NestedText

Release 1.2.0

unknown

Oct 31, 2020

WHY NESTEDTEXT?

Authors: Ken & Kale Kundert
Version: 1.2.0
Released: 2020-10-31
Documentation: nestedtext.org.
Please post all questions, suggestions, and bug reports to: Github.

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

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

```
# Contact information for our officers

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

treasurer:
    -
        name: Fumiko Purvis
        address:
```

(continues on next page)

(continued from previous page)

```
            > 3636 Buffalo Ave
            > Topeka, Kansas 20692
        phone: 1-268-555-0280
        email: fumiko.purvis@hotmail.com
        additional roles:
            - accounting task force
    -
        name: Merrill Eldridge
            # Fumiko's term is ending at the end of the year.
            # She will be replaced by Merrill.
        phone: 1-268-555-3602
        email: merrill.eldridge@yahoo.com
```

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

NestedText is somewhat unique in that the leaf values are always strings. Of course the values start off as strings in the input file, but alternatives like *YAML* or *TOML* aggressively convert those values into the underlying data types such as integers, floats, and Booleans. For example, a value like 2.10 would be converted to a floating point number. But making the decision to do so is based purely on the form of the value, not the context in which it is found. This can lead to misinterpretations. For example, assume that this value is the software version number two point ten. By converting it to a floating point number it becomes two point one, which is wrong. There are many possible versions of this basic issue. But there is also the inverse problem; values that should be converted to particular data types but are not recognized. For example, a value of $2.00 should be converted to a real number but would remain a string instead. There are simply too many values types for a general purpose solution that is only looking at the values themselves to be able to interpret all of them. For example, 12/10/09 is likely a date, but is it in MM/DD/YY, YY/MM/DD or DD/MM/YY form? The fact is, the value alone is often insufficient to reliably determine how to convert values into internal data types. *NestedText* avoids these problems by leaving the values in their original form and allowing the decision to be made by the end application where more context is available to help guide the conversions. If a price is expected for a value, then $2.00 would be checked and converted accordingly. Similarly, local conventions along with the fact that a date is expected for a particular value allows 12/10/09 to be correctly validated and converted. This process of validation and conversion is referred to as applying a schema to the data. There are packages such as Pydantic and Voluptuous available that make this process easy and reliable.

CHAPTER

ONE

RELATED PROJECTS

vim-nestedtext vim syntax files for *NestedText*.

CHAPTER

TWO

CONTRIBUTING

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

2.1 The Zen of *NestedText*

NestedText aspires to be a simple dumb vessel 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 one or two examples and they can use it without without needing to remember any arcane rules and without relying on any of the 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. 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*.

2.2 Alternatives

There are no shortage of well established alternatives to *NestedText* for storing data in a human-readable text file. The features and shortcomings of some of these alternatives are discussed below:

2.2.1 JSON

JSON is a subset of JavaScript suitable for holding data. Like *NestedText*, it consists of a hierarchical collection of dictionaries, lists, and strings, but also allows integers, floats, Booleans and nulls. The fundamental problem with *JSON* in this context is that its meant for serializing and exchanging data between programs; it's not meant for configuration files. Of course, it's used for this purpose anyways, where it has a number of glaring shortcomings.

To begin, it has an excessive amount of syntactic clutter. Dictionary keys and strings both have to be quoted, commas are required between dictionary and list items (but forbidden after the last item), braces are required around dictionaries, etc. Features that would improve clarity are also 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). More conceptually, it is the responsibility of the user to provide data of the correct type (e.g. `32` vs.

`32.0` vs. `"32"`), even though the application already knows what type it expects. All of this results in *JSON* being a frustrating format for humans to read, enter or edit.

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

- text does not require quotes
- data is left in its original form
- comments
- multiline strings
- special characters without escaping them
- commas are not used to separate dictionary and list items

2.2.2 YAML

YAML is considered by many to be a human friendly alternative to *JSON*, but over time it has accumulated too many data types and too many formats. To distinguish between all the various types and formats, a complicated and non-intuitive set of rules developed. *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 like *JSON* it converts data to the underlying data types as appropriate, but unlike with *JSON*, the lack of quoting makes the format ambiguous, which means it has to guess at times, and small seemingly insignificant details can affect the result.

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

- simple
- unambiguous (no implicit typing)
- data is left in its original form
- syntax is insensitive to special characters within text
- safe, no risk of malicious code execution

2.2.3 TOML

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, when it should be the responsibility of the application to convert each field to the correct type.

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.

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

- text does not require quotes

- 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

2.3 Installation

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

2.3.1 Releases

Latest development release

Version: 1.2.0
Released: 2020-10-31

v1.2 (2020-10-31)

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

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.

v1.0 (2020-10-03)

- Production release.

v0.6 (2020-09-26)

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

v0.5 (2020-09-11)

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

v0.4 (2020-09-07)

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

v0.3 (2020-09-03)

- Allow comments to be indented.

v0.2 (2020-09-02)

- Minor enhancements and bug fixes.

v0.1 (2020-08-30)

- Initial release.

2.4 Basic syntax

This is a overview of the syntax of a *NestedText* document, which consists of a *nested collection* of *dictionaries*, *lists*, and *strings*. All leaf values must be simple text. You can find more specifics *later on*.

2.4.1 Dictionaries

A dictionary is an ordered collection of name/value pairs:

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

A dictionary item is introduced by a key followed by a colon at the start of a line. The key is a string and must be quoted if it contains characters that could be misinterpreted. You quote it using either single or double quotes (both have the same meaning). Keys are the only place in *NestedText* where quoting is used to protect text.

The value of a 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 ":␣" 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.

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

2.4.2 Lists

A list is an ordered collection of values:

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

A list item is introduced with a dash at the start of a line. 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.

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

2.4.3 Strings

There are two types of strings: rest-of-line strings and multiline strings. Rest-of-line strings are simply all the remaining characters on the line, including any leading or trailing white space. They can contain any character other than newline:

```
code   : input signed [7:0] level
regex  : [+-]?([0-9]*[.])?[0-9]+\s*\w*
math   : -b + sqrt(b**2 - 4*a*c)
unicode: José and François
```

Multi-line strings are specified on lines prefixed with the greater-than symbol. 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.

2.4.4 Comments

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

```
# this line is ignored
```

2.4.5 Nesting

A value for a dictionary or list item may be a rest-of-line string or it may be a nested dictionary, list or a multiline string. 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
        home: 1-210-555-8470
            # Katheryn prefers that we always call her on her cell phone.
    email: KateMcD@aol.com
    kids:
        - Joanie
        - Terrance

vice president:
    name: Margaret Hodge
    address:
        > 2586 Marigold Land
        > Topeka, Kansas 20697
    phone: 1-470-555-0398
    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 using dictionaries and lists, but the leaf values, the values that are nested most deeply, must all be strings.

2.5 Basic use

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

2.5.1 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, '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', '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'. All but 'any' constrain the data type of the top-level of the *NestedText* content.

2.5.2 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:
...     content = nt.dump(data, 'examples/access.nt')
... except nt.NestedTextError as e:
...     e.terminate()
... except OSError as e:
...     fatal(os_error(e))
```

2.6 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.

2.7 Examples

2.7.1 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
  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',
```

(continues on next page)

(continued from previous page)

```
              'port': 3306,
              'user': 'www'},
 'debug': False,
 'secret_key': 't=)40**y&883y9gdpuw%aiig+wtc033(ui@^1ur72w#zhw3_ch',
 'webmaster_email': 'admin@example.com'}
```

2.7.2 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
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,
})
raw = nt.load('deploy.nt')
config = schema(raw)

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'}
```

2.7.3 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]

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

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

cmdline = docopt(__doc__)
input_filename = cmdline['<filename>']
try:
    indent = int(cmdline['--indent'])
except Exception:
    warn('indent garbled.', culprit=cmdline['--indent'])
    indent = 4

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

    # convert to NestedText
```

(continues on next page)

(continued from previous page)

```
    nestedtext_content = nt.dumps(data, indent=indent) + "\n"

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

The presence of this example should not be taken as a suggestion that *NestedText* is a replacement for *JSON*. Be aware that not all *JSON* data can be converted to *NestedText*, and in the conversion all type information is lost.

2.7.4 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:
```

(continues on next page)

(continued from previous page)

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

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

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


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


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

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

2.7.5 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

from appdirs import user_config_dir
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 = Path(user_config_dir('cc'), 'settings')
    settings_schema = Schema({
        Required('holdings'): All([Coerce(Quantity)], Length(min=1)),
        'currency': str,
        'date format': str,
        'screen width': Coerce(int)
    })
    settings = settings_schema(nt.load(settings_file, top='dict'))
    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(
```

(continues on next page)

(continued from previous page)

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

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

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

except nt.NestedTextError as e:
    e.terminate()
except Invalid as e:
    fatal(full_stop(e.msg), culprit=e.path)
except OSError as e:
    fatal(os_error(e))
except KeyboardInterrupt:
    pass
```

2.7.6 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
```

(continues on next page)

(continued from previous page)

```
    - ~/home/estate/will.pdf
    - ~/home/estate/deed-valhalla.pdf

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

Notice that *estate docs* is defined at the top level. It is not a *PostMortem* setting; it simply defines a value that will be interpolated into 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()}
    if isinstance(value, list):
        return [expand_settings(v) for v in value]
    raise NotImplementedError(value)

try:
```

(continues on next page)

(continued from previous page)

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

        # load from file
        settings = nt.load(config_filepath)

        # 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:
    print(f"ERROR: {', '.join(str(p) for p in e.path)}: {e.msg}")
```

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

2.8 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:␣␣
...         > 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.

2.9 File format

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

Encoding:

A *NestedText* document encoded in UTF-8.

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*, *dict-item*, and *string-item*. Any line that does not fit one of these types is an error.

Comments:

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

Blank lines:

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

Line-type tags:

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

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

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

String items:

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

List items:

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

Dictionary items:

If the line is not a string-item or a list item and it contains a colon followed by either a space (`:␣`) that does not fall within a quoted key or is followed by a newline, the line is considered a *dict-item*. Adjacent dict-items with the same indentation level are combined into a dictionary with their order being retained. Each dict-item consists of a key, the tag (colon), and a value.

A key must be a string and it must not contain a newline. The key must be quoted if it:

1. starts with a *list-item* or *string-item* tag,
2. contains a *dict-item* tag,
3. starts with a quote character, or
4. has leading or trailing spaces or tabs.

A key is quoted by delimiting it with matching single or double quote characters, which are discarded. Unlike traditional programming languages, a quoted key delimited with single quote characters may contain additional single quote characters. Similarly, a quoted key delimited with double quote characters may contain additional double quote characters. Also, backslash is not used as an escape character; backslash has no special meaning anywhere in *NestedText*.

A quoted key starts with the leading quote character and ends when the matching quote character is found along with a trailing colon (there may be white space between the closing quote and the colon). A key is invalid if it contains two or more instances of a quote character separated from `:␣` by zero or more space characters where the quote character in one is a single quote and the quote character in another is the double quote. In this case the key cannot be quoted with either character so that the separator from the key and value can be identified unambiguously.

Values:

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

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

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

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

String values may contain any printing UTF-8 character.

Indentation:

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

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

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

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

Empty document:

A document may be empty. A document is empty if it consists only of comments and blank lines. 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 where all leaf values are strings. All dictionary keys are also strings.

2.10 Python API

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

2.10.1 nestedtext.dumps

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

Recursively convert object to *NestedText* string.

Parameters

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

Returns The *NestedText* content.

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

Examples

```
>>> import nestedtext as nt

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

(continues on next page)

(continued from previous page)

```
>>> 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 and all leaf values must be 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; as do empty dictionaries and lists.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.10.2 nestedtext.dump

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

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

Parameters

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

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

2.10.3 nestedtext.loads

nestedtext.**loads**(*content*, *top='dict'*, ***, *source=None*, *on_dup=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 function, the keys can be de-duplicated. This call-back function returns a new key and takes four arguments:

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

Returns The extracted data. 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 is return of the type specified by top. If top is 'any' None is returned.

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

Examples

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

```
>>> import nestedtext as nt

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

>>> try:
...     data = nt.loads(contents, '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:
...     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'}
```

2.10.4 nestedtext.load

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

Loads *NestedText* from file or stream.

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

Parameters

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

2.10.5 nestedtext.NestedTextError

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

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

The exception provides the following attributes:

source:

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

line:

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

lineno:

The number of the line where the problem was found.

colno:

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

prev_line:

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

template:

The possibly parameterized text used for the error message.

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

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

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

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

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

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

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

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

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

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

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

Exceptions produced by *NestedText* contain a *template* attribute that contains the basic text of the message. You can change this message by overriding the attribute using the *template* argument when using *report*, *terminate*,

or *render*. *render* is like casting the exception to a string except that allows for the passing of arguments. For example, to convert a particular message to Spanish, you could use something like the following.

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

- genindex

INDEX