TOML Configuration File Cheatsheet

A quick reference guide to the TOML (Tom's Obvious, Minimal Language) configuration file format syntax, data types, and structure, including useful tips and examples.

Basic Syntax & Types

Key-Value Pairs

Basic assignment: `key = "value"`	`name = "Tom"`
Keys can contain letters, numbers, underscores, and hyphens.	`first-name = "Tom" last_name = "Preston-Werner"`

Dotted keys for nesting. Equivalent to tables.	`person.name = "Tom" person.age = 30`
	
	Is same as:
	`[person]`
	`name = "Tom"`
	`age = 30`

Keys must be quoted if they contain special characters (like spaces, `.`, `[`, `]`, `#`, `=` etc.) or start with a digit.	`"this is a key" = "value" "192.168.1.1" = "value"`
Keys must be unique within their table/scope.	`[database] server = "192.168.1.1" server = "192.168.1.2" # ERROR: Duplicate key`
Case-sensitive.	`Fruit = 1 fruit = 2 # Valid (different keys)`

Whitespace around keys and values is ignored.	`key = "value"`
	`  key = "value"`
	`key = "value"`
	All are equivalent.

Comments

Start with `#` and continue to the end of the line.
`# This is a full-line comment` `key = "value" # This is an end-of-line comment`
Comments can appear anywhere except within strings.
Used for explanations, disabling lines, etc.
Blank lines and comments are ignored by parsers.
Help improve readability and documentation of your config file.

Strings

Basic Strings: Enclosed in `"`. Escape sequences: `\"`, `\\`, `\b`, `\f`, `\n`, `\r`, `\t`, `\uXXXX`, `\UXXXXXXXX`	`s1 = "I'm a string." s2 = "He said, \"Hello!\""`

Multiline Basic Strings: Enclosed in `"""`.
Newline immediately after opening `"""` is trimmed.
Escape sequences work.
	`multiline_string = """`
	`This is a`
	`multiline`
	`string.`
	`"""`
	`# Result: "This is a\nmultiline\nstring.\n"`

Multiline Basic Strings with `\` at end of line: Concatenates lines, ignoring whitespace after `\`.	`continued_string = """`
	`The quick brown fox jumps over \`
	`  the lazy dog.`
	`"""`
	`# Result: "The quick brown fox jumps over the lazy dog."`

Literal Strings: Enclosed in `'`. No escaping is performed.	`literal_string = 'C:\Users\name\Documents'`

Multiline Literal Strings: Enclosed in `''''`.
Newline immediately after opening `''''` is trimmed.
No escaping.
	`multiline_literal = ''''`
	`This is a`
	`literal`
	`multiline string.`
	`''''`
	`# Result: "This is a\nliteral\nmultiline string.\n"`

Numbers

Integers: Decimal numbers. Underscores allowed for readability.	`i1 = 99 i2 = 1_000 i3 = -5_000_000`
Integers: Hexadecimal (prefix `0x`).	`hex1 = 0xDEADBEEF hex2 = 0xdeadbeef`
Integers: Octal (prefix `0o`).	`oct1 = 0o123 oct2 = 0o755`
Integers: Binary (prefix `0b`).	`bin1 = 0b11010110`
Floats: Standard decimal numbers with a fractional part or exponent. Underscores allowed.	`f1 = 1.0 f2 = 3.14159 f3 = -0.01 f4 = 1_000.00 f5 = 1e20 f6 = -2E-2 f7 = 6.626e-34`
Floats: Special values `inf`, `-inf`, `nan`. Case-insensitive.	`infinite = inf negative_infinite = -inf not_a_number = nan`

Booleans

`true`	Represents the boolean true value.
`false`	Represents the boolean false value.
Case-sensitive. Must be lowercase `true` or `false`.	`enabled = true disabled = false`

Datetimes

Full Datetime (with timezone): RFC 3339 format. `YYYY-MM-DDTHH:MM:SSZ` or `YYYY-MM-DDTHH:MM:SS±HH:MM`. T separator can be lowercase `t`.	`dt1 = 1979-05-27T07:32:00Z dt2 = 1979-05-27T00:32:00-07:00 dt3 = 1979-05-27t07:32:00Z`
Full Datetime (with timezone and fractional seconds): RFC 3339 format.	`dt4 = 1979-05-27T07:32:00.999Z`
Local Datetime (without timezone): `YYYY-MM-DDTHH:MM:SS`	`ldt1 = 1979-05-27T07:32:00`
Local Datetime (without timezone and with fractional seconds):	`ldt2 = 1979-05-27T07:32:00.999`
Local Date (without time): `YYYY-MM-DD`	`date1 = 1979-05-27`
Local Time (without date): `HH:MM:SS`	`time1 = 07:32:00`
Local Time (with fractional seconds):	`time2 = 07:32:00.999`

Data Structures

Arrays

Arrays are enclosed in `[` and `]`.

Elements are separated by commas.

Whitespace is ignored.

```
array1 = [ 1, 2, 3 ]
array2 = [ "red", "yellow", "green" ]
```

Arrays can contain elements of different types, but this is discouraged for consistency.

```
array3 = [ 1, "a", 1.1, true ] # Valid but discouraged
```

Arrays can contain other arrays or tables (inline tables).

```
array4 = [ [1, 2], [3, 4] ]
array5 = [ { name = "apple" }, { name = "banana" } ]
```

Trailing commas are allowed.

```
array6 = [ 1, 2, 3, ]
```

Tables

Tables (also known as sections or hash tables) are defined by a header like `[table-name]`.

Keys below a table header belong to that table until the next table header or EOF.

```
[database]
server = "192.168.1.1"
ports = [ 8000, 8001, 8002 ]
enabled = true
```

Table names follow the same rules as keys.

Quoted table names are allowed for special characters or starting with digits.

```
["table name"]
value = 1
```

Nested tables are created using dotted names.

```
[products.fruit]
apple = { color = "red", taste = "sweet" }
[products.vegetables]
carrot = { color = "orange", taste = "earthy" }
```

This is equivalent to:

```
[products]
  [products.fruit]
  apple = { color = "red", taste = "sweet" }
  [products.vegetables]
  carrot = { color = "orange", taste = "earthy" }
```

If a table or dotted key is defined implicitly (e.g., `[a.b]`), you cannot later redefine it explicitly (`[a]`) or implicitly add keys *before* the explicit/later definition.

```
[a.b]
c = 1 # Defines table 'a' implicitly
[a]
# ERROR: table 'a' already defined implicitly by [a.b]
```

Inline Tables

Inline tables provide a more compact syntax for tables, especially useful within arrays or as values.

Enclosed in `{` and `}`.

Key-value pairs separated by commas.

Must appear on a single line.

```
name = { first = "Tom", last = "Preston-Werner" }
```

Useful inside arrays.

```
points = [ { x = 1, y = 2 }, { x = 3, y = 6 }, { x = 7, y = 8 }
]
```

Contain key-value pairs just like standard tables, but cannot contain other standard tables or arrays of tables (but can contain inline tables).

```
person = { name = "Tom", address = { street = "10 Downing St" }
}
```

Array of Tables

Used to represent a list of tables (or records).

Defined by a header like `[[table-name]]`.

Each `[[table-name]]` entry adds a new table to the array.

```
[[products]]
name = "apple"

[[products]]
name = "banana"
```

This creates an array named `products` containing two tables:

```
[
  { "name": "apple" },
  { "name": "banana" }
]
```

Keys below a `[[table-name]]` header belong to that specific table instance until the next table header or EOF.

Nested arrays of tables are defined with dotted names.

```
[[parent.child]]
key = 1

[[parent.child]]
key = 2
```

Cannot define an array of tables if a standard table with the same name already exists, or vice-versa.

```
[database]
server = "192.168.1.1"

[[database]] # ERROR: 'database' is already a standard table
url = "...
```

Tips & Best Practices

General Readability

Use underscores `_` in large numbers for readability (e.g., `1_000_000`).

Use blank lines to separate logical groups of key-value pairs or tables.

Add comments (`#`) to explain complex sections or provide context.

Prefer standard tables `[table]` for larger sections and inline tables `{...}` for short, simple values or within arrays.

Limit line length for better readability.

Structuring Your File

Conventionally, place top-level key-value pairs first, followed by standard tables, then arrays of tables.

Group related configuration options under appropriate table headers.

Use dotted keys (`[a.b]`) or nested table definitions (`[a] [a.b]`) consistently within a file.

Avoid mixing implicit table definitions (via dotted keys) and explicit definitions in a way that causes errors (see Table section in Page 2).

Keep table and key names descriptive and concise.

Data Types

Use the most specific data type possible (e.g., boolean for flags, integer for counts, datetime for timestamps).

Avoid mixed-type arrays unless absolutely necessary; it often indicates a design issue.

Use multiline strings `"""..."""` for blocks of text or strings containing internal quotes/backslashes to avoid excessive escaping.

Use literal strings `'...'` or `'''...'''` for paths or regex patterns where backslashes should be treated literally.

Tooling & Validation

Always validate your TOML files using a parser or linter before deploying.

Many programming languages have robust TOML parsing libraries.

Editors often have syntax highlighting and basic validation for `.toml` files.

File Naming Convention

Use the `.toml` file extension.