03):

- Allow comments to be indented.

0.2 (2020-09-02):

- Minor enhancements and bug fixes.

0.1 (2020-08-30):

- Initial release.

3.2 Basic syntax

This is a overview of the syntax of a *NestedText* document, which consists of a *nested collection* of *dictionaries*, *lists*, and *strings*. You can find more specifics *later on*.

3.2.1 Dictionaries

A dictionary is a collection of name/value pairs:

```
name 1: value 1
name 2: value 2
...
```

A dictionary item is introduced by a key (the name) and a colon at the start of a line. Anything that follows the space after the colon is the value and is treated as a string.

The key is a string and must be quoted if it contains characters that could be misinterpreted.

A dictionary is all adjacent dictionary items at the same indentation level.

3.2.2 Lists

A list is a collection of simple values:

```
- value 1
- value 2
...
```

A list item is introduced with a dash at the start of a line. Anything that follows the space after the dash is the value and is treated as a string.

A list is all adjacent list items at the same indentation level.

3.2.3 Strings

The values described in the last two sections are all rest-of-line strings; they end at the end of the line. Rest-of-line strings are simply all the remaining characters on the line. They can contain any character other than newline:

```
regex: [+-]?([0-9]*[.])?[0-9]+
math: -b + sqrt(b**2 - 4*a*c)
unicode: José and François
```

It is also possible to specify strings that are alone on a line and they can be combined to form multiline strings. To do so, precede the line with a greater-than symbol:

```
>     this is the first line of a multiline string, it is indented.
> this is the second line, it is not indented.
```

The content of each line starts after the space that follows the greater-than symbol.

You can include empty lines in the string simply by specifying the greater-than symbol alone on a line:

```
>
> The future ain't what it used to be.
>
>                     - Yogi Berra
>
```

3.2.4 Comments

Lines that begin with a hash as the first non-space character, or lines that are empty or consist only of spaces and tabs are ignored. Indentation is not significant on comment lines.

```
# this line is ignored
```

3.2.5 Nesting

A value for a dictionary or list item may be a rest-of-line string as shown above, or it may be a nested dictionary, list or a multiline string. Indentation is used to indicate nesting (or composition). Indentation increases to indicate the beginning of a new nested object, and indentation returns to a prior level to indicate its end. In this way, data can be nested to an arbitrary depth:

```
# Contact information for our officers

president:
    name: Katheryn McDaniel
    address:
        > 138 Almond Street
        > Topika, Kansas 20697
    phone:
        cell: 1-210-555-5297
        home: 1-210-555-8470
            # Katheryn prefers that we always call her on her cell phone.
    email: KateMcD@aol.com
    kids:
        - Joanie
        - Terrance

vice president:
    name: Margaret Hodge
    address:
        > 2586 Marigold Land
        > Topika, Kansas 20697
    phone: 1-470-555-0398
    email: margaret.hodge@uk.edu
    kids:
        - Arnie
        - Zach
        - Maggie
```

It is recommended that each level of indentation be represented by a consistent number of spaces (with the suggested number being 2 or 4). However, it is not required. Any increase in the number of spaces in the indentation represents an indent and and the number of spaces need only be consistent over the length of the nested object.

3.3 Basic use

The *NestedText* API is patterned after that of JSON.

3.3.1 NestedText Reader

The `loads()` function is used to convert *NestedText* into a Python data structure. If there is a problem interpreting the input text, a `NestedTextError` exception is raised.

```
>>> import nestedtext as nt

>>> content = """
... access key id: 8N029N81
... secret access key: 9s83109d3+583493190
... """

>>> try:
...     data = nt.loads(content)
... except nt.NestedTextError as e:
...     e.terminate()

>>> print(data)
{'access key id': '8N029N81', 'secret access key': '9s83109d3+583493190'}
```

You can also read directly from a file or stream using the `load()` function.

```
>>> from inform import fatal, os_error

>>> try:
...     groceries = nt.load('examples/groceries.nt')
... except nt.NestedTextError as e:
...     e.terminate()
... except OSError as e:
...     fatal(os_error(e))

>>> print(groceries)
['Bread', 'Peanut butter', 'Jam']
```

3.3.2 NestedText Writer

The `dumps()` function is used to convert a Python data structure into *NestedText*. As before, if there is a problem converting the input data, a `NestedTextError` exception is raised.

```
>>> try:
...     content = nt.dumps(data)
... except nt.NestedTextError as e:
...     e.terminate()

>>> print(content)
access key id: 8N029N81
secret access key: 9s83109d3+583493190
```

The `dump()` function writes *NestedText* to a file or stream.

```
>>> try:
...     content = nt.dump(data, 'examples/access.nt')
... except nt.NestedTextError as e:
...     e.terminate()
... except OSError as e:
...     fatal(os_error(e))
```

3.4 Structure parser

Parsing data can be a difficult challenge. One way to reduce the challenge is to reduce the scope of what is being parsed. With *NestedText* you can delegate the parsing the of the structure and instead focus on parsing individual values given as strings. A transforming validator like Voluptuous can greatly simply the process.

To use *Voluptuous* you would create a schema and then apply the schema to the data. The schema details what fields are expected, and what what kind of values they should contain. Normally the schema is used to validate the data, but with a little extra plumbing the data can be transformed to the needed form. The following is a very simple example (see *cryptocurrency holdings* for a more complete example).

In order for *Voluptuous* to convert the data to the desired type, a converter function is helpful:

```
>>> import voluptuous

>>> def coerce(type, msg=None):
...     """Coerce a value to a type.
...
...     If the type constructor throws a ValueError, the value will be
...     marked as Invalid.
...     """
...     def f(v):
...         try:
...             return type(v)
...         except ValueError:
...             raise voluptuous.Invalid(msg or ('expected %s' % type.__name__))
...     return f
```

The next step is to define a schema that declares the expected types of the various fields in the configuration file. For example, imagine the configuration file has has three values, *name*, *value*, and *editable*, the first of which must be a string, the second a float, and the third a boolean that is specified using either 'yes' or 'no'. This can be done as follows:

```
>>> import nestedtext as nt

>>> def to_bool(v):
...     try:
...         v = v.lower()
...         assert v in ['yes', 'no']
...         return v == 'yes'
...     except:
...         raise ValueError("expected 'yes' or 'no'.")

>>> config = """
... name: volume
... value: 50
... editable: yes
... """
```

(continues on next page)

(continued from previous page)

```
>>> config_data = nt.loads(config)
>>> print(config_data)
{'name': 'volume', 'value': '50', 'editable': 'yes'}

>>> schema = voluptuous.Schema(
...     dict(name=str, value=coerce(float), editable=coerce(to_bool))
... )

>>> settings = schema(config_data)
>>> print(settings)
{'name': 'volume', 'value': 50.0, 'editable': True}
```

Notice that a dictionary that contains the expected types and conversion functions is passed to *Schema*. Then the raw configuration is parsed for structure by *NestedText*, and the resulting data structure is processed by the schema to and converted to its final form.

3.5 Examples

3.5.1 JSON to NestedText

This example implements a command-line utility that converts a *JSON* file to *NestedText*. It demonstrates the use of `dumps()` and `NestedTextError`.

```
#!/usr/bin/env python3
"""
Read a JSON file and convert it to NestedText.

usage:
    json-to-nestedtext [options] [<filename>]

options:
    -f, --force             force overwrite of output file
    -i <n>, --indent <n>    number of spaces per indent [default: 4]

If <filename> is not given, json input is taken from stdin and NestedText output
is written to stdout.
"""

from docopt import docopt
from inform import fatal, os_error
from pathlib import Path
import json
import nestedtext as nt
import sys
sys.stdin.reconfigure(encoding='utf-8')
sys.stdout.reconfigure(encoding='utf-8')

cmdline = docopt(__doc__)
input_filename = cmdline['<filename>']
try:
    indent = int(cmdline['--indent'])
except Exception:
    warn('indent garbled.', culprit=cmdline['--indent'])
```

(continues on next page)

(continued from previous page)

```
    indent = 4

try:
    # read JSON content; from file or from stdin
    if input_filename:
        input_path = Path(input_filename)
        json_content = input_path.read_text(encoding='utf-8')
    else:
        json_content = sys.stdin.read()
    data = json.loads(json_content)

    # convert to NestedText
    nestedtext_content = nt.dumps(data, indent=indent) + "\n"

    # output NestedText content; to file or to stdout
    if input_filename:
        output_path = input_path.with_suffix('.nt')
        if output_path.exists():
            if not cmdline['--force']:
                fatal('file exists, use -f to force over-write.', culprit=output_path)
        output_path.write_text(nestedtext_content, encoding='utf-8')
    else:
        sys.stdout.write(nestedtext_content)
except OSError as e:
    fatal(os_error(e))
except nt.NestedTextError as e:
    e.terminate(culprit=input_filename)
except json.JSONDecodeError as e:
    # create a nice error message with surrounding context
    msg = e.msg
    culprit = input_filename
    codicil = None
    try:
        lineno = e.lineno
        culprit = (culprit, lineno)
        colno = e.colno
        lines_before = e.doc.split('\n')[lineno-2:lineno]
        lines = []
        for i, l in zip(range(lineno-len(lines_before), lineno), lines_before):
            lines.append(f'{i+1:>4}> {l}')
        lines_before = '\n'.join(lines)
        lines_after = e.doc.split('\n')[lineno:lineno+1]
        lines = []
        for i, l in zip(range(lineno, lineno + len(lines_after)), lines_after):
            lines.append(f'{i+1:>4}> {l}')
        lines_after = '\n'.join(lines)
        codicil = f"{lines_before}\n     {colno*' '}\n{lines_after}"
    except Exception:
        pass
    fatal(full_stop(msg), culprit=culprit, codicil=codicil)
```

3.5.2 NestedText to JSON

This example implements a command-line utility that converts a *NestedText* file to *JSON*. It demonstrates the use of `loads()` and `NestedTextError`.

```
#!/usr/bin/env python3
"""
Read a NestedText file and convert it to JSON.

usage:
    nestedtext-to-json [options] [<filename>]

options:
    -f, --force   force overwrite of output file
    -d, --dedup   de-duplicate keys in dictionaries

If <filename> is not given, NestedText input is taken from stdin and JSON output
is written to stdout.
"""

from docopt import docopt
from inform import fatal, os_error
from pathlib import Path
import json
import nestedtext as nt
import sys
sys.stdin.reconfigure(encoding='utf-8')
sys.stdout.reconfigure(encoding='utf-8')


def de_dup(key, value, data, state):
    if key not in state:
        state[key] = 1
    state[key] += 1
    return f"{key}#{state[key]}"


cmdline = docopt(__doc__)
input_filename = cmdline['<filename>']
on_dup = de_dup if cmdline['--dedup'] else None

try:
    if input_filename:
        input_path = Path(input_filename)
        data = nt.load(input_path, on_dup=de_dup)
        json_content = json.dumps(data, indent=4)
        output_path = input_path.with_suffix('.json')
        if output_path.exists():
            if not cmdline['--force']:
                fatal('file exists, use -f to force over-write.', culprit=output_path)
        output_path.write_text(json_content, encoding='utf-8')
    else:
        data = nt.load(sys.stdin, on_dup=de_dup)
        json_content = json.dumps(data, indent=4)
        sys.stdout.write(json_content)
except OSError as e:
    fatal(os_error(e))
except nt.NestedTextError as e:
```

(continues on next page)

(continued from previous page)

```
    e.terminate()
```

3.5.3 Cryptocurrency holdings

This example implements a command-line utility that displays the current value of cryptocurrency holdings. The program starts by reading a settings file held in `~/.config/cc` that in this case holds:

```
holdings:
    - 5 BTC
    - 50 ETH
    - 50,000 XLM
currency: USD
date format: h:mm A, dddd MMMM D
screen width: 90
```

This file, of course, is in *NestedText* format. After being read by `loads()` it is processed by a Voluptuous schema that does some checking on the form of the values specified and then converts the holdings to a list of QuantiPhy quantities and the screen width to an integer. The latest prices are then downloaded from cryptocompare, the value of the holdings are computed, and then displayed. The result looks like this:

```
Holdings as of 11:18 AM, Wednesday September 2.
5 BTC = $56.8k @ $11.4k/BTC    68.4%
50 ETH = $21.7k @ $434/ETH     26.1%
50 kXLM = $4.6k @ $92m/XLM     5.5%
Total value = $83.1k.
```

And finally, the code:

```
#!/usr/bin/env python3

from appdirs import user_config_dir
import nestedtext as nt
from voluptuous import Schema, Required, All, Length, Invalid
from inform import display, fatal, is_collection, os_error, render_bar
import arrow
import requests
from quantiphy import Quantity
from pathlib import Path

# configure preferences
Quantity.set_prefs(prec=2, ignore_sf = True)
currency_symbols = dict(USD='$', EUR='€', JPY='¥', GBP='£')

# utility functions
def coerce(type):
    def f(value):
        try:
            if is_collection(value):
                return [type(each) for each in value]
            return type(value)
        except ValueError:
            raise Invalid(f'expected {type.__name__}, found {v.__class__.__name__}')
    return f

```

(continues on next page)

(continued from previous page)

```
try:
    # read settings
    settings_file = Path(user_config_dir('cc'), 'settings')
    settings_schema = Schema({
        Required('holdings'): All(coerce(Quantity), Length(min=1)),
        'currency': str,
        'date format': str,
        'screen width': coerce(int)
    })
    settings = settings_schema(nt.load(settings_file))
    currency = settings.get('currency', 'USD')
    currency_symbol = currency_symbols.get(currency, currency)
    screen_width = settings.get('screen width', 80)

    # download latest asset prices from cryptocompare.com
    params = dict(
        fsyms = ','.join(coin.units for coin in settings['holdings']),
        tsyms = currency,
    )
    url = 'https://min-api.cryptocompare.com/data/pricemulti'
    try:
        r = requests.get(url, params=params)
        if r.status_code != requests.codes.ok:
            r.raise_for_status()
    except Exception as e:
        raise Error('cannot access cryptocurrency prices:', codicil=str(e))
    prices = {k: Quantity(v['USD'], currency_symbol) for k, v in r.json().items()}

    # compute total
    total = Quantity(0, currency_symbol)
    for coin in settings['holdings']:
        price = prices[coin.units]
        value = price.scale(coin)
        total = total.add(value)

    # display holdings
    now = arrow.now().format(settings.get('date format', 'h:mm A, dddd MMMM D, YYYY'))
    print(f'Holdings as of {now}.')
    bar_width = screen_width - 37
    for coin in settings['holdings']:
        price = prices[coin.units]
        value = price.scale(coin)
        portion = value/total
        summary = f'{coin} = {value} @ {price}/{coin.units}'
        print(f'{summary:<30} {portion:<5.1%} {render_bar(portion, bar_width)}')
    print(f'Total value = {total}.')

except nt.NestedTextError as e:
    e.terminate()
except Invalid as e:
    fatal(e)
except OSError as e:
    fatal(os_error(e))
except KeyboardInterrupt:
    pass
```

3.6 Common mistakes

When `loads()` complains of errors it is important to look both at the line fingered by the error message and the one above it. The line that is the target of the error message might by an otherwise valid *NestedText* line if it were not for the line above it. For example, consider the following example:

Example:

```
>>> import nestedtext as nt

>>> content = """
... treasurer:
...     name: Fumiko Purvis
...     address: Home
...         > 3636 Buffalo Ave
...         > Topika, Kansas 20692
... """

>>> try:
...     data = nt.loads(content)
... except nt.NestedTextError as e:
...     print(e.get_message())
...     print(e.get_codicil()[0])
invalid indentation.
   4 «    address: Home»
   5 «        > 3636 Buffalo Ave»
```

Notice that the complaint is about line 5, but problem stems from line 4 where *Home* gave a value to *address*. With a value specified for *address*, any further indentation on line 5 indicates a second value is being specified for *address*, which is illegal.

A more subtle version of this same error follows:

Example:

```
>>> content = """
... treasurer:
...     name: Fumiko Purvis
...     address:␣␣
...         > 3636 Buffalo Ave
...         > Topika, Kansas 20692
... """

>>> try:
...     data = nt.loads(content.replace('␣␣', '  '))
... except nt.NestedTextError as e:
...     print(e.get_message())
...     print(e.get_codicil()[0])
invalid indentation.
   4 «    address:  »
   5 «        > 3636 Buffalo Ave»
```

Notice the ␣␣ that follows *address* in *content*. These are replaced by 2 spaces before *content* is processed by *loads*. Thus, in this case there is an extra space at the end of line 4. Anything beyond the ': ' is considered the value for *address*, and in this case that is the single extra space specified at the end of the line. This extra space is taken to be the value of *address*, making the multiline string in lines 5 and 6 a value too many.

3.7 File format

The *NestedText* format follows a small number of simple rules. Here they are.

Encoding:

A *NestedText* document encoded in UTF-8.

Line types:

Each line in a *NestedText* document is assigned one of the following types: *comment*, *blank*, *list-item*, *dict-item*, and *string-item*. Any line that does not fit one of these types is an error.

Comments:

Comments are lines that have # as the first non-space character on the line. Comments are ignored.

Blank lines:

Blank lines are lines that are empty or consist only of white space characters (spaces or tabs). Blank lines are also ignored.

Line-type tags:

The remaining lines are identifying by which one of these ASCII characters are found in an unquoted portion of the line: dash ('-'), colon (':'), or greater-than symbol ('>') when followed immediately by a space or newline. Once the first of one of these pairs has been found in the unquoted portion of the line, any subsequent occurrences of any of the line-type tags are treated as simple text. For example:

```
- And the winner is: {winner}
```

In this case the leading '- ' determines the type of the line and the ': ' is simply treated as part of the remaining text on the line.

String items:

If the first non-space character on a line is a greater-than symbol followed immediately by a space ('>␣') or a newline, the line is a *string-item*. Adjacent string-items with the same indentation level are combined into a multiline string with their order being retained. Any leading white space that follows the space that follows the greater-than symbol is retained, as is any trailing white space.

List items:

If the first non-space character on a line is a dash followed immediately by a space ('-␣') or a newline, the line is a *list-item*. Adjacent list-items with the same indentation level are combined into a list with their order being retained. Each list-item has a single associated value.

Dictionary items:

If the line is not a string-item or a list item and it contains a colon followed by either a space (':␣') that does not fall within a quoted key or is followed by a newline, the line is considered a *dict-item*. Adjacent dict-items with the same indentation level are combined into a dictionary with their order being retained. Each dict-item consists of a key, the colon, and a value. A key must be a string, it must not contain a newline, and it must be quoted if it starts with a line-type or string-type tag or it contains a dict-item tag or if it is delimited by matching quote characters or has leading or trailing spaces. A key is quoted by delimiting it with matching single or double quote characters. Double quotes are used if the key contains a single quote character and a single quotes are used if the key contains a double quote character. A key that requires quoting must not contain both single and double quote characters.

Values:

The value associated with a list and dict item may take one of three forms.

If the line contains further text (characters after the dash-space or colon-space), then the value is that text.

If there is no further text on the line and the next line has greater indentation, then the next line holds the value, which may be a list, a dictionary, or a multiline string.

Otherwise the value is empty; it is taken to be an empty string.

String values may contain any printing UTF-8 character.

Indentation:

An increase in the number of spaces in the indentation signifies the start of a nested object. Indentation must return to a prior level when the nested object ends.

Each level of indentation need not employ the same number of additional spaces, though it is recommended that you choose either 2 or 4 spaces to represent a level of nesting and you use that consistently throughout the document. However, this is not required. Any increase in the number of spaces in the indentation represents an indent and a decrease to return to a prior indentation represents a dedent.

An indent may only follow a list-item or dict-item that does not have a value on the same line.

Only spaces are allowed in the indentation. Specifically, tabs are not allowed.

Empty document:

A document may be empty. A document is empty if it consists only of comments and blank lines.

Result:

When a document is converted from *NestedText* the result takes one of the following forms:

None: The document is empty.

String: The document consists of a single multiline string

List: The top-level of the document is a list. Each value of the list may be a string, list, or a dictionary. The nesting of lists and dictionaries may be arbitrarily deep but the leaf values are all strings as are all keys in all dictionaries.

Dictionary: The top-level of the document is a dictionary. Each value may be a string, list, or a dictionary. The nesting of lists and dictionaries may be arbitrarily deep but the leaf values are all strings as are all keys in all dictionaries.

3.8 Python API

`nestedtext.dumps`(obj, *[, sort_keys, ...])	Recursively convert object to *NestedText* string.
`nestedtext.dump`(obj, f, **kwargs)	Write the *NestedText* representation of the given object to the given file.
`nestedtext.loads`(content[, source, on_dup])	Loads *NestedText* from string.
`nestedtext.load`([f, on_dup])	Loads *NestedText* from file or stream.
`nestedtext.NestedTextError`(*args, **kwargs)	The *load* and *dump* functions all raise *NestedTextError* when they discover an error.

3.8.1 nestedtext.dumps

`nestedtext.dumps(obj, *, sort_keys=False, indent=4, renderers=None, default=None, level=0)`

Recursively convert object to *NestedText* string.

Parameters

- **obj** – The object to convert to *NestedText*.
- **sort_keys** (*bool or func*) – Dictionary items are sorted by their key if *sort_keys* is true. If a function is passed in, it is used as the key function.
- **indent** (*int*) – The number of spaces to use to represent a single level of indentation. Must be one or greater.
- **renderers** (*dict*) – A dictionary where the keys are types and the values are render functions (functions that take an object and convert it to a string). These will be used to convert values to strings during the conversion.
- **default** (*str or func*) – The default renderer. Use to render otherwise unrecognized objects to strings. If not provided an error will be raised for unsupported data types. Typical values are *repr* or *str*. If 'strict' is specified then only dictionaries, lists, strings, and those types specified in *renderers* are allowed. If *default* is not specified then a broader collection of value types are supported, including *None*, *bool*, *int*, *float*, and *list*- and *dict*-like objects.
- **level** (*int*) – The number of indentation levels. When dumps is invoked recursively this is used to increment the level and so the indent. Generally not specified by the user, but can be useful in unusual situations to specify an initial indent.

Returns The *NestedText* content.

Raises *NestedTextError* – if there is a problem in the input data.

Examples

This example writes to a string, but it is common to write to a file. The file name and extension are arbitrary. However, by convention a '.nt' suffix is generally used for *NestedText* files.

```
>>> import nestedtext as nt

>>> data = {
...     'name': 'Kristel Templeton',
...     'sex': 'female',
...     'age': '74',
... }

>>> try:
...     print(nt.dumps(data))
... except nt.NestedTextError as e:
...     print(str(e))
name: Kristel Templeton
sex: female
age: 74
```

The *NestedText* format only supports dictionaries, lists, and strings. By default, *dumps* is configured to be rather forgiving, so it will render many of the base Python data types, such as *None*, *bool*, *int*, *float* and list-like types such as *tuple* and *set* by converting them to the types supported by the format. This implies that a round trip through *dumps* and *loads* could result in the types of values being transformed. You can prevent this by passing *default='strict'* to *dumps*. Doing so means that values that are not dictionaries, lists, or strings generate exceptions.

```
>>> data = {'key': 42, 'value': 3.1415926, 'valid': True}

>>> try:
...     print(nt.dumps(data))
... except nt.NestedTextError as e:
...     print(str(e))
key: 42
value: 3.1415926
valid: True

>>> try:
...     print(nt.dumps(data, default='strict'))
... except nt.NestedTextError as e:
...     print(str(e))
42: unsupported type.
```

Alternatively, you can specify a function to *default*, which is used to convert values to strings. It is used if no other converter is available. Typical values are *str* and *repr*.

```
>>> class Color:
...     def __init__(self, color):
...         self.color = color
...     def __repr__(self):
...         return f'Color({self.color!r})'
...     def __str__(self):
...         return self.color

>>> data['house'] = Color('red')
>>> print(nt.dumps(data, default=repr))
key: 42
value: 3.1415926
valid: True
house: Color('red')

>>> print(nt.dumps(data, default=str))
key: 42
value: 3.1415926
valid: True
house: red
```

You can also specify a dictionary of renderers. The dictionary maps the object type to a render function.

```
>>> renderers = {
...     bool: lambda b: 'yes' if b else 'no',
...     int: hex,
...     float: lambda f: f'{f:0.3}',
...     Color: lambda c: c.color,
... }

>>> try:
...    print(nt.dumps(data, renderers=renderers))
... except nt.NestedTextError as e:
...     print(str(e))
key: 0x2a
value: 3.14
```

(continues on next page)

(continued from previous page)

```
valid: yes
house: red
```

If the dictionary maps a type to *None*, then the default behavior is used for that type. If it maps to *False*, then an exception is raised.

```
>>> renderers = {
...     bool: lambda b: 'yes' if b else 'no',
...     int: hex,
...     float: False,
...     Color: lambda c: c.color,
... }

>>> try:
...    print(nt.dumps(data, renderers=renderers))
... except nt.NestedTextError as e:
...     print(str(e))
3.1415926: unsupported type.
```

Both *default* and *renderers* may be used together. *renderers* has priority over the built-in types and *default*. When a function is specified as *default*, it is always applied as a last resort.

3.8.2 nestedtext.dump

`nestedtext.`**`dump`**(*obj*, *f*, ***kwargs*)

Write the *NestedText* representation of the given object to the given file.

Parameters

- **obj** – The object to convert to *NestedText*.
- **f** (*str, os.PathLike, io.TextIOBase*) – The file to write the *NestedText* content to. The file can be specified either as a path (e.g. a string or a *pathlib.Path*) or as a text IO instance (e.g. an open file). If a path is given, the will be opened, written, and closed. If an IO object is given, it must have been opened in a mode that allows writing (e.g. `open(path, 'w')`), if applicable. It will be written and not closed.

  The name used for the file is arbitrary but it is tradition to use a .nt suffix.
- **kwargs** – See *dumps()* for optional arguments.

Returns The *NestedText* content.

Raises

- ***NestedTextError*** – if there is a problem in the input data.
- **OSError** – if there is a problem opening the file.

Examples

This example writes to a pointer to an open file.

```
>>> import nestedtext as nt
>>> from inform import fatal, os_error

>>> data = {
...     'name': 'Kristel Templeton',
...     'sex': 'female',
...     'age': '74',
... }

>>> try:
...     with open('data.nt', 'w', encoding='utf-8') as f:
...         nt.dump(data, f)
... except nt.NestedTextError as e:
...     fatal(e)
... except OSError as e:
...     fatal(os_error(e))
```

This example writes to a file specified by file name.

```
>>> try:
...     nt.dump(data, 'data.nt')
... except nt.NestedTextError as e:
...     fatal(e)
... except OSError as e:
...     fatal(os_error(e))
>>> data = {'key': 42, 'value': 3.1415926, 'valid': True}
```

3.8.3 nestedtext.loads

`nestedtext.`**`loads`**(*content*, *source=None*, ***, *on_dup=None*)

Loads *NestedText* from string.

Parameters

- **`content`** (*`str`*) – String that contains encoded data.
- **`source`** (*`str or Path`*) – If given, this string is attached to any error messages as the culprit. It is otherwise unused. Is often the name of the file that originally contained the NestedText content.
- **`on_dup`** (*`str or func`*) – Indicates how duplicate keys in dictionaries should be handled. By default they raise exceptions. Specifying 'ignore' causes them to be ignored. Specifying 'replace' results in them replacing earlier items. By specifying a function, the keys can be de-duplicated. This call-back function returns a new key and takes four arguments:

  1. The new key (duplicates an existing key).
  2. The new value.
  3. The entire dictionary as it is at the moment the duplicate key is found.
  4. The state; a dictionary that is created as the *loads* is called and deleted as it returns. Values placed in this dictionary are retained between multiple calls to this call back function.

Returns The extracted data. If content is empty, None is returned.

Raises `NestedTextError` – if there is a problem in the *NextedText* content.

Examples

NestedText is specified to *loads* in the form of a string:

```
>>> import nestedtext as nt

>>> contents = """
... name: Kristel Templeton
... sex: female
... age: 74
... """

>>> try:
...     data = nt.loads(contents)
... except nt.NestedTextError as e:
...     e.terminate()

>>> print(data)
{'name': 'Kristel Templeton', 'sex': 'female', 'age': '74'}
```

loads() takes an optional second argument, *culprit*. If specified, it will be prepended to any error messages. It is often used to designate the source of *contents*. For example, if *contents* were read from a file, *culprit* would be the file name. Here is a typical example of reading *NestedText* from a file:

```
>>> filename = 'examples/duplicate-keys.nt'
>>> try:
...     with open(filename, encoding='utf-8') as f:
...         addresses = nt.loads(f.read(), filename)
... except nt.NestedTextError as e:
...     print(e.render())
...     print(*e.get_codicil(), sep="\n")
examples/duplicate-keys.nt, 5: duplicate key: name.
   4 «name:»
   5 «name:»
```

Notice in the above example the encoding is explicitly specified as 'utf-8'. *NestedText* files should always be read and written using *utf-8* encoding.

The following examples demonstrate the various ways of handling duplicate keys:

```
>>> content = """
... key: value 1
... key: value 2
... key: value 3
... name: value 4
... name: value 5
... """

>>> print(nt.loads(content))
Traceback (most recent call last):
...
nestedtext.NestedTextError: 3: duplicate key: key.

>>> print(nt.loads(content, on_dup='ignore'))
```

(continues on next page)

(continued from previous page)

```
{'key': 'value 1', 'name': 'value 4'}

>>> print(nt.loads(content, on_dup='replace'))
{'key': 'value 3', 'name': 'value 5'}

>>> def de_dup(key, value, data, state):
...     if key not in state:
...         state[key] = 1
...     state[key] += 1
...     return f"{key}#{state[key]}"

>>> print(nt.loads(content, on_dup=de_dup))
{'key': 'value 1', 'key#2': 'value 2', 'key#3': 'value 3', 'name': 'value 4',
↪'name#2': 'value 5'}
```

3.8.4 nestedtext.load

`nestedtext.`**`load`**(*f=None*, *on_dup=None*)

Loads *NestedText* from file or stream.

Is the same as `loads()` except the *NextedText* is accessed by reading a file rather than directly from a string. It does not keep the full contents of the file in memory and so is more memory efficient with large files.

Parameters

- **f** (*str, os.PathLike, io.TextIOBase, collections.abc.Iterator*) – The file to read the *NestedText* content from. This can be specified either as a path (e.g. a string or a *pathlib.Path*), as a text IO object (e.g. an open file), or as an iterator. If a path is given, the file will be opened, read, and closed. If an IO object is given, it will be read and not closed; utf-8 encoding should be used.. If an iterator is given, it should generate full lines in the same manner that iterating on a file descriptor would.
- **on_dup** – See `loads()` description of this argument.

Returns The extracted data. If content is empty, None is returned.

Raises

- ***NestedTextError*** – if there is a problem in the *NextedText* content.
- **OSError** – if there is a problem opening the file.

Examples

Load from a path specified as a string:

```
>>> import nestedtext as nt
>>> print(open('examples/groceries.nt').read())
- Bread
- Peanut butter
- Jam


>>> nt.load('examples/groceries.nt')
['Bread', 'Peanut butter', 'Jam']
```

Load from a *pathlib.Path*:

```
>>> from pathlib import Path
>>> nt.load(Path('examples/groceries.nt'))
['Bread', 'Peanut butter', 'Jam']
```

Load from an open file object:

```
>>> with open('examples/groceries.nt') as f:
...     nt.load(f)
...
['Bread', 'Peanut butter', 'Jam']
```

3.8.5 nestedtext.NestedTextError

exception nestedtext.**NestedTextError**(**args*, ***kwargs*)

The *load* and *dump* functions all raise *NestedTextError* when they discover an error. *NestedTextError* subclasses both the Python *ValueError* and the *Error* exception from *Inform*. You can find more documentation on what you can do with this exception in the Inform documentation.

The exception provides the following attributes:

source:

The source of the *NestedText* content, if given. This is often a filename.

line:

The text of the line of *NestedText* content where the problem was found.

lineno:

The number of the line where the problem was found.

colno:

The number of the character where the problem was found on *line*.

prev_line:

The text of the line immediately before where the problem was found.

template:

The possibly parameterized text used for the error message.

As with most exceptions, you can simply cast it to a string to get a reasonable error message.

```
>>> from textwrap import dedent
>>> import nestedtext as nt

>>> content = dedent("""
...     name1: value1
...     name1: value2
...     name3: value3
... """).strip()

>>> try:
...     print(nt.loads(content))
... except nt.NestedTextError as e:
...     print(str(e))
2: duplicate key: name1.
```

You can also use the *report* method to print the message directly. This is appropriate if you are using *inform* for your messaging as it follows *inform*'s conventions:

```
>> try:
..     print(nt.loads(content))
.. except nt.NestedTextError as e:
..     e.report()
error: 2: duplicate key: name1.
    «name1: value2»
```

The *terminate* method prints the message directly and exits:

```
>> try:
..     print(nt.loads(content))
.. except nt.NestedTextError as e:
..     e.terminate()
error: 2: duplicate key: name1.
    «name1: value2»
```

With exceptions generated from `load()` or `loads()` you may see extra lines at the end of the message that show the problematic lines if you have the exception report itself as above. Those extra lines are referred to as the codicil and they can be very helpful in illustrating the actual problem. You do not get them if you simply cast the exception to a string, but you can access them using `NestedTextError.get_codicil()`. The codicil or codicils are returned as a tuple. You should join them with newlines before printing them.

```
>>> try:
...     print(nt.loads(content))
... except nt.NestedTextError as e:
...     print(e.get_message())
...     print(*e.get_codicil(), sep="\n")
duplicate key: name1.
   1 «name1: value1»
   2 «name1: value2»
```

Note the « and » characters in the codicil. They delimit the extend of the text on each line and help you see troublesome leading or trailing white space.

Exceptions produced by *NestedText* contain a *template* attribute that contains the basic text of the message. You can change this message by overriding the attribute using the *template* argument when using *report*, *terminate*, or *render*. *render* is like casting the exception to a string except that allows for the passing of arguments. For example, to convert a particular message to Spanish, you could use something like the following.

```
>>> try:
...     print(nt.loads(content))
... except nt.NestedTextError as e:
...     template = None
...     if e.template == 'duplicate key: {}.':
...         template = 'llave duplicada: {}.'
...     print(e.render(template=template))
2: llave duplicada: name1.
```

- genindex

INDEX