

Package: fathom (via r-universe)

September 19, 2026

Title Look Before You Parse

Version 0.0.1

Description One small vocabulary for seeing what is in a JSON document and getting a table out of it, spelled the same way in R and in Python. You point it at a document you have never seen and it reports whether the file is sound, what shapes it holds, and what one row could be, with every candidate priced. Output is proportional to the structure rather than to the data, so a 912 MB file and a 12 KB file produce descriptions of similar size. The package is a thin wrapper around the 'fathom' engine, which it invokes as a subprocess; the engine is compiled and bundled at install time, so an installed copy is self-contained and needs nothing else.

License Apache License (≥ 2)

Encoding UTF-8

SystemRequirements Cargo (Rust's package manager), rustc

URL <https://github.com/psychometrician/fathom>,
<https://psychometrician.github.io/fathom-book/>

BugReports <https://github.com/psychometrician/fathom/issues>

Config/pak/sysreqs libclang-dev

Repository <https://psychometrician.r-universe.dev>

Date/Publication 2026-08-20 16:39:45 UTC

RemoteUrl <https://github.com/psychometrician/fathom>

RemoteRef HEAD

RemoteSha 32da152cc87183c2752f30f2118cf1248fdaf470

RemoteSubdir r-pkg/fathom

Contents

fathom	2
fathom_binary	3
find	4
into	5
read_json	7
rows	8
Index	10

fathom	<i>Describe a JSON document</i>
--------	---------------------------------

Description

Reads a JSON document and prints what is in it: whether it is sound, the shapes it holds folded to their structure, the fields that change type, and what one row could be with every candidate priced.

The output is proportional to the **structure**, not to the data. That is the whole claim, and it is why this is worth running on a file too large to open.

Usage

fathom(path)

Arguments

path Path to one file: JSON, NDJSON, or gzipped.

Details

All of the work happens in the fathom command-line binary, which this function invokes as a subprocess. See [fathom_binary](#) for where it is looked for and what to do when it is not found.

Value

A fathom_report: the printed text, with a print method. Returned rather than printed eagerly, so that `x <- fathom(f)` is silent and `fathom(f)` at the console is not. That is R’s own convention, and it makes this behave exactly like the Python binding, which the design requires: the two languages differ by the pipe alone.

See Also

[fathom_binary](#)

Examples

```
## Not run:
fathom("package-lock.json")

# It is text, so it can be kept, written out, or searched.
report <- fathom("package-lock.json")
writeLines(as.character(report), "described.txt")

## End(Not run)
```

fathom_binary

Locate the fathom binary

Description

This package does no JSON work of its own. It runs the `fathom` command-line binary as a subprocess and hands back what it printed, so the binary has to be findable.

Usage

```
fathom_binary(error = TRUE)
```

Arguments

<code>error</code>	Whether to stop when nothing is found. <code>FALSE</code> returns <code>NULL</code> instead, which is what a test harness wants.
--------------------	--

Details

Four places, in order, and the order is the point: an explicit override always wins, a development build outranks the bundled engine, and the `PATH` is last because a `fathom` on it has no reason to match this copy of the package.

1. `$FATHOM_BIN`, if set. If it names a file that does not exist this is an error rather than a fallback: it was set on purpose, and quietly running some other binary is how a measurement gets attributed to the wrong build.
2. `target/release/fathom`, in the working directory or any directory above it. This is the development build, found by walking upward the way `uv run` finds a project, so that scripts run from a subdirectory work without configuration. It comes **BEFORE** the bundled engine deliberately: a bundled copy is written at install time and then stays exactly as old as that install, so preferring it is how a harness spends a day measuring a stale engine while every check passes.
3. The engine bundled inside the installed package, which `configure` compiles and copies to `inst/bin` during `R CMD INSTALL`. This is what makes an installed copy self-contained: nothing else in the package ever creates that file, and without it a user who did not build the project has no engine at all.
4. `fathom` on the `PATH`.

An installed package carries its own engine, so this needs no configuration. In a checkout, build one with `cargo build --release` from the project root.

Value

The path to the binary, or `NULL`.

See Also

[fathom](#)

Examples

```
## Not run:
fathom_binary()
fathom_binary(error = FALSE) # NULL rather than an error when absent

## End(Not run)
```

find	<i>Find every path whose value matches, and take whichever name is there</i>
------	--

Description

`find()` answers *where* – it searches by value and gives you back paths. `whichever()` answers *which of these fields actually exists here*.

Usage

```
find(x, test)

whichever(x, ...)
```

Arguments

x	A view from read_json , or a path.
test	One of "url", "email", "date", "empty".
...	For <code>whichever()</code> , the names to try, in the order you would try them by hand.

Value

`find()`: a `data.frame` of path and count.

`whichever()`: a `data.frame` of key and value, one row per thing where you are standing, with an empty value where none of the names was there.

What find() returns, and why it is folded

Paths, folded, with a count. The unfolded answer on a real document is thousands of rows – the cost-proportional-to-data failure this project exists to name, committed by fathom’s own word. **A word that answers a question by printing the data has not answered it.** So `versions.<key>.dist.tarball` and its 288 matches is one row, not 288.

Why the test is a name and not a function

```
find(view, "url"), not find(view, function(x) grepl("^http", x)).
```

A function cannot cross a subprocess boundary, and it could not be the same function in both languages if it did. A regular expression would need a regex engine in a core that has one dependency on purpose. The tests are `url`, `email`, `date` and `empty` – and a name still reads a year later, which is the bar this vocabulary is held to.

Path variance, which is what whichever is for

A document that spells the same field `Rating` in some records and `rating` in others is answered by asking for both. A null counts as absent, because a field that is present and null has not told you anything, so the next name gets its turn.

See Also

[read_json](#), [into](#), [rows](#)

Examples

```
## Not run:
read_json("package.json") |> find("url")
read_json("package.json") |> find("empty")

read_json("package.json") |>
  into("versions") |>
  whichever("author", "_npmUser")

## End(Not run)
```

into

Go into a part of the document, and come back out

Description

`into()` moves you one name deeper; `back()` moves you out one level. Neither reads the document. A view is a source and a path, so navigating is list arithmetic and costs nothing until you ask a question.

Usage

```
into(x, name)
```

```
back(x)
```

Arguments

x	A view from read_json , or a path.
name	The name to go into.

Details

Going in is how you make the expensive question cheap. The walk, the fold, the classifier and the pricing all run on what you are standing on, so describing one part of a large document costs a fraction of describing the whole of it. Measured on an 18 MB file: the root takes about 6 seconds and one part of it takes 0.09 seconds.

It is also how you look inside a nested cell. [rows](#) hands nested values back as JSON text rather than parsing them, because a parser in the binding would be a second implementation of the one thing the core exists to do – and the two languages' JSON parsers do not agree on numbers past 2^{53} . So the way in is to stand there instead.

A name resolves three ways. On an object that has that field, you get the field. On an array, you get that field from every item. On an object that does not have it, you get that field from every value, which is a keyed collection.

The resolved path is printed by [fathom](#) as at `$.versions.*.dependencies`, so the `*` that means *mapped over every value here* is something you are shown rather than something you type.

Value

A `fathom_view`, one level deeper or shallower.

See Also

[read_json](#), [fathom](#), [rows](#)

Examples

```
## Not run:
read_json("package.json") |> into("versions") |> fathom()

read_json("package.json") |>
  into("versions") |>
  into("dependencies") |>
  rows("an entry of $[]")

read_json("package.json") |> into("versions") |> back() |> fathom()

## End(Not run)
```

read_json	<i>Read a JSON source, and say whether it is sound</i>
-----------	--

Description

The first word. Everything else takes what this returns.

A view cannot hold the parsed document – that lives in the core and ends with the call – so it holds the source and a path. That is what makes navigation free, and what makes a chain possible at all.

Usage

```
read_json(source = NULL, text = NULL)
```

Arguments

source	Path to one file: JSON, NDJSON, or gzipped.
text	A JSON document already in memory, as one string. Give exactly one of source or text. Named rather than guessed, because a string that might be either is the wart every other reader has.

Details

It reports damage rather than raising on it. A truncated file gives you a view that says chopped off; it does not throw. [fathom](#) on that view still prints the full diagnosis, with the parser's line and column. Refusing would replace a good report with a worse exception – and there is no such thing as a broken data frame, so nothing in the rectangular world prepares you for the reply.

Checking costs about 1% of a `fathom()`. A parse is 0.05s on 18 MB where describing the same document is 4.7s, so soundness is answered up front and the expensive question is asked only when you ask it.

Value

A `fathom_view`: a source, and where you are standing in it. Print it to see the source, its soundness, and your position.

See Also

[fathom](#), [rows](#)

Examples

```
## Not run:
read_json("package.json")
read_json(text = '{"a": [1, 2, 3]}')

read_json("package.json") |> fathom()
read_json("package.json") |> rows("an entry of versions")
```

```
## End(Not run)
```

rows	<i>Take a table out of a JSON document</i>
------	--

Description

fathom() prints a menu of row shapes, each one priced. rows() takes one of those labels and returns that table.

You are not guessing at a path. You are naming a row shape the report just showed you and told you the size of, which is why the label is the whole argument and why there is no path language to learn.

Usage

```
rows(path, candidate)
```

Arguments

path	Path to one file: JSON, NDJSON, or gzipped.
candidate	The label to take, exactly as fathom() printed it under ONE ROW COULD BE. The label is the whole argument.

Details

Cells are not flattened. 17 of 19 corpus extracts carry a column whose values are nested, and a data frame has nowhere to put one, so fathom hands it over intact rather than flattening it behind your back. Parse the columns you care about with whatever you already use.

This package has no dependencies, so the table crosses from the binary as tab-separated JSON values and is read with read.delim. Nothing is reshaped or re-typed on the way: a binding that did either would be a second implementation of the answer.

Value

A data.frame, one row per thing, every column character. Each cell holds a JSON value as text: a string arrives quoted, a number bare, and a nested array or object arrives as JSON.

An empty cell means the field was **absent**. A cell reading null means the field was there and null. Those are different things and this keeps them apart.

See Also

[fathom](#)

Examples

```
## Not run:  
fathom("package.json")  
rows("package.json", "an entry of versions")  
  
## End(Not run)
```

Index

`back (into)`, [5](#)

`fathom`, [2](#), [4](#), [6–8](#)

`fathom_binary`, [2](#), [3](#)

`find`, [4](#)

`into`, [5](#), [5](#)

`print.fathom_view (read_json)`, [7](#)

`read_json`, [4–6](#), [7](#)

`rows`, [5–7](#), [8](#)

`whichever (find)`, [4](#)