NestedText

Release 3.2.0

unknown

Jan 17, 2022

LANGUAGE

Authors: Ken & Kale Kundert
Version: 3.2.0
Released: 2022-01-17
Documentation: nestedtext.org.
Please post all questions, suggestions, and bug reports to: Github.

NestedText is a file format for holding structured data to be entered, edited, or viewed by people. It organizes the data into a nested collection of dictionaries, lists, and strings without the need for quoting or escaping. A unique feature of this file format is that it only supports one scalar type: strings. While the decision to eschew integer, real, date, etc. types may seem counter intuitive, it leads to simpler data files and applications that are more robust.

NestedText is convenient for configuration files, address books, account information, and the like. Because there is no need for quoting or escaping, it is particularly nice for holding code fragments. Here is an example of a file that contains a few addresses:

```
# Contact information for our officers

president:
    name: Katheryn McDaniel
    address:
        > 138 Almond Street
        > 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

vice president:
    name: Margaret Hodge
    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

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.

1.1 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* tries not to transform the data in any meaningful way to avoid creating unpleasant surprises. It parses the structure of the data without doing anything that might change how the data is interpreted. Instead, it aims to make it easy for you to interpret the data yourself. After all, you understand what the data is supposed to mean, so you are in the best position to interpret it. There are also many powerful tools available to help with *this exact task*.

1.2 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.

1.2.1 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
```

1.2.2 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: |
        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

      '
```

(continues on next page)

(continued from previous page)

```
  - 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

1.2.3 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
```

(continues on next page)

(continued from previous page)

```
        - 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}
    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

1.2.4 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
```

(continues on next page)

(continued from previous page)

```
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
    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
```

(continues on next page)

(continued from previous page)

```
    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
    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
```

(continues on next page)

(continued from previous page)

```
    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

1.2.5 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. This is the reason that JSON does not distinguish between integers and reals.

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.
`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.

1.3 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*.

1.3.1 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 with a colon (`:`) followed by either a space or a newline. The key must be a string and must not start with a `-`, `>`, `:`, `[`, `{` or space character; or contain newline characters or the `:␣` character sequence. Any spaces between the key and the colon that separates the key from the value 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
```

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 be 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
    - 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.

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}
↪}
```

1.3.2 Lists

A list is an ordered collection of values:

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

A list item is introduced with a dash followed by a space or a newline at the start of a line. 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 `-␣` 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
```

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.

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.

1.3.3 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:

```
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 specified on lines prefixed with the greater-than symbol followed by a space or a newline. 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 greater than tag with the tags removed and the lines joined together with newline characters inserted between each line. Except for the space that separates the tag from the text, 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.

1.3.4 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.
```

1.3.5 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

president:
    name: Katheryn McDaniel
    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

vice president:
    name: Margaret Hodge
    address:
        > 2586 Marigold Lane
        > 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.

1.3.6 NestedText Files

NestedText files should be encoded with UTF-8. 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.

1.4 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 white space characters (spaces or tabs). 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 a space or line break;
2. or a hash {`#`), left bracket (`[`), or left brace (`{`) as the first non-white space character on a line.

Most of these symbols only introduce tags when they are the first non-space character on a line, but colon tags need not start the 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-white-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 a 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 and all newlines except the last.

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

Key items:

If the first non-space character on a line is a colon followed immediately by a 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 and all newlines except the last.

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*.

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 a space (`:␣`) or a newline. The dictionary item consists of the key, the tag, and the trailing value. Any 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 leading spaces (any 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 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.

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 spaces that do follow the colon are 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 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, this includes spaces, tabs, Unicode spaces, etc.

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.

Result:

When a document is converted from *NestedText* the result is a hierarchical collection of dictionaries, lists and strings. All dictionary keys are strings.

1.5 Related projects

1.5.1 Reference Material

nestedtext docs

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.

1.5.2 Implementations

nestex

Go implementation of *NestedText* (supports *NestedText v3.0*).

janet-nested-text

Janet implementation of *NestedText* (supports *NestedText v3.0*).

zig-nestedtext

Zig implementation of *NestedText* (slight subset of *NestedText v2.0*).

1.5.3 Utilities

parametrize from file

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

vim-nestedtext

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

visual studio

Syntax files for *Visual Studio* (supports *NestedText v1.0*).

1.6 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.

1.6.1 Latest development version

Version: 3.2.0
Released: 2022-01-17

1.6.2 v3.1 (2021-07-23)

- No changes.

1.6.3 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 with an that contains an empty string, whereas previously it represented an empty list.

1.6.4 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.

1.6.5 v1.3 (2021-01-02)

- No changes.

1.6.6 v1.2 (2020-10-31)

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

1.6.7 v1.1 (2020-10-13)

- No changes.

1.6.8 v1.0 (2020-10-03)

- Initial release.

1.7 Basic use

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

1.7.1 Installation

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

1.7.2 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*.

More advanced usage is described in *loads()*.

1.7.3 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()

>>> 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))
```

More advanced usage is described in *dumps()*.

1.8 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:

- pydantic: Define schema using type annotations
- voluptuous: Define schema using objects
- 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 *Examples* 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.

1.9 Examples

1.9.1 Validate with *Pydantic*

This example shows how to use pydantic to validate and parse a *NestedText* file. The 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
```

(continues on next page)

(continued from previous page)

```
  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

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

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

pprint(config.dict())
```

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'}
```

1.9.2 Validate with *Voluptuous*

This example shows how to use voluptuous 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 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, Invalid
from inform import fatal, full_stop
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,
})
try:
    keymap = {}
    raw = nt.load('deploy.nt', keymap=keymap)
    config = schema(raw)
except nt.NestedTextError as e:
    e.terminate()
except Invalid as e:
    kind = 'key' if 'key' in e.msg else 'value'
    loc = keymap[tuple(e.path)]
    fatal(full_stop(e.msg), culprit=e.path, codicil=loc.as_line(kind))

pprint(config)
```

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'}
```

This example demonstrates how to use the *keymap* argument from *loads()* or *load()* to add location information to Voluptuous error messages.

1.9.3 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]
    -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
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
```

(continues on next page)

(continued from previous page)

```
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) + "\n"

    # output NestedText content; to file or to stdout
    if input_filename:
        output_path = input_path.with_suffix('.nt')
        if output_path.exists():
            if not cmdline['--force']:
                fatal('file exists, use -f to force over-write.', culprit=output_path)
        output_path.write_text(nestedtext_content, encoding='utf-8')
    else:
        sys.stdout.write(nestedtext_content)

except OSError as e:
    fatal(os_error(e))
except nt.NestedTextError as e:
    e.terminate()
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')[lineno-2:lineno]
        lines = []
        for i, l in zip(range(lineno-len(lines_before), lineno), lines_before):
            lines.append(f'{i+1:>4}> {l}')
        lines_before = '\n'.join(lines)
        lines_after = e.doc.split('\n')[lineno:lineno+1]
        lines = []
        for i, l in zip(range(lineno, lineno + len(lines_after)), lines_after):
            lines.append(f'{i+1:>4}> {l}')
```

(continues on next page)

(continued from previous page)

```
        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
```

1.9.4 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
from pathlib import Path
import json
import nestedtext as nt
import sys
sys.stdin.reconfigure(encoding='utf-8')
sys.stdout.reconfigure(encoding='utf-8')
```

(continues on next page)

(continued from previous page)

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


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

try:
    if input_filename:
        input_path = Path(input_filename)
        data = nt.load(input_path, top='any', on_dup=de_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=de_dup)
        json_content = json.dumps(data, indent=4, ensure_ascii=False)
        sys.stdout.write(json_content + '\n')
except OSError as e:
    fatal(os_error(e))
except nt.NestedTextError as e:
    e.terminate()
except KeyboardInterrupt:
    done()
```

1.9.5 CSV to NestedText

This example implements a command-line utility that converts a *CSV* file to *NestedText*. It demonstrates the use of the *converters* argument to *dumps()*, which is used to cull empty dictionary fields.

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

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

options:
    -n, --names             first row contains column names
    -c, --cull              remove empty fields (only for --names)
    -f, --force             force overwrite of output file
    -i <n>, --indent <n>    number of spaces per indent [default: 4]
```

(continues on next page)

(continued from previous page)

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

If --names is specified, then the first line is assumed to hold the column/field
names with the remaining lines containing the data.  In this case the output is
a list of dictionaries.  Otherwise every line contains data and that data is
output as a list of lists.
"""

from docopt import docopt
from inform import cull, done, fatal, full_stop, os_error, warn
from pathlib import Path
import csv
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

# strip dictionaries of empty fields if requested
converters = {dict: cull} if cmdline['--cull'] else {}

try:
    # read CSV content; from file or from stdin
    if input_filename:
        input_path = Path(input_filename)
        csv_content = input_path.read_text(encoding='utf-8')
    else:
        csv_content = sys.stdin.read()
    if cmdline['--names']:
        data = csv.DictReader(csv_content.splitlines())
    else:
        data = csv.reader(csv_content.splitlines())

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

    # output NestedText content; to file or to stdout
    if input_filename:
        output_path = input_path.with_suffix('.nt')
        if output_path.exists():
            if not cmdline['--force']:
                fatal('file exists, use -f to force over-write.', culprit=output_path)
        output_path.write_text(nt_content, encoding='utf-8')
    else:
```

(continues on next page)

(continued from previous page)

```
        sys.stdout.write(nt_content)

except OSError as e:
    fatal(os_error(e))
except nt.NestedTextError as e:
    e.terminate()
except csv.Error as e:
    fatal(full_stop(e), culprit=(input_filename, data.line_num))
except KeyboardInterrupt:
    done()
```

1.9.6 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
```

(continues on next page)

(continued from previous page)

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

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

1.9.7 Pretty Printing

Besides being a readable file format, *NestedText* makes a reasonable display format for structured data. You can further simplify the output by stripping leading multiline string tags if you so desire.

```
>>> import nestedtext as nt
>>> import re
>>>
>>> def strip_nestedtext(text):
...     return re.sub(r'^(\s*)[>:]\s?(.*)$', r'\1\2', text, flags=re.M)

>>> addresses = nt.load('examples/address.nt')
>>> print(strip_nestedtext(nt.dumps(addresses['treasurer'], default=repr)))
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
```

1.9.8 Cryptocurrency holdings

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

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

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

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

And finally, the code:

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

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

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

try:
    # read settings
    settings_file = 'cryptocurrency.nt'
    settings_schema = Schema({
        Required('holdings'): All([Coerce(Quantity)], Length(min=1)),
        'currency': str,
        'date format': str,
        'screen width': Coerce(int)
    })
    settings = settings_schema(nt.load(settings_file, top='dict', keymap=(keymap:={})))
    currency = settings.get('currency', 'USD')
    currency_symbol = currency_symbols.get(currency, currency)
    screen_width = settings.get('screen width', 80)

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

    # compute total
    total = Quantity(0, currency_symbol)
    for coin in settings['holdings']:
```

(continues on next page)

(continued from previous page)

```
        price = prices[coin.units]
        value = price.scale(coin)
        total = total.add(value)

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

except nt.NestedTextError as e:
    e.terminate()
except Invalid as e:
    kind = 'key' if 'key' in e.msg else 'value'
    loc = keymap[tuple(e.path)]
    fatal(
        full_stop(e.msg),
        culprit = [settings_file] + e.path,
        codicil=loc.as_line(kind)
    )
except OSError as e:
    fatal(os_error(e))
except KeyboardInterrupt:
    pass
```

1.9.9 PostMortem

This example illustrates how one can implement references in *NestedText*. A reference allows you to define some content once and insert that content multiple places in the document. The example also demonstrates a slightly different way to implement validation and conversion on a per field basis with voluptuous.

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. 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:
```

(continues on next page)

(continued from previous page)

```
        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 a setting 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:

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

import nestedtext as nt
from pathlib import Path
from voluptuous import Schema, Invalid, Extra, Required, REMOVE_EXTRA
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('expected text')

def to_ident(arg):
    arg = to_str(arg)
    if len(arg.split()) > 1:
        raise Invalid('expected simple identifier')
    return arg

def to_list(arg):
    if isinstance(arg, str):
        return arg.split()
    if isinstance(arg, dict):
        raise Invalid('expected list')
    return arg

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

def to_email(arg):
    user, _, host = arg.partition('@')
    if '.' in host:
```

(continues on next page)

(continued from previous page)

```
        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, AssertionError):
            raise Invalid('expected GPG id')

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

# 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[value[1:].strip()]
        return value
    if isinstance(value, dict):
        return {k:expand_settings(v) for k, v in value.items()}
```

(continues on next page)

(continued from previous page)

```
    if isinstance(value, list):
        return [expand_settings(v) for v in value]
    raise NotImplementedError(value)

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

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

        # expand references
        settings = expand_settings(settings)

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

        # show the resulting settings
        pprint(settings)

except nt.NestedTextError as e:
    e.report()
except Invalid as e:
    kind = 'key' if 'key' in e.msg else 'value'
    loc = keymap[tuple(e.path)]
    culprit = '.'.join(str(p) for p in [config_filepath] + e.path)
    print(f"ERROR: {culprit}: {e.msg}.")
    print(loc.as_line(kind))
except OSError as e:
    print(f"ERROR: {config_filepath!s}: {e!s}")
```

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 that the above examples. In those, 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 and allows better control of the error messages.

Here are the processed settings:

```
{'my gpg ids': ['odin@norse-gods.com'],
'name template': '{name}-{now:YYMMDD}',
'recipients': {'frigg': {'attach': [PosixPath('.../home/estate/trust.pdf'),
                                    PosixPath('.../home/estate/will.pdf'),
                                    PosixPath('.../home/estate/deed-valhalla.pdf')],
                         'category': 'wife',
                         'email': ['frigg@norse-gods.com'],
                         'networth': 'odin'},
               'loki': {'attach': [PosixPath('.../home/estate/trust.pdf'),
```

(continues on next page)

(continued from previous page)

```
                                  PosixPath('.../home/estate/will.pdf'),
                                  PosixPath('.../home/estate/deed-valhalla.pdf')],
                      'category': 'kids',
                      'email': ['loki@norse-gods.com']},
              'thor': {'attach': [PosixPath('.../home/estate/trust.pdf'),
                                  PosixPath('.../home/estate/will.pdf'),
                                  PosixPath('.../home/estate/deed-valhalla.pdf')],
                      'category': 'kids',
                      'email': ['thor@norse-gods.com']}},
'sign with': 'odin@norse-gods.com'}
```

1.10 Common mistakes

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.
   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:␣␣
```

(continues on next page)

(continued from previous page)

```
...         > 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.

1.11 Python API

1.11.1 dumps

nestedtext.**dumps**(*obj*, ***, *width=0*, *inline_level=0*, *sort_keys=False*, *indent=4*, *converters=None*, *default=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. If a function is passed in, it is used as the key function.
- **indent** (*int*) – The number of spaces to use to represent a single level of indentation. Must be one or greater.
- **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.
- **_level** (*int*) – The number of indentation levels. When dumps is invoked recursively this is used to increment the level and so the indent. This argument is use internally and should not be specified by the user.

Returns The *NestedText* content.

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

Examples

```
>>> import nestedtext as nt

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

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

The *NestedText* format only supports dictionaries, lists, and strings. By default, *dumps* is configured to be rather forgiving, so it will render many of the base Python data types, such as *None*, *bool*, *int*, *float* and list-like types such as *tuple* and *set* by converting them to the types supported by the format. This implies that a round trip through *dumps* and *loads* could result in the types of values being transformed. You can 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
```

(continues on next page)

(continued from previous page)

```
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))
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
```

(continues on next page)

(continued from previous page)

```
>>> 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 = '$'
```

(continues on next page)

(continued from previous page)

```
...     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.

1.11.2 dump

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

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

Parameters

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

  The name used for the file is arbitrary but it is tradition to use a .nt suffix. 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.

Raises

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

Examples

This example writes to a pointer to an open file.

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

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

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

1.11.3 loads

nestedtext.**loads**(*content*, *top='dict'*, ***, *source=None*, *on_dup=None*, *keymap=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. By default they raise exceptions. Specifying 'ignore' causes them to be ignored (first wins). Specifying 'replace' results in them replacing earlier items (last wins). By specifying a func-

tion, the keys can be de-duplicated. This call-back function returns a new key and takes four arguments:

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

- **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.

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* content.

Examples

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

```
>>> import nestedtext as nt

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

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

>>> print(data)
{'name': 'Kristel Templeton', 'sex': '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 *contents*. 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:
```

(continues on next page)

(continued from previous page)

```
...     print(e.render())
...     print(*e.get_codicil(), sep="\n")
examples/duplicate-keys.nt, 5: duplicate key: name.
   4 «name:»
   5 «name:»
```

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

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

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

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

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

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

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

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

1.11.4 load

nestedtext.**load**(*f=None*, *top='dict'*, ***, *on_dup=None*, *keymap=None*)

Loads *NestedText* from file or stream.

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

Parameters

- **f** (*str, os.PathLike, io.TextIOBase, collections.abc.Iterator*) – The file to read the *NestedText* content from. This can be specified either as a path (e.g. a string or a

pathlib.Path), as a text IO object (e.g. an open file), or as an iterator. If a path is given, the file will be opened, read, and closed. If an IO object is given, it will be read and not closed; utf-8 encoding should be used.. If an iterator is given, it should generate full lines in the same manner that iterating on a file descriptor would.

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

Examples

Load from a path specified as a string:

```
>>> import nestedtext as nt
>>> print(open('examples/groceries.nt').read())
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']}
```

1.11.5 Location

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'*)

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** (*str*) – Specify either 'key' or 'value' depending on which token is desired.

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** (*str*) – Specify either 'key' or 'value' depending on which token is desired.

1.11.6 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 would 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

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

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

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

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

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

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

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

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

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

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

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

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.

Returned: 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.

render(*template=None*)
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.

Returned: The formatted message with any culprits.

1.12 Releases

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

1.12.1 Latest development version

Version: 3.2.0
Released: 2022-01-17

1.12.2 v3.2 (2022-01-17)

- add cyclic reference detection and reporting.

1.12.3 v3.1 (2021-07-23)

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

1.12.4 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.

1.12.5 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()`.

1.12.6 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.

1.12.7 v1.2 (2020-10-31)

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

1.12.8 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.

1.12.9 v1.0 (2020-10-03)

- Production release.

INDEX