

Package: god (via r-universe)

September 20, 2026

Title A Grammar of Data

Version 0.2.2

Description One small vocabulary for manipulating tables, spelled the same way in R, in Python, and as plain text. A pipeline is checked whole before any of it runs, so a bad column is reported at the step that names it rather than failing partway through. The grammar owns the words, the checking and the messages; an engine underneath does the work. The manual, live in both languages, is at
<<https://psychometrician.github.io/god-book/>>.

License Apache License (== 2.0)

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

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

SystemRequirements Cargo (Rust's package manager), rustc

Encoding UTF-8

Depends R (>= 4.1)

Imports DBI, duckdb

Suggests dplyr, knitr, pkgload

Config/roxygen2/version 8.0.0

Config/pak/sysreqs xz-utils libclang-dev

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

Date/Publication 2026-08-21 15:00:46 UTC

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

RemoteRef HEAD

RemoteSha 93b7cbcc3de8a5b912643d6b7b680c481afedfa8

RemoteSubdir r-pkg/god

Contents

add	2
add_combinations	3
add_rows	4
collect	5
drop_duplicates	5
drop_missing	6
fill_missing	7
format.god_pipeline	7
format.god_steps	8
god_sql	9
god_table	9
join	10
keep	11
lengthen	12
pick	13
rename	13
run	14
show_as	15
show_steps	16
sort	17
summarize	17
take	18
take_last	19
use_engine	20
widen	21
Index	22

add	<i>Add or replace a column</i>
-----	--------------------------------

Description

Add or replace a column

Usage

`add(.data, ..., by)`

Arguments

- | | |
|--------------------|--|
| <code>.data</code> | A table, or a pipeline. |
| <code>...</code> | Named columns: ‘margin = revenue - cost’. |
| <code>by</code> | Columns to group by. An aggregate is then broadcast back over each group rather than collapsing it: ‘add(share = revenue / total(revenue), by = product)’. |

Value

A pipeline.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
sales |> add(doubled = revenue * 2)
#>   region revenue doubled
#> 1  West      100      200
#> 2  East      120      240
#> 3  West      150      300
```

add_combinations	<i>Make the absent combinations appear</i>
------------------	--

Description

Every combination of the values these columns already hold, as rows. The rows that were there are handed on untouched; the ones that were not arrive with every other column missing, and ‘fill_missing’ is what says otherwise.

Usage

```
add_combinations(.data, ..., by)
```

Arguments

.data	A table, or a pipeline.
...	The columns whose combinations to make. Two or more: one column on its own has no combinations to make.
by	Columns to make the combinations inside. Without it the whole table is one group. With it, a new row keeps these columns filled in rather than going missing, which is the reason to write one.

Details

The values come from the table and nowhere else, so a month with no row anywhere is never invented. A missing value is not a category and makes no combinations, and no row is lost by that: nothing already in the table is touched at all.

Value

A pipeline.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  product = c("Widget", "Widget", "Gadget"),
  revenue = c(100, 120, 150)
)
sales |>
  add_combinations(region, product) |>
  fill_missing(revenue = 0)
#>   region product revenue
#> 1  West  Widget     100
#> 2  East  Widget     120
#> 3  West  Gadget     150
#> 4  East  Gadget       0
```

add_rows

Add another table's rows

Description

Both tables need the same columns. A column on one side only is refused rather than filled in with missing values, because a column that is half empty and says nothing is how a mistake survives.

Usage

```
add_rows(.data, other)
```

Arguments

.data	A table, or a pipeline.
other	The other table.

Value

A pipeline.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
late <- data.frame(region = "North", revenue = 80)
sales |> add_rows(late)
#>   region revenue
#> 1  West     100
#> 2  East     120
#> 3  West     150
#> 4 North      80
```

`collect`*Run a pipeline and get the table*

Description

Nothing runs until the answer is wanted. Printing a pipeline runs it, and so does converting one; this is the explicit form, for when you want the table rather than the look of it.

Usage

```
collect(pipeline)
```

Arguments

`pipeline` A pipeline.

Value

A data frame.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
answer <- collect(sales |> keep(region == "West"))
nrow(answer)
#> [1] 2
```

`drop_duplicates`*Drop repeated rows*

Description

Rows that are identical across every column. The answer comes back in a settled order, because dropping duplicates says nothing about order and an answer that reorders itself between runs is not predictable.

Usage

```
drop_duplicates(.data)
```

Arguments

`.data` A table, or a pipeline.

Value

A pipeline.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  product = c("Widget", "Gadget", "Widget")
)
sales |> drop_duplicates()
#>   region product
#> 1   East  Gadget
#> 2   West  Widget
```

drop_missing	<i>Drop rows with missing values</i>
--------------	--------------------------------------

Description

Drop rows with missing values

Usage

```
drop_missing(.data, ...)
```

Arguments

.data	A table, or a pipeline.
...	Columns to look at. With none, every column.

Value

A pipeline.

Examples

```
patchy <- data.frame(
  product = c("Widget", "Gadget"),
  revenue = c(100, NA),
  listed = c(90, 60)
)
patchy |> drop_missing(revenue)
#>   product revenue listed
#> 1  Widget     100     90
```

fill_missing	<i>Replace missing values in a column</i>
--------------	---

Description

Replace missing values in a column

Usage

```
fill_missing(.data, ...)
```

Arguments

.data	A table, or a pipeline.
...	'column = value' pairs.

Value

A pipeline.

Examples

```
patchy <- data.frame(
  product = c("Widget", "Gadget"),
  revenue = c(100, NA),
  listed = c(90, 60)
)
patchy |> fill_missing(revenue = 0)
#>   product revenue listed
#> 1  Widget     100     90
#> 2  Gadget       0     60
```

format.god_pipeline	<i>The pipeline as the grammar's own text</i>
---------------------	---

Description

The pipeline as the grammar's own text

Usage

```
## S3 method for class 'god_pipeline'
format(x, ...)
```

Arguments

x	A pipeline.
...	Unused.

Value

The text.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
format(sales |> keep(region == "West"))
#> [1] "sales\n  then keep where ([region] is \"West\")"
```

format.god_steps	<i>The drawing as text</i>
------------------	----------------------------

Description

The drawing as text

Usage

```
## S3 method for class 'god_steps'
format(x, as = "text", ...)
```

Arguments

x	A drawing, from [show_steps()].
as	“text” for the ladder, “svg” for the picture.
...	Unused.

Value

The drawing, as one string.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
format(show_steps(sales |> take(1)))
#> [1] "sales      region:text  revenue:number\n take 1  region  revenue\n      at most 1 rows"
```

god_sql

The query a pipeline becomes

Description

The query a pipeline becomes

Usage

```
god_sql(pipeline, columns)
```

Arguments

pipeline The pipeline, as text.
columns The table's columns, as 'name:type' separated by commas.

Value

The query, as text.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
cat(god_sql("sales then take 1", "region:text,revenue:number"))
#> WITH step0 AS (SELECT * FROM "sales"),
#>        step1 AS (SELECT * FROM step0 LIMIT 1)
#> SELECT * FROM step1
```

god_table

Read one of the book's example tables

Description

Returns a table ready to pipe. A 'data/<name>.csv' in the working directory or any directory above it is read first; failing that, the copy published beside the manual is fetched. The cast is declared in the book's preface: 'sales', 'products', 'survey', 'answers', 'marks', 'messy', 'diary' and 'gapminder'.

Usage

```
god_table(name, text = character())
```

Arguments

name	The table's name without the extension, such as "sales".
text	Columns that must stay text. A CSV records what a value is and never what kind of thing it is, so a column of '01', '02', '03' comes back as the numbers 1, 2, 3 unless it is named here.

Value

A data frame.

Examples

```
## Not run:
sales <- god_table("sales")
sales |> keep(region == "West") |> take(3)

## End(Not run)
```

join	<i>Add another table's columns</i>
-------------	------------------------------------

Description

Add another table's columns

Usage

```
join(.data, other, by, unmatched = "this")
```

Arguments

.data	A table, or a pipeline.
other	The other table.
by	The columns that say which rows correspond. Left out, the columns both tables share are used, and god says which it chose. Where the two tables name a key differently, write both with '==' between them and this table's first: 'by = customer_id == id'. The answer keeps this table's name. Several keys go in a 'c()', and the two forms mix: 'by = c(region, customer_id == id)'.
unmatched	Whose unmatched rows survive: "this" keeps this table's and is the default, "none" keeps neither, "both" keeps both. There is no "other", because that is this join with the tables the other way round.

Value

A pipeline.

Examples

```

sales <- data.frame(
  region = c("West", "East", "West", "North"),
  product = c("Widget", "Gadget", "Doohickey", "Widget"),
  revenue = c(100, 120, 120, 150),
  cost    = c(40, 80, 75, 60)
)
products <- data.frame(
  product = c("Widget", "Gadget"),
  maker   = c("Acme", "Globex")
)
sales |>
  pick(product, revenue) |>
  join(products, by = product) |>
  sort(product)
#>   product revenue maker
#> 1 Doohickey     120  <NA>
#> 2   Gadget     120 Globex
#> 3   Widget     100  Acme
#> 4   Widget     150  Acme

```

keep

Keep the rows where a condition holds

Description

Keep the rows where a condition holds

Usage

```
keep(.data, condition)
```

Arguments

`.data` A table, or a pipeline.
`condition` A condition, written in R: `'region == "West"'`.

Value

A pipeline. Print it, or `'collect()'` it, to get a table.

Examples

```

sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
sales |> keep(region == "West")
#>   region revenue
#> 1   West     100
#> 2   West     150

```

lengthen	<i>Turn columns into rows</i>
----------	-------------------------------

Description

The table grows taller, which is what the name says. Direction goes in the name because nobody could ever remember which of ‘melt’ and ‘cast’ did this.

Usage

```
lengthen(.data, ..., name, value)
```

Arguments

<code>.data</code>	A table, or a pipeline.
<code>...</code>	The columns that become rows. ‘all_but(id)’ names the ones to leave where they are, and ‘where(startsWith(name, "q"))’ chooses them by the shape of their name — the same three ways ‘pick’ chooses columns.
<code>name</code>	What the new column of names is called. A bare name, or the shape of the old names in quotes when they hold more than one thing: ‘name = "{question}_{year}"’. Writing ‘{value}’ for a piece says that piece picks which value column a row belongs to.
<code>value</code>	What the new column of values is called. Left out, the two are called ‘name’ and ‘value’, which are the grammar’s own words for them.

Value

A pipeline.

Examples

```
answers <- data.frame(
  student = c("ann", "bob"),
  q1 = c(1, 4), q2 = c(2, 5)
)
answers |> lengthen(q1, q2)
#>   student name value
#> 1     ann   q1     1
#> 2     ann   q2     2
#> 3     bob   q1     4
#> 4     bob   q2     5
```

pick	<i>Just these columns, or all but these</i>
------	---

Description

Just these columns, or all but these

Usage

```
pick(.data, ...)
```

Arguments

.data	A table, or a pipeline.
...	Column names. Wrap them in 'all_but()' to name the ones to drop instead: 'pick(all_but(cost))'.

Value

A pipeline.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West", "North"),
  product = c("Widget", "Gadget", "Doohickey", "Widget"),
  revenue = c(100, 120, 120, 150),
  cost = c(40, 80, 75, 60)
)
sales |> pick(region, revenue)
#>   region revenue
#> 1  West      100
#> 2  East      120
#> 3  West      120
#> 4 North      150
```

rename	<i>Rename a column</i>
--------	------------------------

Description

The new name goes first, the way it does everywhere else in the grammar and the way assignment reads: 'rename(margin = profit)'.

Usage

```
rename(.data, ...)
```

Arguments

`.data` A table, or a pipeline.
`...` ‘new = old’ pairs.

Value

A pipeline.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
sales |> rename(area = region)
#>   area revenue
#> 1 West     100
#> 2 East     120
#> 3 West     150
```

run	<i>Run a pipeline</i>
-----	-----------------------

Description

Run a pipeline

Usage

```
run(pipeline, ...)
```

Arguments

`pipeline` The pipeline, as text.
`...` Tables, named. Usually unnecessary: the table named at the head of the pipeline is looked up where you are calling from, the way ‘`duckdb.sql("SELECT * FROM df")`’ finds ‘df’ in Python.

Value

A data frame.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
run('
  sales
  then keep where [region] is "West"
', sales = sales)
#>   region revenue
#> 1   West     100
#> 2   West     150
```

show_as

The same pipeline, written in a language you already know

Description

A small vocabulary covers most of what people do and never all of it, so the question is not whether you reach its edge but what happens when you do.

Usage

```
show_as(pipeline, as = "dplyr", ...)
```

Arguments

pipeline	The pipeline, as text.
as	Which language. “sql”, “spark”, “dplyr”, “pandas”, “polars”, “pyspark”, or “god” itself. An unknown name is refused and the message lists the real ones, so this list going stale costs nothing.
...	Tables, named, if the one at the head is not in scope.

Value

The text, invisibly, after printing it.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
show_as(sales |> keep(region == "West"), "dplyr")
#> sales |>
#>   filter((region == "West"))
show_as(sales |> keep(region == "West"), "sql")
#> WITH step0 AS (SELECT * FROM "sales"),
#>   step1 AS (SELECT * FROM step0 WHERE ("region" = 'West'))
#> SELECT * FROM step1
```

show_steps

What a pipeline does to the table, step by step

Description

****Nothing runs.**** The grammar checks the whole sentence against the columns before anything is executed, so this is a picture of what would happen, drawn from the same reading that would refuse a column that is not there.

Usage

```
show_steps(pipeline, ...)
```

Arguments

pipeline	The pipeline, as text or as one built from the verbs.
...	Tables, named, if the one at the head is not in scope.

Details

Every step shows the table as it stands once that step has run, with the columns it makes marked and the ones it takes away marked where they leave. A second table gets a row of its own under the step that reads it, so a join shows what crossed over and what matched. A sentence the grammar refuses is still drawn, as far as it checked, with the refusal under the words that stopped it — which is the question an error message on its own cannot answer.

Value

An object that prints as a ladder at the console and draws itself inside a rendered document. ‘format(x, "svg")’ gives the picture as text, for writing to a file.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
show_steps(sales |> keep(region == "West") |> take(1))
#> sales                region:text  revenue:number
#> keep where ([region] is "West")  region  revenue
#> take 1                        region  revenue
#>      at most 1 rows
```

sort	<i>Order the rows</i>
------	-----------------------

Description

Order the rows

Usage

```
sort(.data, ..., missing = "last")
```

Arguments

.data	A table, or a pipeline.
...	Columns to order by. Wrap one in ‘descending()’ to reverse it. There is deliberately no ‘ascending()’: ascending is what happens when you do not ask for anything.
missing	Where the rows with no value go: “last”, the default, or “first”. It applies to the whole sort rather than to one key, because one of the targets can only say it that way.

Value

A pipeline.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
sales |> sort(descending(revenue))
#>   region revenue
#> 1   West     150
#> 2   East     120
#> 3   West     100
```

summarize	<i>Collapse to one row per group</i>
-----------	--------------------------------------

Description

Collapse to one row per group

Usage

```
summarize(.data, ..., by)
```

Arguments

<code>.data</code>	A table, or a pipeline.
<code>...</code>	Named columns, each one an aggregation: ‘margin = total(margin)’.
<code>by</code>	Columns to group by. Without it the whole table is one group.

Value

A pipeline.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
sales |> summarize(sold = total(revenue), by = region)
#>   region sold
#> 1   East  120
#> 2   West  250
```

take

The first n rows

Description

The first n rows

Usage

```
take(.data, n, by, ties = FALSE)
```

Arguments

<code>.data</code>	A table, or a pipeline.
<code>n</code>	How many. An ordinary R value, so a threshold held in a variable works: ‘take(wanted)’.
<code>by</code>	Columns to group by, for the first n rows of each group. It needs a ‘sort’ before it, because “the first rows” means nothing until something says first by what.
<code>ties</code>	Keep every row level with the last one taken, rather than exactly ‘n’ rows. Needs a ‘sort’ before it, since level on what is otherwise unanswerable, and the row count stops being readable off the sentence. dplyr’s ‘slice_max’ defaults the other way.

Value

A pipeline.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
sales |> take(2)
#>   region revenue
#> 1   West     100
#> 2   East     120
```

take_last

The last n rows, or the last n of each group

Description

The other end of ‘take’. It always needs a ‘sort’ before it, where a bare ‘take’ does not: ”the first rows” of an unsorted table is at least the rows the pipeline reached first, and ”the last rows” is a claim about an end that a table does not have until something says which way it runs.

Usage

```
take_last(.data, n, by, ties = FALSE)
```

Arguments

<code>.data</code>	A table, or a pipeline.
<code>n</code>	How many rows.
<code>by</code>	Columns to group by, for the last n rows of each group.
<code>ties</code>	Keep every row level with the last one taken, rather than exactly ‘n’ rows. Needs a ‘sort’ before it.

Value

A pipeline.

Examples

```
sales <- data.frame(
  region = c("West", "East", "West"),
  revenue = c(100, 120, 150)
)
sales |> sort(revenue) |> take_last(2)
```

use_engine

Run pipelines somewhere other than this machine

Description

****The sentences do not change and neither does the vocabulary.**** What changes is which engine answers them. Point god at a warehouse connection and the same pipeline runs there, against tables it already holds, with nothing copied over.

Usage

```
use_engine(connection = NULL, dialect = c("sql", "spark"))
```

Arguments

connection A ‘DBI’ connection, or ‘NULL’ to use the engine here.
dialect Which SQL to write. “sql” for DuckDB, “spark” for Spark and Databricks.

Details

A connection from ‘sparklyr’, or from ‘odbc’ against a warehouse, is what this is for. Both are ‘DBI’ connections, so god needs to know only two things: the connection, and which dialect of SQL to write for it.

Call it with no arguments to go back to the engine on this machine.

Value

The connection that was in use before, invisibly.

Examples

```
## Not run:

# By default a pipeline runs on DuckDB, in this session.
# Point it somewhere else with a DBI connection:
con <- DBI::dbConnect(duckdb::duckdb())
use_engine(con)

# And back again:
use_engine(NULL)

## End(Not run)
```

widen	<i>Turn rows into columns</i>
-------	-------------------------------

Description

The inverse of ‘lengthen’, reading the same two words the other way: ‘name’ points at the column holding column names in both, and the verb says whether it is being made or read.

Usage

```
widen(.data, name, value, by, missing, giving)
```

Arguments

<code>.data</code>	A table, or a pipeline.
<code>name</code>	The column the new column names come from, or the shape to build them in: ‘name = ”{question}_{year}”’.
<code>value</code>	What fills the cells. A bare column means one row per cell, and the query stops and names the cell if two rows want one. An aggregation says what to do about that instead: ‘value = average(answer)’.
<code>by</code>	The columns that say which rows go together. Left out, it is every column not named above — which is convenient until one stray column makes every row unique, so it is worth writing.
<code>missing</code>	What an empty cell holds. Needs ‘giving’, because saying what an empty cell holds means knowing which cells there are.
<code>giving</code>	The columns this makes. Without it the columns come from the data, which nothing can know before the query runs, so nothing may follow.

Value

A pipeline.

Examples

```
marks <- data.frame(
  student = c("ann", "ann", "bob", "bob"),
  question = c("q1", "q2", "q1", "q2"),
  mark     = c(1, 2, 4, 5)
)
marks |> widen(name = question, value = mark, by = student,
              giving = c(q1, q2))
#>   student q1 q2
#> 1     ann  1  2
#> 2     bob  4  5
```

Index

`add`, [2](#)
`add_combinations`, [3](#)
`add_rows`, [4](#)

`collect`, [5](#)

`drop_duplicates`, [5](#)
`drop_missing`, [6](#)

`fill_missing`, [7](#)
`format.god_pipeline`, [7](#)
`format.god_steps`, [8](#)

`god_sql`, [9](#)
`god_table`, [9](#)

`join`, [10](#)

`keep`, [11](#)

`lengthen`, [12](#)

`pick`, [13](#)

`rename`, [13](#)
`run`, [14](#)

`show_as`, [15](#)
`show_steps`, [16](#)
`sort`, [17](#)
`summarize`, [17](#)

`take`, [18](#)
`take_last`, [19](#)

`use_engine`, [20](#)

`widen`, [21](#)