NestedText

Release 3.8

Author name not set

Mar 23, 2026

LANGUAGE

Authors: Ken & Kale Kundert
Version: 3.8
Released: 2025-12-26
Documentation: nestedtext.org
Please post all questions, suggestions, and bug reports to GitHub.

NestedText is a file format for holding structured data. It is similar in concept to JSON, except that *NestedText* is designed to make it easy for people to enter, edit, or view the data directly. It organizes the data into a nested collection of name-value pairs, lists, and strings. The syntax is intended to be very simple and intuitive for most people.

A unique feature of this file format is that it only supports one scalar type: strings. As such, quoting strings is unnecessary, and without quoting there is no need for escaping. While the decision to forego other types (integers, reals, Booleans, etc.) may seem counter productive, it leads to simpler data files and applications that are more robust.

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

```
# Contact information for our officers

Katheryn McDaniel:
    position: president
    address:
        > 138 Almond Street
        > Topeka, 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

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

CHAPTER

ONE

TYPICAL APPLICATIONS

1.1 Configuration

Configuration files are an attractive application for *NestedText*. *NestedText* configuration files tend to be simple, clean and unambiguous. Plus, they handle hierarchy much better than alternatives such as Ini and TOML. Assimilate is an example of an application that uses *NestedText* for configuration.

1.2 Structured Code

One way to build tools to tackle difficult and complex tasks is to provide an application specific language. That can be a daunting challenge. However, in certain cases, such as specifying complex configurations, *NestedText* can help make the task much easier. *NestedText* conveys the structure of the data leaving the end application to interpret the data itself. It can do so with a collection of small parsers that are tailored to the specific piece of data to which they are applied. This generally results in a simpler specification since each piece of data can be given in its natural format, which might otherwise confuse a shared parser. In this way, rather than building one large very general language and parser, a series of much smaller and simpler parsers are needed. These smaller parsers can be as simple as splitters or partitioners, value checkers, or converters for numbers in special forms (numbers with units, times or dates, GPS coordinates, etc.). Or they could be full-blown expression evaluators or mini-languages. Structured code provides a nice middle ground between data and code and its use is growing in popularity.

An example of structured code is provided by GitHub with its workflow specification files. They use YAML. Unfortunately, the syntax of the code snippets held in the various fields can be confused with *YAML* syntax, which leads to unnecessary errors, confusion, and complexity. JSON suffers from similar problems. *NestedText* excels for these applications as it holds code snippets without any need for quoting or escaping. *NestedText* provides simple unambiguous rules for defining the structure of your data and when these rules are followed there is no way for any syntax or special characters in the values of your data to be confused with *NestedText* syntax. In fact, it is possible for *NestedText* to hold *NestedText* snippets without conflict.

Another example of structured code is provided by the files that contain the test cases used by Parametrize From File, a PyTest plugin. *Parametrize From File* simplifies the task of specifying test cases for *PyTest* by separating the test cases from the test code. Here it is being applied to test a command line program. Its response is checked using regular expressions. Each entry includes a shell command to run the program and a regular expression that must match the output for the test to pass:

```
-
    cmd: emborg version
    expected: emborg version: \d+\.\d+(\.\d+(\.?\w+\d+)?)?  \(\d\d\d\d-\d\d-\d\d\)
    expected type: regex
-
    cmd: emborg --quiet files -D
    expected:
```

(continues on next page)

(continued from previous page)

```
        > Archive: home-\d\d\d\d-\d\d-\d\dT\d\d:\d\d:\d\d
        > \d\d\d\d-\d\d-\d\dT\d\d:\d\d:\d\d.\d\d\d\d\d\d configs/subdir/(file|)
        > \d\d\d\d-\d\d-\d\dT\d\d:\d\d:\d\d.\d\d\d\d\d\d configs/subdir/(file|)
            # Unfortunately, we cannot check the order as they were both
            # created at the same time.
    expected type: regex
-
    cmd: emborg due --backup-days 1 --message "{elapsed} since last {action}"
    expected: home: (\d+(\.\d)? (seconds|minutes)) since last backup\.
    expected type: regex
```

Notice that the regular expressions are given clean, without any additional quoting or escaping.

1.3 Composable Utilities

Another attractive use-case for *NestedText* is command line programs whose output is meant to be consumed by either people or other programs. This is another growing trend. Many programs do this by supporting a `--json` command-line flag that indicates the output should be computer readable rather than human readable. But, with *NestedText* it is not necessary to make people choose. Just output the result in *NestedText* and it can be read by people or computers. For example, consider a program that reads your address list and output particular fields on demand:

```
> address --email
Katheryn McDaniel: KateMcD@aol.com
Margaret Hodge: margaret.hodge@ku.edu
```

This output could be fed directly into another program that accepts *NestedText* as input:

```
> address --email | mail-to-list
```

CHAPTER

TWO

CONTRIBUTING

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

Also, please consider using *NestedText* for any applications you create.

2.1 The Zen of *NestedText*

NestedText aspires to be a simple dumb receptacle that holds peoples' structured data and does so in a way that allows people to easily interact with that data.

The desire to be simple is an attempt to minimize the effort required to learn and use the language. Ideally, people can understand it by looking at a few examples. And ideally, they can use it without needing to remember any arcane rules or relying on any knowledge that programmers accumulate through years of experience. One source of simplicity is consistency. As such, *NestedText* uses a small number of rules that it applies with few exceptions.

The desire to be dumb means that *NestedText* does not transform the data in any meaningful way to avoid creating unpleasant surprises. Specifically, when you enter *NestedText* you are specifying a sequence of characters. Beyond extracting the structure of the data, *NestedText* does not interpret what you mean by a particular sequence of characters. All leaf values are left as strings. Any interpretation or conversion of the data must be explicitly done by you after the data is read. After all, you understand what the data is supposed to mean, so you are in the best position to interpret it. There are many powerful tools available to help with *this exact task*, and they turn what may initially seem like a burden into a substantial advantage.

2.1.1 The Genesis of *NestedText*

When people make a list they often start out by numbering the items. For example, if you are making a list of toiletries to take on a trip you might create the following list:

```
1. tooth brush
2. tooth paste
3. floss
4. shaver
```

This is an enumerated list.

But as you start editing the list, you may find that you are spending too much time renumbering the items, and so you often convert to the simpler system of identifying each item with a bullet, such as:

```
- tooth brush
- tooth paste
- floss
- shaver
```

This is also an enumerated list, but here the enumeration is implicit.

This is the form used by *NestedText* to identify the items in a list. It is a very natural form that people tend to use instinctively.

Then if the list grows and you want to add sections. The natural approach is the following:

```
toiletries:
    - tooth brush
    - tooth paste
    - floss
    - shaver
```

This type of list, where a key is specified rather than a bullet, is a keyed list. Again, this is the form used by *NestedText*.

Notice it was natural to nest our first list into this new list by indenting it.

Things become a bit more complicated when list item values grow to be more than one line long. In this case there are two common approaches. This first is to simply continue on the next line while maintaining the indent. For example:

```
toiletries:
    - tooth brush
      the green one
    - tooth paste
      a new tube
    - floss
      mint flavor
    - shaver
      the electric one
```

A second approach is to place all the text on a separate line from the bullet, as in:

```
toiletries:
    -
      tooth brush
      the green one
    -
      tooth paste
      a new tube
    -
      floss
      mint flavor please, its delightful
    -
      shaver
      the electric one
```

If one had to choose one format to use every time, one would generally choose the second because it is more natural for keyed lists.

However, there are more issues. Imagine that the multiline text is indented. How would that be indicated? Or perhaps there are leading or trailing blank lines, how would those be distinguished from the empty space used to make our list more readable. To resolve these issues, *NestedText* deviates from normal convention by asking you to draw the left border of the text. For example:

```
Step 3, Pressure Cook:
    |     Close the lid and seal the vent. Press the “pressure cook” button
```

(continues on next page)

(continued from previous page)

```
    | and set the time to 10 minutes. After this, allow natural pressure
    | release for 5 minutes, then manually release the remaining pressure.
```

Doing so makes any indention and blank lines visible and unambiguous. As such, it is the approach that *NestedText* takes. However it does not use the '|' character to indicate the border. It was found that with most text editors the '>' character works better because the editor treats it as a continuation character, meaning once the multiline text was started, typing a new line automatically brings the next '>' character with an additional space of indent. This allows you to type multiple lines without consciously entering these continuation characters, making the process very efficient. Thus, with *NestedText* you would enter:

```
Step 3, Pressure Cook:
    >     Close the lid and seal the vent. Press the "pressure cook" button
    > and set the time to 10 minutes. After this, allow natural pressure
    > release for 5 minutes, then manually release the remaining pressure.
```

An there you have it. This, plus a few rules to eliminate any remaining ambiguities and to handle some unusual special cases and you have *NestedText*.

2.2 Alternatives

There are no shortage of well established alternatives to *NestedText* for storing data in a human-readable text file. The features and shortcomings of some of these alternatives are discussed next. *NestedText* is intended to be used in situations where people either create, modify, or consume the data directly. It is this perspective that informs these comparisons.

2.2.1 JSON

JSON is a subset of JavaScript suitable for holding data. Like *NestedText*, it consists of a hierarchical collection of objects (dictionaries), lists, and strings, but also allows numbers, Booleans and nulls. In practice, JSON is largely generated and consumed by machines. The data is stored as text, and so can be read, modified, and consumed directly by the end user, but the format is not optimized for this use case and so is often cumbersome or inefficient when used in this manner.

JSON supports all the native data types common to most languages. Syntax is added to values to unambiguously indicate their type. For example, `2`, `2.0`, and `"2"` are three different values with three different types (integer, real, string). This adds two types of complexity. First, the rules for distinguishing various types must be learned and used. Second, all strings must be quoted, and with quoting comes escaping, which is needed to allow quote characters to be included in strings.

JSON was derived as a subset of JavaScript, and so inherits a fair amount of syntactic clutter that can be annoying for users to enter and maintain. In addition, features that would improve clarity are lacking. Comments are not allowed, multiline strings are not supported, and whitespace is insignificant (leading to the possibility that the appearance of the data may not match its true structure).

NestedText only supports three data types (strings, lists and dictionaries) and does not have the baggage of being the subset of a general purpose programming language. The result is a simpler language that has the following clear advantages over JSON as a human readable and writable data file format:

- strings do not require quotes
- comments
- multiline strings
- no need to escape special characters

- commas are not used to separate dictionary and list items

The following examples illustrate the difference between JSON and *NestedText*:

```
{
    "treasurer": {
        "name": "Fumiko Purvis",
        "address": "3636 Buffalo Ave\nTopeka, Kansas 20692",
        "phone": "1-268-555-0280",
        "email": "fumiko.purvis@hotmail.com",
        "additional roles": [
            "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
        > Topeka, Kansas 20692
    phone: 1-268-555-0280
    email: fumiko.purvis@hotmail.com
    additional roles:
        - accounting task force
```

2.2.2 YAML

YAML is considered by many to be a human friendly alternative to JSON. There is less syntactic clutter and the quoting of strings is optional. However, it also supports a wide variety of data types and formats. The optional quoting can result in the type of values being ambiguous. To distinguish between the various types, a complicated and non-intuitive set of rules developed. 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 nine different ways of writing multiline strings.

YAML avoids excessive quoting and supports comments and multiline strings, but the multitude of formats and disambiguation rules make YAML a difficult language to learn, and the ambiguities creates traps for the user. To illustrate these points, the following is a condensation of a YAML document taken from the GitHub documentation that describes how to configure continuous integration using Python:

```
name: Python package
on: [push]
build:
  python-version: [3.6, 3.7, 3.8, 3.9, 3.10]
  steps:
    - name: Install dependencies
      run: |
```

(continues on next page)

(continued from previous page)

```
        python -m pip install --upgrade pip
        pip install pytest
        if [ -f 'requirements.txt' ]; then pip install -r requirements.txt; fi
    - name: Test with pytest
      run: |
        pytest
```

And here is the result of running that document through the Python YAML reader and writer.

```
name: Python package
true:
- push
build:
  python-version:
  - 3.6
  - 3.7
  - 3.8
  - 3.9
  - 3.1
  steps:
  - name: Install dependencies
    run: 'python -m pip install --upgrade pip

      pip install pytest

      if [ -f ''requirements.txt'' ]; then pip install -r requirements.txt; fi

      '
  - name: Test with pytest
    run: 'pytest

      '
```

There are a few things to notice about this second version.

1. `on` key was inappropriately converted to `true`.
2. Python version `3.10` was inappropriately converted to `3.1`.
3. The multiline string was converted to a different representation that added blank lines between each line, greatly confusing the presentation of the string.
4. Escaping was required for the quotes on `'requirements.txt'`.
5. Indentation is not an accurate reflection of nesting (notice that `python-version` and `- 3.6` have the same indentation, but `- 3.6` is contained inside `python-version`).

One might expect that the format might change a bit while the underlying information remains constant. But that is not the case. The ambiguities in the format result in both `on` and `3.10` being changed in value and meaning.

Now consider the *NestedText* version; it is simpler and not subject to misinterpretation.

```
name: Python package
on:
    - push
build:
    python-version:
        - 3.6
        - 3.7
        - 3.8
        - 3.9
        - 3.10
    steps:
        -
            name: Install dependencies
            run:
                > python -m pip install --upgrade pip
                > pip install pytest
                > if [ -f 'requirements.txt' ]; then pip install -r requirements.txt; fi
        -
            name: Test with pytest
            run: pytest
```

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

- simple
- unambiguous (no implicit typing)
- no unexpected conversions of the data
- syntax is insensitive to special characters within text
- safe, no risk of malicious code execution
- round-tripping from *NestedText* does not result in changed values or ugly and confusing presentations

2.2.3 TOML or INI

TOML is a configuration file format inspired by the well-known INI syntax. It supports a number of basic data types (notably including dates and times) using syntax that is more similar to JSON (explicit but verbose) than to YAML (succinct but confusing). As discussed previously, though, this makes it the responsibility of the user to specify the correct type for each field.

Another flaw in TOML is that it is difficult to specify deeply nested structures. The only way to specify a nested dictionary is to give the full key to that dictionary, relative to the root of the entire hierarchy. This is not much a problem if the hierarchy only has 1-2 levels, but any more than that and you find yourself typing the same long keys over and over. A corollary to this is that TOML-based configurations do not scale well: increases in complexity are often accompanied by disproportionate decreases in readability and writability.

Here is an example of a configuration file in TOML and *NestedText*:

```
[plugins]
auth = ['avendesora']
archive = ['ssh', 'gpg', 'avendesora', 'emborg', 'file']
publish = ['scp', 'mount']

[auth.avendesora]
account = 'login'
field = 'passcode'

[archive.file]
src = ['~/src/nfo/contacts']
[archive.avendesora]
[archive.emborg]
config = 'rsync'

[publish.scp]
host = ['backups']
remote_dir = 'archives/{date:YYMMDD}'

[publish.mount]
drive = '/mnt/secrets'
remote_dir = 'sparekeys/{date:YYMMDD}'
```

```
plugins:
    auth:
        - avendesora
    archive:
        - ssh
        - gpg
        - avendesora
        - emborg
        - file
    publish:
        - scp
        - mount
auth:
    avendesora:
        account: login
        field: passcode
archive:
    file:
        src:
            - ~/src/nfo/contacts
    avendesora:
        {}
    emborg:
        config: rsync
publish:
    scp:
        host:
            - backups
        remote_dir: archives/{date:YYMMDD}
```

(continues on next page)

(continued from previous page)

```
    mount:
        drive: /mnt/secrets
        remote_dir: sparekeys/{date:YYMMDD}
```

NestedText has the following clear advantages over TOML and INI as a human readable and writable data file format:

- text does not require quoting or escaping
- data is left in its original form
- indentation used to succinctly represent nested data
- the structure of the file matches the structure of the data
- heavily nested data is represented efficiently

2.2.4 CSV or TSV

CSV (comma-separated values) and the closely related TSV (tab-separated values) are exchange formats for tabular data. Tabular data consists of multiple records where each record is made up of a consistent set of fields. The format separates the records using line breaks and separates the fields using commas or tabs. Quoting and escaping is required when the fields contain line breaks or commas/tabs.

Here is an example data file in CSV and *NestedText*.

```
Year,Agriculture,Architecture,Art and Performance,Biology,Business,Communications and␣
↪Journalism,Computer Science,Education,Engineering,English,Foreign Languages,Health␣
↪Professions,Math and Statistics,Physical Sciences,Psychology,Public Administration,
↪Social Sciences and History
1970,4.22979798,11.92100539,59.7,29.08836297,9.064438975,35.3,13.6,74.53532758,0.8,65.
↪57092343,73.8,77.1,38,13.8,44.4,68.4,36.8
1980,30.75938956,28.08038075,63.4,43.99925716,36.76572529,54.7,32.5,74.98103152,10.3,65.
↪28413007,74.1,83.5,42.8,24.6,65.1,74.6,44.2
1990,32.70344407,40.82404662,62.6,50.81809432,47.20085084,60.8,29.4,78.86685859,14.1,66.
↪92190193,71.2,83.9,47.3,31.6,72.6,77.6,45.1
2000,45.05776637,40.02358491,59.2,59.38985737,49.80361649,61.9,27.7,76.69214284,18.4,68.
↪36599498,70.9,83.5,48.2,41,77.5,81.1,51.8
2010,48.73004227,42.06672091,61.3,59.01025521,48.75798769,62.5,17.6,79.61862451,17.2,67.
↪92810557,69,85,43.1,40.2,77,81.7,49.3
```

```
-
    Year: 1970
    Agriculture: 4.22979798
    Architecture: 11.92100539
    Art and Performance: 59.7
    Biology: 29.08836297
    Business: 9.064438975
    Communications and Journalism: 35.3
    Computer Science: 13.6
    Education: 74.53532758
    Engineering: 0.8
```

(continues on next page)

(continued from previous page)

```
    English: 65.57092343
    Foreign Languages: 73.8
    Health Professions: 77.1
    Math and Statistics: 38
    Physical Sciences: 13.8
    Psychology: 44.4
    Public Administration: 68.4
    Social Sciences and History: 36.8
-
    Year: 1980
    Agriculture: 30.75938956
    Architecture: 28.08038075
    Art and Performance: 63.4
    Biology: 43.99925716
    Business: 36.76572529
    Communications and Journalism: 54.7
    Computer Science: 32.5
    Education: 74.98103152
    Engineering: 10.3
    English: 65.28413007
    Foreign Languages: 74.1
    Health Professions: 83.5
    Math and Statistics: 42.8
    Physical Sciences: 24.6
    Psychology: 65.1
    Public Administration: 74.6
    Social Sciences and History: 44.2
-
    Year: 1990
    Agriculture: 32.70344407
    Architecture: 40.82404662
    Art and Performance: 62.6
    Biology: 50.81809432
    Business: 47.20085084
    Communications and Journalism: 60.8
    Computer Science: 29.4
    Education: 78.86685859
    Engineering: 14.1
    English: 66.92190193
    Foreign Languages: 71.2
    Health Professions: 83.9
    Math and Statistics: 47.3
    Physical Sciences: 31.6
    Psychology: 72.6
    Public Administration: 77.6
    Social Sciences and History: 45.1
-
    Year: 2000
    Agriculture: 45.05776637
    Architecture: 40.02358491
    Art and Performance: 59.2
    Biology: 59.38985737
```

(continues on next page)

(continued from previous page)

```
    Business: 49.80361649
    Communications and Journalism: 61.9
    Computer Science: 27.7
    Education: 76.69214284
    Engineering: 18.4
    English: 68.36599498
    Foreign Languages: 70.9
    Health Professions: 83.5
    Math and Statistics: 48.2
    Physical Sciences: 41
    Psychology: 77.5
    Public Administration: 81.1
    Social Sciences and History: 51.8
-
    Year: 2010
    Agriculture: 48.73004227
    Architecture: 42.06672091
    Art and Performance: 61.3
    Biology: 59.01025521
    Business: 48.75798769
    Communications and Journalism: 62.5
    Computer Science: 17.6
    Education: 79.61862451
    Engineering: 17.2
    English: 67.92810557
    Foreign Languages: 69
    Health Professions: 85
    Math and Statistics: 43.1
    Physical Sciences: 40.2
    Psychology: 77
    Public Administration: 81.7
    Social Sciences and History: 49.3
```

It is hard to beat the compactness of *CSV* for tabular data, however *NestedText* has the following advantages over *CSV* and *TSV* as a human readable and writable data file format that may make it preferable in some situation:

- text does not require quoting or escaping
- arbitrary data hierarchies are supported
- file representation tends to be tall and skinny rather than short and fat
- easier to read

2.2.5 Really, Only Strings?

NestedText and its alternatives are all trying to represent structured data. Of them, only *NestedText* limits you to strings for the scalar values. The alternatives all allow other data types to be represented as well, such as integers, reals, Booleans, etc. Since real applications invariably require all these data types, you might think, "if I use *NestedText*, I'll have to convert all these strings myself, and that will make my application code more complicated". In fact, using *NestedText* will make your application code more robust with little to no increase in complexity:

For robustness, all data should be validated when reading it to assure there are no errors. This is performed conveniently and efficiently with a *schema*. Schemas are used to specify the expected type for each value and are easily extended to perform type conversion as needed. For example, if a particular value should be an integer but a string is provided, as with *NestedText*, the package that implements the schema can be configured to attempt to convert the string to an integer and only report an error if it cannot.

Applications that need to interpret the input data always make assumptions about the data being read. For example, email fields are expected to contain strings that can be interpreted as an email address. In practice, every field can and probably should be checked in some way. Even with *NestedText* that constrains the scalar values to strings, one must assure that a list or dictionary is not given where a string is expected. When every value is being checked there little to no benefit to the underlying data receptacle being aware the type of each value. Rather it is very constraining.

Supporting native data types raises its own issues:

NestedText gains simplicity by jettisoning native support for scalar data types other than strings. However it is important to recognize that the alternatives must do this as well. There are an unlimited number of data types that can be supported and they cannot support them all. Common data types that are generally not supported include dates, times, and quantities (numbers with units, such as $20.00 and 47 k). With all languages there is a decision to be made: what types should be supported natively. Each additional type increases the complexity of the format. If only strings are supported, as with *NestedText*, things are pretty simple. Adding any other data type then requires supporting quoting and escaping, which is a substantial jump up in complexity.

Data types that are not natively supported are generally passed as strings that are later converted to the right type by the end application. This approach actually provides substantial benefits. The end application has context that a general purpose data reader cannot have. For example, the date `10/07/08` could represent either 10 August 2008 or October 7, 2008, or perhaps even July 8, 2010. Only the user and the application would know which.

The type of the value `2` is ambiguous; it may be integer or real. This may cause problems when combined into an array, such as `[1.85, 1.94, 2, 2.09]`. A casually written program may choke on a non-homogeneous array that consists of an integer among the floats.

YAML is notorious for ambiguities because it allows unquoted strings. `2` is a valid integer, real, and string. Similarly, `no` is a valid Boolean and string. In fact, every single value in YAML that is not quoted is also a valid string. Many people that use YAML simply quote every string, but that does not solve all the problems because things that are not intended to be strings can be converted to strings, such as `09`.

There is also the issue of the internal representation of the data. Is the integer represented using 32 bits, 64 bits, or can the integer by arbitrarily large? Is a real number represented as a 64 bit or 128 bit float, or is it represented by a decimal or rational number? Are exceptional values such as infinity or not-a-number supported? Sometimes such things are specified in the definition of the format, but often they are left as details of the implementation. The result could be overflows, underflows, loss of precision, errors, and compatibility issues.

It is common to format real numbers so as to convey the meaningful precision of the number. For example, `2` or `2.` represents a number with one digit of precision, `2.0` represents a number with two digits of precision, `2.00` represents a number with three digits of precision, etc. This information on the precision of the number is lost when these numbers are converted to the float data type.

This same issue also causes problems when representing version numbers. The number `3.10` is used to represent version three point ten, but when converted to a float becomes version three point one.

There are also cases where multiple formats map to the same underlying data type. For example, integers may be given in binary, octal, decimal, or hexadecimal formats. YAML provides almost a dozen different ways to specify strings. This causes problems when round-tripping, which is where you read a file, perhaps process it, and then write it back out. Since the data is converted to an internal data type, the original formatting is lost, meaning that the program that writes out the data cannot know how it was originally specified. Integers are generally written out as decimal number

regardless of how they were specified. In YAML, the writer checks to see if a string contains a newline and if so simply chooses one of the 9 possible multiline string formats arbitrarily. This is why in the round-trip *YAML example* given above the Python script ends up being interleaved with blank lines.

Using *NestedText* also makes life easier for your end-users:

Casual users may not understand that `2` is treated differently than `2.0`, which may cause issues in applications that are not carefully written.

TOML natively accepts dates and times, but only in ISO-8601 formats. Casual users are unlikely to be familiar with this format or may find it awkward or cumbersome.

YAML natively accepts sexagesimal (base 60) numbers in the form `2:30:00`, which YAML converts to 9000. If this is a duration, it would likely imply 2 hours, 30 minutes and 0 seconds, which totals to 9000 seconds. It may be also used for the time of day. Someone that normally uses twelve hour time formatting might write `2:30:00 AM` and get a string. Someone that uses twenty-four hours formatting might write `2:30:00` and get the integer 9000, or they might write `02:30:00` and get a string. However, if they entered a time 12 hours later, `16:30:00`, they would get an integer again.

Native data types are distinguished from each other by using conventions that are second nature to programmers. Conventions such as "you must quote strings", "quote characters in strings must be escaped", "you escape an escape character by doubling it up", "real numbers must contain a decimal point" and "real numbers may not contain units".

Casual users are unlikely to know these conventions, which causes frustration and errors. Forcing them to know and use these conventions represents an undesirable and sometimes overwhelming burden. This is particularly true for YAML, which can be a minefield even for programmers. Consider the following:

`Hey there!` and `"Hey there!"` represent the same string.

`She said, "Hey there!"` is a valid string, but `"She said, "Hey there!""` is an error.

`She said, "Hey there!"` is a valid string, but `She said:  "Hey there!"` is an error (notice the comma is converted to a colon).

`3.10.4` is a string, but `3.10` is a real and `3` is an integer.

`10` is 10, but `010` is 8 and `09` is "09", a string.

`Now` is a string, but `No` is a Boolean.

`(1 + 2)` is a string, but `[1 + 2]` is a list.

`02:30:00` is a string but `2:30:00` is 9000.

Only programmers with substantial experience with YAML can anticipate or even understand this behavior.

Other languages have similar, but less extreme challenges, particularly the need for quoting and escaping.

Every additional supported data type brings a challenge; how to unambiguously distinguish it from the others. The challenge is particularly acute for strings because they consist of any possible sequence of characters and so can be confused with all other data types. *NestedText* addresses this issue by limiting the scalar values to only be strings. That way, there is no need to distinguish the strings from other possible data types.

The alternatives all distinguish strings by surrounding them with quotes. This adds visual clutter and makes them more difficult to type. This is not generally a problem if there are only a few stings, but it becomes a drag if there is are many. However, quoting brings another challenge. Since a string can consist of any sequence of characters, it can include the quote characters. Now the quote characters within the string must be distinguished from the quote characters that delimit the string; a process referred to as escaping the character. This is often done with an special

escape character, generally the backslash, but may be done by duplicating the character to be escaped. The string may naturally contain escape characters and they would need escaping as well. This represents a deep hole. For example, consider the following Python dictionary that contains a collection of regular expressions. The regular expressions are quoted strings that by their very nature generally require a large amount of escaping:

```
regexes = dict(
    double_quoted_string = r'"(?:[^"\\]|\\.)*"',
    single_quoted_string = r"'(?:[^'\\]|\\.)*'",
    identifier = r'[a-zA-Z_][a-zA-Z_0-9]*',
    number = r"[+-]?[0-9]+\.?[0-9]*(?:[eE][+-]?[0-9]+)?",
)
```

Converting this to JSON illustrates the problem:

```
{
    "double_quoted_string": "\"(?:[^\"\\\\]|\\\\.)*\"",
    "single_quoted_string": "'(?:[^'\\\\]|\\\\.)*'",
    "identifier": "[a-zA-Z_][a-zA-Z_0-9]*",
    "number": "[+-]?[0-9]+\\.?[0-9]*(?:[eE][+-]?[0-9]+)?"
}
```

The number of escape characters more than doubled. This problem does not occur in *NestedText*, which is actually cleaner than the original Python:

```
double_quoted_string: "(?:[^"\\]|\\.)*"
single_quoted_string: '(?:[^'\\]|\\.)*'
identifier: [a-zA-Z_][a-zA-Z_0-9]*
number: [+-]?[0-9]+\.?[0-9]*(?:[eE][+-]?[0-9]+)?
```

In general, users that are expected to read, write, or modify structured data benefit from formats tailored to their needs. That only happens when the values are passed as strings that are interpreted by the end application.

Native data types should only be used when both the data generator and the data consumer are machines, preferably using the same software packages to both read and write the data files. In such cases, only programmers would view or edit the files, and only in unusual cases.

Native data types provide little value but many drawbacks. By limiting the scalar values to be only strings, *NestedText* sidesteps all of these issues, and it is unique in that regard.

2.3 Language introduction

This is a overview of the syntax of a *NestedText* document, which consists of a *nested collection* of *dictionaries*, *lists*, and *strings* where indentation is used to indicate nesting. All leaf values must be simple text or empty. You can find more specifics *in the next section*.

2.3.1 Dictionaries

A dictionary is an ordered collection of key value pairs:

```
key 1: value 1
key 2: value 2
key 3: value 3
```

A dictionary item is a single key value pair. A dictionary is all adjacent dictionary items in which the keys all begin at the same level of indentation. There are several different ways to specify dictionaries.

In the first form, the key and value are separated by a dictionary tag, which is a colon followed by a space or newline (`:␣` or `:`). The key must be a string and must not start with a `-␣`, `>␣`, `:␣`, `[`, `{`, `#`, or white space character; or contain newline characters or the `:␣` character sequence. Any spaces between the key and the tag are ignored.

The value of this dictionary item may be a rest-of-line string, a multiline string, a list, or a dictionary. If it is a rest-of-line string, it contains all characters following the tag that separates the key from the value (`:␣`). For all other values, the rest of the line must be empty, with the value given on the next line, which must be further indented.

```
key 1: value 1
key 2:
key 3:
    - value 3a
    - value 3b
key 4:
    key 4a: value 4a
    key 4b: value 4b
key 5:
    > first line of value 5
    > second line of value 5
```

This is equivalent to the following JSON code:

```
{
    "key 1": "value 1",
    "key 2": "",
    "key 3": [
        "value 3a",
        "value 3b"
    ],
    "key 4": {
        "key 4a": "value 4a",
        "key 4b": "value 4b"
    },
    "key 5": "first line of value 5\nsecond line of value 5"
}
```

A second less common form of a dictionary item employs multiline keys. In this case there are no limitations on the key other than it being a string. Each line of a multiline key is introduced with a colon (`:`) followed by a space or newline. The key is all adjacent lines at the same level that start with a colon tag with the tags removed but leading and trailing white space retained, including all newlines except the last.

This form of dictionary does not allow rest-of-line string values; you would use a multiline string value instead:

```
: key 1
:     the first key
    > value 1
: key 2: the second key
```

(continues on next page)

(continued from previous page)

```
    - value 2a
    - value 2b
```

A dictionary may consist of dictionary items of either form.

The final form of a dictionary is the inline dictionary. This is a compact form where all the dictionary items are given on the same line. There is a bit of syntax that defines inline dictionaries, so the keys and values are constrained to avoid ambiguities in the syntax. An inline dictionary starts with an opening brace ({), ends with a matching closing brace (}), and contains inline dictionary items separated by commas (,). An inline dictionary item is a key and value separated by a colon (:). A space need not follow the colon. The keys are inline strings and the values may be inline strings, inline lists, and inline dictionaries. An empty dictionary is represented with {} (there can be no space between the opening and closing braces). Leading and trailing spaces are stripped from keys and string values within inline dictionaries.

For example:

```
{key 1: value 1, key 2: value 2, key 3: value 3}
```

```
{key 1: value 1, key 2: [value 2a, value 2b], key 3: {key 3a: value 3a, key 3b: value 3b}
↪}
```

2.3.2 Lists

A list is an ordered collection of values:

```
- value 1
- value 2
- value 3
```

A list item is introduced with a list tag: a dash followed by a space or a newline at the start of a line (-␣ or -). All adjacent list items at the same level of indentation form the list.

The value of a list item may be a rest-of-line string, a multiline string, a list, or a dictionary. If it is a rest-of-line string, it contains all characters that follow the tag that introduces the list item. For all other values, the rest of the line must be empty, with the value given on the next line, which must be further indented.

```
- value 1
-
-
    - value 3a
    - value 3b
-
    key 4a: value 4a
    key 4b: value 4b
-
    > first line of value 5
    > second line of value 5
```

Which is equivalent to the following JSON code:

```
[
    "value 1",
    "",
    [
```

(continues on next page)

(continued from previous page)

```
        "value 3a",
        "value 3b"
    ],
    {
        "key 4a": "value 4a",
        "key 4b": "value 4b"
    },
    "first line of value 5\nsecond line of value 5"
]
```

Another form of a list is the inline list. This is a compact form where all the list items are given on the same line. There is a bit of syntax that defines the list, so the values are constrained to avoid ambiguities in the syntax. An inline list starts with an opening bracket ([), ends with a matching closing bracket (]), and contains inline values separated by commas. The values may be inline strings, inline lists, and inline dictionaries. An empty list is represented by [] (there should be no space between the opening and closing brackets). Leading and trailing spaces are stripped from string values within inline lists.

For example:

```
[value 1, value 2, value 3]
```

```
[value 1, [value 2a, value 2b], {key 3a: value 3a, key 3b: value 3b}]
```

[ ] is not treated as an empty list as there is space between the brackets, rather this represents a list with a single empty string value. The contents of the brackets, which consists only of white space, is stripped of its padding, leaving an empty string.

2.3.3 Strings

There are three types of strings: rest-of-line strings, multiline strings, and inline strings. Rest-of-line strings are simply all the characters on a line that follow a list tag (-␣) or dictionary tag (:␣), including any leading or trailing white space. They can contain any character other than a newline. The content of the rest-of-line string starts after the first space that follows the dash or colon of the tag:

```
code   : input signed [7:0] level
regex  : [+-]?([0-9]*[.])?[0-9]+\s*\w*
math   : $x = \frac{{-b \pm \sqrt {b^2 - 4ac}}}{2a}$
unicode: José and François
```

Multi-line strings are all adjacent lines that are prefixed with a string tag; the greater-than symbol followed by a space or a newline (>␣ or >). The content of each line starts after the first space that follows the greater-than symbol:

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

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

```
>
> “The worth of a man to his society can be measured by the contribution he
>  makes to it - less the cost of sustaining himself and his mistakes in it.”
>
>                                                - Erik Jonsson
>
```

The multiline string is all adjacent lines that start with a string tag with the tags removed and the lines joined together with newline characters inserted between each line. Except for the space that follows the > in the tag, white space from both the beginning and the end of each line is retained, along with all newlines except the last.

Inline strings are the string values specified in inline dictionaries and lists. They are somewhat constrained in the characters that they may contain; nothing that might be confused with the syntax characters used by the inline list or dictionary that contains it. Specifically, inline strings may not contain newlines or any of the following characters: `[`, `]`, `{`, `}`, or `,`. In addition, inline strings that are contained in inline dictionaries may not contain `:`. Leading and trailing white space are ignored with inline strings.

2.3.4 Comments

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

```
# this line is ignored

# this line is also ignored, as is the blank line above.
```

Comment lines are ignored when determining whether adjacent lines belong to the same dictionary, list, or string. For example, the following represents one multiline string:

```
> this is the first line of a multiline string
# this line is ignored
> this is the second line of the multiline string
```

2.3.5 Nesting

A value for a dictionary or list item may be a rest-of-line string or it may be a nested dictionary, list, multiline string, or inline dictionary or list. Indentation is used to indicate nesting. 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

Katheryn McDaniel:
    position: president
    address:
        > 138 Almond Street
        > Topeka, Kansas 20697
    phone:
        cell: 1-210-555-5297
        work: 1-210-555-3423
        home: 1-210-555-8470
            # Katheryn prefers that we always call her on her cell phone.
    email: KateMcD@aol.com
    kids:
        - Joanie
        - Terrance

Margaret Hodge:
    position: vice president
    address:
        > 2586 Marigold Lane
```

(continues on next page)

(continued from previous page)

```
    > Topeka, Kansas 20697
phone:
    {cell: 1-470-555-0398, home: 1-470-555-7570}
email: margaret.hodge@ku.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 the number of spaces need only be consistent over the length of the nested object.

The data can be nested arbitrarily deeply.

2.3.6 NestedText files

NestedText files should be encoded with UTF-8 and should end with a newline. The top-level object must not be indented.

The name used for the file is arbitrary but it is tradition to use a .nt suffix. If you also wish to further distinguish the file type by giving the schema, it is recommended that you use two suffixes, with the suffix that specifies the schema given first and .nt given last. For example: officers.addr.nt.

2.4 Language reference

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

Encoding:

A *NestedText* document is encoded in UTF-8 and may contain any printing UTF-8 character.

Line breaks:

A *NestedText* document is partitioned into lines where the lines are split by CR LF, CR, or LF where CR and LF are the ASCII carriage return and line feed characters. A single document may employ any or all of these ways of splitting lines.

Line types:

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

Blank lines:

Blank lines are lines that are empty or consist only of ASCII space characters. Blank lines are ignored.

Line-type tags:

Most remaining lines are identified by the presence of tags, where a tag is:

1. the first dash (`-`), colon (`:`), or greater-than symbol (`>`) on a line when followed immediately by an ASCII space or line break;
2. or a hash {`#`), left bracket (`[`), or left brace (`{`) as the first non-ASCII-space character on a line.

These symbols only introduce tags when they are the first non-ASCII-space character on a line, except for the colon (`:`) which introduces a dictionary item with an inline key midway through a line.

The first (left-most) tag on a line determines the line type. Once the first tag has been found on 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.

Comments:

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

String items:

If the first non-space character on a line is a greater-than symbol followed immediately by an ASCII space (`>␣`) or a line break, the line is a *string item*. After comments and blank lines have been removed, adjacent string items with the same indentation level are combined in order into a multiline string. The string value is the multiline string with the tags removed. Any leading white space that follows the tag is retained, as is any trailing white space. The last newline is removed and all other newlines are converted to the default line terminator for the current operating system.

String values may contain any printing UTF-8 character.

List items:

If the first non-space character on a line is a dash followed immediately by an ASCII space (`-␣`) or a line break, the line is a *list item*. Adjacent list items with the same indentation level are combined in order into a list. Each list item has a tag and a value. The tag is only used to determine the type of the line and is discarded leaving the value. The value takes one of three forms.

1. If the line contains further text (characters after the dash-space), then the value is that text. The text ends at the line break and may contain any other printing UTF-8 character.
2. 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; or an inline list or dictionary.
3. Otherwise the value is empty; it is taken to be an empty string.

Key items:

If the first non-ASCII-space character on a line is a colon followed immediately by an ASCII space (`:␣`) or a line break, the line is a *key item*. After comments and blank lines have been removed, adjacent key items with the same indentation level are combined in order into a multiline key. The key itself is the multiline string with the tags removed. Any leading white space that follows the tag is retained, as is any trailing white space. The last newline is removed and all other newlines are converted to the default line terminator for the current operating system.

Key values may contain any printing UTF-8 character.

An indented value must follow a multiline key. The indented value may be either a multiline string, a list or a dictionary. The combination of the key item and its value forms a *dictionary item*; or an inline list or dictionary.

Dictionary items:

Dictionary items take two possible forms.

The first is a *dictionary item with inline key*. In this case the line starts with a key followed by a dictionary tag: a colon followed by either an ASCII space (`:␣`) or a newline. The dictionary item consists of the key, the tag, and the trailing value. Any white space between the key and the tag is ignored.

The inline key precedes the tag. It must be a non-empty string and must not:

1. contain a line break character.
2. start with a list item, string item or key item tag,

3. start with `[` or `{`,
4. contain a dictionary item tag, or
5. contain Unicode leading spaces (any Unicode spaces that follow the key are ignored).

The tag is only used to determine the type of the line and is discarded leaving the key and the value, which follows the tag. The value takes one of three forms.

1. If the line contains further text (characters after the colon-space), then the value is that text. The text ends at the line break and may contain any other printing UTF-8 character.
2. 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.
3. Otherwise the value is empty; it is taken to be an empty string.

The second form of *dictionary item* is the *dictionary item with multiline key*. It consists of a multiline key value followed by an indented value. The value may be a multiline string, list, or dictionary; or an inline list or dictionary.

Adjacent dictionary items of either form with the same indentation level are combined in order into a dictionary.

Inline Lists and Dictionaries:

If the first non-ASCII-space character on a line is either a left bracket (`[`) or a left brace (`{`) the line is an *inline structure*. A bracket introduces an inline list and a brace introduces an inline dictionary.

Inlines are confined to a single line, and so must not contain any line-break white space, such as newlines.

An *inline list* starts with an open bracket (`[`), ends with a matching closed bracket (`]`), contains inline values separated by commas (`,`), and is contained on a single line. The values may be inline strings, inline lists, and inline dictionaries.

An *inline dictionary* starts with an open brace (`{`), ends with a matching closed brace (`}`), contains inline dictionary items separated by commas (`,`), and is contained on a single line. An inline dictionary item is a key and value separated by a colon (`:`). A space need not follow the colon and any white space that does follow the colon is ignored. The keys are inline strings and the values may be inline strings, inline lists, and inline dictionaries.

Inline strings are the string values specified in inline dictionaries and lists. They are somewhat constrained in the characters that they may contain; nothing is allowed that might be confused with the syntax characters used by the inline list or dictionary that contains it. Specifically, inline strings may not include line-break white space characters such as newlines or any of the following characters: `[`, `]`, `{`, `}`, or `,`. In addition, inline strings that are contained in inline dictionaries may not contain `:`. Both leading and trailing white space is ignored with inline strings. This includes all non-line-break white space characters such as ASCII spaces and tabs, as well as the various Unicode white space characters.

Both inline lists and dictionaries may be empty, and represent the only way to represent empty lists or empty dictionaries in *NestedText*. An empty dictionary is represented with `{}` and an empty list with `[]`. In both cases there must be no space between the opening and closing delimiters. An inline list that contains only white spaces, such as `[ ]`, is treated as a list with a single empty string (the whitespace is considered a string value, and string values have leading and trailing spaces removed, resulting in an empty string value). If a list contains multiple values, no white space is required to represent an empty string value. Thus, `[]` represents an empty list, `[ ]` a list with a single empty string value, and `[,]` a list with two empty string values.

Indentation:

Leading spaces on a line represents indentation. Only ASCII spaces are allowed in the indentation. Specifically, tabs and the various Unicode spaces are not allowed.

There is no indentation on the top-level object.

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 indented value may only follow a list item or dictionary item that does not have a value on the same line. An indented value must follow a key item.

Escaping and Quoting:

There is no escaping or quoting in *NestedText*. Once the line has been identified by its tag, and the tag is removed, the remaining text is taken literally.

Empty document:

A document may be empty. A document is empty if it consists only of comments and blank lines. An empty document corresponds to an empty value of unknown type. Implementations may allow a default top-level type of dictionary, list, or string to be specified.

End of file:

The last character in a *NestedText* document file is a newline, though this is generally not enforced when reading a document.

Result:

When a document is converted from *NestedText* the result is a hierarchical collection of dictionaries, lists and strings. The leaf values are all strings, as are all dictionary keys.

2.5 Minimal NestedText

Minimal NestedText is *NestedText* without support for multiline keys and inline dictionaries and lists.

Minimal NestedText is a subset of *NestedText* that foregoes some of the complications of *NestedText*. It sacrifices the completeness of *NestedText* for an even simpler data file format that is still appropriate for a surprisingly wide variety of applications, such as most configuration files. The simplicity of *Minimal NestedText* makes it very easy to create readers and writers. Indeed, writing such functions is good programming exercise for people new to recursion.

If you choose to create a *Minimal NestedText* reader or writer it is important to code it in such a way as to discourage the creation *Minimal NestedText* documents that are invalid *NestedText*. Thus, your implementation should disallow keys that start with `:␣`, `[` or `{`. Also, please clearly indicate that your implementation only supports *Minimal NestedText* to avoid any confusion.

Many of the examples given in this document conform to the *Minimal NestedText* subset. For convenience, here is another. It is a configuration file:

```
default repository: home
report style: tree
compact format: {repo}: {size:{fmt}}.  Last back up: {last_create:ddd, MMM DD}.
normal format: {host:<8} {user:<5} {config:<9} {size:<8.2b} {last_create:ddd, MMM DD}
date format: D MMMM YYYY
size format: .2b

repositories:
    # only the composite repositories need be included
```

(continues on next page)

(continued from previous page)

```
    home:
        children: rsync borgbase
    caches:
        children: cache cache@media cache@files
    servers:
        children:
            - root@dev~root
            - root@mail~root
            - root@media~root
            - root@web~root
    all:
        children: home caches servers
```

Finally, here is a short description of *Minimal NestedText* that you can use to describe to your users if you decide to use it for your application.

***Minimal NestedText*:**

NestedText is a file format for holding structured data. It is intended to be easily entered, edited, or viewed by people. As such, the syntax is very simple and intuitive.

It organizes the data into a nested collection of lists and name-value pairs where the leaf-level values are all simple text. For example, a simple collection of name-value pairs is represented using:

```
Name 1: Value 1
Name 2: Value 2
```

The name and value are separated by a colon followed immediately by a space. The characters that follow the space are the value.

A simple list is represented with:

```
- Value 1
- Value 2
```

A list item is introduced by dash as the first non-blank character on a line followed by a space. The characters that follow the space are the value.

Indentation is used to denote nesting. In this case the colon or dash is the last character on the line and is followed by an indented value. The value may be a collection of name-value pairs, a list, or a multiline string. Every line of a multiline string is introduced by a greater-than symbol followed by a space or newline.

```
Name 1: Value 1
Name 2:
    Name 2a: Value 2a
    Name 2b: Value 2b
Name 3:
    - Value 3a
    - Value 3b
Name 4:
    > Value 4 line 1
    > Value 4 line 2
```

Any line that starts with pound sign (#) as the first non-blank character is ignored and so can be used to add comments.

```
# this line is a comment
Name: Value
```

The name in a name-value pair is referred to as a key. In *Minimal NestedText* keys cannot start with a space, an opening bracket ([) or brace ({), or a dash followed by a space. Nor can it contain a colon followed by a space. Other that that, there are no restrictions on the characters that make up a key or value, and any characters given are taken literally.

2.6 Related projects

2.6.1 Reference Material

NestedText Documentation

NestedText documentation and language specification.

NestedText Source

Source code repository for language documentation and Python implementation. Report any issues here.

NestedText Tests

Official *NestedText* test suite. Also included as submodule in nestedtext.

2.6.2 Implementations

C

nt4c

C implementation of *Minimal NestedText*. Supports *Minimal NestedText v3.7*.

Go

nestedtext

Go implementation of *NestedText* with idiomatic API. Supports *NestedText v3.8*.

nestext

Go implementation of *NestedText*. Supports *NestedText v3.1*.

Janet

janet-nested-text

Janet implementation of *NestedText*. Supports *NestedText v3.0*.

Java

nestedtext-min-java

Java implementation of *Minimal NestedText*. Supports *Minimal NestedText v3.7*.

JavaScript

NestedText

Native JavaScript implementation of *NestedText*. It's not a WASM build of a compiled project, so it is significantly lighter in weight than *@rmw/nestedtext*. Also available from GitHub. Supports *NestedText v3.0*.

@rmw/nestedtext

NodeJS (es module) implementation of *NestedText*. Uses WASM for *NestedText* decode. Supports *NestedText v3.0*.

.NET

NestedText

.NET implementation of *NestedText*. Supports *NestedText v3.7*.

Ruby

nestedtext-ruby

Ruby implementation of *NestedText*. Supports *NestedText v3.0*.

Rust

nested-text

Rust implementation of *NestedText* with serde integration. Supports *NestedText v3.8*.

Zig

zig-nestedtext

Zig implementation of *NestedText* (slight subset of *NestedText v2.0*). Also contains *nt-cli*, an efficient command line utility for converting between *NestedText* and *JSON*.

2.6.3 Utilities

NestedTextTo

Command line utilities for converting between *NestedText*, *JSON*, *YAML*, and *TOML*.

ntLog

ntlog is a *NestedText* logfile aggregation utility.

parametrize from file

Separate your test cases, held in *NestedText*, from your PyTest test code.

pygments

Version of the popular *pygments* Python library that supports *NestedText v3.0*.

sublimetext-nestedtext

Sublime syntax files for *NestedText* (supports *NestedText v3.0*).

vim-nestedtext

Vim syntax files for *NestedText* (supports *NestedText v3.0*).

visual studio code

Syntax files for *Visual Studio Code* (supports *NestedText v3.0*).

2.7 Language changes

Currently the language and the *Python implementation* share version numbers. Since the language is more stable than the implementation, you will see versions that include no changes to the language.

2.7.1 Latest development version

Version: 3.8
Released: 2025-12-26

2.7.2 v3.8 (2025-12-26)

- No changes to language.
- Replaced official test suite.

2.7.3 v3.7 (2024-04-27)

- Clarified policy on white space in inline strings.

2.7.4 v3.6 (2023-05-30)

- No changes.

2.7.5 v3.5 (2022-11-04)

- No changes.

2.7.6 v3.4 (2022-06-15)

- No changes.

2.7.7 v3.3 (2022-06-07)

- Defined *Minimal NestedText*, a subset of *NestedText*.
- *NestedText* document files should end with a newline.

2.7.8 v3.2 (2022-01-17)

- No changes.

2.7.9 v3.1 (2021-07-23)

- No changes.

2.7.10 v3.0 (2021-07-17)

- Deprecate trailing commas in inline lists and dictionaries.

> **Warning**
>
> Be aware that aspects of this version are not backward compatible. Specifically, trailing commas are no longer supported in inline dictionaries and lists. In addition, `[ ]` now represents a list that contains an empty string, whereas previously it represented an empty list.

2.7.11 v2.0 (2021-05-28)

- Deprecate quoted dictionary keys.
- Add multiline dictionary keys to replace quoted keys.
- Add single-line lists and dictionaries.

> **Warning**
>
> Be aware that this version is not backward compatible because it no longer supports quoted dictionary keys.

2.7.12 v1.3 (2021-01-02)

- No changes.

2.7.13 v1.2 (2020-10-31)

- Treat CR LF, CR, or LF as a line break.

2.7.14 v1.1 (2020-10-13)

- No changes.

2.7.15 v1.0 (2020-10-03)

- Initial release.

2.8 Basic use

The *NestedText* Python API is similar to that of *JSON*, *YAML*, *TOML*, etc.

2.8.1 Installation

NestedText is also available from *pip*. Install it with:

```
pip3 install nestedtext
```

Alternately, *NestedText* is also available in *Conda*. Install it with:

```
conda install nestedtext --channel conda-forge
```

2.8.2 NestedText Reader

The *loads()* function is used to convert *NestedText* held in a string 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, top='dict')
... 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', top='dict')
... except nt.NestedTextError as e:
...     e.terminate()
... except OSError as e:
...     fatal(os_error(e))

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

Notice that the type of the return value is specified to be 'dict'. This is the default. You can also specify 'list', 'str', or 'any' (or *dict*, *list*, *str*, or *any*). All but 'any' constrain the data type of the top-level of the *NestedText* content.

The *load* functions provide a *keymap* argument that is useful for adding line numbers to error message. This feature is demonstrated in *Validate with Voluptuous*. They also provide a *normalize_key* argument that can be used to ignore insignificant variation in keys, such as character case, or to convert keys to a desired form, such as to identifiers. These features are described in *loads()*.

2.8.3 NestedText Writer

The *dumps()* function is used to convert a Python data structure into a *NestedText* string. 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()
```

(continues on next page)

(continued from previous page)

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

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

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

The *dump* functions provide arguments that can control the output format and can control the conversion of data types into forms that can be dumped. These features are described in *dumps()*.

2.9 Schemas

Because *NestedText* explicitly does not attempt to interpret the data it parses, it is meant to be paired with a tool that can both validate the data and convert them to the expected types. For example, if you are expecting a date for a particular field, you would want to validate that the input looks like a date (e.g. `YYYY/MM/DD`) and then convert it to a useful type (e.g. `arrow.Arrow`). You can do this on an ad hoc basis, or you can apply a schema.

A schema is the specification of what fields are expected (e.g. "birthday"), what types they should be (e.g. a date), and what values are legal (e.g. must be in the past). There are many libraries available for applying a schema to data such as those parsed by *NestedText*. Because different libraries may be more or less appropriate in different scenarios, *NestedText* avoids favoring any one library specifically:

- voluptuous: Define schema using objects
- pydantic: Define schema using type annotations
- schema: Define schema using objects
- colander: Define schema using classes
- schematics: Define schema using classes
- cerebus : Define schema using strings
- valideer: Define schema using strings
- jsonschema: Define schema using JSON

See the *Techniques* page for examples of how to use some of these libraries with *NestedText*.

The approach of using separate tools for parsing and interpreting the data has two significant advantages that are worth briefly highlighting. First is that the validation tool understands the context and meaning of the data in a way that the parsing tool cannot. For example, "12" can be an integer if it represents a day of a month, a float if it represents the output voltage of a power brick, or a string if represents the version of a software package. Attempting to interpret "12" without this context is inherently unreliable. Second is that when data is interpreted by the parser, it puts the onus on the user to specify the correct types. Going back to the previous example, the user would be required to know whether `12`, `12.0`, or `"12"` should be entered. It does not make sense for this decision to be made by the user instead of the application.

2.10 Techniques

This section documents common patterns of use with examples and suggestions.

2.10.1 Validate with *Voluptuous*

This example shows how to use voluptuous to validate and parse a *NestedText* file and it demonstrates how to use the *keymap* argument from `loads()` or `load()` to add location information to *Voluptuous* error messages.

The input file in this case specifies deployment settings for a web server:

```
debug: false
secret key: t=)40**y&883y9gdpuw%aiig+wtc033(ui@^1ur72w#zhw3_ch

allowed hosts:
  - www.example.com

database:
  engine: django.db.backends.mysql
  host: db.example.com
  port: 3306
  user: www

webmaster email: admin@example.com
```

Below is the code to parse this file. Note how the structure of the data is specified using basic Python objects. The `Coerce()` function is necessary to have Voluptuous convert string input to the given type; otherwise it would simply check that the input matches the given type:

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

import nestedtext as nt
from voluptuous import Schema, Coerce, MultipleInvalid
from voluptuous_errors import report_voluptuous_errors
from inform import terminate
from pprint import pprint

schema = Schema({
    'debug': Coerce(bool),
    'secret_key': str,
    'allowed_hosts': [str],
    'database': {
        'engine': str,
        'host': str,
        'port': Coerce(int),
        'user': str,
    },
    'webmaster_email': str,
})

def normalize_key(key, parent_keys):
    return '_'.join(key.lower().split())

filename = "deploy.nt"
```

(continues on next page)

(continued from previous page)

```
try:
    keymap = {}
    raw = nt.load(filename, keymap=keymap, normalize_key=normalize_key)
    config = schema(raw)
except nt.NestedTextError as e:
    e.terminate()
except MultipleInvalid as e:
    report_voluptuous_errors(e, keymap, filename)
    terminate()

pprint(config)
```

This example uses the following code to adapt error reporting in *Voluptuous* to *NestedText*.

```
from inform import cull, error, full_stop
import nestedtext as nt

voluptuous_error_msg_mappings = {
    "extra keys not allowed": ("unknown key", "key"),
    "expected a dict": ("expected a key-value pair", "value"),
    "required key not provided": ("required key is missing", "value"),
}

def report_voluptuous_errors(multiple_invalid, keymap, source=None, sep="›"):
    source = str(source) if source else ""

    for err in multiple_invalid.errors:

        # convert message to something easier for non-savvy user to understand
        msg, kind = voluptuous_error_msg_mappings.get(
            err.msg, (err.msg, 'value')
        )

        # get metadata about error
        if keymap:
            culprit = nt.get_keys(err.path, keymap=keymap, strict="found", sep=sep)
            line_nums = nt.get_line_numbers(err.path, keymap, kind=kind, sep="-",␣
↪strict=False)
            loc = nt.get_location(err.path, keymap)
            if loc:
                codicil = loc.as_line(kind)
            else:  # required key is missing
                missing = nt.get_keys(err.path, keymap, strict="missing", sep=sep)
                codicil = f"‘{missing}’ was not found."

            file_and_lineno = f"{source!s}@{line_nums}"
            culprit = cull((file_and_lineno, culprit))
        else:
            keys = sep.join(str(c) for c in err.path)
            culprit = cull([source, keys])
            codicil = None
```

(continues on next page)

(continued from previous page)

```
        # report error
        error(full_stop(msg), culprit=culprit, codicil=codicil)
```

This produces the following data structure:

```
{'allowed_hosts': ['www.example.com'],
 'database': {'engine': 'django.db.backends.mysql',
              'host': 'db.example.com',
              'port': 3306,
              'user': 'www'},
 'debug': False,
 'secret_key': 't=)40**y&883y9gdpuw%aiig+wtc033(ui@^1ur72w#zhw3_ch',
 'webmaster_email': 'admin@example.com'}
```

See the *PostMortem* example for a more flexible approach to validating with *Voluptuous*.

2.10.2 Validate with *Pydantic*

This example shows how to use pydantic to validate and parse a *NestedText* file. The input file is the same as in the previous example, i.e. deployment settings for a web server:

```
debug: false
secret key: t=)40**y&883y9gdpuw%aiig+wtc033(ui@^1ur72w#zhw3_ch

allowed hosts:
  - www.example.com

database:
  engine: django.db.backends.mysql
  host: db.example.com
  port: 3306
  user: www

webmaster email: admin@example.com
```

Below is the code to parse this file. Note that basic types like integers, strings, Booleans, and lists are specified using standard type annotations. Dictionaries with specific keys are represented by model classes, and it is possible to reference one model from within another. Pydantic also has built-in support for validating email addresses, which we can take advantage of here:

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

import nestedtext as nt
from pydantic import BaseModel, EmailStr
from typing import List
from pprint import pprint

class Database(BaseModel):
    engine: str
    host: str
    port: int
    user: str
```

(continues on next page)

(continued from previous page)

```
class Config(BaseModel):
    debug: bool
    secret_key: str
    allowed_hosts: List[str]
    database: Database
    webmaster_email: EmailStr

def normalize_key(key, parent_keys):
    return '_'.join(key.lower().split())

obj = nt.load('deploy.nt', normalize_key=normalize_key)
config = Config.parse_obj(obj)

pprint(config.dict())
```

This produces the same result as in the previous example.

2.10.3 Normalizing Keys

With data files created by non-programmers it is often desirable to allow a certain amount of flexibility in the keys. For example, you may wish to ignore case and if you allow multi-word keys you may want to be tolerant of extra spaces between the words. However, the end applications often needs the keys to be specific values. It is possible to normalize the keys using a schema, but this can interfere with error reporting. Imagine there is an error in the value associated with a set of keys, if the keys have been changed by the schema the *keymap* can no longer be used to convert the keys into a line number for an error message. *NestedText* provides the *normalize_key* argument to `load()` and `loads()` to address this issue. It allows you to pass in a function that normalizes the keys before the *keymap* is created, releasing the schema from that task.

The following contact look-up program demonstrates both the normalization of keys and the associated error reporting. In this case, the first level of keys contains the names of the contacts and should not be normalized. Keys at all other levels are considered keywords and so should be normalized.

```
#!/usr/bin/env python3
"""
Display Contact Information

Usage:
    contact <name>...
"""

from docopt import docopt
from inform import codicil, display, error, full_stop, os_error, terminate
import nestedtext as nt
from voluptuous import Schema, Any, MultipleInvalid
import re

contacts_file = "address.nt"

def normalize_key(key, parent_keys):
    if len(parent_keys) == 0:
        return key
    return '_'.join(key.lower().split())
```

(continues on next page)

(continued from previous page)

```
def render_contact(data, keymap=None):
    text = nt.dumps(data, map_keys=keymap)
    return re.sub(r'^(\s*)[>:][ ]?(.*)$', r'\1\2', text, flags=re.M)

cmdline = docopt(__doc__)
names = cmdline['<name>']

try:
    # define structure of contacts database
    contacts_schema = Schema({
        str: {
            'position': str,
            'address': str,
            'phone': Any({str:str},str),
            'email': Any({str:str},str),
            'additional_roles': Any(list,str),
        }
    })

    # read contacts database
    contacts = contacts_schema(
        nt.load(
            contacts_file,
            top = 'dict',
            normalize_key = normalize_key,
            keymap = (keymap:={})
        )
    )

    # display requested contact information, excluding additional_roles
    filtered = {}
    for fullname, contact_info in contacts.items():
        for name in names:
            if name in fullname.lower():
                filtered[fullname] = contact_info
                if 'additional_roles' in contact_info:
                    del contact_info['additional_roles']

    # display contact using normalized keys
    # display(render_contact(filtered))

    # display contact using original keys
    display(render_contact(filtered, keymap))

except nt.NestedTextError as e:
    e.report()
except MultipleInvalid as exception:
    for e in exception.errors:
        kind = 'key' if 'key' in e.msg else 'value'
        keys = tuple(e.path)
        codicil = keymap[keys].as_line(kind) if keys in keymap else None
        line_num, col_num = keymap[keys].as_tuple()
```

(continues on next page)

(continued from previous page)

```
        file_and_lineno = f"{contacts_file!s}@{line_num}"
        key_path = nt.join_keys(keys, keymap=keymap, sep="›")
        error(
            full_stop(e.msg),
            culprit = (file_and_lineno, key_path),
            codicil = codicil
        )
except OSError as e:
    error(os_error(e))
terminate()
```

This program takes a name as a command line argument and prints out the corresponding address. It uses the pretty print idea described below to render the contact information. *Voluptuous* checks the validity of the contacts database, which is shown next. Notice the variability in the keys given in Fumiko's entry:

```
# Contact information for our officers

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

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

Fumiko Purvis:
    Position: Treasurer
        # Fumiko's term is ending at the end of the year.
    Address:
        > 3636 Buffalo Ave
        > Topeka, Kansas 20692
    Phone: 1-268-555-0280
    EMail: fumiko.purvis@hotmail.com
    Additional  Roles:
        - accounting task force
```

There are two display statements near the end of the program, the first of which is commented out. The first outputs

the contact information using normalized keys, and the second outputs the information using the original keys.

Now, requesting Fumiko's contact information gives:

```
Fumiko Purvis:
    Position: treasurer
    Address:
        3636 Buffalo Ave
        Topeka, Kansas 20692
    Phone: 1-268-555-0280
    EMail: fumiko.purvis@hotmail.com
```

Notice that any processing of the information (error checking, deleting *additional_roles*) is performed using the normalized keys, but by choice, the information is output using the original keys.

2.10.4 Duplicate Keys

There are occasions where it is useful to be able to read dictionaries from NestedText that contain duplicate keys. For example, imagine that you have two contacts with the same name, and the name is used as a key. Normally `load()` and `loads()` throw an exception if duplicate keys are detected because the underlying Python dictionaries cannot hold items with duplicate keys. However, you can pass a function to the *on_dup* argument that de-duplicates the keys, making them safe for Python dictionaries. For example the following *NestedText* document that contains duplicate keys:

```
Michael Jordan:
    occupation: basketball player

Michael Jordan:
    occupation: actor

Michael Jordan:
    occupation: football player
```

In the following, the *de_dup* function adds "#*N*" to the end of the key where *N* starts at 2 and increases as more duplicates are found.

```
#!/usr/bin/env python3
from inform import codicil, display, fatal, full_stop, os_error
import nestedtext as nt

filename = "michael_jordan.nt"

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

try:
    # read contacts database
    data = nt.load(filename, 'dict', on_dup=de_dup, keymap=(keymap:={}))

    # display contact using deduplicated keys
    display("DE-DUPLICATED KEYS:")
    display(nt.dumps(data))
```

(continues on next page)

(continued from previous page)

```
    # display contact using original keys
    display()
    display("ORIGINAL KEYS:")
    display(nt.dumps(data, map_keys=keymap))

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

As shown below, this code outputs the data twice, the first time with the de-duplicated keys and the second time using the original keys. Notice that the first contains the duplication markers whereas the second does not.

```
DE-DUPLICATED KEYS:
Michael Jordan:
    occupation: basketball player
Michael Jordan #2:
    occupation: actor
Michael Jordan #3:
    occupation: football player

ORIGINAL KEYS:
Michael Jordan:
    occupation: basketball player
Michael Jordan:
    occupation: actor
Michael Jordan:
    occupation: football player
```

2.10.5 Sorting Keys

The default order of dictionary items in the *NestedText* output of *dump()* and *dumps()* is the natural order of the underlying dictionary, but you can use *sort_keys* argument to change the order. For example, here are two different ways of sorting the address list. The first is a simple alphabetic sort of the keys at each level, which you get by simply specifying *sort_keys=True*.

```
>>> addresses = nt.load( 'examples/addresses/address.nt')
>>> print(nt.dumps(addresses, sort_keys=True))
Fumiko Purvis:
    Additional  Roles:
        - accounting task force
    Address:
        > 3636 Buffalo Ave
        > Topeka, Kansas 20692
    EMail: fumiko.purvis@hotmail.com
    Phone: 1-268-555-0280
    Position: Treasurer
Katheryn McDaniel:
    additional roles:
        - board member
    address:
```

(continues on next page)

(continued from previous page)

```
        > 138 Almond Street
        > Topeka, Kansas 20697
    email: KateMcD@aol.com
    phone:
        cell: 1-210-555-5297
        work: 1-210-555-8470
    position: president
Margaret Hodge:
    additional roles:
        - new membership task force
        - accounting task force
    address:
        > 2586 Marigold Lane
        > Topeka, Kansas 20682
    email: margaret.hodge@ku.edu
    phone: 1-470-555-0398
    position: vice president
```

The second sorts only the first level, by last name then remaining names. It passes a function to *sort_keys*. That function takes two arguments, the key to be sorted and the tuple of parent keys. The key to be sorted is also a tuple that contains the key and the rendered item. The key is the key as specified in the object being dumped, and rendered item is a string that takes the form "mapped_key: value".

The *sort_keys* function is expected to return a string that contains the sort key, the key used by the sort. For example, in this case a first level key "Fumiko Purvis" is mapped to "Purvis Fumiko" for the purposes of determining the sort order. At all other levels any key is mapped to "". In this way the sort keys are all identical, and so the original order is retained.

```
>>> def sort_key(key, parent_keys):
...     if len(parent_keys) == 0:
...         # rearrange names so that last name is given first
...         names = key[0].split()
...         return ' '.join([names[-1]] + names[:-1])
...     return ''  # do not reorder lower levels

>>> print(nt.dumps(addresses, sort_keys=sort_key))
Margaret Hodge:
    position: vice president
    address:
        > 2586 Marigold Lane
        > Topeka, Kansas 20682
    phone: 1-470-555-0398
    email: margaret.hodge@ku.edu
    additional roles:
        - new membership task force
        - accounting task force
Katheryn McDaniel:
    position: president
    address:
        > 138 Almond Street
        > Topeka, Kansas 20697
    phone:
        cell: 1-210-555-5297
```

(continues on next page)

(continued from previous page)

```
        work: 1-210-555-8470
    email: KateMcD@aol.com
    additional roles:
        - board member
Fumiko Purvis:
    Position: Treasurer
    Address:
        > 3636 Buffalo Ave
        > Topeka, Kansas 20692
    Phone: 1-268-555-0280
    EMail: fumiko.purvis@hotmail.com
    Additional  Roles:
        - accounting task force
```

2.10.6 Key Presentation

When generating a *NestedText* document, it is sometimes desirable to transform the keys upon output. Generally one transforms the keys in order to change the presentation of the key, not the meaning. For example, you may want change its case, rearrange it (ex: swap first and last names), translate it, etc. These are done by passing a function to the *map_keys* argument. This function takes two arguments: the key after it has been rendered to a string and the tuple of parent keys. It is expected to return the transformed string. For example, lets print the address book again, this time with names printed with the last name first.

```
>>> def last_name_first(key, parent_keys):
...     if len(parent_keys) == 0:
...         # rearrange names so that last name is given first
...         names = key.split()
...         return f"{names[-1]}, {' '.join(names[:-1])}"

>>> def sort_key(key, parent_keys):
...     return key if len(parent_keys) == 0 else ''  # only sort first level keys

>>> print(nt.dumps(addresses, map_keys=last_name_first, sort_keys=sort_key))
Hodge, Margaret:
    position: vice president
    address:
        > 2586 Marigold Lane
        > Topeka, Kansas 20682
    phone: 1-470-555-0398
    email: margaret.hodge@ku.edu
    additional roles:
        - new membership task force
        - accounting task force
McDaniel, Katheryn:
    position: president
    address:
        > 138 Almond Street
        > Topeka, Kansas 20697
    phone:
        cell: 1-210-555-5297
        work: 1-210-555-8470
    email: KateMcD@aol.com
```

(continues on next page)

(continued from previous page)

```
    additional roles:
        - board member
Purvis, Fumiko:
    Position: Treasurer
    Address:
        > 3636 Buffalo Ave
        > Topeka, Kansas 20692
    Phone: 1-268-555-0280
    EMail: fumiko.purvis@hotmail.com
    Additional  Roles:
        - accounting task force
```

When round-tripping a *NestedText* document (reading the document and then later writing it back out), one often wants to undo any changes that were made to the keys when reading the documents. These modifications would be due to key normalization or key de-duplication. This is easily accomplished by simply retaining the keymap from the original load and passing it to the dumper by way of the *map_keys* argument.

```
>>> def normalize_key(key, parent_keys):
...     if len(parent_keys) == 0:
...         return key
...     return '_'.join(key.lower().split())

>>> keymap = {}
>>> addresses = nt.load(
...     'examples/addresses/address.nt',
...     normalize_key=normalize_key,
...     keymap=keymap
... )
>>> filtered = {k:v for k,v in addresses.items() if 'fumiko' in k.lower()}

>>> print(nt.dumps(filtered))
Fumiko Purvis:
    position: Treasurer
    address:
        > 3636 Buffalo Ave
        > Topeka, Kansas 20692
    phone: 1-268-555-0280
    email: fumiko.purvis@hotmail.com
    additional_roles:
        - accounting task force

>>> print(nt.dumps(filtered, map_keys=keymap))
Fumiko Purvis:
    Position: Treasurer
    Address:
        > 3636 Buffalo Ave
        > Topeka, Kansas 20692
    Phone: 1-268-555-0280
    EMail: fumiko.purvis@hotmail.com
    Additional  Roles:
        - accounting task force
```

Notice that the keys differ between the two. The normalized key are output in the former and original keys in the latter.

Finally consider the case where you want to do both things; you want to return to the original keys but you also want to change the presentation. For example, imagine wanting to display the original keys in blue. That can be done as follows:

```
>>> from inform import Color
>>> blue = Color('blue', enable=Color.isTTY())

>>> def format_key(key, parent_keys):
...     orig_keys = nt.get_original_keys(parent_keys + (key,), keymap)
...     return blue(orig_keys[-1])

>>> print(nt.dumps(filtered, map_keys=format_key))
Fumiko Purvis:
    Position: Treasurer
    Address:
        > 3636 Buffalo Ave
        > Topeka, Kansas 20692
    Phone: 1-268-555-0280
    EMail: fumiko.purvis@hotmail.com
    Additional  Roles:
        - accounting task force
```

The result looks identical in the documentation, but if you ran this program in a terminal you would see the keys in blue.

2.10.7 References

A reference allows you to define some content once and insert that content multiple places in the document. A reference is also referred to as a macro. Both simple and parametrized references can be easily implemented. For parametrized references, the arguments list is treated as an embedded *NestedText* document.

The technique is demonstrated with an example. This example is a fragment of a diet program. It reads two *NestedText* documents, one containing the known foods, and the other that documents the actual meals as consumed. The foods may be single ingredient, like *steel cut oats*, or it may contain multiple ingredients, like *oatmeal*. The use of parametrized references allows one to override individual ingredients in a composite ingredient. In this example, the user simply specifies the composite ingredient *oatmeal* on 21 March. On 22 March, they specify it as a simple reference, meaning that they end up with the same ingredients, but this time they are listed separately in the final summary. Finally, on 23 March they specify oatmeal using a parametrized reference so as to override the number of tangerines consumed and add some almonds.

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

from inform import Error, display, dedent
import nestedtext as nt
import re

foods = nt.loads(dedent("""
    oatmeal:
        steel cut oats: 1/4 cup
        tangerines: 1 each
        whole milk: 1/4 cup
    steel cut oats:
        calories by weight: 150/40 cals/gram
    tangerines:
```

(continues on next page)

(continued from previous page)

```
        calories each: 40 cals
        calories by weight: 53/100 cals/gram
    whole milk:
        calories by weight: 149/255 cals/gram
        calories by volume: 149 cals/cup
    almonds:
        calories each: 40 cals
        calories by weight: 822/143 cals/gram
        calories by volume: 822 cals/cup
"""), dict)

meals = nt.loads(dedent("""
    21 March 2023:
        breakfast: oatmeal
    22 March 2023:
        breakfast: @oatmeal
    23 March 2023:
        breakfast: @oatmeal(tangerines: 0 each, almonds: 10 each)
"""), dict)

def expand_foods(value):
    # allows macro values to be defined as a top-level food.
    # allows macro reference to be found anywhere.
    if isinstance(value, str):
        value = value.strip()
        if value[:1] == '@':
            value =  parse_macro(value[1:].strip())
        return value
    if isinstance(value, dict):
        return {k:expand_foods(v) for k, v in value.items()}
    if isinstance(value, list):
        return [expand_foods(v) for v in value]
    raise NotImplementedError(value)

def parse_macro(macro):
    match = re.match(r'(\w+)(?:\((.*)\))?', macro)
    if match:
        name, args = match.groups()
        try:
            food = foods[name].copy()
        except KeyError:
            raise Error("unknown food.", culprit=name)
        if args:
            args = nt.loads('{' + args + '}', dict)
            food.update(args)
        return food
    raise Error("unknown macro.", culprit=macro)


try:
    meals = expand_foods(meals)
    display(nt.dumps(meals))
```

(continues on next page)

(continued from previous page)

```
except Error as e:
    e.terminate()
```

It produces the following output:

```
21 March 2023:
    breakfast: oatmeal
22 March 2023:
    breakfast:
        steel cut oats: 1/4 cup
        tangerines: 1 each
        whole milk: 1/4 cup
23 March 2023:
    breakfast:
        steel cut oats: 1/4 cup
        tangerines: 0 each
        whole milk: 1/4 cup
        almonds: 10 each
```

In this example the content for the references was pulled from a different *NestedText* document. See the *PostMortem* as an example that pulls the referenced content from the same document.

2.10.8 Accumulation

This example demonstrates how to used *NestedText* so that it supports some common common aspects of settings files; specifically you can override or accumulate to previously specified settings by repeating their names.

It implements an example settings file reader that supports a small variety of settings. *NestedText* is configured to normalize and de-duplicate the keys (the names of the settings) with the result being processed to identify and report errors and to implement overrides, accumulations, and simple conversions. Accumulation is indicated adding a plus sign to the beginning of the key. The keys are normalized by converting them to snake case (all lower case, contiguous spaces replaced by a single underscore).

```
from inform import Error, full_stop, os_error
import nestedtext as nt

schema = dict(
    name = str,
    limit = float,
    actions = dict,
    patterns = list,
)
list_settings = set(k for k, v in schema.items() if v == list)
dict_settings = set(k for k, v in schema.items() if v == dict)

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

def normalize_key(key, parent_keys):
    return '_'.join(key.lower().split())  # convert key to snake case
```

(continues on next page)

(continued from previous page)

```
def read_settings(path, processed=None):
    if processed is None:
        processed = {}

    try:
        keymap = {}
        settings = nt.load(
            path,
            top = dict,
            normalize_key = normalize_key,
            on_dup = de_dup,
            keymap = keymap
        )
    except OSError as e:
        raise Error(os_error(e))

    def report_error(msg):
        keys = key_as_given,
        offset = key_as_given.index(key)
        raise Error(
            full_stop(msg),
            culprit = path,
            codicil = nt.get_location(keys, keymap=keymap).as_line('key', offset=offset)
        )

    # process settings
    for key_as_given, value in settings.items():

        # remove any decorations on the key
        key = key_as_given

        accumulate = '+' in key_as_given
        if accumulate:
            cruft, _, key = key_as_given.partition('+')
            if cruft:
                report_error("'+' must precede setting name")

        if '#' in key:  # get original name for duplicate key
            key, _, _ = key.partition('#')

        key = key.strip('_')
        if not key.isidentifier():
            report_error("expected identifier")

        # check the type of the value
        if key in list_settings:
            if isinstance(value, str):
                value = value.split()
            if not isinstance(value, list):
                report_error(f"expected list, found {value.__class__.__name__}")
            if accumulate:
```

(continues on next page)

(continued from previous page)

```
                base = processed.get(key, [])
                value = base + value
        elif key in dict_settings:
            if value == "":
                value = {}
            if not isinstance(value, dict):
                report_error(f"expected dict, found {value.__class__.__name__}")
            if accumulate:
                base = processed.get(key, {})
                base.update(value)
                value = base
        elif key in schema:
            if accumulate:
                report_error("setting is unsuitable for accumulation")
            value = schema[key](value)  # cast to desired type
        else:
            report_error("unknown setting")
        processed[key] = value

    return processed
```

It would interpret this settings file:

```
name: trantor
actions:
    default: clean
patterns: ..
limit: 60

name: terminus
+patterns: ../**/{name}.nt
+patterns: ../**/*.{name}:*.nt

+ actions:
    final: archive
```

as equivalent to this settings file:

```
name: terminus
actions:
    default: clean
    final: archive
patterns:
    - ..
    - ../**/{name}.nt
    - ../**/*.{name}:*.nt
limit: 60.0
```

2.10.9 Pretty Printing

Besides being a readable file format, *NestedText* makes a reasonable display format for structured data. This example further simplifies the output by stripping leading multiline string tags.

```
>>> import nestedtext as nt
>>> import re
>>>
>>> def pp(data):
...     try:
...         text = nt.dumps(data, default=repr)
...         print(re.sub(r'^(\s*)[>:][ ]?(.*)$', r'\1\2', text, flags=re.M))
...     except nt.NestedTextError as e:
...         e.report()

>>> addresses = nt.load('examples/addresses/address.nt')

>>> pp(addresses['Katheryn McDaniel'])
position: president
address:
    138 Almond Street
    Topeka, Kansas 20697
phone:
    cell: 1-210-555-5297
    work: 1-210-555-8470
email: KateMcD@aol.com
additional roles:
    - board member
```

Stripping leading multiline string tags results in the output no longer being valid *NestedText* and so should not be done if the output needs to be readable later as *NestedText*..

2.10.10 Long Lines

One of the benefits of *NestedText* is that no escaping of special characters is ever needed. However, you might find it helpful to add your own support for removing escaped newlines in multiline strings. Doing so allows you to keep your lines short in the source document so as to make them easier to interpret in windows of limited width.

This example uses the pretty-print function from the previous example.

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

>>> document = dedent(r"""
...     lorum ipsum:
...         > Lorem ipsum dolor sit amet, \
...         > consectetur adipiscing elit.
...         > Sed do eiusmod tempor incididunt \
...         > ut labore et dolore magna aliqua.
... """)

>>> def reverse_escaping(text):
...     return text.replace("\\\n", "")
```

(continues on next page)

(continued from previous page)

```
>>> schema = Schema({str: reverse_escaping})
>>> data = schema(nt.loads(document))
>>> pp(data)
lorum ipsum:
    Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
```

An alternative to using a backslash to escape the newline is to simply join lines that end with a space. This might be more natural for non-programmers and can work well for prose.

```
>>> document = dedent(r"""
...     lorum ipsum:
...         > Lorem ipsum dolor sit amet,␣
...         > consectetur adipiscing elit.
...         > Sed do eiusmod tempor incididunt␣
...         > ut labore et dolore magna aliqua.
... """).replace('␣', ' ')

>>> def reverse_escaping(text):
...     return text.replace(" \n", " ")

>>> schema = Schema({str: reverse_escaping})
>>> data = schema(nt.loads(document))
>>> pp(data)
lorum ipsum:
    Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
```

2.11 Examples

2.11.1 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]
    -s, --sort             sort the keys
    -w <n>, --width <n>    desired maximum line width; specifying enables
                           use of single-line lists and dictionaries as long
                           as the fit in given width [default: 0]

If <filename> is not given, JSON input is taken from stdin and NestedText output
```

(continues on next page)

(continued from previous page)

```
is written to stdout.
"""

from docopt import docopt
from inform import done, fatal, full_stop, os_error, warn
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('expected positive integer for indent.', culprit=cmdline['--indent'])
    indent = 4
try:
    width = int(cmdline['--width'])
except Exception:
    warn('expected non-negative integer for width.', culprit=cmdline['--width'])
    width = 0

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,
        width = width,
        sort_keys = cmdline['--sort']
    )

    # 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 + "\n")
```

(continues on next page)

(continued from previous page)

```
except OSError as e:
    fatal(os_error(e))
except nt.NestedTextError as e:
    e.terminate()
except UnicodeError as e:
    fatal(full_stop(e))
except KeyboardInterrupt:
    done()
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')[max(lineno-2, 0):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)
```

Be aware that not all *JSON* data can be converted to *NestedText*, and in the conversion much of the type information is lost.

json-to-nestedtext can be used as a JSON pretty printer:

```
> json-to-nestedtext < fumiko.json
treasurer:
    name: Fumiko Purvis
    address:
        > 3636 Buffalo Ave
        > Topeka, Kansas 20692
    phone: 1-268-555-0280
    email: fumiko.purvis@hotmail.com
    additional roles:
        - accounting task force
```

2.11.2 NestedText to JSON

This example implements a command-line utility that converts a *NestedText* file to *JSON*. It demonstrates the use of *load()* 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 done, fatal, os_error, full_stop
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, 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, top='any', on_dup=on_dup)
        json_content = json.dumps(data, indent=4, ensure_ascii=False)
        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, top='any', on_dup=on_dup)
        json_content = json.dumps(data, indent=4, ensure_ascii=False)
```

(continues on next page)

(continued from previous page)

```
        sys.stdout.write(json_content + '\n')
except OSError as e:
    fatal(os_error(e))
except nt.NestedTextError as e:
    e.terminate()
except UnicodeError as e:
    fatal(full_stop(e))
except KeyboardInterrupt:
    done()
```

2.11.3 PyTest

This example highlights a PyTest package parametrize_from_file that allows you to neatly separate your test code from your test cases; the test cases being held in a *NestedText* file. Since test cases often contain code snippets, the ability of *NestedText* to hold arbitrary strings without the need for quoting or escaping results in very clean and simple test case specifications. Also, use of the *eval* function in the test code allows the fields in the test cases to be literal Python code.

The test cases:

```
# test_expr.nt
test_substitution:
  -
    given:   first  second
    search: ^\s*(\w+)\s*(\w+)\s*$
    replace: \2 \1
    expected: second first
  -
    given: 4 * 7
    search: ^\s*(\d+)\s*([-+*/])\s*(\d+)\s*$
    replace: \1 \3 \2
    expected: 4 7 *

test_expression:
  -
    given: 1 + 2
    expected: 3
  -
    given: "1" + "2"
    expected: "12"
  -
    given: pathlib.Path("/") / "tmp"
    expected: pathlib.Path("/tmp")
```

And the corresponding test code:

```
# test_misc.py
import parametrize_from_file
import re
import pathlib

@parametrize_from_file
def test_substitution(given, search, replace, expected):
    assert re.sub(search, replace, given) == expected
```

(continues on next page)

(continued from previous page)

```
@parametrize_from_file
def test_expression(given, expected):
    assert eval(given) == eval(expected)
```

2.11.4 PostMortem

PostMortem is a program that generates a packet of information that is securely shared with your dependents in case of your death. Only the settings processing part of the package is shown here.

This example includes *references*, *key normalization*, and different way to implement validation and conversion on a per field basis with voluptuous. References allow you to define some content once and insert that content multiple places in the document. Key normalization allows the keys to be case insensitive and contain white space even though the program that uses the data prefers the keys to be lower case identifiers.

Here is a configuration file that Odin might use to generate packets for his wife and kids:

```
my GPG ids: odin@norse-gods.com
sign with: @ my gpg ids
name template: {name}-{now:YYMMDD}
estate docs:
    - ~/home/estate/trust.pdf
    - ~/home/estate/will.pdf
    - ~/home/estate/deed-valhalla.pdf

recipients:
    Frigg:
        email: frigg@norse-gods.com
        category: wife
        attach: @ estate docs
        networth: odin
    Thor:
        email: thor@norse-gods.com
        category: kids
        attach: @ estate docs
    Loki:
        email: loki@norse-gods.com
        category: kids
        attach: @ estate docs
```

Notice that *estate docs* is defined at the top level. It is not a *PostMortem* setting; it simply defines a value that will be interpolated into settings later. The interpolation is done by specifying `@` along with the name of the reference as a value. So for example, in *recipients attach* is specified as `@ estate docs`. This causes the list of estate documents to be used as attachments. The same thing is done in *sign with*, which interpolates *my gpg ids*.

Here is the code for validating and transforming the *PostMortem* settings. For more on *report_voluptuous_errors*, see *Validate with Voluptuous*.

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

from inform import error, os_error, terminate
import nestedtext as nt
from pathlib import Path
```

(continues on next page)

(continued from previous page)

```
from voluptuous import (
    Schema, Invalid, MultipleInvalid, Extra, Required, REMOVE_EXTRA
)
from voluptuous_errors import report_voluptuous_errors
from pprint import pprint

# Settings schema
# First define some functions that are used for validation and coercion
def to_str(arg):
    if isinstance(arg, str):
        return arg
    raise Invalid(f"expected text, found {arg.__class__.__name__}")

def to_ident(arg):
    arg = to_str(arg)
    if arg.isidentifier():
        return arg
    raise Invalid('expected simple identifier')

def to_list(arg):
    if isinstance(arg, str):
        return arg.split()
    if isinstance(arg, dict):
        raise Invalid(f"expected list, found {arg.__class__.__name__}")
    return arg

def to_paths(arg):
    return [Path(p).expanduser() for p in to_list(arg)]

def to_email(arg):
    email = to_str(arg)
    user, _, host = email.partition('@')
    if '.' in host and '@' not in host:
        return arg
    raise Invalid('expected email address')

def to_emails(arg):
    return [to_email(e) for e in to_list(arg)]

def to_gpg_id(arg):
    try:
        return to_email(arg)      # gpg ID may be an email address
    except Invalid:
        try:
            int(arg, base=16)     # if not an email, it must be a hex key
            assert len(arg) >= 8  # at least 8 characters long
            return arg
        except (ValueError, TypeError, AssertionError):
            raise Invalid('expected GPG id')

def to_gpg_ids(arg):
    return [to_gpg_id(i) for i in to_list(arg)]
```

(continues on next page)

(continued from previous page)

```
def to_snake_case(key):
    return '_'.join(key.strip().lower().split())

# provide user-friendly error messages
voluptuous_error_msg_mappings = {
    "extra keys not allowed": ("unknown key", "key"),
    "expected a dictionary": ("expected key-value pairs", "value"),
}

# define the schema for the settings file
schema = Schema(
    {
        Required('my_gpg_ids'): to_gpg_ids,
        'sign_with': to_gpg_id,
        'avendesora_gpg_passphrase_account': to_str,
        'avendesora_gpg_passphrase_field': to_str,
        'name_template': to_str,
        Required('recipients'): {
            Extra: {
                Required('category'): to_ident,
                Required('email'): to_emails,
                'gpg_id': to_gpg_id,
                'attach': to_paths,
                'networth': to_ident,
            }
        },
    },
    extra = REMOVE_EXTRA
)

# this function implements references
def expand_settings(value):
    # allows macro values to be defined as a top-level setting.
    # allows macro reference to be found anywhere.
    if isinstance(value, str):
        value = value.strip()
        if value[:1] == '@':
            value = settings.get(to_snake_case(value[1:]))
        return value
    if isinstance(value, dict):
        return {k:expand_settings(v) for k, v in value.items()}
    if isinstance(value, list):
        return [expand_settings(v) for v in value]
    raise NotImplementedError(value)

def normalize_key(key, parent_keys):
    if parent_keys != ('recipients',):
        # normalize all keys except the recipient names
        return to_snake_case(key)
    return key
```

(continues on next page)

(continued from previous page)

```
try:
    # Read settings
    config_filepath = Path('postmortem.nt')
    if config_filepath.exists():

        # load from file
        settings = nt.load(
            config_filepath,
            keymap = (keymap:={}),
            normalize_key = normalize_key
        )

        # expand references
        settings = expand_settings(settings)

        # check settings and transform to desired types
        settings = schema(settings)

        # show the resulting settings
        print(nt.dumps(settings, default=str))

except nt.NestedTextError as e:
    e.report()
except MultipleInvalid as e:
    report_voluptuous_errors(e, keymap, config_filepath)
except OSError as e:
    error(os_error(e))
terminate()
```

This code uses *expand_settings* to implement references, and it uses the *Voluptuous* schema to clean and validate the settings and convert them to convenient forms. For example, the user could specify *attach* as a string or a list, and the members could use a leading ~ to signify a home directory. Applying *to_paths* in the schema converts whatever is specified to a list and converts each member to a pathlib path with the ~ properly expanded.

Notice that the schema is defined in a different manner than in the :ref:` Volupuous example <voluptuous example>`. In that example, you simply state which type you are expecting for the value and you use the *Coerce* function to indicate that the value should be cast to that type if needed. In this example, simple functions are passed in that perform validation and coercion as needed. This is a more flexible approach that allows better control of the conversions and the error messages.

This code does not do any thing useful, it just reads in and expands the information contained in the input file. It simply represents the beginnings of a program that would use the specified information to generate the postmortem reports. In this case it simply prints the expanded information in the form of a *NestedText* document, which is easier to read that if it were pretty-printed as *Python* or *JSON*.

Here are the processed settings:

```
my_gpg_ids:
    - odin@norse-gods.com
sign_with: odin@norse-gods.com
name_template: {name}-{now:YYMMDD}
recipients:
    Frigg:
        email:
```

(continues on next page)

(continued from previous page)

```
            - frigg@norse-gods.com
        category: wife
        attach:
            - ~/home/estate/trust.pdf
            - ~/home/estate/will.pdf
            - ~/home/estate/deed-valhalla.pdf
        networth: odin
    Thor:
        email:
            - thor@norse-gods.com
        category: kids
        attach:
            - ~/home/estate/trust.pdf
            - ~/home/estate/will.pdf
            - ~/home/estate/deed-valhalla.pdf
    Loki:
        email:
            - loki@norse-gods.com
        category: kids
        attach:
            - ~/home/estate/trust.pdf
            - ~/home/estate/will.pdf
            - ~/home/estate/deed-valhalla.pdf
```

2.12 Common mistakes

2.12.1 Two values for one key

When *load()* or *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
...         > Topeka, Kansas 20692
... """

>>> try:
...     data = nt.loads(content)
... except nt.NestedTextError as e:
...     print(e.get_message())
...     print(e.get_codicil()[0])
invalid indentation.
An indent may only follow a dictionary or list item that does not
already have a value.
```

(continues on next page)

(continued from previous page)

```
   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
...         > Topeka, Kansas 20692
... """

>>> try:
...     data = nt.loads(content.replace('␣␣', '  '))
... except nt.NestedTextError as e:
...     print(e.get_message())
...     print(e.get_codicil()[0])
invalid indentation.
An indent may only follow a dictionary or list item that does not
already have a value, which in this case consists only of whitespace.
   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.

This mistake is easier to see in advance if you configure your editor to show trailing whitespace. To do so in Vim, add:

```
set listchars=trail:␣
```

to your ~/.vimrc file.

2.12.2 Lists or strings at the top level

Most *NestedText* files start with key-value pairs at the top-level and we noticed that many developers would simply assume this in their code, which would result in unexpected crashes when their programs read legal *NestedText* files that had either a list or a string at the top level. To avoid this, the *load()* and *loads()* functions are configured to expect a dictionary at the top level by default, which results in an error being reported if a dictionary key is not the first token found:

```
>>> import nestedtext as nt

>>> content = """
```

(continues on next page)

(continued from previous page)

```
... - a
... - b
... """

>>> try:
...     print(nt.loads(content))
... except nt.NestedTextError as e:
...     e.report()
error: 2: content must start with key or brace ({).
       2 - a
```

This restriction is easily removed using *top*:

```
>>> try:
...     print(nt.loads(content, top=list))
... except nt.NestedTextError as e:
...     e.report()
['a', 'b']
```

The *top* argument can take any of the values shown in the table below. The default value is *dict*. The value given for *top* also determines the value returned by `load()` and `loads()` if the *NestedText* document is empty.

top	value returned for empty document
"dict", *dict*	`{}`
"list", *list*	`[]`
"str", *str*	`""`
"any", *any*	*None*

2.13 Python API

2.13.1 Convert Data to *NestedText*

`nestedtext.`**`dumps`**(*obj*, ** (Keyword-only parameters separator (PEP 3102))*, *width=0*, *inline_level=0*, *sort_keys=False*, *indent=4*, *converters=None*, *map_keys=None*, *default=None*, *dialect=None*)

Recursively convert object to *NestedText* string.

Parameters

- **`obj`** – The object to convert to *NestedText*.
- **`width`** (*`int`*) – Enables inline lists and dictionaries if greater than zero and if resulting line would be less than or equal to given width.
- **`inline_level`** (*`int`*) – Recursion depth must be equal to this value or greater to be eligible for inlining.
- **`sort_keys`** (*`bool or func`*) – Dictionary items are sorted by their key if *sort_keys* is *True*. In this case, keys at all level are sorted alphabetically. If *sort_keys* is *False*, the natural order of dictionaries is retained.

  If a function is passed in, it is expected to return the sort key. The function is passed two tuples, each consists only of strings. The first contains the mapped key, the original key, and the rendered item. So it takes the form:

```
('<mapped_key>', '<orig_key>', '<mapped_key>: <value>')
```

The second contains the keys of the parent.

- **indent** (*int*) – The number of spaces to use to represent a single level of indentation. Must be one or greater.
- **converters** (*dict*) – A dictionary where the keys are types and the values are converter functions (functions that take an object and return it in a form that can be processed by *NestedText*). If a value is False, an unsupported type error is raised.

  An object may provide its own converter by defining the `__nestedtext_converter__` attribute. It may be False, or it may be a method that converts the object to a supported data type for *NestedText*. A matching converter specified in the *converters* argument dominates over this attribute.
- **default** (*func or "strict"*) – The default converter. Use to convert otherwise unrecognized objects to a form that can be processed. 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 that have converters 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. In this case Booleans are rendered as "True" and "False" and None is rendered as an empty string. If *default* is a function, it acts as the default converter. If it raises a TypeError, the value is reported as an unsupported type.
- **map_keys** (*func or keymap*) – This argument is used to modify the way keys are rendered. It may be a keymap that was created by *load()* or *loads()*, in which case keys are rendered into their original form, before any normalization or de-duplication was performed by the load functions.

  It may also be a function that takes two arguments: the key after any needed conversion has been performed, and the tuple of parent keys. The value returned is used as the key and so must be a string. If no value is returned, the key is not modified.
- **dialect** (*str*) – Specifies support for particular variations in *NestedText*.

  In general you are discouraged from using a dialect as it can result in *NestedText* documents that are not compliant with the standard.

  The following deviant dialects are supported.

  ***support inlines*:**
  If "i" is included in *dialect*, support for inline lists and dictionaries is dropped. The default is "I", which enables support for inlines. The main effect of disabling inlines in the dump functions is that empty lists and dictionaries are output using an empty value, which is normally interpreted by *NestedText* as an empty string.

Returns

The *NestedText* content without a trailing newline. *NestedText* files should end with a newline, but unlike *dump()*, this function does not output that newline. You will need to explicitly add that newline when writing the output *dumps()* to a file.

Raises

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

Examples

```
>>> import nestedtext as nt

>>> data = {
...     "name": "Kristel Templeton",
...     "gender": "female",
...     "age": "74",
... }

>>> try:
...     print(nt.dumps(data))
... except nt.NestedTextError as e:
...     print(str(e))
name: Kristel Templeton
gender: 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 restrict *dumps* to only supporting the native types of *NestedText* 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))
key: unsupported type (int).
```

Alternatively, you can specify a function to *default*, which is used to convert values to recognized types. It is used if no suitable 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))
```

(continues on next page)

(continued from previous page)

```
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
```

If *Color* is consistently used with *NestedText*, you can include the converter in *Color* itself.

```
>>> class Color:
...     def __init__(self, color):
...         self.color = color
...     def __nestedtext_converter__(self):
...         return self.color.title()

>>> data["house"] = Color("red")
>>> print(nt.dumps(data))
key: 42
value: 3.1415926
valid: True
house: Red
```

You can also specify a dictionary of converters. The dictionary maps the object type to a converter function.

```
>>> class Info:
...     def __init__(self, **kwargs):
...         self.__dict__ = kwargs

>>> converters = {
...     bool: lambda b: "yes" if b else "no",
...     int: hex,
...     float: lambda f: f"{f:0.3}",
...     Color: lambda c: c.color,
...     Info: lambda i: i.__dict__,
... }

>>> data["attributes"] = Info(readable=True, writable=False)

>>> try:
...    print(nt.dumps(data, converters=converters))
... except nt.NestedTextError as e:
...     print(str(e))
key: 0x2a
value: 3.14
valid: yes
house: red
attributes:
    readable: yes
    writable: no
```

The above example shows that *Color* in the *converters* argument dominates over the `__nestedtest__converter__` class.

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.

```
>>> converters = {
...     bool: lambda b: "yes" if b else "no",
...     int: hex,
...     float: False,
...     Color: lambda c: c.color,
...     Info: lambda i: i.__dict__,
... }

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

converters need not actually change the type of a value, it may simply transform the value. In the following example, *converters* is used to transform dictionaries by removing empty dictionary fields. It is also converts dates and physical quantities to strings.

```
>>> import arrow
>>> from inform import cull
>>> import quantiphy

>>> class Dollars(quantiphy.Quantity):
...     units = "$"
...     form = "fixed"
...     prec = 2
...     strip_zeros = False
...     show_commas = True

>>> converters = {
...     dict: cull,
...     arrow.Arrow: lambda d: d.format("D MMMM YYYY"),
...     quantiphy.Quantity: lambda q: str(q)
... }

>>> transaction = dict(
...     date = arrow.get("2013-05-07T22:19:11.363410-07:00"),
...     description = "Incoming wire from Publisher's Clearing House",
...     debit = 0,
...     credit = Dollars(12345.67)
... )

>>> print(nt.dumps(transaction, converters=converters))
date: 7 May 2013
description: Incoming wire from Publisher's Clearing House
credit: $12,345.67
```

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

Use the *map_keys* argument to format the keys as you wish. For example, you may wish to render the keys at the first level of hierarchy in upper case:

```
>>> def map_keys(key, parent_keys):
...     if len(parent_keys) == 0:
...         return key.upper()

>>> print(nt.dumps(transaction, converters=converters, map_keys=map_keys))
DATE: 7 May 2013
DESCRIPTION: Incoming wire from Publisher's Clearing House
CREDIT: $12,345.67
```

It can also be used map the keys back to their original form when round-tripping a dataset when using key normalization or key de-duplication:

```
>>> content = """
... Michael Jordan:
...     occupation: basketball player
... Michael Jordan:
...     occupation: actor
... Michael Jordan:
...     occupation: football player
... """

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

>>> keymap = {}
>>> people = nt.loads(content, dict, on_dup=de_dup, keymap=keymap)
>>> print(nt.dumps(people))
Michael Jordan:
    occupation: basketball player
Michael Jordan  #2:
    occupation: actor
Michael Jordan  #3:
    occupation: football player

>>> print(nt.dumps(people, map_keys=keymap))
Michael Jordan:
    occupation: basketball player
Michael Jordan:
    occupation: actor
Michael Jordan:
    occupation: football player
```

nestedtext.**dump**(*obj*, *dest*, ***kwargs*)

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

Parameters

- **obj** – The object to convert to *NestedText*.
- **dest** (*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, or 1 for stdout). 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. If you also wish to further distinguish the file type by giving the schema, it is recommended that you use two suffixes, with the suffix that specifies the schema given first and .nt given last. For example: flicker.sig.nt.

- **kwargs** – See *dumps()* for optional arguments.

Returns

The *NestedText* content with a trailing newline. This differs from *dumps()*, which does not add the trailing newline.

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",
...     "gender": "female",
...     "age": "74",
... }

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

This example writes to a file specified by file name. In general, the file name and extension are arbitrary. However, by convention a '.nt' suffix is generally used for *NestedText* files.

```
>>> try:
...     nt.dump(data, "data.nt")
... except nt.NestedTextError as e:
...     e.terminate()
... except OSError as e:
...     fatal(os_error(e))
```

2.13.2 Convert *NestedText* to Data

nestedtext.**loads**(*content*, *top='dict'*, ***, *source=None*, *on_dup=None*, *keymap=None*, *normalize_key=None*, *dialect=None*)

Loads *NestedText* from string.

Parameters

- **content** (*str*) – String that contains encoded data.
- **top** (*str*) – Top-level data type. The NestedText format allows for a dictionary, a list, or a string as the top-level data container. By specifying top as "dict", "list", or "str" you constrain both the type of top-level container and the return value of this function. By specifying "any" you enable support for all three data types, with the type of the returned value matching that of top-level container in content. As a short-hand, you may specify the *dict*, *list*, *str*, and *any* built-ins rather than specifying *top* with a string.
- **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. Specifying "error" causes them to raise exceptions (the default behavior). Specifying "ignore" causes them to be ignored (first wins). Specifying "replace" results in them replacing earlier items (last wins). By specifying a function, the keys can be de-duplicated. This callback function returns a new key and takes two arguments:

  key:
  The new key (duplicates an existing key).

  state:
  A dictionary containing other possibly helpful information:

  dictionary:
  The entire dictionary as it is at the moment the duplicate key is found. You should not change it.

  keys:
  The keys that identify the dictionary.

  This dictionary is created as *loads* is called and deleted as it returns. Any values placed in it are retained and available on subsequent calls to this function during the load operation.

  This function should return a new key. If the key duplicates an existing key, the value associated with that key is replaced. If *None* is returned, this key is ignored. If a *KeyError* is raised, the duplicate key is reported as an error.

  Be aware that key de-duplication occurs after key normalization. As such you should generate keys during de-duplication that are consistent with your normalization scheme.
- **keymap** (*dict*) – Specify an empty dictionary or nothing at all for the value of this argument. If you give an empty dictionary it will be filled with location information for the values that are returned. Upon return the dictionary maps a tuple containing the keys for the value of interest to the location of that value in the *NestedText* source document. The location is contained in a *Location* object. You can access the line and column number using the *Location.as_tuple()* method, and the line that contains the value annotated with its location using the *Location.as_line()* method.
- **normalize_key** (*func*) – A function that takes two arguments; the original key for a value and the tuple of normalized keys for its parent values. It then transforms the given key into

the desired normalized form. Only called on dictionary keys, so the key will always be a string.

- **dialect** (*str*) – Specifies support for particular variations in *NestedText*.

  In general you are discouraged from using a dialect as it can result in *NestedText* documents that are not compliant with the standard.

  The following deviant dialects are supported.

  ***support inlines*:**
  If "i" is included in *dialect*, support for inline lists and dictionaries is dropped. The default is "I", which enables support for inlines. The main effect of disabling inlines in the load functions is that keys may begin with `[` or `{`.

Returns
The extracted data. The type of the return value is specified by the top argument. If top is "any", then the return value will match that of top-level data container in the input content. If content is empty, an empty data value of the type specified by top is returned. If top is "any" None is returned.

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

Examples

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

```
>>> import nestedtext as nt

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

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

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

loads() takes an optional argument, *source*. If specified, it is added to any error messages. It is often used to designate the source of *NestedText* document. For example, if *contents* were read from a file, *source* 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(), source=filename)
... except nt.NestedTextError as e:
...     print(e.render())
examples/duplicate-keys.nt, 5: duplicate key: name.
       4 name:
```

(continues on next page)

(continued from previous page)

```
       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.nestedtext.NestedTextError: 3: duplicate key: key.
       2 key: value 1
       3 key: value 2


>>> print(nt.loads(content, on_dup="ignore"))
{'key': 'value 1', 'name': 'value 4'}

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

>>> def de_dup(key, 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'}
```

nestedtext.**load**(*f*, *top='dict'*, ***, *source=None*, *on_dup=None*, *keymap=None*, *normalize_key=None*, *dialect=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 0 for stdin), 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.

- **kwargs** – See *loads()* for optional arguments.

Returns
The extracted data. See *loads()* description of the return value.

Raises

- ***NestedTextError*** – if there is a problem in the *NextedText* document.
- **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())
groceries:
  - Bread
  - Peanut butter
  - Jam


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

Load from a *pathlib.Path*:

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

Load from an open file object:

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

2.13.3 Location

Location objects are returned from *load()* and *loads()* as the values in a *keymap*. They are also returned by *get_location()*. Objects of this class holds the line and column numbers of the key and value tokens.

class nestedtext.**Location**(*line=None*, *col=None*, *key_line=None*, *key_col=None*)

Holds information about the location of a token.

Returned from *load()* and *loads()* as the values in a *keymap*. Objects of this class holds the line and column numbers of the key and value tokens.

as_line(*kind='value'*, *offset=0*)

Returns a string containing two lines that identify the token in context. The first line contains the line number and text of the line that contains the token. The second line contains a pointer to the token.

Parameters

- **kind** – Specify either "key" or "value" depending on which token is desired.

- **offset** – If *offset* is None, the error pointer is not added to the line. If *offset* is an integer, the pointer is moved to the right by this many characters. The default is 0. If *offset* is a tuple, it must have two values. The first is the row offset and the second is the column offset. This is useful for annotating errors in multiline strings.

Raises
***IndexError* if row offset is out of range.** –

as_tuple(*kind='value'*)

Returns the location of either the value or the key token as a tuple that contains the line number and the column number. The line and column numbers are 0 based.

Parameters
kind – Specify either "key" or "value" depending on which token is desired.

get_line_numbers(*kind='value'*, *sep=None*)

Returns the line numbers of a token either as a pair of integers or as a string.

Parameters

- **kind** – Specify either "key" or "value" depending on which token is desired.
- **sep** – The separator string.

  If given a string is returned and *sep* is inserted between two line numbers. In this case the line numbers start at 1.

  If *sep* is not given, a tuple of integers is returned. In this case the line numbers start at 0, but the second number returned is the last line number plus 1. This form is suitable to use with the Python slice function to extract the lines from the *NestedText* source.

2.13.4 Utilities

Extras that are useful when using *NestedText*.

nestedtext.**get_keys**(*keys*, *keymap*, *, *original=True*, *strict=True*, *sep=None*)

Returns a key sequence given a normalized key sequence.

Keys in the dataset output by the load functions are referred to as normalized keys, even though no key normalization may have occurred. This distinguishes them from the original keys, which are the keys given in the NestedText document read by the load functions. The original keys are mapped to normalized keys by the *normalize_key* argument to the load function. If normalization is not performed, the normalized keys are the same as the original keys.

By default this function returns the original key sequence that corresponds to *keys*, a normalized key sequence.

Parameters

- **keys** – The sequence of normalized keys that identify a value in the dataset.
- **keymap** – The keymap returned from *load()* or *loads()*.
- **original** – If true, return keys as originally given in the NestedText document (pre-normalization). Otherwise return keys as they exist in the dataset (post-normalization). The value of this argument has no effect if the keys were not normalized.
- **strict** – *strict* controls what happens if the given keys are not found in *keymap*.

  The various options can be helpful when reporting errors on key sequences that do not exist in the data set. Since they are not in the dataset, the original keys are not available.

  True or "error":
  A *KeyError* is raised.

False or "all":
All keys given in *keys* are returned.

"found":
Only the initial available keys are returned.

"missing":
Only the missing final keys are returned.

When returning keys, the initial available keys are converted to their original form if *original* is true, The missing keys are always returned as given.

- **sep** – A join string. If given the keys are interleaved with *sep* and joined into a string before being returned.

Returns
A tuple containing the desired keys if *sep* is not given. A string formed by joining the keys with *sep* if *sep* is given.

Examples

```
>>> import nestedtext as nt

>>> contents = """
... Names:
...     Given: Fumiko
... """

>>> def normalize_key(key, keys):
...     return key.lower()

>>> data = nt.loads(contents, "dict", normalize_key=normalize_key, keymap=(keymap:=
↪{}))

>>> print(get_keys(("names", "given"), keymap))
('Names', 'Given')

>>> print(get_keys(("names", "given"), keymap, sep=""))
NamesGiven

>>> print(get_keys(("names", "given"), keymap, original=False))
('names', 'given')

>>> keys = get_keys(("names", "surname"), keymap, strict=True)
Traceback (most recent call last):
...
KeyError: ('names', 'surname')

>>> print(get_keys(("names", "surname"), keymap, strict="found"))
('Names',)

>>> print(get_keys(("names", "surname"), keymap, strict="missing"))
('surname',)

>>> print(get_keys(("names", "surname"), keymap, strict="all"))
('Names', 'surname')
```

nestedtext.**get_value**(*data*, *keys*)

Get value from keys.

Parameters

- **data** – Your data set as returned by *load()* or *loads()*.
- **keys** – The sequence of normalized keys that identify a value in the dataset.

Returns

The value that corresponds to a tuple of keys from a keymap.

Examples

```
>>> import nestedtext as nt

>>> contents = """
... names:
...     given: Fumiko
...     surname: Purvis
... """

>>> data = nt.loads(contents, "dict")

>>> get_value_from_keys(data, ("names", "given"))
'Fumiko'
```

nestedtext.**get_line_numbers**(*keys*, *keymap*, *kind='value'*, ***, *strict=True*, *sep=None*)

Get line numbers from normalized key sequence.

This function returns the line numbers of the key or value selected by *keys*. It is used when reporting an error in a value that is possibly a multiline string. If the location contained in a keymap were used the user would only see the line number of the first line, which may confuse some users into believing the error is actually contained in the first line. Using this function gives both the starting and ending line number so the user focuses on the whole string and not just the first line. This only happens for multiline keys and multiline strings.

If *sep* is given, either one line number or both the beginning and ending line numbers are returned, joined with the separator. In this case the line numbers start from line 1.

If *sep* is not given, the line numbers are returned as a tuple containing a pair of integers that is tailored to be suitable to be arguments to the Python slice function (see example). The beginning line number and 1 plus the ending line number is returned as a tuple. In this case the line numbers start at 0.

If *keys* corresponds to a composite value (a dictionary or list), the line on which it ends cannot be easily determined, so the value is treated as if it consists of a single line. This is considered tolerable as it is expected that this function is primarily used to return the line number of leaf values, which are always strings.

Parameters

- **keys** – The sequence of normalized keys that identify a value in the dataset.
- **keymap** – The keymap returned from *load()* or *loads()*.
- **kind** – Specify either "key" or "value" depending on which token is desired.
- **strict** – If *strict* is true, a *KeyError* is raised if *keys* is not found. Otherwise the line number that corresponds to composite value that would contain *keys* if it existed. This composite value corresponds to the largest sequence of keys that does actually exist in the dataset.
- **sep** – The separator string. If given a string is returned and *sep* is inserted between two line numbers. Otherwise a tuple is returned.

Raises

KeyError – If keys are not in *keymap* and *strict* is true.

Example

```
>>> import nestedtext as nt
```

```
>>> doc = """
... key:
...     > this is line 1
...     > this is line 2
...     > this is line 3
... """
```

```
>>> data = nt.loads(doc, keymap=(keymap:={}))
>>> keys = ("key",)
>>> lines = nt.get_line_numbers(keys, keymap, sep="-")
>>> text = doc.splitlines()
>>> print(
...     f"Lines {lines}:",
...     *text[slice(*nt.get_line_numbers(keys, keymap))],
...     sep="\n"
... )
Lines 3-5:
    > this is line 1
    > this is line 2
    > this is line 3
```

nestedtext.**get_location**(*keys*, *keymap*)

Returns *Location* information from the keys. None is returned if location is unknown.

Parameters

- **keys** – The sequence of normalized keys that identify a value in the dataset.
- **keymap** – The keymap returned from *load()* or *loads()*.

Deprecated functions

These functions are to be removed in future versions.

nestedtext.**get_value_from_keys**(*data*, *keys*)

Get value from keys.

Deprecated in version 3.7 and is to be removed in future versions. *get_value_from_keys()* is replaced *get_value()*, which is identical.

Parameters

- **data** – Your data set as returned by *load()* or *loads()*.
- **keys** – The sequence of normalized keys that identify a value in the dataset.

Returns

The value that corresponds to a tuple of keys from a keymap.

Examples

```
>>> import nestedtext as nt

>>> contents = """
... names:
...     given: Fumiko
...     surname: Purvis
... """

>>> data = nt.loads(contents, "dict")

>>> get_value_from_keys(data, ("names", "given"))
'Fumiko'
```

nestedtext.**get_lines_from_keys**(*data*, *keys*, *keymap*, *kind='value'*, *sep=None*)

Get line numbers from normalized keys.

Deprecated in version 3.7 and is to be removed in future versions. Use *get_line_numbers()* as a replacement.

This function returns the line numbers of the key or value selected by *keys*. It is used when reporting an error in a value that is possibly a multiline string. If the location contained in a keymap were used the user would only see the line number of the first line, which may confuse some users into believing the error is actually contained in the first line. Using this function gives both the starting and ending line number so the user focuses on the whole string and not just the first line.

If *sep* is given, either one line number or both the beginning and ending line numbers are returned, joined with the separator. In this case the line numbers start from line 1.

If *sep* is not given, the line numbers are returned as a tuple containing a pair of integers that is tailored to be suitable to be arguments to the Python *slice* function (see example). The beginning line number and 1 plus the ending line number is returned as a tuple. In this case the line numbers start at 0.

If the value is requested and it is a composite (a dictionary or list), the line on which it ends cannot be easily determined, so the value is treated as if it consists of a single line. This is considered tolerable as it is expected that this function is primarily used to return the line number of leaf values, which are always strings.

Parameters

- **data** – Your data set as returned by *load()* or *loads()*.
- **keys** – The collection of keys that identify a value in the dataset.
- **keymap** – The keymap returned from *load()* or *loads()*.
- **kind** (*str*) – Specify either "key" or "value" depending on which token is desired.
- **sep** – The separator string. If given a string is returned and *sep* is inserted between two line numbers. Otherwise a tuple is returned.

Example

```
>>> import nestedtext as nt
```

```
>>> doc = """
... key:
...     > this is line 1
...     > this is line 2
```

(continues on next page)

(continued from previous page)

```
...     > this is line 3
... """
```

```
>>> data = nt.loads(doc, keymap=(keymap:={}))
>>> keys = ("key",)
>>> lines = nt.get_lines_from_keys(data, keys, keymap, sep="-")
>>> text = doc.splitlines()
>>> print(
...     f"Lines {lines}:",
...     *text[slice(*nt.get_lines_from_keys(data, keys, keymap))],
...     sep="\n"
... )
Lines 3-5:
    > this is line 1
    > this is line 2
    > this is line 3
```

nestedtext.**get_original_keys**(*keys*, *keymap*, *strict=False*)

Get original keys from normalized keys.

Deprecated in version 3.7 and is to be removed in future versions. Use *get_keys()* as a replacement.

This function is used when the *normalize_key* argument is used with *load()* or *loads()* to transform the keys to a standard form. Given a set of normalized keys that point to a particular value in the returned dataset, this function returns the original keys for that value.

Parameters

- **keys** – The collection of normalized keys that identify a value in the dataset.
- **keymap** – The keymap returned from *load()* or *loads()*.
- **strict** – If true, a *KeyError* is raised if the given keys are not found in *keymap*. Otherwise, the given normalized keys will be returned rather than the original keys. This is helpful when reporting errors on required keys that do not exist in the data set. Since they are not in the dataset, the original keys are not available.

Returns

A tuple containing the original keys names.

Examples

```
>>> import nestedtext as nt

>>> contents = """
... Names:
...     Given: Fumiko
... """

>>> def normalize_key(key, keys):
...     return key.lower()

>>> data = nt.loads(contents, "dict", normalize_key=normalize_key, keymap=(keymap:=
↪{}))
```

(continues on next page)

(continued from previous page)

```
>>> print(get_original_keys(("names", "given"), keymap))
('Names', 'Given')

>>> print(get_original_keys(("names", "surname"), keymap))
('Names', 'surname')

>>> keys = get_original_keys(("names", "surname"), keymap, strict=True)
Traceback (most recent call last):
...
KeyError: ('names', 'surname')
```

nestedtext.**join_keys**(*keys*, *sep=', '*, *keymap=None*, *strict=False*)

Joins the keys into a string.

Deprecated in version 3.7 and is to be removed in future versions. Use *get_keys()* as a replacement.

Parameters

- **keys** – A tuple of keys.
- **sep** – The separator string. It is inserted between each key during the join.
- **keymap** – The keymap returned from *load()* or *loads()*. It is optional. If given the given keys are converted to the original keys before the joining.
- **strict** – If true, a *KeyError* is raised if the given keys are not found in *keymap*. Otherwise, the given normalized keys will be returned rather than the original keys. This is helpful when reporting errors on required keys that do not exist in the data set. Since they are not in the dataset, the original keys are not available.

Returns

A string containing the joined keys.

Examples

```
>>> import nestedtext as nt

>>> contents = """
... Names:
...     Given: Fumiko
... """

>>> def normalize_key(key, keys):
...     return key.lower()

>>> data = nt.loads(contents, "dict", normalize_key=normalize_key, keymap=(keymap:=
↪{}))

>>> join_keys(("names", "given"))
'names, given'

>>> join_keys(("names", "given"), sep=".")
'names.given'

>>> join_keys(("names", "given"), keymap=keymap)
```

(continues on next page)

(continued from previous page)

```
'Names, Given'

>>> join_keys(("names", "surname"), keymap=keymap)
'Names, surname'

>>> join_keys(("names", "surname"), keymap=keymap, strict=True)
Traceback (most recent call last):
    ...
KeyError: ('names', 'surname')
```

2.13.5 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.

All exceptions provide the following attributes:

args
: The exception arguments. A tuple that usually contains the problematic value.

template
: The possibly parameterized text used for the error message.

Exceptions raised by the *loads()* or *load()* functions provide the following additional 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.

prev_line
: The text of the meaningful line immediately before where the problem was found. This will not be a comment or blank line.

lineno
: The number of the line where the problem was found. Line numbers are zero based except when included in messages to the end user.

colno
: The number of the character where the problem was found on *line*. Column numbers are zero based.

codicil
: The line that contains the error decorated with the location of the error.

The exception culprit is the tuple that indicates where the error was found. With exceptions from *loads()* or *load()*, the culprit consists of the source name, if available, and the line number. With exceptions from *dumps()* or *dump()*, the culprit consists of the keys that lead to the problematic value.

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
```

(continues on next page)

(continued from previous page)

```
>>> 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.
       1 name1: value1
       2 name1: value2
```

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 extent of the text on each line and help you see trouble-

some 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.
       1 name1: value1
       2 name1: value2
```

get_message(*template=None*)

Get exception message.

Parameters

template (*str*) – This argument is treated as a format string and is passed both the unnamed and named arguments. The resulting string is treated as the message and returned.

If not specified, the *template* keyword argument passed to the exception is used. If there was no *template* argument, then the positional arguments of the exception are joined using *sep* and that is returned.

Returns

The formatted message without the culprits.

get_culprit(*culprit=None*)

Get the culprits.

Culprits are extra pieces of information attached to an error that help to identify the source of the error. For example, file name and line number where the error was found are often attached as culprits.

Return the culprit as a tuple. If a culprit is specified as an argument, it is appended to the exception's culprit without modifying it.

Parameters

culprit (*string, number or tuple of strings and numbers*) – A culprit or collection of culprits that is appended to the return value without modifying the cached culprit.

Returns

The culprit argument is prepended to the exception's culprit and the combination is returned. The return value is always in the form of a tuple even if there is only one component.

get_codicil(*codicil=None*)

Get the codicils.

A codicil is extra text attached to an error that can clarify the error message or to give extra context.

Return the codicil as a tuple. If a codicil is specified as an argument, it is appended to the exception's codicil without modifying it.

Parameters
codicil (*string or tuple of strings*) – A codicil or collection of codicils that is appended to the return value without modifying the cached codicil.

Returns
The codicil argument is appended to the exception's codicil and the combination is returned. The return value is always in the form of a tuple even if there is only one component.

`report`(***new_kwargs*)

Report exception to the user.

Prints the error message on the standard output.

The `inform.error()` function is called with the exception arguments.

Parameters
****kwargs** – *report()* takes any of the normal keyword arguments normally allowed on an informant (culprit, template, etc.). Any keyword argument specified here overrides those that were specified when the exception was first raised.

`terminate`(***new_kwargs*)

Report exception and terminate.

Prints the error message on the standard output and exits the program.

The `inform.fatal()` function is called with the exception arguments.

Parameters
****kwargs** – *report()* takes any of the normal keyword arguments normally allowed on an informant (culprit, template, etc.). Any keyword argument specified here overrides those that were specified when the exception was first raised.

`reraise`(***new_kwargs*)

Re-raise the exception with replaced arguments.

`render`(*template=None*, *include_codicil=True*)

Convert exception to a string for use in an error message.

Parameters

- **template** (*str*) – This argument is treated as a format string and is passed both the unnamed and named arguments. The resulting string is treated as the message and returned.

  If not specified, the *template* keyword argument passed to the exception is used. If there was no *template* argument, then the positional arguments of the exception are joined using *sep* and that is returned.

- **include_codicil** (*bool*) – Include the codicil in the rendered message.

Returns
The formatted message with any culprits.

2.14 Releases

This page documents the changes in the Python implementation of *NestedText*. Changes to the *NestedText* language are shown in *Language changes*.

2.14.1 Latest development version

Version: 3.8
Released: 2025-12-26

2.14.2 v3.8 (2025-12-26)

Bug Fixes:

- Catch multiline key not followed by indented value.
- Make line-ending recognition consistent for files and strings.
- Fix bug in `get_value()`.
- Report error if there is content that follows top-level inline dictionary.

Enhancements:

- Support files that have utf8 byte-order marker (BOM).
- Allow binary files to be passed to `load()` and `dump()`.
- Allow byte strings to be passed to `loads()`.

Tests:

In version 3.8 a new implementation independent suite of tests has replaced the original suite. These tests should be more comprehensive and more independent of the Python implementation of *NestedText*.

2.14.3 v3.7 (2024-04-27)

- Added ability to disable support for inlines using *dialect* argument to `load()` and `loads()`.
- Added `get_keys()`, `get_value()`, `get_line_numbers()`, and `get_location()`.
- Deprecated `get_value_from_keys()`, `get_lines_from_keys()`, `get_original_keys()`, and `join_keys()`.
- Added *offset* argument to `Location.as_line()`.
- Add ability to specify *source* to `load()`.
- Clarified policy on white space in inline strings.

2.14.4 v3.6 (2023-05-30)

- De-duplicating with the *on_dup* argument to `loads()` now works well for error reporting with keymaps.
- The *map_keys* argument has been added to `dump()` and `dumps()`.

> **Warning**
>
> The *sort_keys* argument to `dump()` and `dumps()` has changed. When passing a call-back function to *sort_keys*, that call-back function now has a second argument, *parent_keys*. In addition, the first argument has changed. It is now a tuple with three members rather than two, with the new and leading member being the mapped key rather than the original key.

Warning

The state passed to the *on_dup* functions of `dump()` and `dumps()` no longer contains the value associated with the duplicate key.

2.14.5 v3.5 (2022-11-04)

- Minor refinements and fixes.

2.14.6 v3.4 (2022-06-15)

- Improved the *on_dup* parameter to `load()` and `loads()`.
- Added *strict* argument to `join_keys()`.

Warning

Be aware that the new version of the *on_dup* parameter is not compatible with previous versions.

2.14.7 v3.3 (2022-06-07)

- Add *normalize_key* argument to `load()` and `loads()`.
- Added utility functions for operating on keys and keymaps:
  - `get_value_from_keys()`
  - `get_lines_from_keys()`
  - `get_original_keys()`
  - `join_keys()`
- None passed as key is now converted to an empty string rather than "None".

2.14.8 v3.2 (2022-01-17)

- Add circular reference detection and reporting.

2.14.9 v3.1 (2021-07-23)

- Change error reporting for `dumps()` and `dump()` functions; culprit is now the keys rather than the value.

2.14.10 v3.0 (2021-07-17)

- Deprecate trailing commas in inline lists and dictionaries.
- Adds *keymap* argument to `load()` and `loads()`.
- Adds *inline_level* argument to `dump()` and `dumps()`.
- Implement *on_dup* argument to `load()` and `loads()` in inline dictionaries.
- Apply *convert* and *default* arguments of `dump()` and `dumps()` to dictionary keys.

Warning

Be aware that aspects of this version are not backward compatible. Specifically, trailing commas are no longer supported in inline dictionaries and lists. In addition, `[ ]` now represents a list that contains an empty string, whereas previously it represented an empty list.

2.14.11 v2.0 (2021-05-28)

- Deprecate quoted keys.
- Add multiline keys to replace quoted keys.
- Add inline lists and dictionaries.
- Move from *renderers* to *converters* in `dump()` and `dumps()`. Both allow you to support arbitrary data types. With *renderers* you provide functions that are responsible for directly creating the text to be inserted in the *NestedText* output. This can be complicated and error prone. With *converters* you instead convert the object to a known *NestedText* data type (dict, list, string, ...) and the *dump* function automatically formats it appropriately.
- Restructure documentation.

Warning

Be aware that aspects of this version are not backward compatible.

1. It no longer supports quoted dictionary keys.
2. The *renderers* argument to `dump()` and `dumps()` has been replaced by *converters*.
3. It no longer allows one to specify *level* in `dump()` and `dumps()`.

2.14.12 v1.3 (2021-01-02)

- Move the test cases to a submodule.

Note

When cloning the *NestedText* repository you should use the –recursive flag to get the *official_tests* submodule:

```
git clone --recursive https://github.com/KenKundert/nestedtext.git
```

When updating an existing repository, you need to initialize the submodule after doing a pull:

```
git submodule update --init --remote tests/official_tests
```

This only need be done once.

2.14.13 v1.2 (2020-10-31)

- Treat CR LF, CR, or LF as a line break.
- Always quote keys that start with a quote.

2.14.14 v1.1 (2020-10-13)

- Add ability to specify return type of `load()` and `loads()`.
- Quoted keys are now less restricted.
- Empty dictionaries and lists are rejected by `dump()` and `dumps()` except as top-level object if *default* argument is specified as 'strict'.

Warning

Be aware that this version is not fully backward compatible. Unlike previous versions, this version allows you to restrict the type of the return value of the `load()` and `loads()` functions, and the default is 'dict'. The previous behavior is still supported, but you must explicitly specify *top='any'* as an argument.

This change results in a simpler return value from `load()` and `loads()` in most cases. This substantially reduces the chance of coding errors. It was noticed that it was common to simply assume that the top-level was a dictionary when writing code that used these functions, which could result in unexpected errors when users hand-create the input data. Specifying the return value eliminates this type of error.

There is another small change that is not backward compatible. The source argument to these functions is now a keyword only argument.

2.14.15 v1.0 (2020-10-03)

- Production release.

INDEX