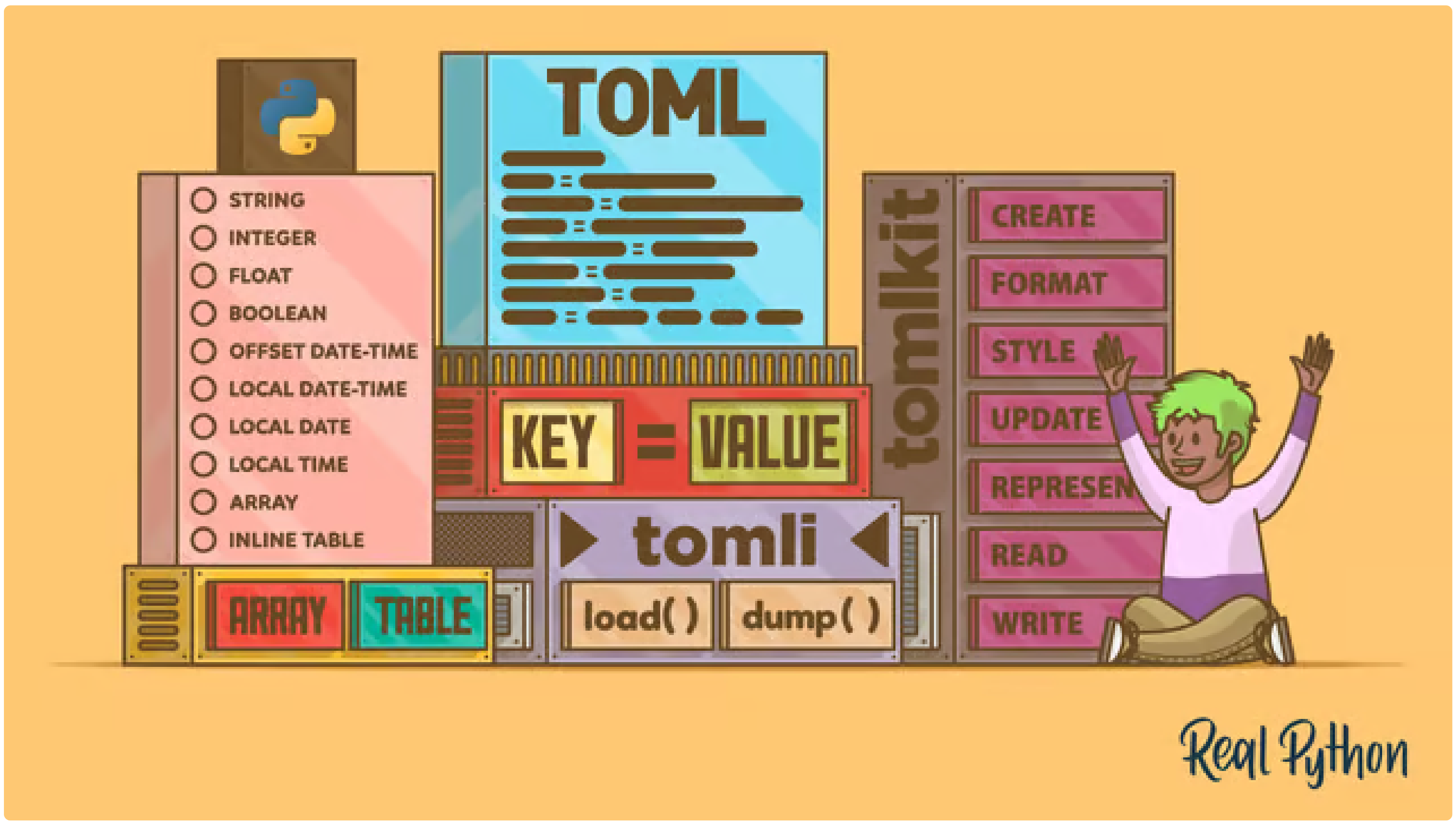


Python and TOML: New Best Friends

by Geir Arne Hjelle

Jul 11, 2022 11 Comments

data-structures intermediate

Mark as Completed

Tweet Share Email

Table of Contents

Master Real-World Python Skills
With a Community of Experts
Level Up With Unlimited Access to Our Vast Library
of Python Tutorials and Video Lessons
Watch Now »

Help

Remove ads

TOML—Tom's Obvious Minimal Language—is a reasonably new configuration file format that the Python community has embraced over the last couple of years. TOML plays an essential part in the Python ecosystem. Many of your favorite tools rely on TOML for configuration, and you'll use `pyproject.toml` when you build and distribute your own packages.

In this tutorial, you'll learn more about TOML and how you can use it. In particular, you'll:

- Learn and understand the **syntax** of TOML
- Use `tomli` and `tomllib` to **parse** TOML documents
- Use `tomli_w` to **write** data structures as TOML
- Use `tomlkit` when you need **more control** over your TOML files

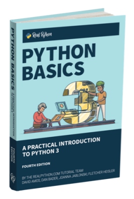
A new module for TOML parsing is being added to Python's standard library in Python 3.11. Later in this tutorial, you'll learn how to use this new module. If you want to know more about why `tomllib` was added to Python, then have a look at the companion tutorial, Python 3.11 Preview: TOML and `tomllib`.

Free Download: Get a sample chapter from Python Tricks: The Book that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

Use TOML as a Configuration Format

TOML is short for **Tom's Obvious Minimal Language** and is humbly named after its creator, Tom Preston-Werner. It was designed expressly to be a **configuration file format** that should be "easy to parse into data structures in a wide variety of languages" (Source).

In this section, you'll start thinking about configuration files and look at what TOML brings to the table.


Your **Practical Introduction to Python 3** »

Remove ads

Configurations and Configuration Files

A configuration is an important part of almost any application or system. It'll allow you to change settings or behavior without changing the source code. Sometimes you'll use a configuration to specify information needed to connect to another service like a database or cloud storage. Other times you'll use configuration settings to allow your users to customize their experience with your project.

Using a configuration file for your project is a good way to separate your code from its settings. It also encourages you to be conscious about which parts of your system are genuinely configurable, giving you a tool to name magic values in your source code. For now, consider this configuration file for a hypothetical tic-tac-toe game:

Config File

```
player_x_color = blue
player_o_color = green
board_size     = 3
server_url     = https://tictactoe.example.com/
```

You could potentially code this directly in your source code. However, by moving the settings into a separate file, you achieve a few things:

- You give explicit **names** to values.
- You provide these values more **visibility**.
- You make it simpler to **change** the values.

Look more closely at your hypothetical configuration file. Those values are conceptually different. The colors are values that your framework probably supports changing. In other words, if you replaced `blue` with `red`, that would be honored without any special handling in your code. You could even consider if it's worth exposing this configuration to your end users through your front end.

However, the board size may or may not be configurable. A tic-tac-toe game is played on a three-by-three grid. It's not certain that your logic would still work for other board sizes. It may still make sense to keep the value in your configuration file, both to give a name to the value and to make it visible.

Finally, the project URL is usually essential when deploying your application. It's not something that a typical user will change, but a power user may want to redeploy your game to a different server.

To be more explicit about these different use cases, you may want to add some organization to your configuration. One popular option is to separate your configuration into additional files, each dealing with a different concern. Another option is to group your configuration values somehow. For example, you can organize your hypothetical configuration file as follows:

Config File

```
[user]
player_x_color = blue
player_o_color = green

[constant]
board_size = 3

[server]
url = https://tictactoe.example.com
```

The organization of the file makes the role of each configuration item clearer. You can also add comments to the configuration file with instructions to anyone thinking about making changes to it.

Note: The actual format of your configuration file isn't important for this discussion. The above principles hold independently of how you specify your configuration values. As it happens, the examples that you've seen so far can be parsed by Python's `ConfigParser` class.

There are many ways for you to specify a configuration. Windows has traditionally used INI files, which resemble your configuration file from above. Unix systems have also relied on plain-text, human-readable configuration files, although the actual format varies between different services.

Over time, more and more applications have come to use well-defined formats like XML, JSON, or YAML for their configuration needs. These formats were designed as data **interchange** or **serialization** formats, usually meant for computer communication.

On the other hand, configuration files are often written or edited by humans. Many developers have gotten frustrated with JSON's strict comma rules when updating their Visual Studio Code settings or with YAML's nested indentations when setting up a cloud service. Despite their ubiquity, these file formats aren't the easiest to write by hand.

TOML: Tom's Obvious Minimal Language

TOML is a fairly new format. The first format specification, version 0.1.0, was released in 2013. From the beginning, it focused on being a minimal configuration file format that's human-readable. According to the TOML web page, TOML's goals are the following:

> 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. (Source, emphasis added)

As you work through this tutorial, you'll see how well TOML hits these targets. It's clear, though, that TOML has gotten quite popular over its short life span. More and more Python tools, including Black, pytest, mypy, and isort, use TOML for their configuration. TOML parsers are available for most popular programming languages.

Recall your configuration from the previous subsection. One way to express it in TOML is the following:

TOML

```toml
[user]
player_x.color = "blue"
player_o.color = "green"

[constant]
board_size = 3

[server]
url = "https://tictactoe.example.com"
```

You'll learn more about the details of the TOML format in the next section. For now, just try to read and parse the information yourself. Note that it's not much different from earlier. The biggest change is the addition of quotation marks (`"`) in some of the values.

TOML's syntax is inspired by traditional configuration files. Its one major advantage over Windows INI files and Unix configuration files is that TOML has a **specification** that spells out precisely what's allowed in a TOML document and how different values should be interpreted. The specification is stable and mature after reaching version 1.0.0 in early 2021.

In contrast, the INI format doesn't have a formal specification. Instead, there are many variants and dialects, most of them defined by an implementation. Python comes bundled with support for reading INI files in the standard library. While `ConfigParser` is quite lenient, it doesn't support all kinds of INI files.

Another difference between TOML and many traditional formats is that TOML values have types. In the example above, `"blue"` is interpreted as a string, while `3` is a number. One potential criticism of TOML is that humans writing TOML need to be aware of types. In simpler formats, that responsibility lies with the programmer parsing the configuration.

TOML is *not* meant to be a data serialization format like JSON or YAML. In other words, you shouldn't try to store general data in TOML to recover it later. TOML is restrictive in a few aspects:

- All keys are interpreted as strings. You can't easily use, say, a number as a key.
- TOML has no null type.
- Some whitespace is important, which makes it less efficient to compress the size of TOML documents.

Even though TOML is a good hammer, not all data files are nails. You should primarily use TOML for configurations.

Real Python

The **Real Python Podcast** »

Remove ads

TOML Schema Validation

You'll dive deeper into TOML syntax in the next section. There you'll learn about some of the syntax requirements of TOML files. However, in practice, a given TOML file may also come with some non-syntactical requirements.

These are **schema requirements**. For example, your tic-tac-toe application may require that the configuration file contain the server URL. On the other hand, the player colors may be optional because the application defines a default color.

Currently, TOML doesn't include a schema language that can specify required and optional fields in a TOML document. Several proposals exist, although it's not clear if any of them will be accepted anytime soon.

In simple applications, you can validate your TOML configuration manually. For example, you can use structural pattern matching, which was introduced in Python 3.10. Assume that you've parsed the configuration into Python and named it `config`. You can then check its structure as follows:

Python

```python
match config:
    case {
        "user": {"player_x": {"color": str()}, "player_o": {"color": str()}},
        "constant": {"board_size": int()},
        "server": {"url": str()},
    }:
        pass
    case _:
        raise ValueError(f"invalid configuration: {config}")
```

The first `case` statement spells out the structure that you expect. If `config` matches, then you use `pass` to continue your code. Otherwise, you raise an error.

This approach may not scale well if your TOML document is more complicated. You also need to do more work if you want to provide good error messages. A better alternative is to use pydantic which utilizes type annotations to do data validation at runtime. One advantage of pydantic is that it has precise and helpful error messages built in.

There are also tools that take advantage of the existing schema validations that exist for formats like JSON. For example, Taplo is a TOML tool kit that can validate TOML documents against JSON schemas. Taplo is also available for Visual Studio Code, bundled into the Even Better TOML extension.

You won't worry about schema validation in the rest of this tutorial. Instead, you'll get more familiar with the TOML syntax and see all the different data types that are available to you. Later on, you'll see examples of how you can interact with TOML in Python, and you'll explore some of the use cases where TOML is a good fit.

Get to Know TOML: Key-Value Pairs

TOML is built around key-value pairs that map nicely to hash table data structures. TOML values have different types. Each value must have one of the following types:

- String
- Integer
- Float
- Boolean
- Offset date-time
- Local date-time
- Local date
- Local time
- Array
- Inline table

Additionally, you can use **tables** and **arrays of tables** as collections that organize several key-value pairs. You'll learn more about all of these—and how you can specify them in TOML—in the rest of this section.

Note: TOML supports comments with the same syntax as Python. A hash symbol (#) marks the rest of the line as a comment. Use comments to make your configuration files easier to understand and use for yourself and your users.

You'll see all the different elements of TOML in this tutorial. However, some details and edge cases will be glossed over. Check out the documentation if you're interested in the fine print.

As noted, **key-value pairs** are your basic building blocks in a TOML document. You specify them with a `<key> = <value>` syntax, where the key is separated from the value with an equal sign. The following is a valid TOML document with one key-value pair:

TOML

```toml
greeting = "Hello, TOML!"
```

In this example, `greeting` is the key, while `"Hello, TOML!"` is the value. Values have types. In this example, the value is a text string. You'll learn about the different value types in the following subsections.

Keys are always interpreted as strings, even if quotation marks don't surround them. Consider the following example:

TOML

```toml
greeting = "Hello, TOML!"
42 = "Life, the universe, and everything"
```

Here, `42` is a valid key, but it's interpreted as a string, not a number. Usually, you want to use **bare keys**. These are keys that consist only of ASCII letters and numbers as well as underscores and dashes. All such keys can be written without quotation marks, as in the examples above.

TOML documents must be encoded in UTF-8 Unicode. This gives you great flexibility when representing your values. Despite the restrictions on bare keys, you can also use Unicode when spelling out your keys. This comes at a cost, though. To use Unicode keys, you must add quotation marks around them:

TOML

```toml
"realpython.com" = "Real Python"
"blåbærsyltetøy" = "blueberry jam"
"Tom Preston-Werner" = "creator"
```

All these keys contain characters that aren't allowed in bare keys: the dot (`.`), Norwegian characters (`å`, `æ`, and `ø`), and a space. You're allowed to use quotation marks around any key, but in general, you want to stick to bare keys that don't use or require quotation marks.

Dots (`.`) play a special role in TOML keys. You can use dots in unquoted keys, but in that case, they'll trigger grouping by splitting the dotted key at each dot. Consider the following example:

TOML

```toml
player_x.symbol = "X"
player_x.color = "purple"
```

Here, you specify two dotted keys. Since they both start with `player_x`, the keys `symbol` and `color` will be grouped together inside a section named `player_x`. You'll learn more about dotted keys when you start exploring tables.

Next, turn your attention to the values. In the next section, you'll learn about the most basic data types in TOML.

Write Cleaner & More Pythonic Code

realpython.com

Remove ads

Strings, Numbers, and Booleans

TOML uses familiar syntax for the basic data types. Coming from Python, you'll recognize strings, integers, floats, and Booleans:

TOML

```toml
string = "Text with quotes"
integer = 42
float = 3.11
boolean = true
```

The immediate difference between TOML and Python is that TOML's Boolean values are lowercase: `true` and `false`.

A TOML **string** should typically use double quotation marks (`"`). Inside strings, you can escape special characters with the help of backslashes: `"\u03c0 is less than four"`. Here, `\u03c0` denotes the Unicode character with codepoint U+03c0, which happens to be the Greek letter π. This string will be interpreted as `"π is less than four"`.

You can also specify TOML strings using single quotation marks ('). Single-quoted strings are called **literal strings** and behave similarly to raw strings in Python. Nothing is escaped and interpreted in a literal string, so `'\u03c0 is the Unicode codepoint of π'` starts with the literal `\u03c0` characters.

Finally, TOML strings can also be specified using **triple quotation marks** (`"""` or `'''`). Triple-quoted strings allow you to write a string over multiple lines, similar to Python multiline strings:

TOML

```toml
partly_zen = """
Flat is better than nested.
Sparse is better than dense.
"""
```

Control characters, including literal newlines, aren't allowed in basic strings. You can use `\n` to represent a newline inside a basic string, though. You must use a multiline string if you want to format your strings over several lines. You can also use triple-quoted literal strings. In addition to being multiline, these are the only way to include a single quotation mark inside a literal string: `'''Use '\u03c0' to represent π'''`.

Note: Be careful with special characters when you create TOML documents inside your Python code, because Python will also interpret the special characters. For example, the following is a valid TOML document:

TOML

```toml
numbers = "one\ntwo\nthree"
```

Here, the value of `numbers` is a string that's split over three lines. You may try to represent the same document in Python as follows:

Python >>>

```python
>>> 'numbers = "one\ntwo\nthree"'
'numbers = "one\ntwo\nthree"'
```

This won't work, because Python parses the `\n` characters and creates an invalid TOML document. You need to keep the special characters away from Python, for example by using raw strings:

Python >>>

```python
>>> r'numbers = "one\ntwo\nthree"'
'numbers = "one\\ntwo\\nthree"'
```

This string represents the same TOML document as your original one.

Numbers in TOML are either integers or floating-point numbers. **Integers** represent whole numbers and are specified as plain, numeric characters. As in Python, you can use underscores to enhance readability:

TOML

```toml
number = 42
negative = -8
large = 60_481_729
```

Floating-point numbers represent decimal numbers and include an integer part, a dot representing the decimal point, and a fractional part. Floats can use scientific notation to represent very small or very large numbers. TOML also supports special float values like infinity and not a number (NaN):

TOML

```toml
number = 3.11
googol = 1e100
mole = 6.22e23
negative_infinity = -inf
not_a_number = nan
```

Note that the TOML specification requires that integers at least are represented as 64-bit signed integers. Python handles arbitrarily large integers, but only integers with up to about 19 digits are guaranteed to work on all TOML implementations.

Note: TOML is a configuration file format, not a programming language. Expressions like `1 + 2` aren't supported, only literal numbers.

Non-negative integer values may also be represented as hexadecimal, octal, or binary values by using a `0x`, `0o`, or `0b` prefix, respectively. For example, `0xffff00` is a hexadecimal representation, and `0b00101010` is a binary representation.

Boolean values are represented as `true` and `false`. These must be lowercase.

TOML also includes several time and date types. Before you explore those, though, you'll see how you can use tables to organize and structure your key-value pairs.

Tables

You've learned that a TOML document consists of one or more key-value pairs. When represented in a programming language, these should be stored in a hash table data structure. In Python, that would be a dictionary or another dictionary-like data structure. To organize your key-value pairs, you can use **tables**.

TOML supports three different ways of specifying tables. You'll see examples of each of these shortly. The end results will be the same, independently of how you represent your tables. Still, the different tables do have slightly different use cases:

- Use regular **tables** with **headers** in most cases.
- Use **dotted key tables** when you need to specify a few key-value pairs that are closely tied to their parent table.
- Use **inline tables** only for very small tables with up to three key-value pairs, where the data makes up a clearly defined entity.

The different table representations are mostly interchangeable. You should default to regular tables and only switch to dotted key tables or inline tables if you think it improves your configuration's readability or clarifies your intent.

How do these different table types look in practice? Start with the regular tables. They're defined by adding a table header above your key-value pairs. A header is a **key** without a value, wrapped inside square brackets (`[]`). The following example, which you encountered earlier, defines three tables:

TOML
```toml
[user]
player_x.color = "blue"
player_o.color = "green"

[constant]
board_size = 3

[server]
url = "https://tictactoe.example.com"
```

The three highlighted lines are the table headers. They specify three tables, named `user`, `constant`, and `server`, respectively. The contents or value of a table are all the key-value pairs listed below the header and above the next header. For example, `constant` and `server` contain one nested key-value pair each.

You can also find **dotted key tables** in the configuration above. Inside `user`, you have the following:

TOML
```toml
[user]
player_x.color = "blue"
player_o.color = "green"
```

The period, or dot (`.`), in the keys creates a table named by the part of the key before the dot. You can also represent the same part of the configuration by nesting regular tables:

TOML

```toml
[user]

    [user.player_x]
    color = "blue"

    [user.player_o]
    color = "green"
```

Indentation isn’t important in TOML. You use it here to represent the nesting of the tables. You can see that the `user` table contains two sub-tables, `player_x` and `player_o`. Each of those sub-tables contains one key-value pair.

Note: You can nest TOML tables arbitraily deep. For example, a key or table header like `player.x.color.name` represents `name` within a `color` table within an `x` table within a `player` table.

Note that you need to use a dotted key in the header of nested tables and name all the intermediate tables. This makes TOML header specifications quite verbose. In a similar specification in, for example, JSON or YAML, you’d only specify the sub-table name without repeating the names of the outer tables. At the same time, this makes TOML very explicit, and it’s harder to lose track inside deeply nested structures.

Now, you’ll expand a bit on the `user` table by including a **label** or **symbol** for each player as well. You’ll represent this table in three different forms, first using only regular tables, then using dotted key tables, and finally using inline tables. You haven’t seen the latter yet, so this will be an introduction to inline tables and how those are represented.

Start with nested, regular tables:

TOML

```toml
[user]

    [user.player_x]
    symbol = "X"
    color = "blue"

    [user.player_o]
    symbol = "O"
    color = "green"
```

This representation makes it very clear that you have two different player tables. You don’t need to explicitly define tables that only contain sub-tables and not any regular keys. In the previous example, you could remove the line `[user]`.

Compare the nested tables with the dotted key configuration:

TOML

```toml
[user]
player_x.symbol = "X"
player_x.color = "blue"
player_o.symbol = "O"
player_o.color = "green"
```

This is shorter and more concise than the nested tables above. However, the structure is now less clear, and you need to expend some effort on parsing the keys before you realize that there are two player tables nested within `user`. The dotted key tables are more beneficial when you have a few nested tables with one key each, like in the earlier example with only `color` sub-keys.

Next, you’ll represent `user` with **inline tables**:

TOML

```toml
[user]
player_x = { symbol = "X", color = "blue" }
player_o = { symbol = "O", color = "green" }
```

An inline table is defined with curly braces ({}) wrapped around comma-separated key-value pairs. In this example, the inline table brings a nice balance of readability and compactness, as the grouping of the player tables becomes clear.

Still, you should use inline tables sparingly, and mostly in cases like this where a table represents a small and well-defined entity, like a player. Inline tables are intentionally limited compared to regular tables. In particular, an inline table must be written on one line in the TOML file, and you can't use conveniences like trailing commas.

To wrap up your tour of tables in TOML, you'll have a brief look at a few minor points. In general, you can define your tables in any order, and you should strive to order your configuration in a way that makes sense to your users.

A TOML document is represented by a nameless root table that contains all other tables and key-value pairs. Key-value pairs that you write at the top of your TOML configuration, before any table header, are stored directly in the root table:

TOML

```toml
title = "Tic-Tac-Toe"

[constant]
board_size = 3
```

In this example, `title` is a key in the root table, `constant` is a table that's nested within the root table, and `board_size` is a key in the `constant` table.

Be aware that a table includes all key-value pairs written between its header and the next table header. In practice, this means that you must define nested sub-tables below the key-value pairs belonging to the table. Consider this document:

TOML

```toml
[user]

    [user.player_x]
    color = "blue"

    [user.player_o]
    color = "green"

background_color = "white"
```

The indentation suggests that `background_color` is supposed to be a key in the `user` table. However, TOML ignores the indentation and checks only the table headers. In this example, `background_color` is part of the `user.player_o` table. To correct this, `background_color` should be defined before the nested tables:

TOML

```toml
[user]
background_color = "white"

    [user.player_x]
    color = "blue"

    [user.player_o]
    color = "green"
```

In this case, `background_color` is a key in `user`, as intended. If you use a dotted key table, then you can more freely use any order of the keys:

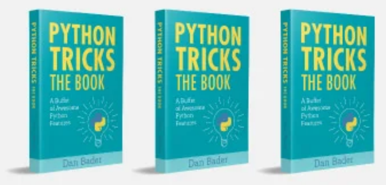
TOML

```toml
[user]
player_x.color = "blue"
player_o.color = "green"
background_color = "white"
```

Now there are no explicit table headers except for `[user]`, so `background_color` will be a key in the `user` table.

You've learned about the basic data types in TOML and how you can use tables to organize your data. In the next subsections, you'll see the final data types that you can play with in your TOML documents.

Python Tricks The Book
A Buffet of Awesome Python Features
Get Your Free Sample Chapter



Remove ads

Times and Dates

TOML supports defining times and dates directly in your documents. You can choose between four different representations, each with its own specific use case:

- An **offset date-time** is a timestamp with time zone information, representing a specific instant in time.
- A **local date-time** is a timestamp without time zone information.
- A **local date** is a date without any time zone information. You typically use this to represent a full day.
- A **local time** is a time with any date or time zone information. You use a local time to represent a time of day.

TOML bases its representation of times and dates on RFC 3339. This document defines a time and date format that is commonly used to represent timestamps on the Internet. A fully defined timestamp would look something like this: `2021-01-12T01:23:45.654321+01:00`. The timestamp is composed of several fields, split by different separators:

Field	Example	Details
Year	`2021`	
Month	`01`	Two digits from `01` (January) to `12` (December)
Day	`12`	Two digits, zero padded when below ten
Hour	`01`	Two digits from `00` to `23`
Minute	`23`	Two digits from `00` to `59`
Second	`45`	Two digits from `00` to `59`
Microsecond	`654321`	Six digits from `000000` to `999999`
Offset	`+01:00`	Time zone as offset from UTC, with `Z` representing UTC

An offset date-time is a timestamp that includes the offset information. A local date-time is a timestamp that doesn't include this. Local timestamps are also called naive timestamps.

In TOML, the microsecond field is optional for all date-time and time types. You're also allowed to replace the `T` that separates the date and time with a space. Here, you can see examples of each of the timestamp-related types:

TOML
```toml
offset_date-time     = 2021-01-12 01:23:45+01:00
offset_date-time_utc = 2021-01-12 00:23:45Z
local_date-time      = 2021-01-12 01:23:45
local_date           = 2021-01-12
local_time           = 01:23:45
local_time_with_us   = 01:23:45.654321
```

Note that you can't wrap the timestamp values within quotation marks, since that would turn them into text strings.

These different time and date types give you a fair amount of flexibility. If you have use cases that aren't covered by these—for example, if you want to specify a time interval like `1 day`—then you can use strings and use your application to properly process those.

The final data type that TOML supports is arrays. These allow you to combine several other values in a list. Read on to learn more.

Arrays

TOML arrays represent an ordered list of values. You specify them using square brackets (`[]`), so that they resemble Python's lists:

TOML

```toml
packages = ["tomllib", "tomli", "tomli_w", "tomlkit"]
```

In this example, the value of `packages` is an array containing four string elements: `"tomllib"`, `"tomli"`, `"tomli_w"`, and `"tomlkit"`.

> **Note:** You'll learn more about `tomllib`, `tomli`, `tomli_w`, and `tomlkit`, as well as the roles they play in Python's TOML landscape, in the practial sections later in the tutorial.

You can use any TOML data type, including other arrays, inside arrays, and one array can contain different data types. You're allowed to specify an array over several lines, and you can use a trailing comma after the last element in the array. All the following examples are valid TOML arrays:

TOML

```toml
potpourri = ["flower", 1749, { symbol = "X", color = "blue" }, 1994-02-14]
skiers = ["Thomas", "Bjørn", "Mika"]
players = [
    { symbol = "X", color = "blue", ai = true },
    { symbol = "O", color = "green", ai = false },
]
```

This defines three arrays. `potpourri` is an array with four elements with different data types, while `skiers` is an array containing three strings. The final array, `players`, adapts your earlier example to represent two inline tables as elements in an array. Note that `players` is defined over four lines and that there's an optional comma after the last inline table.

The last example shows one way that you can create arrays of tables. You can put inline tables inside square brackets. However, as you saw previously, inline tables don't scale well. If you want to represent an array of tables where the tables are bigger, you should use a different syntax.

In general, you should express an **array of tables** by writing table headers inside double square brackets (`[[]]`). The syntax isn't necessarily pretty, but it's quite effective. You can represent `players` from the example above as follows:

TOML

```toml
[[players]]
symbol = "X"
color = "blue"
ai = true

[[players]]
symbol = "O"
color = "green"
ai = false
```

This array of tables is equivalent to the array of inline tables that you wrote above. The double square brackets define an array of tables instead of a regular table. You need to repeat the array name for each nested table inside the array.

For a more extensive example, consider the following excerpt from a TOML document listing questions for a quiz application:

TOML

```toml
[python]
label = "Python"

[[python.questions]]
question = "Which built-in function can get information from the user"
answers = ["input"]
alternatives = ["get", "print", "write"]

[[python.questions]]
question = "What's the purpose of the built-in zip() function"
answers = ["To iterate over two or more sequences at the same time"]
alternatives = [
    "To combine several strings into one",
    "To compress several files into one archive",
    "To get information from the user",
]
```

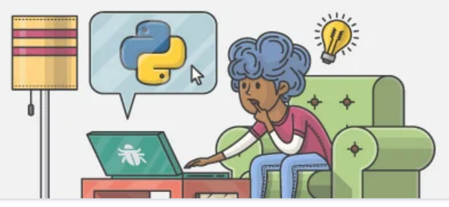
In this example, the `python` table has two keys, `label` and `questions`. The value of `questions` is an array of tables with two elements. Each element is a table with three keys: `question`, `answers`, and `alternatives`.

You've now seen all the data types that TOML has to offer. In addition to simple data types like strings, numbers, Booleans, times, and dates, you can also combine and organize your keys and values with tables and arrays. There are some details and edge cases that you've glossed over in this overview. You can learn all the details in the TOML specification.

In the upcoming sections, you'll get more practical as you learn how you can use TOML in Python. You'll learn about how you can read and write TOML documents and explore how you can organize your applications to use configuration files effectively.

Learn Python Programming, By Example

realpython.com



Remove ads

Load TOML With Python

It's time to get your hands dirty. In this section, you'll fire up your Python interpreter and load TOML documents into Python. You've seen how the main use case for the TOML format is configuration files. These are often written by hand, so in this section, you'll look at how you can read such configuration files with Python and work with them inside your project.

Read TOML Documents With `tomli` and `tomllib`

Since the TOML specification first arrived in 2013, there have been several packages available for working with the format. Over time, some of these packages have become unmaintained. Some libraries that used to be popular are no longer compliant with the latest version of TOML.

In this section, you'll work with a relatively new package called `tomli` and its sibling `tomllib`. These are great libraries when you only want to load a TOML document into Python. You'll also explore `tomlkit` in a future section. That package brings more advanced functionality to the table, and opens up some new use cases for you.

Note: TOML support is added to the Python standard library in Python 3.11. The new `tomllib` module can help you read and parse TOML documents. See Python 3.11 Preview: TOML and `tomllib` for details about the motivation and reasoning for adding the library.

This new `tomllib` module was essentially created by copying the existing `tomli` library into the CPython codebase. The upshot of this is that you can use `tomli` as a compatible backport on Python versions 3.7, 3.8, 3.9, and 3.10.

By following the instructions below, you'll learn how to use `tomli`. If you're using Python 3.11, then you can skip the installation of `tomli` and replace any code that mentions `tomli` with `tomllib`.

It's time to explore how you can read TOML files. Start by creating the following TOML file and saving it as `tic_tac_toe.toml`:

TOML

```toml
# tic_tac_toe.toml

[user]
player_x.color = "blue"
player_o.color = "green"

[constant]
board_size = 3

[server]
url = "https://tictactoe.example.com"
```

This is the same configuration that you worked with in the previous section. Next, use `pip` to install `tomli` into your virtual environment:

Shell

```shell
(venv) $ python -m pip install tomli
```

The `tomli` module only exposes two functions: `load()` and `loads()`. You use these to load a TOML document from a file object and from a string, respectively. Start by using `load()` to read the file that you created above:

Python >>>

```python
>>> import tomli
>>> with open("tic_tac_toe.toml", mode="rb") as fp:
...     config = tomli.load(fp)
...
```

You first open the file, using a context manager to handle any issues that may show up. Importantly, you need to open the file in binary mode by specifying `mode="rb"`. This allows `tomli` to correctly handle the encoding of your TOML file.

You stored the TOML configuration in a variable named `config`. Go ahead and explore its contents:

Python >>>

```python
>>> config
{'user': {'player_x': {'color': 'blue'}, 'player_o': {'color': 'green'}},
 'constant': {'board_size': 3},
 'server': {'url': 'https://tictactoe.example.com'}}

>>> config["user"]["player_o"]
{'color': 'green'}

>>> config["server"]["url"]
'https://tictactoe.example.com'
```

The TOML document is represented as a dictionary in Python. All the tables and sub-tables in the TOML file show up as nested dictionaries in `config`. You can pick out individual values by following the keys into the nested dictionary.

If you already have the TOML document represented as a string, then you can use `loads()` instead of `load()`. Think of the trailing `s` in the function name as a mnemonic for *string*. The following example parses the TOML document stored as `toml_str`:

Python >>>

```
>>> import tomli
>>> toml_str = """
... offset_date-time_utc = 2021-01-12 00:23:45Z
... potpourri = ["flower", 1749, { symbol = "X", color = "blue" }, 1994-02-14]
... """

>>> tomli.loads(toml_str)
{'offset_date-time_utc': datetime.datetime(2021, 1, 12, 0, 23, 45,
                                           tzinfo=datetime.timezone.utc),
 'potpourri': ['flower',
               1749,
               {'symbol': 'X', 'color': 'blue'},
               datetime.date(1994, 2, 14)]}
```

Again, you'll produce a dictionary with keys and values corresponding to the key-value pairs in the TOML document. Note that the TOML time and date types are represented by Python's `datetime` types, and the TOML array is turned into a Python list. You can see that time zone information, expressed in the `.tzinfo` attribute, is attached to `offset_date-time_utc`, as expected.

> **Note:** An offset date-time is a date-time with a specified time zone. Adding the time zone to a `datetime` means that you provide enough information to describe an exact moment in time, which is important in many applications that handle real-world data.
>
> Have a look at Python 3.9: Cool New Features for You to Try to read more about how Python handles time zones and check out the `zoneinfo` module that was added in version 3.9 of Python.

Both `load()` and `loads()` convert TOML documents to Python dictionaries, and you can use them interchangably. Pick the one that's most convenient for your use case. As a final example, you'll combine `loads()` with `pathlib` to reconstruct the tic-tac-toe configuration example:

Python >>>

```
>>> from pathlib import Path
>>> import tomli
>>> tomli.loads(Path("tic_tac_toe.toml").read_text(encoding="utf-8"))
{'user': {'player_x': {'color': 'blue'}, 'player_o': {'color': 'green'}},
 'constant': {'board_size': 3},
 'server': {'url': 'https://tictactoe.example.com'}}
```

One difference between `load()` and `loads()` is that you use regular strings and not bytes when you use the latter. In this case, `tomli` assumes that you've correctly handled the encoding.

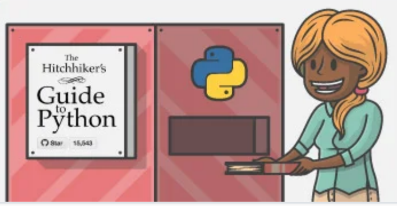
> **Note:** These examples have used `tomli`. However, as noted above, you can replace any code mentioning `tomli` with `tomllib` if you're using Python 3.11 or a newer version.
>
> You may want to automate this decision in your applications. You can do so by adding the following line to your `requirements.txt` dependency specification:
>
> Python Requirements
>
> ```
> tomli >= 1.1.0 ; python_version < "3.11"
> ```
>
> This will make sure that `tomli` is only installed on Python versions prior to 3.11. Additionally, you should replace your imports of `tomli` with a slightly more complicated incantation:
>
> Python
>
> ```
> try:
>     import tomllib
> except ModuleNotFoundError:
>     import tomli as tomllib
> ```

This code will first try to import `tomllib`. If that fails, then it'll import `tomli` instead but alias the `tomli` module to the `tomllib` name. Since the two libraries are compatible, you can now refer to `tomllib` in your code and it'll work on all Python versions 3.7 and up.

You've gotten started and have loaded and parsed your first TOML documents in Python. In the next subsection, you'll look a bit closer at the correspondence between TOML data types and the output from `tomli`.

A Python Best Practices Handbook

python-guide.org



Remove ads

Compare TOML Types and Python Types

In the previous subsection, you loaded some TOML documents and learned how `tomli` and `tomllib` represent, for example, a TOML string as a Python string and a TOML array as a Python list. The TOML specification doesn't explicitly define how Python should represent TOML objects, as that's outside of its scope. However, the TOML specification mentions some requirements on its own types. For example:

- A TOML file must be a valid UTF-8 encoded Unicode document.
- Arbitrary 64-bit signed integers (from −2^63 to 2^63−1) should be accepted and handled losslessly.
- Floats should be implemented as IEEE 754 binary64 values.

In general, TOML's requirements match well with Python's implementation of the corresponding types. Python usually defaults to using UTF-8 when handling files, and a Python `float` follows IEEE 754. Python's `int` class implements arbitrary-precision integers, which handle the required range and much larger numbers as well.

For basic libraries like `tomli` and `tomllib`, the mapping between TOML's data types and Python's data types is quite natural. You can find the following conversion table in the documentation of `tomllib`:

TOML	**Python**
string	`str`
integer	`int`
float	`float`
boolean	`bool`
table	`dict`
offset date-time	`datetime.datetime` (`.tzinfo` is an instance of `datetime.timezone`)
local date-time	`datetime.datetime` (`.tzinfo` is `None`)
local date	`datetime.date`
local time	`datetime.time`
array	`list`

All the Python data types are either built in or part of `datetime` in the standard library. To reiterate, it's not a requirement that TOML types must map to native Python types. This is a convenience `tomli` and `tomllib` have chosen to implement.

Using only standard types is also a limitation. In practice, you can then only represent values and not other information encoded in the TOML document like comments or indentation. Your Python representation also doesn't differentiate between values defined inside a regular table or an inline table.

In many use cases, this metainformation is irrelevant, so nothing is lost. However, sometimes it's important. For example, if you're trying to insert a table into an existing TOML file, then you don't want all the comments to disappear. You'll learn about `tomlkit` later. This library represents TOML types as custom Python objects that retain the information necessary to restore the complete TOML document.

The `load()` and `loads()` functions have one parameter that you can use to customize the TOML parsing. You can supply an argument to `parse_float` to specify how floating-point numbers should be parsed. The default implementation fulfills the requirement of using 64-bit floats, which will usually be precise to about 16 significant digits.

If you have an application that relies on very precise numbers, 16 digits may not be enough, though. As an example, consider the concept of Julian days used in astronomy. This is a representation of a timestamp as a number counting the number of days since the beginning of the Julian period, which is more than 6700 years ago. For example, noon UTC on July 11, 2022, is Julian day 2,459,772.

Astronomers sometimes need to work with very small timescales, like nanoseconds or even picoseconds. To represent a time of day to nanosecond precision, you'd need about 14 digits after the decimal point in a fractional number. For example, 2:01pm UTC on July 11, 2022, represented as a Julian date with nanosecond precision is 2459772.084027777777778.

Numbers like this, which are both large in value and precise to many decimal places, aren't well represented as floats. How much precision do you lose if you read this Julian date with `tomli`? Open a REPL and try it out:

Python >>>

```python
>>> import tomli
>>> ts = tomli.loads("ts = 2_459_772.084027777777778")["ts"]
>>> ts
2459772.084027778

>>> seconds = (ts - int(ts)) * 86_400
>>> seconds
7260.000009834766

>>> seconds - 7260
9.834766387939453e-06
```

You first use `tomli` to parse your Julian date, pick out the value, and name it `ts`. You can see that the value of `ts` has been truncated by several decimal places. To figure out how bad the effect of the truncation is, you calculate the number of seconds represented by the fractional part of `ts` and compare it to 7260.

An integer Julian date represents noon on some day. 2:01pm is two hours and one minute after noon, and two hours and one minute equals 7260 seconds, so `seconds - 7260` shows you how big of an error is introduced by your parsing.

In this case, your timestamp is about ten microseconds off the mark. That may not sound like much, but in many astronomical applications, signals travel at the speed of light. In that case, ten microseconds might cause an error of about three kilometers!

One common solution to this issue is to not store very precise timestamps as Julian dates. Instead, many variants with more inherent precision exist. However, you can also fix your example by using Python's `Decimal` class, which provides arbitrary-precision decimal numbers.

Go back to your REPL and redo the example from above:

Python >>>

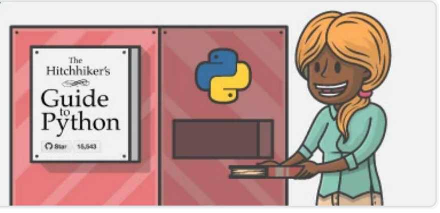
```python
>>> import tomli
>>> from decimal import Decimal
>>> ts = tomli.loads(
...     "ts = 2_459_772.084027777777778",
...     parse_float=Decimal,
... )["ts"]
>>> ts
Decimal('2459772.084027777777778')

>>> seconds = (ts - int(ts)) * 86_400
>>> seconds
Decimal('7260.000000000019200')

>>> seconds - 7260
Decimal('1.9200E-11')
```

Now, the small error that's left comes from your original representation and is about nineteen picoseconds, which translates to subcentimeter errors at the speed of light.

You can use `Decimal` when you know that you require precise floating-point numbers. In more specific use cases, you may also store your data as strings and parse the strings in your application after you've read the TOML file.

So far, you've seen how you can read TOML files with Python. Next, you'll address how you can incorporate a configuration file into your own projects.

Your Guide to the Python Programming Language and a Best Practices Handbook
python-guide.org



Remove ads

Use Configuration Files in Your Projects

You have a project with some settings that you want to extract into a configuration file. Recall that there are several ways that a configuration can improve your codebase:

- It **names** values and concepts.
- It provides more **visibility** for specific values.
- It makes values simpler to **change**.

A configuration file can help you get an overview of your source code and add flexibility to how your users interact with your application. You know how to read a TOML-based configuration file, but how do you use it in your project?

In particular, how do you make sure that the configuration file is only **parsed once**, and how do you access the configuration **from different modules**?

It turns out that Python's import system already supports both of these features out of the box. When you import a module, it's cached for later use. In other words, if you wrap your configuration in a module, you know that the configuration will only be read one time, even if you import that module from several places.

It's time for a concrete example. Recall the `tic_tac_toe.toml` configuration file from earlier:

TOML

```toml
# tic_tac_toe.toml

[user]
player_x.color = "blue"
player_o.color = "green"

[constant]
board_size = 3

[server]
url = "https://tictactoe.example.com"
```

Create a directory named `config/` and save `tic_tac_toe.toml` inside that directory. Additionally, create an empty file named `__init__.py` inside `config/`. Your small directory structure should look like this:

```
config/
├── __init__.py
└── tic_tac_toe.toml
```

Files named `__init__.py` play a special role in Python. They mark the containing directory as a package. Additionally, names defined inside `__init__.py` are exposed through the package. You'll shortly see what this means in practice.

Now, add code to `__init__.py` to read the configuration file:

Python

```python
# __init__.py

import pathlib
import tomli

path = pathlib.Path(__file__).parent / "tic_tac_toe.toml"
with path.open(mode="rb") as fp:
    tic_tac_toe = tomli.load(fp)
```

You read the TOML file using `load()`, as earlier, and store the TOML data to the name `tic_tac_toe`. You use `pathlib` and the special `__file__` variable to set up `path`, the full path to the TOML file. In practice, this specifies that the TOML file is stored in the same directory as the `__init__.py` file.

Try out your little package by starting a REPL session from the parent directory of `config/`:

Python >>>

```python
>>> import config
>>> config.path
PosixPath('/home/realpython/config/tic_tac_toe.toml')

>>> config.tic_tac_toe
{'user': {'player_x': {'color': 'blue'}, 'player_o': {'color': 'green'}},
 'constant': {'board_size': 3},
 'server': {'url': 'https://tictactoe.example.com'}}
```

You can check the path of the configuration and access the configuration itself. To read particular values, you can use regular item access:

Python >>>

```
>>> config.tic_tac_toe["server"]["url"]
'https://tictactoe.example.com'

>>> config.tic_tac_toe["constant"]["board_size"]
3

>>> config.tic_tac_toe["user"]["player_o"]
{'color': 'green'}

>>> config.tic_tac_toe["user"]["player_o"]["color"]
'green'
```

You can now integrate a configuration into your existing projects by copying the `config/` directory into your project and replacing the tic-tac-toe configuration with your own settings.

Inside your code files, you may want to alias the configuration import to make it more convenient to access your settings:

Python >>>

```
>>> from config import tic_tac_toe as CFG

>>> CFG["user"]["player_x"]["color"]
'blue'
```

Here you name the configuration `CFG` during import, which makes it both efficient and readable to access configuration settings.

This recipe gives you a quick and reliable way of working with a configuration in your own projects.

A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You
pythonistacafe.com

Remove ads

Dump Python Objects as TOML

You now know how you can read TOML files with Python. How can you do the opposite? TOML documents are often written by hand because they're mainly used as configurations. Still, sometimes you may need to convert a nested dictionary into a TOML document.

In this section, you'll start by coding up a basic TOML writer by hand. Then, you'll have a look at which tools are already available, and use the third-party `tomli_w` library to dump your data to TOML.

Convert Dictionaries to TOML

Recall the tic-tac-toe configuration that you were working with earlier. You can represent a slightly changed version of it as a nested Python dictionary:

Python

```
{
    "user": {
        "player_x": {"symbol": "X", "color": "blue", "ai": True},
        "player_o": {"symbol": "O", "color": "green", "ai": False},
        "ai_skill": 0.85,
    },
    "board_size": 3,
    "server": {"url": "https://tictactoe.example.com"},
}
```

In this subsection, you'll write a simplified TOML writer that's able to write this dictionary as a TOML document. You won't implement all the features of TOML. In particular, you're leaving out some value types like times, dates, and arrays of tables. You're also not handling keys that need to be quoted or multiline strings.

Still, your implementation will handle many of the typical use cases for TOML. In the next subsection, you'll see how you can use a library that also handles the rest of the specification. Open your editor and create a new file that you'll call `to_toml.py`.

First, code up a helper function named `_dumps_value()`. This function will take some value and return its TOML representation based on the value type. You can do this with `isinstance()` checks:

Python

```python
# to_toml.py

def _dumps_value(value):
    if isinstance(value, bool):
        return "true" if value else "false"
    elif isinstance(value, (int, float)):
        return str(value)
    elif isinstance(value, str):
        return f'"{value}"'
    elif isinstance(value, list):
        return f"[{', '.join(_dumps_value(v) for v in value)}]"
    else:
        raise TypeError(f"{type(value).__name__} {value!r} is not supported")
```

You return `true` or `false` for Boolean values and add double quotation marks around strings. If your value is a list, you create a TOML array by calling `_dumps_value()` recursively. If you're using Python 3.10 or newer, then you can replace your `isinstance()` checks with a `match` … `case` statement.

Next, you'll add the code that handles the tables. Your main function loops over a dictionary and converts each item into a key-value pair. If the value happens to be a dictionary, then you'll add a table header and fill out the table recursively:

Python

```python
# to_toml.py

# ...

def dumps(toml_dict, table=""):
    toml = []
    for key, value in toml_dict.items():
        if isinstance(value, dict):
            table_key = f"{table}.{key}" if table else key
            toml.append(f"\n[{table_key}]\n{dumps(value, table_key)}")
        else:
            toml.append(f"{key} = {_dumps_value(value)}")
    return "\n".join(toml)
```

For convenience, you use a list that keeps track of each table or key-value pair as you add it. You convert this list to a string just before you return it.

In addition to the limitations mentioned earlier, there's one subtle bug hiding in this function. Consider what happens if you try to dump the example from earlier:

Python

```python
>>> import to_toml
>>> config = {
...     "user": {
...         "player_x": {"symbol": "X", "color": "blue", "ai": True},
...         "player_o": {"symbol": "O", "color": "green", "ai": False},
...         "ai_skill": 0.85,
...     },
...     "board_size": 3,
...     "server": {"url": "https://tictactoe.example.com"},
... }

>>> print(to_toml.dumps(config))

[user]

[user.player_x]
symbol = "X"
color = "blue"
ai = true

[user.player_o]
symbol = "O"
color = "green"
ai = false
ai_skill = 0.85
board_size = 3

[server]
url = "https://tictactoe.example.com"
```

Pay special attention to the highlighted lines. It looks like `ai_skill` and `board_size` are keys in the `user.player_o` table. But according to the original data, they should be members of the `user` and the root tables, respectively.

The issue is that there's no way to mark the end of a TOML table. Instead, regular keys must be listed before any sub-tables. One way to fix your code is to sort your dictionary items so that dictionary values come after all other values. Update your function as follows:

Python

```python
# to_toml.py

# ...

def dumps(toml_dict, table=""):
    def tables_at_end(item):
        _, value = item
        return isinstance(value, dict)

    toml = []
    for key, value in sorted(toml_dict.items(), key=tables_at_end):
        if isinstance(value, dict):
            table_key = f"{table}.{key}" if table else key
            toml.append(f"\n[{table_key}]\n{dumps(value, table_key)}")
        else:
            toml.append(f"{key} = {_dumps_value(value)}")
    return "\n".join(toml)
```

In practice, `tables_at_end()` returns `False` or `0` for all non-dictionary values and `True`, which is equivalent to `1`, for all dictionary values. Using this as a sorting key ensures that nested dictionaries are handled after other kinds of values.

You can now redo the example from above. When you print the result to your terminal screen, you'll see the following TOML document:

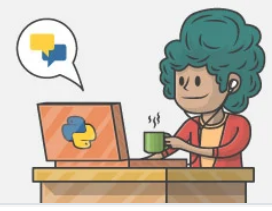
TOML

```toml
board_size = 3

[user]
ai_skill = 0.85

[user.player_x]
symbol = "X"
color = "blue"
ai = true

[user.player_o]
symbol = "O"
color = "green"
ai = false

[server]
url = "https://tictactoe.example.com"
```

Here, `board_size` is listed at the top as part of the root table, as expected. Additionally, `ai_skill` is now a key in `user`, like it's supposed to be.

Even though TOML isn't a complicated format, there are some finer points that you need to take into account when creating your own TOML writer. Instead of pursuing this task further, you'll switch gears and look into how you can use an existing library to dump your data into TOML.

A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You
pythonistacafe.com



Remove ads

Write TOML Documents With `tomli_w`

In this section, you'll work with the `tomli_w` library. As the name indicates, `tomli_w` is related to `tomli`. It comes with two functions, `dump()` and `dumps()`, that are designed to be more or less the opposite of `load()` and `loads()`.

Note: The new `tomllib` library in Python 3.11 doesn't include `dump()` and `dumps()`, and there's no `tomllib_w`. Instead, you can use `tomli_w` to write TOML on all versions of Python since Python 3.7.

You must install `tomli_w` into your virtual environment before you can use it:

Shell

```shell
(venv) $ python -m pip install tomli_w
```

Now, try to redo the example from the previous subsection:

Python >>>

```python
>>> import tomli_w
>>> config = {
...     "user": {
...         "player_x": {"symbol": "X", "color": "blue", "ai": True},
...         "player_o": {"symbol": "O", "color": "green", "ai": False},
...         "ai_skill": 0.85,
...     },
...     "board_size": 3,
...     "server": {"url": "https://tictactoe.example.com"},
... }

>>> print(tomli_w.dumps(config))
board_size = 3

[user]
ai_skill = 0.85

[user.player_x]
symbol = "X"
color = "blue"
ai = true

[user.player_o]
symbol = "O"
color = "green"
ai = false

[server]
url = "https://tictactoe.example.com"
```

No surprises there: `tomli_w` writes the same TOML document that your handwritten `dumps()` function did in the previous section. Additionally, the third-party library supports all the features that you didn't implement, including times and dates, inline tables, and arrays of tables.

`dumps()` writes to a string that you can continue to process. If you want to store your new TOML document directly to disk, then you can call `dump()` instead. As with `load()`, you need to pass in a file pointer opened in binary mode. Continue the example from above:

Python >>>

```python
>>> with open("tic-tac-toe-config.toml", mode="wb") as fp:
...     tomli_w.dump(config, fp)
...
```

This stores the `config` data structure to the file `tic-tac-toe-config.toml`. Have a look at your newly created file:

TOML

```toml
# tic-tac-toe-config.toml

board_size = 3

[user]
ai_skill = 0.85

[user.player_x]
symbol = "X"
color = "blue"
ai = true

[user.player_o]
symbol = "O"
color = "green"
ai = false

[server]
url = "https://tictactoe.example.com"
```

You find all the familiar tables and key-value pairs where you expect them.

Both `tomli` and `tomli_w` are quite basic with somewhat limited functionality, while implementing full support for TOML v1.0.0. In general, you can round-trip your data structures through TOML as long as they're compatible:

Python >>>

```
>>> import tomli, tomli_w
>>> data = {"fortytwo": 42}
>>> tomli.loads(tomli_w.dumps(data)) == data
True
```

Here, you confirm that you're able to recover `data` after first dumping to TOML and then loading back into Python.

Note: One reason that you shouldn't use TOML for data serialization is that there are many data types that aren't supported. For example, if you have a dictionary with numeric keys, then `tomli_w` rightfully refuses to convert it to TOML:

Python >>>

```
>>> import tomli, tomli_w
>>> data = {1: "one", 2: "two"}
>>> tomli.loads(tomli_w.dumps(data)) == data
Traceback (most recent call last):
  ...
TypeError: 'int' object is not iterable
```

The error message isn't very descriptive, but the problem is that non-string keys like `1` and `2` aren't supported in TOML.

Earlier, you learned that `tomli` discards comments. In addition, you can't distinguish between literal strings, multiline strings, and regular strings in the dictionary that's returned by `load()` or `loads()`. Altogether, this means that you lose some metainformation when you parse a TOML document and then write it back:

Python >>>

```
>>> import tomli, tomli_w
>>> toml_data = """
... [nested]  # Not necessary
...
...     [nested.table]
...     string       = "Hello, TOML!"
...     weird_string = '''Literal
...         Multiline'''
... """
>>> print(tomli_w.dumps(tomli.loads(toml_data)))
[nested.table]
string = "Hello, TOML!"
weird_string = "Literal\n        Multiline"
```

The TOML content remains the same, but your output is quite different from what you passed in! The parent table, `nested`, isn't explicitly included in the output, and the comment is gone. Furthermore, the equal signs inside `nested.table` aren't aligned any longer, and `weird_string` isn't represented as a literal multiline string.

Note: You can use the `multiline_strings` parameter to instruct `tomli_w` to use multiline strings when appropriate.

In conclusion, `tomli_w` is a great option for writing TOML documents as long as you don't need a lot of control over the output. In the next section, you'll work with `tomlkit`, which gives you much more control in case you need it. Instead of simply dumping a dictionary to TOML, you'll create a dedicated TOML document object from scratch.

A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You
pythonistacafe.com

Remove ads

Create New TOML Documents

You know how you can quickly read and write TOML documents with `tomli` and `tomli_w`. You've also noticed some of the limitations of `tomli_w`, in particular when it comes to formatting in the resulting TOML files.

In this section, you'll first explore how you can format TOML documents to make them easier to use for the users. Then, you'll try out another library, called `tomlkit`, that you can use to take full control over your TOML documents.

Format and Style TOML Documents

In general, whitespace is ignored in TOML files. You can take advantage of this to make your configuration files well organized, readable, and intuitive. Additionally, a hash symbol (`#`) marks the rest of the line as a comment. Use them liberally.

There's no style guide for TOML documents in the sense that PEP 8 is a style guide for Python code. However, the specification does include some recommendations, while leaving some style aspects open for you to choose.

Some features in TOML are quite flexible. For example, you can define tables in any order you wish. Because the table names are fully qualified, you can even define a sub-table before its parent. Furthermore, whitespace is ignored around keys. The headers `[nested.table]` and `[ nested .  table]` start the same nested table.

The recommendations in the TOML specification can be summarized as **don't abuse flexibility.** Keep your focus on consistency and readability, and you and your users will be happier!

To see a list of styling options where you can reasonably make choices based on your personal preferences, check out the available configuration options for the Taplo formatter. Here are some questions that you can ponder:

- **Indent sub-tables** or rely only on **table headers** to indicate structure?
- **Align the equal signs** in key-value pairs within each table or always stick with **one space on each side** of the equal signs?
- **Split long arrays** to multiple lines or always keep them together on **one line**?
- **Add a trailing comma** after the last value in a multiline array or **leave it bare**?
- Order tables and keys **semantically** or **alphabetically**?

Each of these choices comes down to personal taste, so feel free to experiment to find something that you're comfortable with.

Still, it's good to strive to be consistent. For consistency, you can use a formatter like Taplo in your projects and include its configuration file in your version control. You may be able to integrate it into your editor as well.

Look back to the questions above. If you use `tomli_w` to write your TOML document, then the only question where you have a choice is how to order your tables and keys. If you want more control over your documents, then you need a different tool. In the next subsection, you'll start looking at `tomlkit` which gives you both more power and more responsibility.

Create TOML From Scratch With `tomlkit`

TOML Kit was originally built for the Poetry project. As part of its dependency management, Poetry manipulates the `pyproject.toml` file. However, since this file is used for many purposes, Poetry must preserve the style and comments within the file.

In this subsection, you'll create a TOML document from scratch with `tomlkit` in order to play with some of its functionality. First, you need to install the package into your virtual environment:

Shell

```
(venv) $ python -m pip install tomlkit
```

You can start by confirming that `tomlkit` is more powerful than `tomli` and `tomli_w`. Redo the round-trip example from earlier, and note that all your formatting is preserved:

Python >>>

```
>>> import tomlkit
>>> toml_data = """
... [nested]  # Not necessary
...
...     [nested.table]
...     string       = "Hello, TOML!"
...     weird_string = '''Literal
...         Multiline'''
... """
>>> print(tomlkit.dumps(tomlkit.loads(toml_data)))

[nested]  # Not necessary

    [nested.table]
    string       = "Hello, TOML!"
    weird_string = '''Literal
        Multiline'''

>>> tomlkit.dumps(tomlkit.loads(toml_data)) == toml_data
True
```

You can use `loads()` and `dumps()`—and `load()` and `dump()`—to read and write TOML as earlier. However, now all your string types, indentations, comments, and alignments are preserved.

To achieve this, `tomlkit` uses custom data types that behave more or less like your native Python types. You'll learn more about these data types later. First, you'll see how you can create a TOML document from scratch:

Python >>>

```
>>> from tomlkit import comment, document, nl, table

>>> toml = document()
>>> toml.add(comment("Written by TOML Kit"))
>>> toml.add(nl())
>>> toml.add("board_size", 3)
```

In general, you need to start by calling `document()` to create a TOML document instance. You can then use `.add()` to add different objects to this document, like comments, newlines, key-value pairs, and tables.

Note: Calling `.add()` returns the updated object. You won't see this in the examples in this section, as the extra output would distract from the flow of the examples. Later you'll see how you can take advantage of this design, though, and chain together several calls to `.add()`.

You convert `toml` to an actual TOML document by using `dump()` or `dumps()` as above, or you can use the `.as_string()` method:

Python >>>

```
>>> print(toml.as_string())
# Written by TOML Kit

board_size = 3
```

In this example, you're starting to re-create parts of the tic-tac-toe configuration that you've worked with previously. Note how each line in the output corrensponds to an `.add()` method in your code. First, you have the comment, then `nl()` representing a blank line, and then the key-value pair.

Continue your example by adding a few tables:

Python >>>

```python
>>> player_x = table()
>>> player_x.add("symbol", "X")
>>> player_x.add("color", "blue")
>>> player_x.comment("Start player")
>>> toml.add("player_x", player_x)

>>> player_o = table()
>>> player_o.update({"symbol": "O", "color": "green"})
>>> toml["player_o"] = player_o
```

You create tables by calling `table()` and adding content to them. After you've created a table, you add it to the TOML document. You can stick to using `.add()` to assemble your document, but the example also shows a few alternative ways that you can add content. For example, you can use `.update()` to add keys and values directly from a dictionary.

When you convert your document to a TOML string, it'll look as follows:

Python >>>

```python
>>> print(toml.as_string())
# Written by TOML Kit

board_size = 3

[player_x] # Start player
symbol = "X"
color = "blue"

[player_o]
symbol = "O"
color = "green"
```

Compare this output with the commands that you used to create the document. If you're creating a TOML document with a fixed structure, then it's probably easier to write the document as a TOML string and load it with `tomlkit`. However, the commands you've seen above give you a lot of flexibility when dynamically putting together your configuration.

In the next section, you'll dig deeper into `tomlkit` and see how you can use it to update existing configurations.

Update Existing TOML Documents

Imagine that you've spent some time putting together a well-organized configuration with good comments that instruct your users how they can change it. Then some other application comes along and stores its configuration in the same file, at the same time destroying your carefully crafted work of art.

This might be an argument for keeping your configuration in a dedicated file that no one else will touch. However, sometimes it's convenient to use a common configuration file as well. The `pyproject.toml` file is used as such a common file, especially for tools that you use when developing and building packages.

In this section, you'll dive deeper into how `tomlkit` represents TOML objects and how you can use the package to update existing TOML files.

Represent TOML as `tomlkit` Objects

Earlier, you saw that `tomli` and `tomllib` parse a TOML document into native Python types like strings, integers, and dictionaries. You've seen some indications that `tomlkit` is different. Now, it's time to look closer at how `tomlkit` represents a TOML document.

First, copy and save the following TOML document as `tic-tac-toe-config.toml`:

TOML

```toml
# tic-tac-toe-config.toml

board_size = 3

[user]
ai_skill = 0.85  # A number between 0 (random) and 1 (expert)

    [user.player_x]
    symbol = "X"
    color = "blue"
    ai = true

    [user.player_o]
    symbol = "O"
    color = "green"
    ai = false

# Settings used when deploying the application
[server]
url = "https://tictactoe.example.com"
```

Open a REPL session and load this document with `tomlkit`:

Python >>>

```python
>>> import tomlkit
>>> with open("tic-tac-toe-config.toml", mode="rt", encoding="utf-8") as fp:
...     config = tomlkit.load(fp)
...
>>> config
{'board_size': 3, 'user': {'ai_skill': 0.85, 'player_x': { ... }}}

>>> type(config)
<class 'tomlkit.toml_document.TOMLDocument'>
```

You use `load()` to load the TOML document from a file. When you look at `config`, it looks like a dictionary at first glance. However, digging deeper, you find that it's a special `TOMLDocument` type.

Note: Unlike `tomli`, `tomlkit` expects you to open the file in text mode. You should also remember to specify that the file should be opened using the `utf-8` encoding.

These custom data types behave more or less like your native Python types. For example, you can access sub-tables and values in your document using square brackets (`[]`), just like dictionaries. Continue the example from above:

Python >>>

```python
>>> config["user"]["player_o"]["color"]
'green'

>>> type(config["user"]["player_o"]["color"])
<class 'tomlkit.items.String'>

>>> config["user"]["player_o"]["color"].upper()
'GREEN'
```

Even though the values are also special `tomlkit` data types, you can work with them as if they're regular Python types. For example, you can use the `.upper()` string method.

One advantage of the special data types is that they give you access to metainformation about the document, including comments and indentation:

Python >>>

```
>>> config["user"]["ai_skill"]
0.85

>>> config["user"]["ai_skill"].trivia.comment
'# A number between 0 (random) and 1 (expert)'

>>> config["user"]["player_x"].trivia.indent
'    '
```

For example, you can recover comment and indentation information through the `.trivia` accessor.

As you saw above, you can mostly treat these special objects as if they were native Python objects. In fact, they inherit from their native counterparts. However, if you really need to, you can use `.unwrap()` to convert them to plain Python:

Python >>>

```
>>> config["board_size"] ** 2
9

>>> isinstance(config["board_size"], int)
True

>>> config["board_size"].unwrap()
3

>>> type(config["board_size"].unwrap())
<class 'int'>
```

After `.unwrap()` is called, the `3` is now a regular Python integer. Altogether, this investigation gives you some insight into how `tomlkit` is able to preserve the style of TOML documents.

In the next subsection, you'll learn how you can use the `tomlkit` data types to customize a TOML document without affecting the existing style.

Read and Write TOML Losslessly

You know that `tomlkit` represents a TOML document using custom classes, and you've seen how you can create these objects from scratch and how you can read existing TOML documents. In this subsection, you'll load an existing TOML file and make some changes to it before writing it back to disk.

Start by loading the same TOML file that you used in the previous subsection:

Python >>>

```
>>> import tomlkit
>>> with open("tic-tac-toe-config.toml", mode="rt", encoding="utf-8") as fp:
...     config = tomlkit.load(fp)
...
```

As you saw earlier, `config` is now a `TOMLDocument`. You can use `.add()` to add new elements to it, exactly as you did when you created a document from scratch. However, you can't use `.add()` to update the value of existing keys:

Python >>>

```
>>> config.add("app_name", "Tic-Tac-Toe")
{'board_size': 3, 'app_name': 'Tic-Tac-Toe', 'user': { ... }}

>>> config["user"].add("ai_skill", 0.6)
Traceback (most recent call last):
  ...
KeyAlreadyPresent: Key "ai_skill" already exists.
```

You try to lower the skill of the AI so that you'll have an easier opponent to play against. However, you can't do this with `.add()`. Instead, you can assign the new value as if `config` were a regular dictionary:

Python >>>

```
>>> config["user"]["ai_skill"] = 0.6
>>> print(config["user"].as_string())
ai_skill = 0.6  # A number between 0 (random) and 1 (expert)

    [user.player_x]
    symbol = "X"
    color = "blue"
    ai = true

    [user.player_o]
    symbol = "O"
    color = "green"
    ai = false
```

When you update a value like this, `tomlkit` still takes care to preserve the style and comments. As you can see, the comment about `ai_skill` is left untouched.

Parts of the `tomlkit` supports what's known as a fluent interface. In practice, this means that operations like `.add()` return the updated object so that you can chain another call to `.add()` onto it. You can take advantage of this when you need to construct tables with several fields:

Python >>>

```
>>> from tomlkit import aot, comment, inline_table, nl, table
>>> player_data = [
...     {"user": "gah", "first_name": "Geir Arne", "last_name": "Hjelle"},
...     {"user": "tompw", "first_name": "Tom", "last_name": "Preston-Werner"},
... ]

>>> players = aot()
>>> for player in player_data:
...     players.append(
...         table()
...         .add("username", player["user"])
...         .add("name",
...             inline_table()
...             .add("first", player["first_name"])
...             .add("last", player["last_name"])
...         )
...     )
...
>>> config.add(nl()).add(comment("Players")).add("players", players)
```

In this example, you create an array of tables with information about players. You start by creating an empty array of tables with the `aot()` constructor. Then you loop over your player data to append each player to the array.

You use method chaining to create each player table. In practice, your call is `table().add().add()` which adds two elements to a new table. Finally, you add the new array of player tables at the bottom of your configuration, below a short comment.

With your updates to the configuration done, you're now ready to write it back to the same file:

Python >>>

```
>>> with open("tic-tac-toe-config.toml", mode="wt", encoding="utf-8") as fp:
...     tomlkit.dump(config, fp)
```

Open up `tic-tac-toe-config.toml` and note that your updates are included. At the same time, the preexisting style has been preserved:

TOML

```toml
# tic-tac-toe-config.toml

board_size = 3
app_name = "Tic-Tac-Toe"

[user]
ai_skill = 0.6  # A number between 0 (random) and 1 (expert)

    [user.player_x]
    symbol = "X"
    color = "blue"
    ai = true

    [user.player_o]
    symbol = "O"
    color = "green"
    ai = false

# Settings used when deploying the application
[server]
url = "https://tictactoe.example.com"

# Players

[[players]]
username = "gah"
name = {first = "Geir Arne", last = "Hjelle"}

[[players]]
username = "tompw"
name = {first = "Tom", last = "Preston-Werner"}
```

Note that `app_name` has been added, the value of `user.ai_skill` has been updated, and the array of `players` tables has been appended to the end of your configuration. You've successfully updated your configuration programmatically.

Conclusion

This is the end of your extensive tour of the TOML format and the ways that you can use it in Python. You've seen some of the features that make TOML a flexible and convenient format for configuration files. At the same time, you've uncovered some of the limitations that restrict its usefullness in other applications, like data serialization.

In this tutorial, you've:

- Learned about the **TOML syntax** and which data types it supports
- **Parsed** TOML documents with `tomli` and `tomllib`
- **Written** TOML documents with `tomli_w`
- Losslessly **updated** TOML files with `tomlkit`

Do you have applications where you need to have a convenient configuration? TOML might be just what you're looking for.

Free Download: Get a sample chapter from Python Tricks: The Book that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

Mark as Completed

🐍 Python Tricks 💌