Text Version

TOML v1.0.0

Tom's Obvious, Minimal Language.

By Tom Preston-Werner, Pradyun Gedam, et al.

Objectives

TOML aims to be a minimal configuration file format that's easy to read due to obvious semantics. TOML is designed to map unambiguously to a hash table. TOML should be easy to parse into data structures in a wide variety of languages.

Spec

- TOML is case-sensitive.
- A TOML file must be a valid UTF-8 encoded Unicode document.
- Whitespace means tab (0x09) or space (0x20).
- Newline means LF (0x0A) or CRLF (0x0D 0x0A).

Comment

A hash symbol marks the rest of the line as a comment, except when inside a string.

```
# This is a full-line comment
key = "value"  # This is a comment at the end of a line
another = "# This is not a comment"
```

The primary building block of a TOML document is the key/value pair.

Keys are on the left of the equals sign and values are on the right. Whitespace is ignored around key names and values. The key, equals sign, and value must be on the same line (though some values can be broken over multiple lines).

```
key = "value"
```

Values must have one of the following types.

- String
- Integer
- Float
- Boolean
- Offset Date-Time
- Local Date-Time
- Local Date
- Local Time
- Array
- Inline Table

Unspecified values are invalid.

```
key = # INVALID
```

There must be a newline (or EOF) after a key/value pair. (See Inline Table for exceptions.)

```
first = "Tom" last = "Preston-Werner" # INVALID
```

Keys

A key may be either bare, quoted, or dotted.

Bare keys may only contain ASCII letters, ASCII digits, underscores, and dashes (`A-Za-z0-9_-`). Note that bare keys are allowed to be composed of only ASCII digits, e.g. `1234` , but are always interpreted as strings.

```toml
key = "value"
bare_key = "value"
bare-key = "value"
1234 = "value"
```

Quoted keys follow the exact same rules as either basic strings or literal strings and allow you to use a much broader set of key names. Best practice is to use bare keys except when absolutely necessary.

```toml
"127.0.0.1" = "value"
"character encoding" = "value"
"ʎǝʞ" = "value"
'key2' = "value"
'quoted "value"' = "value"
```

A bare key must be non-empty, but an empty quoted key is allowed (though discouraged).

```toml
= "no key name"  # INVALID
"" = "blank"     # VALID but discouraged
'' = 'blank'     # VALID but discouraged
```

```
physical.color = "orange"
physical.shape = "round"
site."google.com" = true
```

In JSON land, that would give you the following structure:

```
{
  "name": "Orange",
  "physical": {
    "color": "orange",
    "shape": "round"
  },
  "site": {
    "google.com": true
  }
}
```

For details regarding the tables that dotted keys define, refer to the Table section below.

Whitespace around dot-separated parts is ignored. However, best practice is to not use any extraneous whitespace.

```
fruit.name = "banana"     # this is best practice
fruit. color = "yellow"    # same as fruit.color
fruit . flavor = "banana"   # same as fruit.flavor
```

Indentation is treated as whitespace and ignored.

Defining a key multiple times is invalid.

Note that bare keys and quoted keys are equivalent:

```toml
# THIS WILL NOT WORK
spelling = "favorite"
"spelling" = "favourite"
```

As long as a key hasn't been directly defined, you may still write to it and to names within it.

```toml
# This makes the key "fruit" into a table.
fruit.apple.smooth = true

# So then you can add to the table "fruit" like so:
fruit.orange = 2
```

```toml
# THE FOLLOWING IS INVALID

# This defines the value of fruit.apple to be an integer.
fruit.apple = 1

# But then this treats fruit.apple like it's a table.
# You can't turn an integer into a table.
fruit.apple.smooth = true
```

Defining dotted keys out-of-order is discouraged.

```toml
# VALID BUT DISCOURAGED

apple.type = "fruit"
```

```
apple.color = "red"
orange.color = "orange"
```

```
# RECOMMENDED

apple.type = "fruit"
apple.skin = "thin"
apple.color = "red"

orange.type = "fruit"
orange.skin = "thick"
orange.color = "orange"
```

Since bare keys can be composed of only ASCII integers, it is possible to write dotted keys that look like floats but are 2-part dotted keys. Don't do this unless you have a good reason to (you probably don't).

```
3.14159 = "pi"
```

The above TOML maps to the following JSON.

```
{ "3": { "14159": "pi" } }
```

String

There are four ways to express strings: basic, multi-line basic, literal, and multi-line literal. All strings must contain only valid UTF-8 characters.

```
str = "I'm a string. \"You can quote me\". Name\tJos\u00E9\nLocation
```

For convenience, some popular characters have a compact escape sequence.

```
\b         - backspace       (U+0008)
\t         - tab             (U+0009)
\n         - linefeed        (U+000A)
\f         - form feed       (U+000C)
\r         - carriage return (U+000D)
\"         - quote           (U+0022)
\\         - backslash       (U+005C)
\uXXXX     - unicode         (U+XXXX)
\UXXXXXXXX - unicode         (U+XXXXXXXX)
```

Any Unicode character may be escaped with the `\uXXXX` or `\UXXXXXXXX` forms. The escape codes must be valid Unicode scalar values.

All other escape sequences not listed above are reserved; if they are used, TOML should produce an error.

Sometimes you need to express passages of text (e.g. translation files) or would like to break up a very long string into multiple lines. TOML makes this easy.

Multi-line basic strings are surrounded by three quotation marks on each side and allow newlines. A newline immediately following the opening delimiter will be trimmed. All other whitespace and newline characters remain intact.

```
str1 = """
Roses are red
Violets are blue"""
```

```toml
str2 = "Roses are red\nViolets are blue"

# On a Windows system, it will most likely be equivalent to:
str3 = "Roses are red\r\nViolets are blue"
```

For writing long strings without introducing extraneous whitespace, use a "line ending backslash". When the last non-whitespace character on a line is an unescaped `\`, it will be trimmed along with all whitespace (including newlines) up to the next non-whitespace character or closing delimiter. All of the escape sequences that are valid for basic strings are also valid for multi-line basic strings.

```toml
# The following strings are byte-for-byte equivalent:
str1 = "The quick brown fox jumps over the lazy dog."

str2 = """
The quick brown \


  fox jumps over \
    the lazy dog."""

str3 = """\
       The quick brown \
       fox jumps over \
       the lazy dog.\
       """
```

Any Unicode character may be used except those that must be escaped: backslash and the control characters other than tab, line feed, and carriage return (U+0000 to U+0008, U+000B, U+000C, U+000E to U+001F, U+007F).

```
# str5 = """Here are three quotation marks: """."""  # INVALID
str5 = """Here are three quotation marks: ""\"."""
str6 = """Here are fifteen quotation marks: ""\"""\"""\"""\"""\"."""

# "This," she said, "is just a pointless statement."
str7 = """"This," she said, "is just a pointless statement.""""
```

If you're a frequent specifier of Windows paths or regular expressions, then having to escape backslashes quickly becomes tedious and error-prone. To help, TOML supports literal strings which do not allow escaping at all.

Literal strings are surrounded by single quotes. Like basic strings, they must appear on a single line:

```
# What you see is what you get.
winpath  = 'C:\Users\nodejs\templates'
winpath2 = '\\ServerX\admin$\system32\'
quoted   = 'Tom "Dubs" Preston-Werner'
regex    = '<\i\c*\s*>'
```

Since there is no escaping, there is no way to write a single quote inside a literal string enclosed by single quotes. Luckily, TOML supports a multi-line version of literal strings that solves this problem.

Multi-line literal strings are surrounded by three single quotes on each side and allow newlines. Like literal strings, there is no escaping whatsoever. A newline immediately following the opening delimiter will be trimmed. All other content between the delimiters is interpreted as-is without modification.

```
regex2 = '''I [dw]on't need \d{2} apples'''
lines  = '''
```

You can write 1 or 2 single quotes anywhere within a multi-line literal string, but sequences of three or more single quotes are not permitted.

```
quot15 = '''Here are fifteen quotation marks: """""""""""""""'''

# apos15 = '''Here are fifteen apostrophes: ''''''''''''''''  # IN
apos15 = "Here are fifteen apostrophes: ''''''''''''''"

# 'That,' she said, 'is still pointless.'
str = ''''That,' she said, 'is still pointless.''''
```

Control characters other than tab are not permitted in a literal string. Thus, for binary data, it is recommended that you use Base64 or another suitable ASCII or UTF-8 encoding. The handling of that encoding will be application-specific.

Integer

Integers are whole numbers. Positive numbers may be prefixed with a plus sign. Negative numbers are prefixed with a minus sign.

```
int1 = +99
int2 = 42
int3 = 0
int4 = -17
```

For large numbers, you may use underscores between digits to enhance readability. Each underscore must be surrounded by at least one digit on each side.

Leading zeros are not allowed. Integer values `-0` and `+0` are valid and identical to an unprefixed zero.

Non-negative integer values may also be expressed in hexadecimal, octal, or binary. In these formats, leading `+` is not allowed and leading zeros are allowed (after the prefix). Hex values are case-insensitive. Underscores are allowed between digits (but not between the prefix and the value).

```
# hexadecimal with prefix `0x`
hex1 = 0xDEADBEEF
hex2 = 0xdeadbeef
hex3 = 0xdead_beef

# octal with prefix `0o`
oct1 = 0o01234567
oct2 = 0o755 # useful for Unix file permissions

# binary with prefix `0b`
bin1 = 0b11010110
```

Arbitrary 64-bit signed integers (from −2^63 to 2^63−1) should be accepted and handled losslessly. If an integer cannot be represented losslessly, an error must be thrown.

Float

Floats should be implemented as IEEE 754 binary64 values.

A float consists of an integer part (which follows the same rules as decimal integer values) followed by a fractional part and/or an exponent part. If both a fractional part and exponent part are present, the fractional part must precede the exponent part.

```toml
# exponent
flt4 = 5e+22
flt5 = 1e06
flt6 = -2E-2

# both
flt7 = 6.626e-34
```

A fractional part is a decimal point followed by one or more digits.

An exponent part is an E (upper or lower case) followed by an integer part (which follows the same rules as decimal integer values but may include leading zeros).

The decimal point, if used, must be surrounded by at least one digit on each side.

```toml
# INVALID FLOATS
invalid_float_1 = .7
invalid_float_2 = 7.
invalid_float_3 = 3.e+20
```

Similar to integers, you may use underscores to enhance readability. Each underscore must be surrounded by at least one digit.

```toml
flt8 = 224_617.445_991_228
```

Float values `-0.0` and `+0.0` are valid and should map according to IEEE 754.

Special float values can also be expressed. They are always lowercase.

```toml
# not a number
sf4 = nan  # actual sNaN/qNaN encoding is implementation-specific
sf5 = +nan # same as `nan`
sf6 = -nan # valid, actual encoding is implementation-specific
```

Boolean

Booleans are just the tokens you're used to. Always lowercase.

```toml
bool1 = true
bool2 = false
```

Offset Date-Time

To unambiguously represent a specific instant in time, you may use an RFC 3339 formatted date-time with offset.

```toml
odt1 = 1979-05-27T07:32:00Z
odt2 = 1979-05-27T00:32:00-07:00
odt3 = 1979-05-27T00:32:00.999999-07:00
```

For the sake of readability, you may replace the T delimiter between date and time with a space character (as permitted by RFC 3339 section 5.6).

```toml
odt4 = 1979-05-27 07:32:00Z
```

Local Date-Time

If you omit the offset from an RFC 3339 formatted date-time, it will represent the given date-time without any relation to an offset or timezone. It cannot be converted to an instant in time without additional information. Conversion to an instant, if required, is implementation-specific.

```
ldt1 = 1979-05-27T07:32:00
ldt2 = 1979-05-27T00:32:00.999999
```

Millisecond precision is required. Further precision of fractional seconds is implementation-specific. If the value contains greater precision than the implementation can support, the additional precision must be truncated, not rounded.

Local Date

If you include only the date portion of an RFC 3339 formatted date-time, it will represent that entire day without any relation to an offset or timezone.

```
ld1 = 1979-05-27
```

Local Time

If you include only the time portion of an RFC 3339 formatted date-time, it will represent that time of day without any relation to a specific day or any offset or timezone.

```
lt1 = 07:32:00
```

Array

Arrays are square brackets with values inside. Whitespace is ignored. Elements are separated by commas. Arrays can contain values of the same data types as allowed in key/value pairs. Values of different types may be mixed.

```
integers = [ 1, 2, 3 ]
colors = [ "red", "yellow", "green" ]
nested_arrays_of_ints = [ [ 1, 2 ], [3, 4, 5] ]
nested_mixed_array = [ [ 1, 2 ], ["a", "b", "c"] ]
string_array = [ "all", 'strings', """are the same""", '''type''' ]

# Mixed-type arrays are allowed
numbers = [ 0.1, 0.2, 0.5, 1, 2, 5 ]
contributors = [
  "Foo Bar <foo@example.com>",
  { name = "Baz Qux", email = "bazqux@example.com", url = "https://e
]
```

Arrays can span multiple lines. A terminating comma (also called a trailing comma) is permitted after the last value of the array. Any number of newlines and comments may precede values, commas, and the closing bracket. Indentation between array values and commas is treated as whitespace and ignored.

```
integers2 = [
  1, 2, 3
]

integers3 = [
```

Table

Tables (also known as hash tables or dictionaries) are collections of key/value pairs. They are defined by headers, with square brackets on a line by themselves. You can tell headers apart from arrays because arrays are only ever values.

```
[table]
```

Under that, and until the next header or EOF, are the key/values of that table. Key/value pairs within tables are not guaranteed to be in any specific order.

```
[table-1]
key1 = "some string"
key2 = 123

[table-2]
key1 = "another string"
key2 = 456
```

Naming rules for tables are the same as for keys (see definition of Keys above).

```
[dog."tater.man"]
type.name = "pug"
```

In JSON land, that would give you the following structure:

```
{ "dog": { "tater.man": { "type": { "name": "pug" } } } }
```

```
[ d.e.f ]          # same as [d.e.f]
[ g .  h  . i ]    # same as [g.h.i]
[ j . "ʞ" . 'l' ]  # same as [j."ʞ".'l']
```

Indentation is treated as whitespace and ignored.

You don't need to specify all the super-tables if you don't want to. TOML knows how to do it for you.

```
# [x] you
# [x.y] don't
# [x.y.z] need these
[x.y.z.w] # for this to work

[x] # defining a super-table afterward is ok
```

Empty tables are allowed and simply have no key/value pairs within them.

Like keys, you cannot define a table more than once. Doing so is invalid.

```
# DO NOT DO THIS

[fruit]
apple = "red"

[fruit]
orange = "orange"
```

```
# DO NOT DO THIS EITHER
```

Defining tables out-of-order is discouraged.

```toml
# VALID BUT DISCOURAGED
[fruit.apple]
[animal]
[fruit.orange]
```

```toml
# RECOMMENDED
[fruit.apple]
[fruit.orange]
[animal]
```

The top-level table, also called the root table, starts at the beginning of the document and ends just before the first table header (or EOF). Unlike other tables, it is nameless and cannot be relocated.

```toml
# Top-level table begins.
name = "Fido"
breed = "pug"

# Top-level table ends.
[owner]
name = "Regina Dogman"
member_since = 1999-08-04
```

Dotted keys create and define a table for each key part before the last one, provided that such tables were not previously created.

```
# Defines a table named fruit.apple.taste
# fruit and fruit.apple were already created
```

Since tables cannot be defined more than once, redefining such tables using a `[table]` header is not allowed. Likewise, using dotted keys to redefine tables already defined in `[table]` form is not allowed. The `[table]` form can, however, be used to define sub-tables within tables defined via dotted keys.

```
[fruit]
apple.color = "red"
apple.taste.sweet = true

# [fruit.apple]  # INVALID
# [fruit.apple.taste]  # INVALID

[fruit.apple.texture]  # you can add sub-tables
smooth = true
```

Inline Table

Inline tables provide a more compact syntax for expressing tables. They are especially useful for grouped data that can otherwise quickly become verbose. Inline tables are fully defined within curly braces: `{` and `}`. Within the braces, zero or more comma-separated key/value pairs may appear. Key/value pairs take the same form as key/value pairs in standard tables. All value types are allowed, including inline tables.

Inline tables are intended to appear on a single line. A terminating comma (also called trailing comma) is not permitted after the last key/value pair in an inline table. No newlines are allowed between the curly braces unless they are valid within a value. Even so, it is strongly discouraged

```
point = { x = 1, y = 2 }
animal = { type.name = "pug" }
```

The inline tables above are identical to the following standard table definitions:

```
[name]
first = "Tom"
last = "Preston-Werner"

[point]
x = 1
y = 2

[animal]
type.name = "pug"
```

Inline tables are fully self-contained and define all keys and sub-tables within them. Keys and sub-tables cannot be added outside the braces.

```
[product]
type = { name = "Nail" }
# type.edible = false  # INVALID
```

Similarly, inline tables cannot be used to add keys or sub-tables to an already-defined table.

```
[product]
type.name = "Nail"
# type = { edible = false }  # INVALID
```

Array of Tables

The last syntax that has not yet been described allows writing arrays of tables. These can be expressed by using a header with a name in double brackets. The first instance of that header defines the array and its first table element, and each subsequent instance creates and defines a new table element in that array. The tables are inserted into the array in the order encountered.

```toml
[[products]]
name = "Hammer"
sku = 738594937

[[products]]  # empty table within the array

[[products]]
name = "Nail"
sku = 284758393

color = "gray"
```

In JSON land, that would give you the following structure.

```json
{
  "products": [
    { "name": "Hammer", "sku": 738594937 },
    { },
    { "name": "Nail", "sku": 284758393, "color": "gray" }
  ]
}
```

Any reference to an array of tables points to the most recently defined table element of the array. This allows you to define sub-tables, and even sub-arrays of tables, inside the most

```toml
[fruits.physical]  # subtable
color = "red"
shape = "round"

[[fruits.varieties]]  # nested array of tables
name = "red delicious"

[[fruits.varieties]]
name = "granny smith"


[[fruits]]
name = "banana"

[[fruits.varieties]]
name = "plantain"
```

The above TOML maps to the following JSON.

```json
{
  "fruits": [
    {
      "name": "apple",
      "physical": {
        "color": "red",
        "shape": "round"
      },
      "varieties": [
        { "name": "red delicious" },
        { "name": "granny smith" }
      ]
    },
```

```
    }
  ]
}
```

If the parent of a table or array of tables is an array element, that element must already have been defined before the child can be defined. Attempts to reverse that ordering must produce an error at parse time.

```
# INVALID TOML DOC
[fruit.physical]  # subtable, but to which parent element should it
color = "red"
shape = "round"

[[fruit]]  # parser must throw an error upon discovering that "fruit
           # an array rather than a table
name = "apple"
```

Attempting to append to a statically defined array, even if that array is empty, must produce an error at parse time.

```
# INVALID TOML DOC
fruits = []

[[fruits]] # Not allowed
```

Attempting to define a normal table with the same name as an already established array must produce an error at parse time. Attempting to redefine a normal table as an array must likewise produce a parse-time error.

```toml
[[fruits.varieties]]
name = "red delicious"

# INVALID: This table conflicts with the previous array of tables
[fruits.varieties]
name = "granny smith"

[fruits.physical]
color = "red"
shape = "round"

# INVALID: This array of tables conflicts with the previous table
[[fruits.physical]]
color = "green"
```

You may also use inline tables where appropriate:

```toml
points = [ { x = 1, y = 2, z = 3 },
           { x = 7, y = 8, z = 9 },
           { x = 2, y = 4, z = 8 } ]
```

Filename Extension

TOML files should use the extension `.toml`.

MIME Type

When transferring TOML files over the internet, the appropriate MIME type is `application/toml`.

ABNF Grammar

A formal description of TOML's syntax is available, as a separate ABNF file.