Package ‘googlesheets4’

October 13, 2022

Title   Access Google Sheets using the Sheets API V4
Version 1.0.1
Description Interact with Google Sheets through the Sheets API v4

<https://developers.google.com/sheets/api>. "API" is an acronym for "application programming interface"; the Sheets API allows users to
interact with Google Sheets programmatically, instead of via a web
browser. The "v4" refers to the fact that the Sheets API is currently
at version 4. This package can read and write both the metadata and
the cell data in a Sheet.

License MIT + file LICENSE

URL https://googlesheets4.tidyverse.org,
https://github.com/tidyverse/googlesheets4

BugReports https://github.com/tidyverse/googlesheets4/issues

Depends  R (>= 3.4)
Imports cellranger, cli (>= 3.0.0), curl, gargle (>= 1.2.0), glue (>=
1.3.0), googledrive (>= 2.0.0), httr, ids, magrittr, methods,
purr, rematch2, rlang (>= 1.0.2), tibble (>= 2.1.1), utils,
vctrs (>= 0.2.3)

Suggests covr, readr, rmarkdown, sodium, spelling, testthat (>=
3.1.3), withr

ByteCompile true

Config/Needs/website tidyverse, tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

Language en-US

RoxygenNote 7.2.1

NeedsCompilation no

Author Jennifer Bryan [cre, aut] (<https://orcid.org/0000-0002-6983-2759>),
RStudio [cph, fnd]

Maintainer Jennifer Bryan <jenny@rstudio.com>
## R topics documented:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell-specification</td>
<td>3</td>
</tr>
<tr>
<td>googlesheets4-configuration</td>
<td>3</td>
</tr>
<tr>
<td>gs4_auth</td>
<td>5</td>
</tr>
<tr>
<td>gs4_auth_configure</td>
<td>7</td>
</tr>
<tr>
<td>gs4_browse</td>
<td>9</td>
</tr>
<tr>
<td>gs4_create</td>
<td>9</td>
</tr>
<tr>
<td>gs4_deauth</td>
<td>11</td>
</tr>
<tr>
<td>gs4_endpoints</td>
<td>12</td>
</tr>
<tr>
<td>gs4_examples</td>
<td>12</td>
</tr>
<tr>
<td>gs4_find</td>
<td>13</td>
</tr>
<tr>
<td>gs4_fodder</td>
<td>14</td>
</tr>
<tr>
<td>gs4_formula</td>
<td>15</td>
</tr>
<tr>
<td>gs4_get</td>
<td>16</td>
</tr>
<tr>
<td>gs4_has_token</td>
<td>17</td>
</tr>
<tr>
<td>gs4_random</td>
<td>17</td>
</tr>
<tr>
<td>gs4_token</td>
<td>18</td>
</tr>
<tr>
<td>gs4_user</td>
<td>19</td>
</tr>
<tr>
<td>range_autofit</td>
<td>21</td>
</tr>
<tr>
<td>range_delete</td>
<td>22</td>
</tr>
<tr>
<td>range_flood</td>
<td>24</td>
</tr>
<tr>
<td>range_read</td>
<td>27</td>
</tr>
<tr>
<td>range_read_cells</td>
<td>29</td>
</tr>
<tr>
<td>range_speedread</td>
<td>31</td>
</tr>
<tr>
<td>range_write</td>
<td>33</td>
</tr>
<tr>
<td>request_generate</td>
<td>35</td>
</tr>
<tr>
<td>request_make</td>
<td>36</td>
</tr>
<tr>
<td>sheets_id</td>
<td>37</td>
</tr>
<tr>
<td>sheet_add</td>
<td>39</td>
</tr>
<tr>
<td>sheet_append</td>
<td>40</td>
</tr>
<tr>
<td>sheet_delete</td>
<td>42</td>
</tr>
<tr>
<td>sheet_properties</td>
<td>44</td>
</tr>
<tr>
<td>sheet_relocate</td>
<td>45</td>
</tr>
<tr>
<td>sheet_rename</td>
<td>46</td>
</tr>
<tr>
<td>sheet_resize</td>
<td>48</td>
</tr>
<tr>
<td>sheet_write</td>
<td>49</td>
</tr>
<tr>
<td>spread_sheet</td>
<td>51</td>
</tr>
</tbody>
</table>

### Index

53
cell-specification

Specify cells

Description

Many functions in googlesheets4 use a range argument to target specific cells. The Sheets v4 API expects user-specified ranges to be expressed via its A1 notation, but googlesheets4 accepts and converts a few alternative specifications provided by the functions in the cellranger package. Of course, you can always provide A1-style ranges directly to functions like `read_sheet()` or `range_read_cells()`. Why would you use the cellranger helpers? Some ranges are practically impossible to express in A1 notation, specifically when you want to describe rectangles with some bounds that are specified and others determined by the data.

Examples

```r
ss <- gs4_example("mini-gap")

# Specify only the rows or only the columns
read_sheet(ss, range = cell_rows(1:3))
read_sheet(ss, range = cell_cols("C:D"))
read_sheet(ss, range = cell_cols(1))

# Specify upper or lower bound on row or column
read_sheet(ss, range = cell_rows(c(NA, 4)))
read_sheet(ss, range = cell_cols(c(NA, "D")))
read_sheet(ss, range = cell_rows(c(3, NA)))
read_sheet(ss, range = cell_cols(c(2, NA)))
read_sheet(ss, range = cell_cols(c("C", NA)))

# Specify a partially open rectangle
read_sheet(ss, range = cell_limits(c(2, 3), c(NA, NA)), col_names = FALSE)
read_sheet(ss, range = cell_limits(c(1, 2), c(NA, 4)))
```

googlebooks4-configuration

Description

Some aspects of googlesheets4 behaviour can be controlled via an option.

Usage

```
local_gs4_quiet(env = parent.frame())
with_gs4_quiet(code)
```
Arguments

env  The environment to use for scoping

code  Code to execute quietly

Messages

The googlesheets4_quiet option can be used to suppress messages from googlesheets4. By default, googlesheets4 always messages, i.e. it is not quiet.

Set googlesheets4_quiet to TRUE to suppress messages, by one of these means, in order of decreasing scope:

- Put options(googlesheets4_quiet = TRUE) in a start-up file, such as .Rprofile, or in your R script
- Use local_gs4_quiet() to silence googlesheets4 in a specific scope
- Use with_gs4_quiet() to run a small bit of code silently

local_gs4_quiet() and with_gs4_quiet() follow the conventions of the the withr package (https://withr.r-lib.org).

Auth

Read about googlesheets4’s main auth function, gs4_auth(). It is powered by the gargle package, which consults several options:

- Default Google user or, more precisely, email: see gargle::gargle_oauth_email()
- Whether or where to cache OAuth tokens: see gargle::gargle_oauth_cache()
- Whether to prefer "out-of-band" auth: see gargle::gargle_oob_default()
- Application Default Credentials: see gargle::credentials_app_default()

Examples

```r
# message: "Creating new Sheet ...
(ss <- gs4_create("gs4-quiet-demo", sheets = "alpha"))

# message: "Editing ..., Writing ...
range_write(ss, data = data.frame(x = 1, y = "a"))

# suppress messages for a small amount of code
with_gs4_quiet(
  ss %>% sheet_append(data.frame(x = 2, y = "b"))
)

# message: "Writing ..., Appending ...
ss %>% sheet_append(data.frame(x = 3, y = "c"))

# suppress messages until end of current scope
local_gs4_quiet()
ss %>% sheet_append(data.frame(x = 4, y = "d"))
```
Authorize googlesheets4

Authorize googlesheets4 to view and manage your Google Sheets. This function is a wrapper around gargle::token_fetch().

By default, you are directed to a web browser, asked to sign in to your Google account, and to grant googlesheets4 permission to operate on your behalf with Google Sheets. By default, with your permission, these user credentials are cached in a folder below your home directory, from where they can be automatically refreshed, as necessary. Storage at the user level means the same token can be used across multiple projects and tokens are less likely to be synced to the cloud by accident.

If you are interacting with R within a browser (applies to RStudio Server, RStudio Workbench, and RStudio Cloud), you need a variant of this flow, known as out-of-band auth ("oob"). If this does not happen automatically, you can request it yourself with use_oob = TRUE or, more persistently, by setting an option via options(gargle_oob_default = TRUE).

Usage

```r
gs4_auth(
  email = gargle::gargle_oauth_email(),
  path = NULL,
  scopes = "https://www.googleapis.com/auth/spreadsheets",
  cache = gargle::gargle_oauth_cache(),
  use_oob = gargle::gargle_oob_default(),
  token = NULL
)
```

Arguments

- `email` Optional. Allows user to target a specific Google identity. If specified, this is used for token lookup, i.e. to determine if a suitable token is already available in the cache. If no such token is found, email is used to pre-select the targeted Google identity in the OAuth chooser. Note, however, that the email associated with a token when it’s cached is always determined from the token itself, never from this argument. Use NA or FALSE to match nothing and force the OAuth dance in the browser. Use TRUE to allow email auto-discovery, if exactly one matching token is found in the cache. Specify just the domain
with a glob pattern, e.g. "*@example.com", to create code that "just works" for both alice@example.com and bob@example.com. Defaults to the option named "gargle_oauth_email", retrieved by `gargle_oauth_email()`.

**path**

JSON identifying the service account, in one of the forms supported for the `txt` argument of `jsonlite::fromJSON()` (typically, a file path or JSON string).

**scopes**

A character vector of scopes to request. Pick from those listed at https://developers.google.com/identity/protocols/oauth2/scopes. For certain token flows, the "https://www.googleapis.com/auth/userinfo.email" scope is unconditionally included. This grants permission to retrieve the email address associated with a token; gargle uses this to index cached OAuth tokens. This grants no permission to view or send email and is generally considered a low-value scope.

**cache**

Specifies the OAuth token cache. Defaults to the option named "gargle_oauth_cache", retrieved via `gargle_oauth_cache()`.

**use_oob**

Whether to prefer "out of band" authentication. Defaults to the option named "gargle_oob_default", retrieved via `gargle_oob_default()`.

**token**

A token with class `Token2.0` or an object of `httr`'s class `request`, i.e. a token that has been prepared with `httr::config()` and has a `Token2.0` in the `auth_token` component.

**Details**

Most users, most of the time, do not need to call `gs4_auth()` explicitly – it is triggered by the first action that requires authorization. Even when called, the default arguments often suffice. However, when necessary, this function allows the user to explicitly:

- Declare which Google identity to use, via an email address. If there are multiple cached tokens, this can clarify which one to use. It can also force googlesheets4 to switch from one identity to another. If there's no cached token for the email, this triggers a return to the browser to choose the identity and give consent. You can specify just the domain by using a glob pattern. This means that a script containing `email = "*@example.com"` can be run without further tweaks on the machine of either alice@example.com or bob@example.com.
- Use a service account token or workload identity federation.
- Bring their own `Token2.0`.
- Specify non-default behavior re: token caching and out-of-bound authentication.
- Customize scopes.

For details on the many ways to find a token, see `gargle::token_fetch()`. For deeper control over auth, use `gs4_auth_configure()` to bring your own OAuth app or API key. Read more about gargle options, see `gargle::gargle_options`.

**See Also**

Other auth functions: `gs4_auth_configure()`, `gs4_deauth()`
Examples

```r
# load/refresh existing credentials, if available
# otherwise, go to browser for authentication and authorization
gs4_auth()

# force use of a token associated with a specific email
gs4_auth(email = "jenny@example.com")

# use a 'read only' scope, so it's impossible to edit or delete Sheets
gs4_auth(
  scopes = "https://www.googleapis.com/auth/spreadsheets.readonly"
)

# use a service account token
gs4_auth(path = "foofy-83ee9e7c9c48.json")
```

 gs4_auth_configure  Edit and view auth configuration

Description

These functions give more control over and visibility into the auth configuration than gs4_auth() does. gs4_auth_configure() lets the user specify their own:

- OAuth app, which is used when obtaining a user token.
- API key. If googlesheets4 is de-authorized via gs4_deauth(), all requests are sent with an API key in lieu of a token. See the vignette How to get your own API credentials for more. If the user does not configure these settings, internal defaults are used. gs4_oauth_app() and gs4_api_key() retrieve the currently configured OAuth app and API key, respectively.

Usage

```r
gs4_auth_configure(app, path, api_key)

gs4_api_key()

gs4_oauth_app()
```

Arguments

- **app**: OAuth app, in the sense of `httr::oauth_app()`.
- **path**: JSON downloaded from Google Cloud Platform Console, containing a client id (aka key) and secret, in one of the forms supported for the `txt` argument of `jsonlite::fromJSON()` (typically, a file path or JSON string).
- **api_key**: API key.
gs4_auth_configure

Value

- `gs4_auth_configure()`: An object of R6 class `gargle::AuthState`, invisibly.
- `gs4_oauth_app()`: the current user-configured `httr::oauth_app()`.
- `gs4_api_key()`: the current user-configured API key.

See Also

Other auth functions: `gs4_auth()`, `gs4_deauth()`

Examples

```r
# see and store the current user-configured OAuth app (probably 'NULL')
(original_app <- gs4_oauth_app())

# see and store the current user-configured API key (probably 'NULL')
(original_api_key <- gs4_api_key())

if (require(httr)) {
  # bring your own app via client id (aka key) and secret
  google_app <- httr::oauth_app(
    "my-awesome-google-api-wrapping-package",
    key = "YOUR_CLIENT_ID_GOES_HERE",
    secret = "YOUR_SECRET_GOES_HERE"
  )
  google_key <- "YOUR_API_KEY"
  gs4_auth_configure(app = google_app, api_key = google_key)

  # confirm the changes
  gs4_oauth_app()
  gs4_api_key()

  # bring your own app via JSON downloaded from Google Developers Console
  # this file has the same structure as the JSON from Google
  app_path <- system.file(
    "extdata", "fake-oauth-client-id-and-secret.json",
    package = "googlesheets4"
  )
  gs4_auth_configure(path = app_path)

  # confirm the changes
  gs4_oauth_app()
}

# restore original auth config
gs4_auth_configure(app = original_app, api_key = original_api_key)
```
gs4_browse Visit a Sheet in a web browser

Description
Visits a Google Sheet in your default browser, if session is interactive.

Usage

\[
gs4\textunderscore browse(ss)
\]

Arguments

\[
ss \quad \text{Something that identifies a Google Sheet:}
\]

- its file id as a string or \texttt{drive\_id}
- a URL from which we can recover the id
- a one-row \texttt{dribble}, which is how googledrive represents Drive files
- an instance of \texttt{googlesheets4\_spreadsheet}, which is what \texttt{gs4\_get()} returns

Processed through \texttt{as\_sheets\_id()}.

Value
The Sheet’s browser URL, invisibly.

Examples

\[
gs4\_example("mini\_gap") \%>% gs4\_browse()
\]

gs4_create Create a new Sheet

Description
Creates an entirely new (spread)Sheet (or, in Excel-speak, workbook). Optionally, you can also provide names and/or data for the initial set of (work)sheets. Any initial data provided via \texttt{sheets} is styled as a table, as described in \texttt{sheet\_write()}.

Usage

\[
gs4\_create(name = gs4\_random(), ..., sheets = NULL)
\]
Arguments

name
   The name of the new spreadsheet.

... Optional spreadsheet properties that can be set through this API endpoint, such
   as locale and time zone.

sheets
   Optional input for initializing (work)sheets. If unspecified, the Sheets API auto-
   matically creates an empty "Sheet1". You can provide a vector of sheet names,
   a data frame, or a (possibly named) list of data frames. See the examples.

Value

The input ss, as an instance of sheets_id

See Also

Wraps the spreadsheets.create endpoint:


There is an article on writing Sheets:

- https://googlesheets4.tidyverse.org/articles/articles/write-sheets.html

Other write functions: gs4_formula(), range_delete(), range_flood(), range_write(), sheet_append(), sheet_write()

Examples

gs4_create("gs4-create-demo-1")

gs4_create("gs4-create-demo-2", locale = "en_CA")

gs4_create(
   "gs4-create-demo-3",
   locale = "fr_FR",
   timeZone = "Europe/Paris"
)

gs4_create(
   "gs4-create-demo-4",
   sheets = c("alpha", "beta")
)

my_data <- data.frame(x = 1)
ge4_create(
   "gs4-create-demo-5",
   sheets = my_data
)

gs4_create(
   "gs4-create-demo-6",
gs4_deauth

    sheets = list(chickwts = head(chickwts), mtcars = head(mtcars))

    # Clean up
gs4_find("gs4-create-demo") %>%
    googledrive::drive_trash()

 gs4_deauth  Suspend authorization

Description

Put googlesheets4 into a de-authorized state. Instead of sending a token, googlesheets4 will send an
API key. This can be used to access public resources for which no Google sign-in is required. This
is handy for using googlesheets4 in a non-interactive setting to make requests that do not require
a token. It will prevent the attempt to obtain a token interactively in the browser. The user can
configure their own API key via \texttt{gs4_auth_configure()} and retrieve that key via \texttt{gs4_api_key()}. In the absence of a user-configured key, a built-in default key is used.

Usage

\texttt{gs4_deauth()}

See Also

Other auth functions: \texttt{gs4_auth_configure()}, \texttt{gs4_auth()}

Examples

\texttt{gs4_deauth()}
\texttt{gs4_user()}

\texttt{# get metadata on the public 'deaths' spreadsheet}
\texttt{gs4_example("deaths") %>%
  gs4_get()}

\texttt{# Load the Sheet "deaths" of the public Google spreadsheet "deaths"}
\texttt{gs4_example("deaths") %>%
  gs4_get()}
gs4_endpoints  List Sheets endpoints

Description

Returns a list of selected Sheets API v4 endpoints, as stored inside the googlesheets4 package. The names of this list (or the id sub-elements) are the nicknames that can be used to specify an endpoint in `request_generate()`. For each endpoint, we store its nickname or id, the associated HTTP method, the path, and details about the parameters. This list is derived programmatically from the Sheets API v4 Discovery Document (https://www.googleapis.com/discovery/v1/apis/sheets/v4/rest).

Usage

gs4_endpoints(i = NULL)

Arguments

i  The name(s) or integer index(ices) of the endpoints to return. Optional. By default, the entire list is returned.

Value

A list containing some or all of the subset of the Sheets API v4 endpoints that are used internally by googlesheets4.

Examples

str(gs4_endpoints(), max.level = 2)
gs4_endpoints("sheets.spreadsheets.values.get")
gs4_endpoints(4)

gs4_examples  Example Sheets

Description

googlesheets4 makes a variety of world-readable example Sheets available for use in documentation and reproxes. These functions help you access the example Sheets. See `vignette("example-sheets", package = "googlesheets4")` for more.

Usage

```
gs4_examples(matches)
gs4_example(matches)
```
Arguments

matches

A regular expression that matches the name of the desired example Sheet(s). matches is optional for the plural gs4_examples() and, if provided, it can match multiple Sheets. The singular gs4_example() requires matches and it must match exactly one Sheet.

Value

• gs4_example(): a sheets_id
• gs4_examples(): a named vector of all built-in examples, with class drive_id

Examples

gs4_examples()
gs4_examples("gap")

gs4_example("gapminder")
gs4_example("deaths")

Description

Finds your Google Sheets. This is a very thin wrapper around googledrive::drive_find(), that specifies you want to list Drive files where type = "spreadsheet". Therefore, note that this will require auth for googledrive! See the article Using googlesheets4 with googledrive if you want to coordinate auth between googlesheets4 and googledrive. This function will emit an informational message if you are currently logged in with both googlesheets4 and googledrive, but as different users.

Usage

gs4_find(...)

Arguments

Arguments (other than type, which is hard-wired as type = "spreadsheet") that are passed along to googledrive::drive_find().

Value

An object of class dribble, a tibble with one row per file.
Examples

```r
# see all your Sheets
gs4_find()

# see 5 Sheets, prioritized by creation time
x <- gs4_find(order_by = "createdTime desc", n_max = 5)
x

# hoist the creation date, using other packages in the tidyverse
# x %>%
#   tidyr::hoist(drive_resource, created_on = "createdTime") %>%
#   dplyr::mutate(created_on = as.Date(created_on))
```

---

**gs4_fodder**  
*Create useful spreadsheet filler*

---

**Description**

Creates a data frame that is useful for filling a spreadsheet, when you just need a sheet to experiment with. The data frame has \(n\) rows and \(m\) columns with these properties:

- Inner cell values reflect the coordinates where each value will land in the sheet, in A1-notation. So the first row is "B2", "C2", and so on. Note that this \(n\)-row data frame will occupy \(n + 1\) rows in the sheet, because the column names occupy the first row.

**Usage**

```r
gs4_fodder(n = 10, m = n)
```

**Arguments**

- \(n\)  
  Number of rows.
- \(m\)  
  Number of columns.

**Value**

A data frame of character vectors.

**Examples**

```r
gs4_fodder()
gs4_fodder(5, 3)
```
Class for Google Sheets formulas

**Description**

In order to write a formula into Google Sheets, you need to store it as an object of class `googlesheets4_formula`. This is how we distinguish a “regular” character string from a string that should be interpreted as a formula. `googlesheets4_formula` is an S3 class implemented using the `vctrs` package.

**Usage**

```r
gs4_formula(x = character())
```

**Arguments**

- `x`  
  Character.

**Value**

An S3 vector of class `googlesheets4_formula`.

**See Also**

Other write functions: `gs4_create()`, `range_delete()`, `range_flood()`, `range_write()`, `sheet_append()`, `sheet_write()`

**Examples**

```r
dat <- data.frame(x = c(1, 5, 3, 2, 4, 6))

ss <- gs4_create("gs4-formula-demo", sheets = dat)

summaries <- tibble::tribble(
  ~desc, ~summaries,
  "max", "=max(A:A)",
  "sum", "=sum(A:A)",
  "min", "=min(A:A)",
  "sparkline", "=SPARKLINE(A:A, \\
  "{color", \\
  "blue\}")"
)

# explicitly declare a column as `googlesheets4_formula`
summaries$summaries <- gs4_formula(summaries$summaries)

range_write(ss, data = summaries, range = "C1", reformat = FALSE)

miscellany <- tibble::tribble(
  ~desc, ~miscellany,
  "one", "=sum(A:A)"
)
```
-desc, -example,
"hyperlink", "=HYPERLINK("http://www.google.com/", "Google")",
"image", "=IMAGE("https://www.google.com/images/srpr/logo3w.png")"
}
miscellany.example <- gs4_formula(miscellany.example)
miscellany

sheet_write(miscellany, ss = ss)

# clean up
gs4_find("gs4-formula-demo") %>%
googledrive::drive_trash()

---

**gs4_get**

*Get Sheet metadata*

**Description**

Retrieve spreadsheet-specific metadata, such as details on the individual (work)sheets or named ranges.

- `gs4_get()` complements `googledrive::drive_get()`, which returns metadata that exists for any file on Drive.

**Usage**

```r
gs4_get(ss)
```

**Arguments**

- **ss**
  
  Something that identifies a Google Sheet:
  
  - its file id as a string or `drive_id`
  - a URL from which we can recover the id
  - a one-row `dribble`, which is how googledrive represents Drive files
  - an instance of `googlesheets4_spreadsheet`, which is what `gs4_get()` returns
  
  Processed through `as_sheets_id()`.

**Value**

A list with S3 class `googlesheets4_spreadsheet`, for printing purposes.

**See Also**

Wraps the `spreadsheets.get` endpoint:

Examples

```r
gs4_get(gs4_example("mini-gap"))
```

---

**Description**

Reports whether googlesheets4 has stored a token, ready for use in downstream requests.

**Usage**

```r
gs4_has_token()
```

**Value**

Logical.

**See Also**

Other low-level API functions: `gs4_token()`, `request_generate()`, `request_make()`

**Examples**

```r
gs4_has_token()
```

---

**gs4_random**

*Generate a random Sheet name*

---

**Description**

Generates a random name, suitable for a newly created Sheet, using `ids::adjective_animal()`.

**Usage**

```r
gs4_random(n = 1)
```

**Arguments**

- `n` Number of names to generate.

**Value**

A character vector.
Examples

gs4_random()

Description

For internal use or for those programming around the Sheets API. Returns a token pre-processed
with `httr::config()`. Most users do not need to handle tokens "by hand" or, even if they need
some control, `gs4_auth()` is what they need. If there is no current token, `gs4_auth()` is called to
either load from cache or initiate OAuth2.0 flow. If auth has been deactivated via `gs4_deauth()`,
gs4_token() returns NULL.

Usage

gs4_token()

Value

A request object (an S3 class provided by `httr`).

See Also

Other low-level API functions: `gs4_has_token()`, `request_generate()`, `request_make()`

Examples

```r
req <- request_generate(
    "sheets.spreadsheets.get",
    list(spreadsheetId = "abc"),
    token = gs4_token()
)
req
```
gs4_user  

Get info on current user

Description
Reveals the email address of the user associated with the current token. If no token has been loaded yet, this function does not initiate auth.

Usage

```r
gs4_user()
```

Value
An email address or, if no token has been loaded, NULL.

See Also
gargle::token_userinfo(), gargle::token_email(), gargle::token_tokeninfo()

Examples

```r
gs4_user()
```

range_autofit  

Auto-fit columns or rows to the data

Description
Applies automatic resizing to either columns or rows of a (work)sheet. The width or height of targeted columns or rows, respectively, is determined from the current cell contents. This only affects the appearance of a sheet in the browser and doesn't affect its values or dimensions in any way.

Usage

```r
range_autofit(ss, sheet = NULL, range = NULL, dimension = c("columns", "rows"))
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| ss       | Something that identifies a Google Sheet:  
- its file id as a string or `drive_id`  
- a URL from which we can recover the id  
- a one-row `dribble`, which is how googledrive represents Drive files  
- an instance of `googlesheets4_spreadsheet`, which is what `gs4_get()` returns |
Processed through `as_sheets_id()`.

**sheet**
Sheet to modify, in the sense of "worksheet" or "tab". You can identify a sheet by name, with a string, or by position, with a number. Ignored if the sheet is specified via `range`. If neither argument specifies the sheet, defaults to the first visible sheet.

**range**
Which columns or rows to resize. Optional. If you want to resize all columns or all rows, use `dimension` instead. All the usual range specifications are accepted, but the targeted range must specify only columns (e.g. "B:F") or only rows (e.g. "2:7").

**dimension**
Ignored if `range` is given. If consulted, `dimension` must be either "columns" (the default) or "rows". This is the simplest way to request auto-resize for all columns or all rows.

**Value**
The input `ss`, as an instance of `sheets_id`.

**See Also**
Makes an `AutoResizeDimensionsRequest`:
- [https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#autoresizedimensionsrequest](https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#autoresizedimensionsrequest)

**Examples**

```r
dat <- tibble::tibble(
  fruit = c("date", "lime", "pear", "plum")
)

ss <- gs4_create("range-autofit-demo", sheets = dat)
ss

# open in the browser
gs4_browse(ss)

# shrink column A to fit the short fruit names
range_autofit(ss)
# in the browser, notice how the column width shrank

# send some longer fruit names
dat2 <- tibble::tibble(
  fruit = c("cucumber", "honeydew")
)
ss %>% sheet_append(dat2)
# in the browser, see that column A is now too narrow to show the data

range_autofit(ss)
# in the browser, see the column A reveals all the data now
```
range_delete

# clean up
gs4_find("range-autofit-demo") %>%
googledrive::drive_trash()

---

range_delete  Delete cells

Description

Deletes a range of cells and shifts other cells into the deleted area. There are several related tasks that are implemented by other functions:

- To clear cells of their value and/or format, use `range_clear()`.
- To delete an entire (work)sheet, use `sheet_delete()`.
- To change the dimensions of a (work)sheet, use `sheet_resize()`.

Usage

```r
range_delete(ss, sheet = NULL, range, shift = NULL)
```

Arguments

- **ss**: Something that identifies a Google Sheet:
  - its file id as a string or `drive_id`
  - a URL from which we can recover the id
  - a one-row `dribble`, which is how googledrive represents Drive files
  - an instance of `googlesheets4_spreadsheet`, which is what `gs4_get()` returns
    Processed through `as_sheets_id()`.

- **sheet**: Sheet to delete, in the sense of "worksheet" or "tab". You can identify a sheet by name, with a string, or by position, with a number. Ignored if the sheet is specified via `range`. If neither argument specifies the sheet, defaults to the first visible sheet.

- **range**: Cells to delete. There are a couple differences between `range` here and how it works in other functions (e.g. `range_read()`):
  - range must be specified.
  - range must not be a named range.
  - range must not be the name of a (work)sheet. Instead, use `sheet_delete()` to delete an entire sheet. Row-only and column-only ranges are especially relevant, such as "2:6" or "D". Remember you can also use the helpers in `cell-specification`, such as `cell_cols(4:6)`, or `cell_rows(5)`.

- **shift**: Must be one of "up" or "left", if specified. Required if `range` is NOT a rows-only or column-only range (in which case, we can figure it out for you). Determines whether the deleted area is filled by shifting surrounding cells up or to the left.
Value

The input ss, as an instance of `sheets_id`

See Also

Makes a DeleteRangeRequest:

- [https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#DeleteRangeRequest](https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#DeleteRangeRequest)

Other write functions: `gs4_create()`, `gs4_formula()`, `range_flood()`, `range_write()`, `sheet_append()`, `sheet_write()`

Examples

```r
# create a data frame to use as initial data
df <- gs4_fodder(10)

# create Sheet
ss <- gs4_create("range-delete-example", sheets = list(df))

# delete some rows
range_delete(ss, range = "2:4")

# delete a column
range_delete(ss, range = "C")

# delete a rectangle and specify how to shift remaining cells
range_delete(ss, range = "B3:F4", shift = "left")

# clean up
gs4_find("range-delete-example") %>%
googledrive::drive_trash()
```

---

**range_flood**

*Flood or clear a range of cells*

Description

`range_flood()` "floods" a range of cells with the same content. `range_clear()` is a wrapper that handles the common special case of clearing the cell value. Both functions, by default, also clear the format, but this can be specified via `reformat`.

Usage

```r
range_flood(ss, sheet = NULL, range = NULL, cell = NULL, reformat = TRUE)

range_clear(ss, sheet = NULL, range = NULL, reformat = TRUE)
```
Arguments

**ss**
Something that identifies a Google Sheet:
- its file id as a string or `drive_id`
- a URL from which we can recover the id
- a one-row `dribble`, which is how googledrive represents Drive files
- an instance of `googlesheets4_spreadsheet`, which is what `gs4_get()` returns

Processed through `as_sheets_id()`.

**sheet**
Sheet to write into, in the sense of "worksheet" or "tab". You can identify a sheet by name, with a string, or by position, with a number.

**range**
A cell range to read from. If `NULL`, all non-empty cells are read. Otherwise specify `range` as described in Sheets A1 notation or using the helpers documented in `cell-specification`. Sheets uses fairly standard spreadsheet range notation, although a bit different from Excel. Examples of valid ranges: "Sheet1!A1:B2", "Sheet1!A:A", "Sheet1!:1:2", "Sheet1!A5:A", "A1:B2", "Sheet1". Interpreted strictly, even if the range forces the inclusion of leading, trailing, or embedded empty rows or columns. Takes precedence over `skip`, `n_max` and `sheet`.

Note `range` can be a named range, like "sales_data", without any cell reference.

**cell**
The value to fill the cells in the `range` with. If unspecified, the default of `NULL` results in clearing the existing value.

**reformat**
Logical, indicates whether to reformat the affected cells. Currently googlesheets4 provides no real support for formatting, so `reformat = TRUE` effectively means that edited cells become unformatted.

Value

The input `ss`, as an instance of `sheets_id`

See Also

Makes a `RepeatCellRequest`:

- https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#repeatcellrequest

Other write functions: `gs4_create()`, `gs4_formula()`, `range_delete()`, `range_write()`, `sheet_append()`, `sheet_write()`

Examples

```r
# create a data frame to use as initial data
df <- gs4_fodder(10)

# create Sheet
ss <- gs4_create("range-flood-demo", sheets = list(df))
```
range_read

Read a Sheet into a data frame

Description

This is the main "read" function of the googlesheets4 package. It goes by two names, because we want it to make sense in two contexts:

- \texttt{read_sheet()} evokes other table-reading functions, like \texttt{readr::read_csv()} and \texttt{readxl::read_excel()}. The sheet in this case refers to a Google (spread)Sheet.
- \texttt{range_read()} is the right name according to the naming convention used throughout the googlesheets4 package.

\texttt{read_sheet()} and \texttt{range_read()} are synonyms and you can use either one.
Usage

range_read(
  ss,
  sheet = NULL,
  range = NULL,
  col_names = TRUE,
  col_types = NULL,
  na = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  guess_max = min(1000, n_max),
  .name_repair = "unique"
)

read_sheet(
  ss,
  sheet = NULL,
  range = NULL,
  col_names = TRUE,
  col_types = NULL,
  na = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  guess_max = min(1000, n_max),
  .name_repair = "unique"
)

Arguments

ss Something that identifies a Google Sheet:
  • its file id as a string or drive_id
  • a URL from which we can recover the id
  • a one-row dribble, which is how googledrive represents Drive files
  • an instance of googlesheets4_spreadsheet, which is what gs4_get() returns
  Processed through as_sheets_id().

sheet Sheet to read, in the sense of "worksheet" or "tab". You can identify a sheet by name, with a string, or by position, with a number. Ignored if the sheet is specified via range. If neither argument specifies the sheet, defaults to the first visible sheet.

range A cell range to read from. If NULL, all non-empty cells are read. Otherwise specify range as described in Sheets A1 notation or using the helpers documented in cell-specification. Sheets uses fairly standard spreadsheet range notation, although a bit different from Excel. Examples of valid ranges: "Sheet1!A1:B2", 
"Sheet1!A:A", "Sheet1!1:2", "Sheet1!A5:A", "A1:B2", "Sheet1". Interpreted strictly, even if the range forces the inclusion of leading, trailing, or embedded empty rows or columns. Takes precedence over skip, n_max and sheet. Note range can be a named range, like "sales_data", without any cell reference.

col_names
TRUE to use the first row as column names, FALSE to get default names, or a character vector to provide column names directly. If user provides col_types, col_names can have one entry per column or one entry per unskipped column.

col_types
Column types. Either NULL to guess all from the spreadsheet or a string of readr-style shortcodes, with one character or code per column. If exactly one col_type is specified, it is recycled. See Column Specification for more.

na
Character vector of strings to interpret as missing values. By default, blank cells are treated as missing data.

trim_ws
Logical. Should leading and trailing whitespace be trimmed from cell contents?

skip
Minimum number of rows to skip before reading anything, be it column names or data. Leading empty rows are automatically skipped, so this is a lower bound. Ignored if range is given.

n_max
Maximum number of data rows to parse into the returned tibble. Trailing empty rows are automatically skipped, so this is an upper bound on the number of rows in the result. Ignored if range is given. n_max is imposed locally, after reading all non-empty cells, so, if speed is an issue, it is better to use range.

guess_max
Maximum number of data rows to use for guessing column types.

.name_repair
Handling of column names. By default, googlesheets4 ensures column names are not empty and are unique. There is full support for .name_repair as documented in tibble::tibble().

Value
A tibble

Column Specification

Column types must be specified in a single string of readr-style short codes, e.g. "cci?!" means "character, character, integer, guess, logical". This is not where googlesheets4’s col spec will end up, but it gets the ball rolling in a way that is consistent with readr and doesn’t reinvent any wheels.

Shortcodes for column types:

• _ or -. Skip. Data in a skipped column is still requested from the API (the high-level functions in this package are rectangle-oriented), but is not parsed into the data frame output.
• ?: Guess. A type is guessed for each cell and then a consensus type is selected for the column. If no atomic type is suitable for all cells, a list-column is created, in which each cell is converted to an R object of "best" type. If no column types are specified, i.e. col_types = NULL, all types are guessed.
• 1: Logical.
• i: Integer. This type is never guessed from the data, because Sheets have no formal cell type for integers.
• d or n: Numeric, in the sense of "double".
• D: Date. This type is never guessed from the data, because date cells are just serial datetimes that bear a "date" format.
• t: Time of day. This type is never guessed from the data, because time cells are just serial datetimes that bear a "time" format. Not implemented yet; returns POSIXct.
• T: Datetime, specifically POSIXct.
• c: Character.
• C: Cell. This type is unique to googlesheets4. This returns raw cell data, as an R list, which consists of everything sent by the Sheets API for that cell. Has S3 type of "CELL_SOMETHING" and "SHEETS_CELL". Mostly useful internally, but exposed for those who want direct access to, e.g., formulas and formats.
• L: List, as in "list-column". Each cell is a length-1 atomic vector of its discovered type.
• Still to come: duration (code will be : ) and factor (code will be f).

Examples

ss <- gs4_example("deaths")
read_sheet(ss, range = "A5:F15")
read_sheet(ss, range = "other!A5:F15", col_types = "ccilDD")
read_sheet(ss, range = "arts_data", col_types = "ccilDD")

read_sheet(gs4_example("mini-gap"))
read_sheet(
  gs4_example("mini-gap"),
  sheet = "Europe",
  range = "A:D",
  col_types = "ccid"
)

range_read_cells  Read cells from a Sheet

Description

This low-level function returns cell data in a tibble with one row per cell. This tibble has integer variables row and col (referring to location with the Google Sheet), an A1-style reference loc, and a cell list-column. The flagship function read_sheet(), a.k.a. range_read(), is what most users are looking for, rather than range_read_cells(). read_sheet() is basically range_read_cells() (this function), followed by spread_sheet(), which looks after reshaping and column typing. But if you really want raw cell data from the API, range_read_cells() is for you!
range_read_cells

Usage

```r
range_read_cells(
  ss,
  sheet = NULL,
  range = NULL,
  skip = 0,
  n_max = Inf,
  cell_data = c("default", "full"),
  discard_empty = TRUE
)
```

Arguments

- **ss**: Something that identifies a Google Sheet:
  - its file id as a string or `drive_id`
  - a URL from which we can recover the id
  - a one-row `dribble`, which is how googledrive represents Drive files
  - an instance of `googlesheets4_spreadsheet`, which is what `gs4_get()` returns
    Processed through `as_sheets_id()`.
- **sheet**: Sheet to read, in the sense of "worksheet" or "tab". You can identify a sheet by name, with a string, or by position, with a number. Ignored if the sheet is specified via `range`. If neither argument specifies the sheet, defaults to the first visible sheet.
- **range**: A cell range to read from. If `NULL`, all non-empty cells are read. Otherwise specify `range` as described in Sheets A1 notation or using the helpers documented in cell-specification. Sheets uses fairly standard spreadsheet range notation, although a bit different from Excel. Examples of valid ranges: "Sheet1!A1:B2", "Sheet1!A:A", "Sheet1!!2", "Sheet1!!A5:A", "A1:B2", "Sheet1". Interpreted strictly, even if the range forces the inclusion of leading, trailing, or embedded empty rows or columns. Takes precedence over `skip`, `n_max` and `sheet`. Note range can be a named range, like "sales_data", without any cell reference.
- **skip**: Minimum number of rows to skip before reading anything, be it column names or data. Leading empty rows are automatically skipped, so this is a lower bound. Ignored if range is given.
- **n_max**: Maximum number of data rows to parse into the returned tibble. Trailing empty rows are automatically skipped, so this is an upper bound on the number of rows in the result. Ignored if range is given. `n_max` is imposed locally, after reading all non-empty cells, so, if speed is an issue, it is better to use `range`.
- **cell_data**: How much detail to get for each cell. "default" retrieves the fields actually used when googlesheets4 guesses or imposes cell and column types. "full" retrieves all fields in the `CellData` schema. The main differences relate to cell formatting.
- **discard_empty**: Whether to discard cells that have no data. Literally, we check for an `effectiveValue`, which is one of the fields in the `CellData` schema.
**range_speedread**

**Value**

A tibble with one row per cell in the range.

**See Also**

Wraps the spreadsheets.get endpoint:


**Examples**

```r
range_read_cells(gs4_example("deaths"), range = "arts_data")

# if you want detailed and exhaustive cell data, do this
range_read_cells(
    gs4_example("formulas-and-formats"),
    cell_data = "full",
    discard_empty = FALSE
)
```

**range_speedread** Read Sheet as CSV

**Description**

This function uses a quick-and-dirty method to read a Sheet that bypasses the Sheets API and, instead, parses a CSV representation of the data. This can be much faster than `range_read()` — noticeably so for "large" spreadsheets. There are real downsides, though, so we recommend this approach only when the speed difference justifies it. Here are the limitations we must accept to get faster reading:

- Only formatted cell values are available, not underlying values or details on the formats.
- We can’t target a named range as the range.
- We have no access to the data type of a cell, i.e. we don’t know that it’s logical, numeric, or datetime. That must be re-discovered based on the CSV data (or specified by the user).
- Auth and error handling have to be handled a bit differently internally, which may lead to behaviour that differs from other functions in googlesheets4.

Note that the Sheets API is still used to retrieve metadata on the target Sheet, in order to support range specification. `range_speedread()` also sends an auth token with the request, unless a previous call to `gs4_deauth()` has put googlesheets4 into a de-authorized state.

**Usage**

```r
range_speedread(ss, sheet = NULL, range = NULL, skip = 0, ...)
```
Arguments

ss  Something that identifies a Google Sheet:
    • its file id as a string or drive_id
    • a URL from which we can recover the id
    • a one-row dribble, which is how googledrive represents Drive files
    • an instance of googlesheets4_spreadsheet, which is what gs4.get() returns

Processed through as_sheets_id().

sheet  Sheet to read, in the sense of "worksheet" or "tab". You can identify a sheet
        by name, with a string, or by position, with a number. Ignored if the sheet is
        specified via range. If neither argument specifies the sheet, defaults to the first
        visible sheet.

range  A cell range to read from. If NULL, all non-empty cells are read. Otherwise specify
        range as described in Sheets A1 notation or using the helpers documented
        in cell-specification. Sheets uses fairly standard spreadsheet range notation, al-
        preted strictly, even if the range forces the inclusion of leading, trailing, or em-
        bedded empty rows or columns. Takes precedence over skip, n_max and sheet.
        Note range can be a named range, like "sales_data", without any cell reference.

skip  Minimum number of rows to skip before reading anything, be it column names
        or data. Leading empty rows are automatically skipped, so this is a lower bound.
        Ignored if range is given.

...  Passed along to the CSV parsing function (currently readr::read_csv()).

Value

A tibble

Examples

```r
if (require("readr")) {
  # since cell type is not available, use readr's col type specification
  range_speedread(
    gs4_example("deaths"),
    sheet = "other",
    range = "A5:F15",
    col_types = cols(
      Age = col_integer(),
      'Date of birth' = col_date("%m/%d/%Y"),
      'Date of death' = col_date("%m/%d/%Y")
    )
  )
}
```
# write a Sheet that, by default, is NOT world-readable
(ss <- sheet_write(chickwts))

# demo that range_speedread() sends a token, which is why we can read this
range_speedread(ss)

# clean up
googledrive::drive_trash(ss)

---

**range_write** *(Over)write new data into a range*

**Description**

Writes a data frame into a range of cells. Main differences from `sheet_write()` (a.k.a. `write_sheet()`):

- Narrower scope. `range_write()` literally targets some cells, not a whole (work)sheet.
- The edited rectangle is not explicitly styled as a table. Nothing special is done re: formatting a header row or freezing rows.
- Column names can be suppressed. This means that, although data must be a data frame (at least for now), `range_write()` can actually be used to write arbitrary data.
- The target (spread)Sheet and (work)sheet must already exist. There is no ability to create a Sheet or add a worksheet.
- The target sheet dimensions are not "trimmed" to shrink-wrap the data. However, the sheet might gain rows and/or columns, in order to write data to the user-specified range.

If you just want to add rows to an existing table, the function you probably want is `sheet_append()`.

**Usage**

```r
range_write(
  ss, 
  data, 
  sheet = NULL, 
  range = NULL, 
  col_names = TRUE, 
  reformat = TRUE 
)
```

**Arguments**

- **ss**: Something that identifies a Google Sheet:
  - its file id as a string or `drive_id`
  - a URL from which we can recover the id
  - a one-row `dribble`, which is how googledrive represents Drive files
• an instance of googlesheets4_spreadsheet, which is what gs4_get() returns

Processed through as_sheets_id().

data
A data frame.
sheet
Sheet to write into, in the sense of "worksheet" or "tab". You can identify a sheet by name, with a string, or by position, with a number. Ignored if the sheet is specified via range. If neither argument specifies the sheet, defaults to the first visible sheet.

range
Where to write. This range argument has important similarities and differences to range elsewhere (e.g. range_read()):

• Similarities: Can be a cell range, using A1 notation ("A1:D3") or using the helpers in cell-specification. Can combine sheet name and cell range ("Sheet1!A5:A") or refer to a sheet by name (range = "Sheet1", although sheet = "Sheet1" is preferred for clarity).
• Difference: Can NOT be a named range.
• Difference: range can be interpreted as the start of the target rectangle (the upper left corner) or, more literally, as the actual target rectangle. See the "Range specification" section for details.

col_names
Logical, indicates whether to send the column names of data.

reformat
Logical, indicates whether to reformat the affected cells. Currently googlesheets4 provides no real support for formatting, so reformat = TRUE effectively means that edited cells become unformatted.

Value
The input ss, as an instance of sheets_id

Range specification

The range argument of range_write() is special, because the Sheets API can implement it in 2 different ways:

• If range represents exactly 1 cell, like "B3", it is taken as the start (or upper left corner) of the targeted cell rectangle. The edited cells are determined implicitly by the extent of the data we are writing. This frees you from doing fiddly range computations based on the dimensions of the data.
• If range describes a rectangle with multiple cells, it is interpreted as the actual rectangle to edit. It is possible to describe a rectangle that is unbounded on the right (e.g. "B2:4"), on the bottom (e.g. "A4:C"), or on both the right and the bottom (e.g. cell_limits(c(2, 3), c(NA, NA)). Note that all cells inside the rectangle receive updated data and format. Important implication: if the data object isn’t big enough to fill the target rectangle, the cells that don’t receive new data are effectively cleared, i.e. the existing value and format are deleted.

See Also

If sheet size needs to change, makes an UpdateSheetPropertiesRequest:
The main data write is done via an UpdateCellsRequest:

- https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#updatecellsrequest

Other write functions: gs4_create(), gs4_formula(), range_delete(), range_flood(), sheet_append(), sheet_write()

Examples

# create a Sheet with some initial, empty (work)sheets
(ss <- gs4_create("range-write-demo", sheets = c("alpha", "beta")))

df <- data.frame(
  x = 1:3,
  y = letters[1:3]
)

# write df somewhere other than the "upper left corner"
range_write(ss, data = df, range = "D6")

# view your magnificent creation in the browser
gs4_browse(ss)

# send data of disparate types to a 1-row rectangle
dat <- tibble::tibble(
  string = "string",
  logical = TRUE,
  datetime = Sys.time()
)
range_write(ss, data = dat, sheet = "beta", col_names = FALSE)

# send data of disparate types to a 1-column rectangle
dat <- tibble::tibble(
  x = list(Sys.time(), FALSE, "string")
)
range_write(ss, data = dat, range = "beta!C5", col_names = FALSE)

# clean up
gs4_find("range-write-demo") %>%
  googledrive::drive_trash()
Description

Generate a request, using knowledge of the Sheets API from its Discovery Document (https://www.googleapis.com/discovery/v1/apis/sheets/v4/rest). Use request_make() to execute the request. Most users should, instead, use higher-level wrappers that facilitate common tasks, such as reading or writing worksheets or cell ranges. The functions here are intended for internal use and for programming around the Sheets API.

request_generate() lets you provide the bare minimum of input. It takes a nickname for an endpoint and:

- Uses the API spec to look up the method, path, and base_url.
- Checks params for validity and completeness with respect to the endpoint. Uses params for URL endpoint substitution and separates remaining parameters into those destined for the body versus the query.
- Adds an API key to the query if and only if token = NULL.

Usage

request_generate(
    endpoint = character(),
    params = list(),
    key = NULL,
    token = gs4_token()
)

Arguments

- **endpoint** Character. Nickname for one of the selected Sheets API v4 endpoints built into googlesheets4. Learn more in gs4_endpoints().
- **params** Named list. Parameters destined for endpoint URL substitution, the query, or the body.
- **key** API key. Needed for requests that don’t contain a token. The need for an API key in the absence of a token is explained in Google’s document "Credentials, access, security, and identity" (https://support.google.com/googleapi/answer/6158857?hl=en&ref_topic=7013279). In order of precedence, these sources are consulted: the formal key argument, a key parameter in params, a user-configured API key set up with gs4_auth_configure() and retrieved with gs4_api_key().
- **token** Set this to NULL to suppress the inclusion of a token. Note that, if auth has been de-activated via gs4_deauth(), gs4_token() will actually return NULL.

Value

list()
Components are method, url, body, and token, suitable as input for request_make().

See Also

gargle::request_develop(), gargle::request_build(), gargle::request_make()
Other low-level API functions: gs4_has_token(), gs4_token(), request_make()
request_make

Examples

```r
req <- request_generate(
  "sheets.spreadsheets.get",
  list(spreadsheetId = gs4_example("deaths")),
  key = "PRETEND_I_AM_AN_API_KEY",
  token = NULL
)
req
```

Description

Low-level function to execute a Sheets API request. Most users should, instead, use higher-level wrappers that facilitate common tasks, such as reading or writing worksheets or cell ranges. The functions here are intended for internal use and for programming around the Sheets API.

Usage

```r
request_make(x, ..., encode = "json")
```

Arguments

- **x** List. Holds the components for an HTTP request, presumably created with `request_generate()` or `gargle::request_build()`. Must contain a method and url. If present, body and token are used.
- **...** Optional arguments passed through to the HTTP method.
- **encode** If the body is a named list, how should it be encoded? This has the same meaning as encode in all the `httr::VERB()`s, such as `httr::POST()`. Note, however, that we default to encode = "json", which is what you want most of the time when calling the Sheets API. The httr default is "multipart". Other acceptable values are "form" and "raw".

Details

`make_request()` is a very thin wrapper around `gargle::request_retry()`, only adding the googlesheets4 user agent. Typically the input has been created with `request_generate()` or `gargle::request_build()` and the output is processed with `process_response()`.

`gargle::request_retry()` retries requests that error with 429 RESOURCE_EXHAUSTED. Its basic scheme is exponential backoff, with one tweak that is very specific to the Sheets API, which has documented usage limits:

"a limit of 500 requests per 100 seconds per project and 100 requests per 100 seconds per user"

Note that the "project" here means everyone using googlesheets4 who hasn’t configured their own OAuth app. This is potentially a lot of users, all acting independently.
If you hit the "100 requests per 100 seconds per user" limit (which really does mean YOU), the first wait time is a bit more than 100 seconds, then we revert to exponential backoff.

If you experience lots of retries, especially with 100 second delays, it means your use of googlesheets4 is more than casual and it's time for you to get your own OAuth app or use a service account token. This is explained in the gargle vignette vignette("get-api-credentials", package = "gargle").

Value

Object of class response from http.

See Also

Other low-level API functions: gs4_has_token(), gs4_token(), request_generate()

<table>
<thead>
<tr>
<th>sheets_id</th>
<th>sheets_id class</th>
</tr>
</thead>
</table>

Description

sheets_id is an S3 class that marks a string as a Google Sheet's id, which the Sheets API does refer to as spreadsheetId.

Any object of class sheets_id also has the drive_id class, which is used by googledrive for the same purpose. This means you can provide a sheets_id to googledrive functions, in order to do anything with your Sheet that has nothing to do with it being a spreadsheet. Examples: change the Sheet's name, parent folder, or permissions. Read more about using googlesheets4 and googledrive together in vignette("drive-and-sheets"). Note that a sheets_id object is intended to hold just one id, while the parent class drive_id can be used for multiple ids.

as_sheets_id() is a generic function that converts various inputs into an instance of sheets_id. See more below.

When you print a sheets_id, we attempt to reveal the Sheet's current metadata, via gs4_get(). This can fail for a variety of reasons (e.g. if you're offline), but the input sheets_id is always revealed and returned, invisibly.

Usage

as_sheets_id(x, ...)

Arguments

xSomething that contains a Google Sheet id: an id string, a drive_id, a URL, a one-row dribble, or a googlesheets4_spreadsheet.

...Other arguments passed down to methods. (Not used.)
as_sheets_id()

These inputs can be converted to a sheets_id:

- Spreadsheet id, "a string containing letters, numbers, and some special characters", typically 44 characters long, in our experience. Example: 1qpyC0XZvTcKT6EISywvqESX3A0MwQoFDE8p-B1l4hps.
- A URL, from which we can excavate a spreadsheet or file id. Example: "https://docs.google.com/spreadsheets/d/1BzfL0kZUz1TsI5zxJF1WNF01IxvC67FbOJUiiGMZ_mQ/edit#gid=1150108545".
- A one-row dribble, a "Drive tibble" used by the googledrive package. In general, a dribble can represent several files, one row per file. Since googlesheets4 is not vectorized over spreadsheets, we are only prepared to accept a one-row dribble.
  - googledrive::drive_get("YOUR_SHEET_NAME") is a great way to look up a Sheet via its name.
  - gs4_find("YOUR_SHEET_NAME") is another good way to get your hands on a Sheet.
- Spreadsheet meta data, as returned by, e.g., gs4_get(). Literally, this is an object of class googlesheets4_spreadsheet.

See Also
gogledrive::as_id

Examples

mini_gap_id <- gs4_example("mini-gap")
class(mini_gap_id)
mini_gap_id

as_sheets_id("abc")

Description

Add one or more (work)sheets

Usage

sheet_add(ss, sheet = NULL, ..., .before = NULL, .after = NULL)

Arguments

ss Something that identifies a Google Sheet:
  • its file id as a string or drive_id
  • a URL from which we can recover the id
  • a one-row dribble, which is how googledrive represents Drive files
• an instance of googlesheets4_spreadsheet, which is what gs4_get()
returns
Processed through as_sheets_id().

sheet
One or more new sheet names. If unspecified, one new sheet is added and Sheets
autogenerates a name of the form "SheetN".
...
Optional parameters to specify additional properties, common to all of the new
sheet(s). Not relevant to most users. Specify fields of the SheetProperties
schema in name = value form.
.before, .after
Optional specification of where to put the new sheet(s). Specify, at most, one
of .before and .after. Refer to an existing sheet by name (via a string) or by
position (via a number). If unspecified, Sheets puts the new sheet(s) at the end.

Value
The input ss, as an instance of sheets_id

See Also
Makes a batch of AddSheetRequests (one per sheet):

• https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#
addsheetrequest

Other worksheet functions: sheet_append(), sheet_copy(), sheet_delete(), sheet_properties(),
sheet_relocate(), sheet_rename(), sheet_resize(), sheet_write()

Examples

ss <- gs4_create("add-sheets-to-me")

# the only required argument is the target spreadsheet
ss %>% sheet_add()

# but you CAN specify sheet name and/or position
ss %>% sheet_add("apple", .after = 1)
ss %>% sheet_add("banana", .after = "apple")

# add multiple sheets at once
ss %>% sheet_add(c("coconut", "dragonfruit"))

# keeners can even specify additional sheet properties
ss %>%
sheet_add(
    sheet = "eggplant",
    .before = 1,
    gridProperties = list(
        rowCount = 3, columnCount = 6, frozenRowCount = 1
    )
)
# get an overview of the sheets
sheet_properties(ss)

# clean up
gs4_find("add-sheets-to-me") %>%
googledrive::drive_trash()

---

**sheet_append**  
*Append rows to a sheet*

### Description

Adds one or more new rows after the last row with data in a (work)sheet, increasing the row dimension of the sheet if necessary.

### Usage

```r
sheet_append(ss, data, sheet = 1)
```

### Arguments

- **ss**  
  Something that identifies a Google Sheet:  
  - its file id as a string or `drive_id`  
  - a URL from which we can recover the id  
  - a one-row `dribble`, which is how googledrive represents Drive files  
  - an instance of `googlesheets4_spreadsheet`, which is what `gs4_get()` returns  
  Processed through `as_sheets_id()`.

- **data**  
  A data frame.

- **sheet**  
  Sheet to append to, in the sense of "worksheet" or "tab". You can identify a sheet by name, with a string, or by position, with a number.

### Value

The input `ss`, as an instance of `sheets_id`

### See Also

Makes an `AppendCellsRequest`:

- [https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#AppendCellsRequest](https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#AppendCellsRequest)

Other write functions: `gs4_create()`, `gs4_formula()`, `range_delete()`, `range_flood()`, `range_write()`, `sheet_write()`

Other worksheet functions: `sheet_add()`, `sheet_copy()`, `sheet_delete()`, `sheet_properties()`, `sheet_relocate()`, `sheet_rename()`, `sheet_resize()`, `sheet_write()`
Examples

# we will recreate the table of "other" deaths from this example Sheet
(deaths <- gs4_example("deaths") %>%
  range_read(range = "other_data", col_types = "????DD"))

# split the data into 3 pieces, which we will send separately
deaths_one <- deaths[1:5, ]
debaths_two <- deaths[6, ]
debaths_three <- deaths[7:10, ]

# create a Sheet and send the first chunk of data
ss <- gs4_create("sheet-append-demo", sheets = list(deaths = deaths_one))

# append a single row
ss %>% sheet_append(deaths_two)

# append remaining rows
ss %>% sheet_append(deaths_three)

# read and check against the original
deaths_replica <- range_read(ss, col_types = "????DD")
identical(deaths, deaths_replica)

# clean up
gs4_find("sheet-append-demo") %>%
googledrive::drive_trash()

---

sheet_copy

Copy a (work)sheet

Description

Copies a (work)sheet, within its current (spread)Sheet or to another Sheet.

Usage

sheet_copy(
  from_ss,
  from_sheet = NULL,
  to_ss = from_ss,
  to_sheet = NULL,
  .before = NULL,
  .after = NULL
)
Arguments

from_ss  Something that identifies a Google Sheet:
  • its file id as a string or drive_id
  • a URL from which we can recover the id
  • a one-row dribble, which is how googledrive represents Drive files
  • an instance of googlesheets4_spreadsheet, which is what gs4_get() returns

Processed through as_sheets_id().

from_sheet  Sheet to copy, in the sense of "worksheet" or "tab". You can identify a sheet by
  name, with a string, or by position, with a number. Defaults to the first visible
  sheet.

to_ss The Sheet to copy to. Accepts all the same types of input as from_ss, which is
  also what this defaults to, if unspecified.

to_sheet Optional. Name of the new sheet, as a string. If you don’t specify this, Google
  generates a name, along the lines of "Copy of blah". Note that sheet names must
  be unique within a Sheet, so if the automatic name would violate this, Google
  also de-duplicates it for you, meaning you could conceivably end up with "Copy
  of blah 2". If you have better ideas about sheet names, specify to_sheet.

.before, .after
  Optional specification of where to put the new sheet. Specify, at most, one of
  .before and .after. Refer to an existing sheet by name (via a string) or by
  position (via a number). If unspecified, Sheets puts the new sheet at the end.

Value

The receiving Sheet, to_ ss, as an instance of sheets_id.

See Also

If the copy happens within one Sheet, makes a DuplicateSheetRequest:

  • https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#
    duplicatesheetrequest

If the copy is from one Sheet to another, wraps the
  spreadsheets.sheets/copyTo endpoint:

  • https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets.sheets/
    copyTo

and possibly makes a subsequent UpdateSheetPropertiesRequest:

  • https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#
    UpdateSheetPropertiesRequest

Other worksheet functions: sheet_add(), sheet_append(), sheet_delete(), sheet_properties(),
  sheet_relocate(), sheet_rename(), sheet_resize(), sheet_write()
Examples

```r
ss_aaa <- gs4_create(
  "sheet-copy-demo-aaa",
  sheets = list(mtcars = head(mtcars), chickwts = head(chickwts))
)

# copy 'mtcars' sheet within existing Sheet, accept autogenerated name
ss_aaa %>%
  sheet_copy()

# copy 'mtcars' sheet within existing Sheet
# specify new sheet's name and location
ss_aaa %>%
  sheet_copy(to_sheet = "mtcars-the-sequel", .after = 1)

# make a second Sheet
ss_bbb <- gs4_create("sheet-copy-demo-bbb")

# copy 'chickwts' sheet from first Sheet to second
# accept auto-generated name and default location
ss_aaa %>%
  sheet_copy("chickwts", to_ss = ss_bbb)

# copy 'chickwts' sheet from first Sheet to second,
# WITH a specific name and into a specific location
ss_aaa %>%
  sheet_copy(
    "chickwts",
    to_ss = ss_bbb, to_sheet = "chicks-two", .before = 1
  )

# clean up
gs4_find("sheet-copy-demo") %>%
  googledrive::drive_trash()
```

---

**sheet_delete**  
*Delete one or more (work)sheets*

**Description**

Deletes one or more (work)sheets from a (spread)Sheet.

**Usage**

```r
sheet_delete(ss, sheet)
```
Arguments

ss  Something that identifies a Google Sheet:
   • its file id as a string or drive_id
   • a URL from which we can recover the id
   • a one-row dribble, which is how googledrive represents Drive files
   • an instance of googlesheets4_spreadsheet, which is what gs4_get() returns

Sheet to delete, in the sense of "worksheet" or "tab". You can identify a sheet by name, with a string, or by position, with a number. You can pass a vector to delete multiple sheets at once or even a list, if you need to mix names and positions.

Value

The input ss, as an instance of sheets_id

See Also

Makes an DeleteSheetsRequest:

- https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#DeleteSheetRequest

Other worksheet functions: sheet_add(), sheet_append(), sheet_copy(), sheet_properties(), sheet_relocate(), sheet_rename(), sheet_resize(), sheet_write()

Examples

ss <- gs4_create("delete-sheets-from-me")
sheet_add(ss, c("alpha", "beta", "gamma", "delta"))

# get an overview of the sheets
sheet_properties(ss)

# delete sheets
sheet_delete(ss, 1)
sheet_delete(ss, "gamma")
sheet_delete(ss, list("alpha", 2))

# get an overview of the sheets
sheet_properties(ss)

# clean up
gs4_find("delete-sheets-from-me") %>%
googledrive::drive_trash()
Description
Reveals full metadata or just the names for the (work)sheets inside a (spread)Sheet.

Usage

sheet_properties(ss)
sheet_names(ss)

Arguments

ss Something that identifies a Google Sheet:
- its file id as a string or drive_id
- a URL from which we can recover the id
- a one-row dribble, which is how googledrive represents Drive files
- an instance of googlesheets4_spreadsheet, which is what gs4_get() returns
  Processed through as_sheets_id().

Value

- sheet_properties(): A tibble with one row per (work)sheet.
- sheet_names(): A character vector of (work)sheet names.

See Also

Other worksheet functions: sheet_add(), sheet_append(), sheet_copy(), sheet_delete(), sheet_relocate(), sheet_rename(), sheet_resize(), sheet_write()

Examples

ss <- gs4_example("gapminder")
sheet_properties(ss)
sheet_names(ss)
Description
Move (work)sheets around within a (spread)Sheet. The outcome is most predictable for these common and simple use cases:

- Reorder and move one or more sheets to the front.
- Move a single sheet to a specific (but arbitrary) location.
- Move multiple sheets to the back with .after = 100 (.after can be any number greater than or equal to the number of sheets).

If your relocation task is more complicated and you are puzzled by the result, break it into a sequence of simpler calls to `sheet_relocate()`.

Usage
```
sheet_relocate(ss, sheet, .before = if (is.null(.after)) 1, .after = NULL)
```

Arguments
- **ss**: Something that identifies a Google Sheet:
  - its file id as a string or `drive_id`
  - a URL from which we can recover the id
  - a one-row `dribble`, which is how googledrive represents Drive files
  - an instance of `googlesheets4_spreadsheet`, which is what `gs4_get()` returns

  Processed through `as_sheets_id()`.
- **sheet**: Sheet to relocate, in the sense of "worksheet" or "tab". You can identify a sheet by name, with a string, or by position, with a number. You can pass a vector to move multiple sheets at once or even a list, if you need to mix names and positions.
- **.before, .after**: Specification of where to locate the sheets(s) identified by `sheet`. Exactly one of `.before` and `.after` must be specified. Refer to an existing sheet by name (via a string) or by position (via a number).

Value
The input `ss`, as an instance of `sheets_id`
See Also

Constructs a batch of UpdateSheetPropertiesRequests (one per sheet):

- https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#

Other worksheet functions: `sheet_add()`, `sheet_append()`, `sheet_copy()`, `sheet_delete()`, `sheet_properties()`, `sheet_rename()`, `sheet_resize()`, `sheet_write()`

Examples

```r
sheet_names <- c("alfa", "bravo", "charlie", "delta", "echo", "foxtrot")
ss <- gs4_create("sheet-relocate-demo", sheets = sheet_names)
sheet_names(ss)

# move one sheet, forwards then backwards
ss %>%
  sheet_relocate("echo", .before = "bravo") %>%
  sheet_names()

ss %>%
  sheet_relocate("echo", .after = "delta") %>%
  sheet_names()

# reorder and move multiple sheets to the front
ss %>%
  sheet_relocate(list("foxtrot", 4)) %>%
  sheet_names()

# put the sheets back in the original order
ss %>%
  sheet_relocate(sheet_names) %>%
  sheet_names()

# reorder and move multiple sheets to the back
ss %>%
  sheet_relocate(c("bravo", "alfa", "echo"), .after = 10) %>%
  sheet_names()

# clean up
gs4_find("sheet-relocate-demo") %>%
googledrive:::drive_trash()
```

---

### `sheet_rename`

**Rename a (work)sheet**

**Description**

Changes the name of a (work)sheet.
**Usage**

`sheet_rename(ss, sheet = NULL, new_name)`

**Arguments**

- **ss**: Something that identifies a Google Sheet:
  - its file id as a string or `drive_id`
  - a URL from which we can recover the id
  - a one-row `dribble`, which is how googledrive represents Drive files
  - an instance of `googlesheets4_spreadsheet`, which is what `gs4_get()` returns
    Processed through `as_sheets_id()`.
- **sheet**: Sheet to rename, in the sense of "worksheet" or "tab". You can identify a sheet by name, with a string, or by position, with a number. Defaults to the first visible sheet.
- **new_name**: New name of the sheet, as a string. This is required.

**Value**

The input `ss`, as an instance of `sheets_id`

**See Also**

Makes an `UpdateSheetPropertiesRequest`:


Other worksheet functions: `sheet_add()`, `sheet_append()`, `sheet_copy()`, `sheet_delete()`, `sheet_properties()`, `sheet_relocate()`, `sheet_resize()`, `sheet_write()`

**Examples**

```r
ss <- gs4_create(
  "sheet-rename-demo",
  sheets = list(cars = head(cars), chickwts = head(chickwts))
)
sheet_names(ss)

ss %>%
  sheet_rename(1, new_name = "automobiles") %>%
  sheet_rename("chickwts", new_name = "poultry")

# clean up
gs4_find("sheet-rename-demo") %>%
googledrive::drive_trash()
```
**sheet_resize**

*Change the size of a (work)sheet*

**Description**

Changes the number of rows and/or columns in a (work)sheet.

**Usage**

```r
sheet_resize(ss, sheet = NULL, nrow = NULL, ncol = NULL, exact = FALSE)
```

**Arguments**

- `ss`:
  - Something that identifies a Google Sheet:
  - its file id as a string or `drive_id`
  - a URL from which we can recover the id
  - a one-row `dribble`, which is how googledrive represents Drive files
  - an instance of `googlesheets4_spreadsheet`, which is what `gs4_get()` returns
  - Processed through `as_sheets_id()`.

- `sheet`:
  - Sheet to resize, in the sense of "worksheet" or "tab". You can identify a sheet by name, with a string, or by position, with a number.

- `nrow`, `ncol`:
  - Desired number of rows or columns, respectively. The default of `NULL` means to leave unchanged.

- `exact`:
  - Logical, indicating whether to impose `nrow` and `ncol` exactly or to treat them as lower bounds. If `exact = FALSE`, `sheet_resize()` can only add cells. If `exact = TRUE`, cells can be deleted and their contents are lost.

**Value**

The input `ss`, as an instance of `sheets_id`

**See Also**

Makes an `UpdateSheetPropertiesRequest`:

- `<# https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/request#UpdateSheetPropertiesRequest>`

Other worksheet functions: `sheet_add()`, `sheet_append()`, `sheet_copy()`, `sheet_delete()`, `sheet_properties()`, `sheet_relocate()`, `sheet_rename()`, `sheet_write()`
Examples

```r
# create a Sheet with the default initial worksheet
(ss <- gs4_create("sheet-resize-demo"))

# see (work)sheet dims
sheet_properties(ss)

# no resize occurs
sheet_resize(ss, nrow = 2, ncol = 6)

# reduce sheet size
sheet_resize(ss, nrow = 5, ncol = 7, exact = TRUE)

# add rows
sheet_resize(ss, nrow = 7)

# add columns
sheet_resize(ss, ncol = 10)

# add rows and columns
sheet_resize(ss, nrow = 9, ncol = 12)

# re-inspect (work)sheet dims
sheet_properties(ss)

# clean up
gs4_find("sheet-resize-demo") %>%
googledrive::drive_trash()
```

---

**Description**

This is one of the main ways to write data with googlesheets4. This function writes a data frame into a (work)sheet inside a (spread)Sheet. The target sheet is styled as a table:

- Special formatting is applied to the header row, which holds column names.
- The first row (header row) is frozen.
- The sheet’s dimensions are set to "shrink wrap" the data.

If no existing Sheet is specified via `ss`, this function delegates to `gs4_create()` and the new Sheet’s name is randomly generated. If that’s undesirable, call `gs4_create()` directly to get more control.

If no sheet is specified or if sheet doesn’t identify an existing sheet, a new sheet is added to receive the data. If sheet specifies an existing sheet, it is effectively overwritten! All pre-existing values, formats, and dimensions are cleared and the targeted sheet gets new values and dimensions from data.
This function goes by two names, because we want it to make sense in two contexts:

- `write_sheet()` evokes other table-writing functions, like `readr::write_csv()`. The sheet here technically refers to an individual (work)sheet (but also sort of refers to the associated Google (spread)Sheet).
- `sheet_write()` is the right name according to the naming convention used throughout the googlesheets4 package.

`write_sheet()` and `sheet_write()` are equivalent and you can use either one.

Usage

```r
sheet_write(data, ss = NULL, sheet = NULL)
write_sheet(data, ss = NULL, sheet = NULL)
```

Arguments

- **data**: A data frame. If it has zero rows, we send one empty pseudo-row of data, so that we can apply the usual table styling. This empty row goes away (gets filled, actually) the first time you send more data with `sheet_append()`.
- **ss**: Something that identifies a Google Sheet:
  - its file id as a string or `drive_id`
  - a URL from which we can recover the id
  - a one-row `dribble`, which is how googledrive represents Drive files
  - an instance of `googlesheets4_spreadsheet`, which is what `gs4_get()` returns
    Processed through `as_sheets_id()`.
- **sheet**: Sheet to write into, in the sense of "worksheet" or "tab". You can identify a sheet by name, with a string, or by position, with a number.

Value

The input `ss`, as an instance of `sheets_id`

See Also

Other write functions: `gs4_create()`, `gs4_formula()`, `range_delete()`, `range_flood()`, `range_write()`, `sheet_append()`

Other worksheet functions: `sheet_add()`, `sheet_append()`, `sheet_copy()`, `sheet_delete()`, `sheet_properties()`, `sheet_relocate()`, `sheet_rename()`, `sheet_resize()`

Examples

```r
df <- data.frame(
  x = 1:3,
  y = letters[1:3]
)
# specify only a data frame, get a new Sheet, with a random name
ss <- write_sheet(df)
read_sheet(ss)

# clean up
googledrive::drive_trash(ss)

# create a Sheet with some initial, placeholder data
ss <- gs4_create(
  "sheet-write-demo",
  sheets = list(alpha = data.frame(x = 1), omega = data.frame(x = 1))
)

# write df into its own, new sheet
sheet_write(df, ss = ss)

# write mtcars into the sheet named "omega"
sheet_write(mtcars, ss = ss, sheet = "omega")

# get an overview of the sheets
sheet_properties(ss)

# view your magnificent creation in the browser
gs4_browse(ss)

# clean up
gs4_find("sheet-write-demo") %>%
  googledrive::drive_trash()

---

spread_sheet

Spread a data frame of cells into spreadsheet shape

Description

Reshapes a data frame of cells (presumably the output of `range_read_cells()` into another data frame, i.e., puts it back into the shape of the source spreadsheet. This function exists primarily for internal use and for testing. The flagship function `range_read()`, a.k.a. `read_sheet()`, is what most users are looking for. It is basically `range_read_cells()` + `spread_sheet()`.

Usage

```r
spread_sheet(
  df,
  col_names = TRUE,
  col_types = NULL,
  na = "",
)```
trim_ws = TRUE,
    guess_max = min(1000, max(df$row)),
    .name_repair = "unique"
)

Arguments

df A data frame with one row per (nonempty) cell, integer variables row and column (probably referring to location within the spreadsheet), and a list-column cell of SHEET_CELL objects.

col_names TRUE to use the first row as column names, FALSE to get default names, or a character vector to provide column names directly. If user provides col_types, col_names can have one entry per column or one entry per unskipped column.

col_types Column types. Either NULL to guess all from the spreadsheet or a string of readr-style shortcodes, with one character or code per column. If exactly one col_type is specified, it is recycled. See Column Specification for more.

na Character vector of strings to interpret as missing values. By default, blank cells are treated as missing data.

trim_ws Logical. Should leading and trailing whitespace be trimmed from cell contents?

guess_max Maximum number of data rows to use for guessing column types.

.name_repair Handling of column names. By default, googlesheets4 ensures column names are not empty and are unique. There is full support for .name_repair as documented in tibble::tibble().

Value

A tibble in the shape of the original spreadsheet, but enforcing user’s wishes regarding column names, column types, NA strings, and whitespace trimming.

Examples

df <- gs4_example("mini-gap") %>%
    range_read_cells()
spread_sheet(df)

# ^^ gets same result as ...
read_sheet(gs4_example("mini-gap"))
Index

* auth functions
  gs4_auth, 5
  gs4_auth_configure, 7
  gs4_deauth, 11

* formatting functions
  range_autofit, 19

* low-level API functions
  gs4_has_token, 17
  gs4_token, 18
  request_generate, 33
  request_make, 35

* worksheet functions
  sheet_add, 37
  sheet_append, 39
  sheet_copy, 40
  sheet_delete, 42
  sheet_properties, 44
  sheet_relocate, 45
  sheet_rename, 46
  sheet_resize, 48
  sheet_write, 49

* write functions
  gs4_create, 9
  gs4_formula, 15
  request_generate, 33
  request_make, 35
  request_retry(), 35
  request_build(), 34, 35
  request_develop(), 34
  range_delete, 21
  range_flood, 22
  range_write, 31
  sheet_append, 39
  sheet_write, 49

 anchored(cell-specification), 3
 as_sheets_id(sheets_id), 36
 as_sheets_id(), 9, 16, 20, 21, 23, 25, 28, 30, 32, 38, 39, 41, 43–45, 47, 48, 50

 cell-specification, 3, 23, 25, 28, 30
 cell_cols(cell-specification), 3
 cell_limits(cell-specification), 3
 cell_rows(cell-specification), 3
 cellranger, 3

dribble, 9, 13, 16, 19, 21, 23, 25, 28, 30, 31, 36, 37, 39, 41, 43–45, 47, 48, 50
drive_id, 9, 13, 16, 19, 21, 23, 25, 28, 30, 31, 36, 37, 39, 41, 43–45, 47, 48, 50
gargle::AuthState, 8
gargle::credentials_app_default(), 4
gargle::gargle_oauth_cache(), 4
gargle::gargle_oauth_email(), 4
gargle::gargle_oob_default(), 4
gargle::gargle_options, 6
gargle::request_build(), 34, 35
gargle::request_develop(), 34
gargle::request_make(), 34
gargle::request_retry(), 35
gargle::token_email(), 19
gargle::token_fetch(), 5, 6
gargle::token_tokeninfo(), 19
gargle::token_userinfo(), 19
gargle_oauth_cache(), 6
gargle_oauth_email(), 6
gargle_oob_default(), 6

googledrive, 36, 37

google::as_id, 37
google::drive_find(), 13

google::drive_get(), 16

google::drive_get(YOUR_SHEET_NAME), 37

googlesheets4, 36
googlesheets4-configuration, 3

gs4_api_key(gs4_auth_configure), 7

gs4_api_key(), 11, 34

gs4_auth, 5, 8, 11

gs4_auth(), 4, 7, 18

gs4_auth_configure(), 6, 7, 11

gs4_auth_configure(), 6, 11, 34

gs4_browse, 9

gs4_create, 9, 15, 22, 23, 33, 39, 50

gs4_create(), 49

gs4_deauth, 6, 8, 11