

Package: multitool (via r-universe)

October 13, 2026

Title Run Multiverse Style Analyses

Version 0.1.5.9000

Description Run the same analysis over a range of arbitrary data processing decisions. 'multitool' provides an interface for creating alternative analysis pipelines and turning them into a grid of all possible pipelines. Using this grid as a blueprint, you can model your data across all possible pipelines and summarize the results.

License MIT + file LICENSE

Imports arrow, cli, correlation, DiagrammeR, dplyr, furr, forcats, glue, moments, purrr, rlang, stringr, tibble, tidyr, patchwork, parameters, performance, lifecycle, ragg, styler

Suggests broom, broom.mixed, clipr, carrier, flextable, future, ggdist, ggplot2, ggtext, knitr, mirai, rmarkdown, testthat (>= 3.0.0), tidyverse

Config/testthat/edition 3

Encoding UTF-8

```
Roxygen list(markdown = TRUE)
```

VignetteBuilder knitr

URL <https://ethan-young.github.io/multitool/>,
<https://github.com/ethan-young/multitool>

BugReports <https://github.com/ethan-young/multitool/issues>

Depends R ($\geq 4.2.0$)

LazyData true

Config/roxygen2/version 8.1.0

```
Config/pak/sysreqs cmake libfontconfig1-dev libfreetype6-dev libfribidi-dev libglpk-  
dev make libharfbuzz-dev libicu-dev libjpeg-dev libpng-dev libtiff-dev libuv1-  
dev libwebp-dev libxml2-dev libssl-dev libx11-dev
```

Repository <https://ethan-young.r-universe.dev>

Date/Publication 2026-10-13 17:45:05 UTC

RemoteUrl <https://github.com/ethan-young/multitool>

RemoteRef HEAD

RemoteSha aa4e07ac4332c385b625ea7b5fd3112fbfd4455f

Contents

add_filters	3
add_model	4
add_model_descriptives	6
add_parameter_keys	7
add_postprocess	8
add_preprocess	10
add_section	11
add_subgroups	14
add_variables	15
analyze_grid	16
analyze_grid_parallel	19
assess_decisions	21
assess_robustness	23
coffee_quality	25
compose_view	27
condense	28
configure_labels	30
create_blueprint_graph	31
detect_multiverse_n	32
detect_n_filters	33
detect_n_models	34
detect_n_subgroups	35
detect_n_variables	36
expand_decisions	37
generate_docs	39
initialize_doc	41
layout_section	43
preview_section	45
show_code	48
show_labels	49
show_section_content	50
summarize_filter_ns	51
unpack_results	52
unpack_specs	54
view_real_size	56
visualize_pipeline	58

Index	61
--------------	-----------

add_filters*Add filtering/exclusion criteria to a multiverse pipeline*

Description

Add filtering/exclusion criteria to a multiverse pipeline

Usage

```
add_filters(.df, ..., remove_do_nothing = FALSE)
```

Arguments

.df The original `data.frame`(e.g., base data set). If part of set of `add_*` decision functions in a pipeline, the base data will be passed along as an attribute.

... logical expressions to be used with `filter` separated by commas. Expressions should not be quoted.

remove_do_nothing logical, `FALSE` by default. Indicates whether to include an specification where no filters are applied to the data. Typically this is desirable, but on a occasion you may wan to turn this functionally off.

Value

a `data.frame` with three columns: type, group, and code. Type indicates the decision type, group is a decision, and the code is the actual code that will be executed. If part of a pipe, the current set of decisions will be appended as new rows.

Examples

```
library(tidyverse)
library(multitool)

# Simulate some data
the_data <-
  data.frame(
    id   = 1:500,
    iv1  = rnorm(500),
    iv2  = rnorm(500),
    iv3  = rnorm(500),
    mod1 = rnorm(500),
    mod2 = rnorm(500),
    mod3 = rnorm(500),
    cov1 = rnorm(500),
    cov2 = rnorm(500),
    dv1  = rnorm(500),
    dv2  = rnorm(500),
```

```

    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

the_data |>
  add_filters(include1 == 0, include2 != 3, include2 != 2, include3 > -2.5)

```

add_model

Add a model and formula to a multiverse pipeline

Description

Add a model and formula to a multiverse pipeline

Usage

```

add_model(
  .df,
  model_desc,
  code,
  model_coefs = parameters::parameters(),
  model_fit = performance::performance(),
  model_standardize = parameters::standardize_parameters()
)

```

Arguments

<code>.df</code>	The original <code>data.frame</code> (e.g., base data set). If part of set of <code>add_*</code> decision functions in a pipeline, the base data will be passed along as an attribute.
<code>model_desc</code>	a human readable name you would like to give the model.
<code>code</code>	literal model syntax you would like to run. You can use <code>glue</code> inside formulas to dynamically generate variable names based on a variable grid. For example, if you make variable grid with two versions of your IVs (e.g., <code>iv1</code> and <code>iv2</code>), you can write your formula like so: <code>lm(happiness ~ {iv} + control_var)</code> . The only requirement is that the variables written in the formula actually exist in the underlying data. You are also responsible for loading any packages that run a particular model (e.g., <code>lme4</code> for mixed-models)
<code>model_coefs</code>	a function to extract coefficients from the model object. The default is to use <code>parameters::parameters()</code> but this could be also be <code>broom::tidy()</code> or any other function that summarizes model output. Whichever function you choose must take a model object as the first argument and return a <code>data.frame</code> .

model_fit a function to summarize model fit statistics. The default is to use `performance::performance()` but this could be also be `broom::glance()` or any other function that summarizes model output. Whichever function you choose must take a model object as the first argument and return a `data.frame`.

model_standardize a function to calculate standardized coefficients from the model object. The default is to use `parameters::standardize_parameters()` but this could be also be some other function that standardizes model output. Whichever function you choose must take a model object as the first argument and return a `data.frame`.

Value

a `data.frame` with three columns: type, group, and code. Type indicates the decision type, group is a decision, and the code is the actual code that will be executed. If part of a pipe, the current set of decisions will be appended as new rows.

Examples

```
library(tidyverse)
library(multitool)

the_data <-
  data.frame(
    id    = 1:500,
    iv1   = rnorm(500),
    iv2   = rnorm(500),
    iv3   = rnorm(500),
    mod1  = rnorm(500),
    mod2  = rnorm(500),
    mod3  = rnorm(500),
    cov1  = rnorm(500),
    cov2  = rnorm(500),
    dv1   = rnorm(500),
    dv2   = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

the_data |>
  add_filters(include1 == 0, include2 != 3, include2 != 2, include3 > -2.5) |>
  add_variables("ivs", iv1, iv2, iv3) |>
  add_variables("dvs", dv1, dv2) |>
  add_variables("mods", starts_with("mod")) |>
  add_preprocess("scale_iv", 'mutate({ivs} = scale({ivs}))') |>
  add_model("linear model", lm({dvs} ~ {ivs} * {mods}))
```

```
add_model_descriptives
```

Add arbitrary summary statistics to a multiverse pipeline

Description

Add arbitrary summary statistics to a multiverse pipeline

Usage

```
add_model_descriptives(.df, desc_name, code)
```

Arguments

<code>.df</code>	The original <code>data.frame</code> (e.g., base data set). If part of set of <code>add_*</code> decision functions in a pipeline, the base data will be passed along as an attribute.
<code>desc_name</code>	a character string. A descriptive name for what the summary statistics you want to compute over the data passed to your model.
<code>code</code>	the literal code you would like to execute. For summary statistics, <code>model.frame()</code> will be called on the model object fit in the prior step. Your code should thus work with the variables that are used in your model.

Value

a `data.frame` with three columns: type, group, and code. Type indicates the decision type, group is a decision, and the code is the actual code that will be executed. If part of a pipe, the current set of decisions will be appended as new rows.

Examples

```
library(tidyverse)
library(multitool)

the_data <-
  data.frame(
    id   = 1:500,
    iv1  = rnorm(500),
    iv2  = rnorm(500),
    iv3  = rnorm(500),
    mod1 = rnorm(500),
    mod2 = rnorm(500),
    mod3 = rnorm(500),
    cov1 = rnorm(500),
    cov2 = rnorm(500),
    dv1  = rnorm(500),
    dv2  = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
```

```

      include2 = sample(1:3, size = 500, replace = TRUE),
      include3 = rnorm(500)
    )

the_data |>
  add_filters(include1 == 0, include2 != 3, include2 != 2, include3 > -2.5) |>
  add_variables("ivs", iv1, iv2, iv3) |>
  add_variables("dvs", dv1, dv2) |>
  add_variables("mods", starts_with("mod")) |>
  add_preprocess("scale_iv", 'mutate({ivs} = scale({ivs}))') |>
  add_model("linear model", lm({dvs} ~ {ivs} * {mods})) |>
  add_model_descriptives(
    "descriptives",
    summarize(body_mass_mean = mean({dvs}), .by = c(include2))
  )

```

add_parameter_keys	<i>Add parameter keys names for later use in summarizing model effects</i>
--------------------	--

Description

Add parameter keys names for later use in summarizing model effects

Usage

```
add_parameter_keys(.df, parameter_group, parameter_name)
```

Arguments

<code>.df</code>	The original <code>data.frame</code> (e.g., base data set). If part of set of <code>add_*</code> decision functions in a pipeline, the base data will be passed along as an attribute.
<code>parameter_group</code>	character, a name for the parameter of interest
<code>parameter_name</code>	quoted or unquoted names of variables involved in a particular parameter of interest. Usually this is just a variable in your model (e.g., a main effect of your iv). However, it could also be an interaction term or some other term. You can use <code>glue</code> syntax to specify an effect that might use alternative versions of the same variable.

Value

a `data.frame` with three columns: type, group, and code. Type indicates the decision type, group is a decision, and the code is the actual code that will be executed. If part of a pipe, the current set of decisions will be appended as new rows.

Examples

```
library(tidyverse)
library(multitool)

# Simulate some data
the_data <-
  data.frame(
    id   = 1:500,
    iv1  = rnorm(500),
    iv2  = rnorm(500),
    iv3  = rnorm(500),
    mod1 = rnorm(500),
    mod2 = rnorm(500),
    mod3 = rnorm(500),
    cov1 = rnorm(500),
    cov2 = rnorm(500),
    dv1  = rnorm(500),
    dv2  = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

the_data |>
  add_variables("ivs", iv1, iv2, iv3) |>
  add_variables("dvs", dv1, dv2) |>
  add_variables("mods", starts_with("mod")) |>
  add_model("linear model", lm({dvs} ~ {ivs} * {mods})) |>
  add_parameter_keys("my_interaction", "{ivs}:{mods}") |>
  add_parameter_keys("my_main_effect", {ivs})
```

add_postprocess

Add arbitrary postprocessing code to a multiverse pipeline

Description

Add arbitrary postprocessing code to a multiverse pipeline

Usage

```
add_postprocess(.df, postprocess_name, code)
```

Arguments

.df The original `data.frame`(e.g., base data set). If part of set of `add_*` decision functions in a pipeline, the base data will be passed along as an attribute.

postprocess_name a character string. A descriptive name for what the postprocessing step accomplishes.

code the literal code you would like to execute after each analysis.
 The code should be written to work with pipes (i.e., `|>` or `%>%`). Because the post-processing code comes last in each multiverse analysis step, the chosen model object will be passed to the post-processing code.
 For example, if you fit a simple linear model like: `lm(y ~ x1 + x2)`, and your post-processing code executes a call to `anova`, you would simply pass `anova()` to `add_postprocess()`. The underlying code would be:
`data |> filters |> lm(y ~ x1 + x2, data = _) |> anova()`

Value

a `data.frame` with three columns: `type`, `group`, and `code`. `Type` indicates the decision type, `group` is a decision, and the `code` is the actual code that will be executed. If part of a pipe, the current set of decisions will be appended as new rows.

Examples

```
library(tidyverse)
library(multitool)

the_data <-
  data.frame(
    id = 1:500,
    iv1 = rnorm(500),
    iv2 = rnorm(500),
    iv3 = rnorm(500),
    mod1 = rnorm(500),
    mod2 = rnorm(500),
    mod3 = rnorm(500),
    cov1 = rnorm(500),
    cov2 = rnorm(500),
    dv1 = rnorm(500),
    dv2 = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

the_data |>
  add_filters(include1 == 0, include2 != 3, include2 != 2, include3 > -2.5) |>
  add_variables("ivs", iv1, iv2, iv3) |>
  add_variables("dvs", dv1, dv2) |>
  add_variables("mods", starts_with("mod")) |>
  add_preprocess("scale_iv", 'mutate({ivs} = scale({ivs}))') |>
  add_model("linear model", lm({dvs} ~ {ivs} * {mods})) |>
  add_postprocess("analysis of variance", aov())
```

add_preprocess	<i>Add arbitrary preprocessing code to a multiverse analysis pipeline</i>
----------------	---

Description

Add arbitrary preprocessing code to a multiverse analysis pipeline

Usage

```
add_preprocess(.df, process_name, code)
```

Arguments

<code>.df</code>	The original <code>data.frame</code> (e.g., base data set). If part of set of <code>add_*</code> decision functions in a pipeline, the base data will be passed along as an attribute.
<code>process_name</code>	a character string. A descriptive name for what the preprocessing step accomplishes.
<code>code</code>	the literal code you would like to execute after data are filtered. <code>glue</code> syntax is allowed. An example might be centering or scaling a predictor after the appropriate filters are applied to the data. The code should be written to work with pipes (i.e., <code> ></code> or <code>%>%</code>). Pre-processing code will eventually take the base data along with any filters applied to the data. This means <code>mutate</code> calls are the most natural but other functions that take a <code>data.frame</code> as the first argument should work as well (as long as they also return a <code>data.frame</code>).

Value

a `data.frame` with three columns: type, group, and code. Type indicates the decision type, group is a decision, and the code is the actual code that will be executed. If part of a pipe, the current set of decisions will be appended as new rows.

Examples

```
library(tidyverse)
library(multitool)

the_data <-
  data.frame(
    id   = 1:500,
    iv1  = rnorm(500),
    iv2  = rnorm(500),
    iv3  = rnorm(500),
    mod1 = rnorm(500),
    mod2 = rnorm(500),
    mod3 = rnorm(500),
```

```

      cov1 = rnorm(500),
      cov2 = rnorm(500),
      dv1  = rnorm(500),
      dv2  = rnorm(500),
      include1 = rbinom(500, size = 1, prob = .1),
      include2 = sample(1:3, size = 500, replace = TRUE),
      include3 = rnorm(500)
    )

the_data |>
  add_filters(include1 == 0, include2 != 3, include2 != 2, include3 > -2.5) |>
  add_variables("ivs", iv1, iv2, iv3) |>
  add_variables("dvs", dv1, dv2) |>
  add_variables("mods", starts_with("mod")) |>
  add_preprocess("scale_iv", 'mutate({ivs} = scale({ivs}))')

```

add_section	<i>Add a content section to a report</i>
-------------	--

Description

Defines a section of a report by attaching content-generating code to an analysis grid. A section bundles up to three kinds of content — text, a table, and a figure — each produced by a function applied to the section's gathered data. Sections accumulate: piping one `add_section()` into the next builds a multi-section report grid.

Usage

```

add_section(
  .report,
  id = "",
  title = "",
  description = "",
  report_data,
  txt.fn = NULL,
  tbl.fn = NULL,
  fig.fn = NULL,
  .by = NULL
)

```

Arguments

<code>.report</code>	A named analysis grid (e.g. an analyzed decision grid), or the result of a previous <code>add_section()</code> call to append to. <code>add_section()</code> re-evaluates against the grid by name, so the grid must be a named object in the calling environment.
<code>id</code>	A short identifier for the section, used later to lay it out and assemble it (e.g. <code>"robustness"</code>). Should be unique within a report.

title, description	Section heading text. Processed with <code>glue::glue()</code> , so they may interpolate <code>.by</code> columns to vary per subsection.
report_data	A code expression — usually a <code>compose_view()</code> call — applied to the grid to gather this section’s data into a single data frame. This data frame is what each content function receives.
txt.fn, tbl.fn, fig.fn	Content functions, each taking the <code>report_data</code> data frame as its first argument and returning one rendered object: <code>txt.fn</code> a text string, <code>tbl.fn</code> a table (<code>data.frame</code> , <code>gt</code> , or <code>flextable</code>), <code>fig.fn</code> a figure (<code>ggplot</code> or <code>patchwork</code>). Must be written as a call (with parentheses), since the data is piped in. Any may be <code>NULL</code> ; an omitted content type yields an empty slot and is skipped when the section is rendered.
.by	Columns that split the section into subsections, one per unique combination. When <code>NULL</code> , the section is a single unit.

Details

`add_section()` records the content functions (`txt.fn`, `tbl.fn`, `fig.fn`) as code; it does not run them. Realization is deferred to the moment a section is previewed (`preview_section()`) or the document is generated (`generate_docs()`), so defining a section is cheap no matter how many subsections it fans out into. The stored code *is* the section — it is read back when inspecting, and evaluated when rendering.

`add_section()` stores code, not rendered output. The `sec_txt`, `sec_tbl`, and `sec_fig` columns each hold a complete pipeline — the analysis grid, through `report_data`, through the content function — as an evaluable string. Nothing is rendered here; realization happens downstream. This is the package’s code-as-artifact principle in its strongest form: a section is fully described by readable, re-runnable code, inspectable with `show_section_content()` and rendered only at preview or generation time.

Deferring realization also keeps `add_section()` fast: a section fanned out into many subsections costs no rendering at definition time, since the content is rendered later — once — when the document is built.

Content functions have access only to the gathered section data, not to the section’s meta-data fields — a figure title, for instance, belongs inside `fig.fn`, while a section- or slide-level heading belongs in `title`.

Value

A report grid (a tibble) with one row per subsection. Each row carries the section’s id, the per-subsection title and description, and the content-generating code for each channel in the `sec_txt`, `sec_tbl`, and `sec_fig` columns — full pipelines from the analysis grid through the content function, stored as code and evaluated only when the section is previewed or generated. Empty channels hold `"NULL"`. The originating grid name is recorded as an `"analysis_grid"` attribute so successive `add_section()` calls can chain.

Section content

gather, then distil: A section is built in two steps. First, `report_data` gathers the data the section needs from the grid — typically a `compose_view()` call that joins the relevant

unpacked results into a single data frame. Second, each content function (`txt.fn`, `tbl.fn`, `fig.fn`) receives that gathered data frame as its first argument and distils it into one rendered object: a text string, a table, or a figure.

So the flow is always: `report_data` produces the section's data frame, and each content function consumes that same data frame. A content function is written as code that takes the `compose_view` output as its first argument — for example `fig.fn = ggplot(aes(...)) + geom_point()`, or a call to a named function whose first argument is the data, written with parentheses as `fig.fn = make_spec_curve()` so it sits correctly after the pipe. Because the data is piped into the content function, the function must be written as a call (with parentheses), not a bare name.

A section need not have all three content types. Any of `txt.fn`, `tbl.fn`, or `fig.fn` may be left `NULL`, in which case that content slot is empty and simply omitted wherever the section is rendered. A figure-only section, a table-plus-text section, or any combination is valid.

Sections and subsections

When `.by` is supplied, the section is *fanned out* into one subsection per unique combination of the `.by` columns. Each subsection receives its own filtered slice of the grid and its own title and description. Because `title` and `description` are processed with `glue::glue()`, they can reference the `.by` columns to produce per-subsection labels — for example, `title = "Results for {outcome}"` yields a distinct title per `outcome`.

Without `.by`, the section is a single unit with one title and description, and glue interpolation of the `.by` columns does not apply (there are no subsections to vary over).

See Also

`compose_view()` for gathering results for section reporting; `show_section_content()` to inspect one section's content code and render it; `preview_section()` to preview a section's composed layout; `layout_section()` and `generate_docs()` to assemble sections into a document.

Examples

```
## Not run:
# A figure-only section, fanned out by outcome
report <-
  analyzed_grid |>
  add_section(
    id      = "estimates",
    title    = "Effect estimates for {outcome}",
    description = "Distribution across specifications",
    report_data = compose_view(model_parameters, model_performance),
    fig.fn    = make_spec_curve(),
    .by       = outcome
  )

# Chain a table-only section onto the same report
report <-
  report |>
  add_section(
```

```

      id          = "robustness",
      title       = "Robustness summary",
      report_data = compose_view(rob = assess_robustness),
      tbl.fn      = gt::gt()
    )

## End(Not run)

```

add_subgroups	<i>Add sub groups to the multiverse pipeline</i>
---------------	--

Description

Add sub groups to the multiverse pipeline

Usage

```
add_subgroups(.df, ..., .use_partitions = FALSE)
```

Arguments

<code>.df</code>	The original <code>data.frame</code> (e.g., base data set). If part of set of <code>add_*</code> decision functions in a pipeline, the base data will be passed along as an attribute.
<code>...</code>	sub group variable(s) in your data whose values specify groupings.
<code>.use_partitions</code>	whether or not to search an arrow pointer table's partitioning for subgroup names and values. Defaults to <code>FALSE</code> .

Value

a `data.frame` with three columns: type, group, and code. Type indicates the decision type, group is a decision, and the code is the actual code that will be executed. If part of a pipe, the current set of decisions will be appended as new rows.

Examples

```

library(tidyverse)
library(multitool)

# Simulate some data
the_data <-
  data.frame(
    id   = 1:500,
    iv1  = rnorm(500),
    iv2  = rnorm(500),
    iv3  = rnorm(500),

```

```

    mod1 = rnorm(500),
    mod2 = rnorm(500),
    mod3 = rnorm(500),
    cov1 = rnorm(500),
    cov2 = rnorm(500),
    dv1  = rnorm(500),
    dv2  = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500),
    group    = sample(1:3, size = 500, replace = TRUE)
  )

the_data |>
  add_subgroups(group)

the_data |>
  add_subgroups(group, .only = c(1,3))

```

add_variables

Add a set of variable alternatives to a multiverse pipeline

Description

Add a set of variable alternatives to a multiverse pipeline

Usage

```
add_variables(.df, var_group, ...)
```

Arguments

<code>.df</code>	The original <code>data.frame</code> (e.g., base data set). If part of set of <code>add_*</code> decision functions in a pipeline, the base data will be passed along as an attribute.
<code>var_group</code>	a character string. Indicates the name of the current set. For example, "primary_iv" could indicate this set are alternatives of the main predictor in an analysis.
<code>...</code>	the bare unquoted names of the variables to include as alternative options for this variable set. You can also use <code>tidyselect</code> to select variables.

Value

a `data.frame` with three columns: `type`, `group`, and `code`. `Type` indicates the decision type, `group` is a decision, and the `code` is the actual code that will be executed. If part of a pipe, the current set of decisions will be appended as new rows.

Examples

```
library(tidyverse)
library(multitool)

# Simulate some data
the_data <-
  data.frame(
    id   = 1:500,
    iv1  = rnorm(500),
    iv2  = rnorm(500),
    iv3  = rnorm(500),
    mod1 = rnorm(500),
    mod2 = rnorm(500),
    mod3 = rnorm(500),
    cov1 = rnorm(500),
    cov2 = rnorm(500),
    dv1  = rnorm(500),
    dv2  = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

the_data |>
  add_variables("ivs", iv1, iv2, iv3) |>
  add_variables("dvs", dv1, dv2) |>
  add_variables("mods", starts_with("mod"))
```

analyze_grid

Perform all analyses over a complete decision grid

Description

Executes the analysis pipeline for every row of an expanded decision grid, returning tidied results for each. Execution is parallel-ready: if `mirai::daemons()` have been provisioned, the rows are distributed across the daemons; otherwise they run sequentially. `analyze_grid()` never sets up daemons itself — you provision them, and the function uses them if present.

Usage

```
analyze_grid(
  .grid,
  show_progress = TRUE,
  ship_base_df = TRUE,
  libraries = NULL,
  ...
)
```

Arguments

<code>.grid</code>	a tibble produced by <code>expand_decisions</code> .
<code>show_progress</code>	logical, whether to show a progress bar while running.
<code>ship_base_df</code>	logical (default TRUE). Controls how the base data frame reaches the workers when running in parallel. When TRUE, the base data frame named in the grid's "base_df" attribute is resolved in the calling environment, shipped to each worker, and assigned there by name — the simplest option, suited to small-to-moderate in-memory data. When FALSE, the base data frame is not shipped; you are responsible for making it available on the workers under the name the pipeline code references, typically by establishing it on the daemons beforehand with <code>mirai::everywhere()</code> (e.g. opening an Arrow dataset on each worker). Use FALSE for large or externally-stored data you do not want copied to every worker. Has no effect when running sequentially.
<code>libraries</code>	a character vector naming packages to load on each worker. Internally this calls <code>library()</code> dynamically so that functions from the packages your pipeline uses are available during execution on the workers. Relevant when running in parallel via <code>mirai::daemons()</code> ; include here any package your pipeline code calls that is not attached by default.
<code>...</code>	Custom functions your pipeline references (e.g. a custom post-processing step), passed as <code>name = function</code> pairs (for example <code>my_step = my_step</code>). Each is made available on the workers under the given name, so pipeline code that calls it by name resolves. The name must match the name used in the pipeline.

Value

a single tibble containing tidied results for the model and any post-processing tests/tasks. For each unique test (e.g., an `lm` or `aov` called on an `lm`), a list column with the function name is created with `parameters` and `performance` and any warnings or messages printed while fitting the models. A `timing_logs` list column records the start, end, and duration of each row's run. The grid's "pipeline" attribute is carried through to the result.

Parallel execution

To run in parallel, provision daemons before calling, e.g. `mirai::daemons(6)`. Each worker loads `multitool`, `dplyr`, and any packages named in `libraries`; receives any custom functions passed through `...`; and (when `ship_base_df = TRUE`) the base data frame. With no daemons set, execution falls back to sequential and these provisions still apply locally. For large data, prefer `ship_base_df = FALSE` with the data established on the daemons via `mirai::everywhere()`, or an Arrow partition path baked into the pipeline so each worker reads its own slice from storage.

Establishing the data on workers yourself (`ship_base_df = FALSE`)

With `ship_base_df = FALSE`, `analyze_grid()` does not ship the base data frame — you are responsible for making it available on each worker, under the same name your pipeline references. This avoids copying large data to every worker: instead each worker holds its

own handle (for example an Arrow dataset opened from storage), established once with `mirai::everywhere()`.

Two rules make this work. First, the object must live in each worker's global environment, because that is where the pipeline code is resolved; the reliable way to put it there is `evalq(..., envir = .GlobalEnv)` inside `everywhere()`. Second, the name must match the grid's "base_df" attribute exactly — the pipeline code refers to the data by that name, so the object you establish must carry it.

```
# name must match attr(.grid, "base_df"), e.g. "coffee_analysis_df"
mirai::daemons(6)
mirai::everywhere(
  evalq(
    {
      library(arrow)
      coffee_analysis_df <- open_dataset("/absolute/path/to/data/")
    },
    envir = .GlobalEnv
  )
)
analyzed_grid <- analyze_grid(pipeline_grid, ship_base_df = FALSE)
mirai::daemons(0)
```

If the object is missing or misnamed on the workers, the pipeline code fails to find it; because the failure occurs inside worker evaluation it may surface as an opaque error rather than a clear "object not found", so check the name match first if a parallel run with `ship_base_df = FALSE` fails.

Examples

```
library(tidyverse)
library(multitool)

# Simulate some data
the_data <-
  data.frame(
    id    = 1:500,
    iv1   = rnorm(500),
    iv2   = rnorm(500),
    iv3   = rnorm(500),
    mod1  = rnorm(500),
    mod2  = rnorm(500),
    mod3  = rnorm(500),
    cov1  = rnorm(500),
    cov2  = rnorm(500),
    dv1   = rnorm(500),
    dv2   = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )
```

```

# Decision pipeline
full_pipeline <-
  the_data |>
  add_filters(include1 == 0,include2 != 3,include2 != 2,scale(include3) > -2.5) |>
  add_variables("ivs", iv1, iv2, iv3) |>
  add_variables("dvs", dv1, dv2) |>
  add_variables("mods", starts_with("mod")) |>
  add_preprocess(process_name = "scale_iv", 'mutate({ivs} = scale({ivs}))') |>
  add_preprocess(process_name = "scale_mod", mutate({mods} := scale({mods}))) |>
  add_model("no covariates",lm({dvs} ~ {ivs} * {mods})) |>
  add_model("covariate", lm({dvs} ~ {ivs} * {mods} + cov1)) |>
  add_postprocess("ptp", predict())

pipeline_grid <- expand_decisions(full_pipeline)

# analyze the grid (sequential)
analyzed_grid <- analyze_grid(pipeline_grid[1:10,])

## Not run:
# analyze in parallel: provision daemons first
mirai::daemons(6)
analyzed_grid <- analyze_grid(pipeline_grid)
mirai::daemons(0)

## End(Not run)

```

analyze_grid_parallel

Analyze a complete decision grid in parallel

Description

Analyze a complete decision grid in parallel

Usage

```

analyze_grid_parallel(
  .grid,
  save_model = FALSE,
  show_progress = TRUE,
  furr_r_globals = NULL,
  furr_r_packages = c("multitool", "dplyr", "tidyr")
)

```

Arguments

`.grid` a tibble produced by [expand_decisions](#)

save_model logical, indicates whether to save the model object in its entirety. The default is **FALSE** because model objects are usually large and under the hood, **parameters** and **performance** is used to summarize the most useful model information.

show_progress logical, whether to show a progress bar while running.

furrr_globals any global objects to pass to **furrr_options**

furrr_packages character vector, any packages to load inside parallel environments

Value

a single **tibble** containing tidied results for the model and any post-processing tests/tasks. For each unique test (e.g., an **lm** or **aov** called on an **lm**), a list column with the function name is created with **parameters** and **performance** and any warnings or messages printed while fitting the models.

Examples

```
library(tidyverse)
library(multitool)
library(furrr)

# Simulate some data
the_data <-
  data.frame(
    id = 1:500,
    iv1 = rnorm(500),
    iv2 = rnorm(500),
    iv3 = rnorm(500),
    mod1 = rnorm(500),
    mod2 = rnorm(500),
    mod3 = rnorm(500),
    cov1 = rnorm(500),
    cov2 = rnorm(500),
    dv1 = rnorm(500),
    dv2 = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

# Decision pipeline
full_pipeline <-
  the_data |>
  add_filters(include1 == 0, include2 != 3, include2 != 2, scale(include3) > -2.5) |>
  add_variables("ivs", iv1, iv2, iv3) |>
  add_variables("dvs", dv1, dv2) |>
  add_variables("mods", starts_with("mod")) |>
  add_preprocess(process_name = "scale_iv", 'mutate({ivs} = scale({ivs}))') |>
  add_preprocess(process_name = "scale_mod", mutate({mods} := scale({mods}))) |>
  add_model("no covariates", lm({dvs} ~ {ivs} * {mods})) |>
```

```
add_model("covariate", lm({dvs} ~ {ivs} * {mods} + cov1))

pipeline_grid <- expand_decisions(full_pipeline)

# Run the whole multiverse
plan(multisession, workers = 4)
the_multiverse <- analyze_grid_parallel(pipeline_grid[4,])
plan(sequential)
```

assess_decisions	<i>Decompose decision variance using Sobol sensitivity indices and variance dispersal</i>
-------------------------	---

Description

Quantifies how much each decision type (e.g., filters, variables, models) contributes to the total variance in a focal estimand across all decision specifications. Uses variance-based sensitivity analysis to partition variance into main effects, interaction effects, and total effects for each decision set.

Usage

```
assess_decisions(.unpacked, .estimand, .by = NULL)
```

Arguments

.unpacked	A <code>data.frame</code> with unpacked multiverse results, typically produced by unpack_model_parameters or unpack_model_performance . Must contain columns for each decision type (filters, variables, models, etc.) and the focal estimand.
.estimand	The numeric outcome variable to decompose. Defaults to <code>std_coef</code> (standardized coefficients). Use unquoted column names with tidy evaluation.
.by	Optional grouping variable(s) for stratified decomposition. The variance decomposition will be computed separately for each group. Useful for examining whether decision importance varies across different model variables or subgroups. Use unquoted column names.

Details

This function implements a Sobol-style decomposition where "decision sets" (e.g., all filter decisions) are treated as factors whose combinations produce different specifications. The decomposition reveals which analytical choices have the strongest influence on results.

The function computes four complementary variance measures:

Main effect (first-order Sobol): How much does this decision matter on average, ignoring interactions? Computed by averaging the estimand over all combinations of other decisions, then computing the variance of those conditional means.

Total effect (total-order Sobol): How much variance remains when we fix all decisions *except* this one? Includes the decision's main effect plus all interactions involving it.

Interaction effect: The gap between total and main effects, showing how much the decision's impact depends on other choices.

Variance reduction: How much would total variance decrease if we picked one option for this decision? Useful for prioritizing which decisions to "fix" to reduce result instability.

Interpretation: A decision with high main effect drives results independently. A decision with high interaction effect matters, but differently depending on other choices. A decision with low total effect is relatively inconsequential.

Value

A `data.frame` with one row per decision set, containing:

decision_set Name of the decision type (e.g., "filters", "variables", "model")

main_effect First-order Sobol index. Proportion of total variance explained by this decision set alone, averaging over all other decisions. Ranges from 0 (no effect) to 1 (explains all variance)

total_effect Total Sobol index. Proportion of total variance explained by this decision set including all its interactions with other decisions. Always `main_effect`

interaction_effect Total effect minus main effect. Proportion of variance due to interactions between this decision and others

variance_reduction Proportion of variance eliminated by fixing this decision to a single option. Also called "expected reduction in variance" or EVPPI (Expected Value of Perfect Parameter Information)

If `.by` is specified, grouping columns appear first.

Examples

```
library(tidyverse)
library(multitool)

# Simulate some data
the_data <-
  data.frame(
    id   = 1:500,
    iv1  = rnorm(500),
    iv2  = rnorm(500),
    dv1  = rnorm(500),
    dv2  = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE)
  )

# Run a multiverse analysis
results <-
  the_data |>
```

```

add_filters(include1 == 0, include2 != 3) |>
add_variables("ivs", iv1, iv2) |>
add_variables("dvs", dv1, dv2) |>
add_model("linear", lm({dvs} ~ {ivs})) |>
expand_decisions() |>
analyze_grid()

# Decompose variance in standardized coefficients
unpacked <- unpack_model_parameters(results)
assess_decisions(unpacked, .estimand = std_coefficient)

# Which decisions matter most for p-values?
assess_decisions(unpacked, .estimand = p)

# Decompose separately for each parameter
assess_decisions(unpacked, .estimand = p, .by = dvs)

```

assess_robustness	<i>Assess the robustness of multiverse analysis results</i>
--------------------------	---

Description

Summarizes the distribution of a particular model parameter, fit statistics, or any other values returned by the focal modeling process or a post-processing step. The summaries are computed over all specifications in the analysis grid. This function helps evaluate whether results are robust to analytical decisions by computing key distributional properties and sign consistency metrics.

Usage

```
assess_robustness(.multi, .estimand, zero_threshold = 0.01, .by = NULL)
```

Arguments

.multi	An object returned by analyze_grid containing results from a multitool pipeline.
.estimand	The parameter or coefficient to assess. Defaults to <code>std_coef</code> (standardized coefficients). Can be any numeric column from the model parameters, performance, or a post-processing analysis (e.g., <code>unstd_coef</code> , <code>r2</code> , <code>p_value</code>). Use unquoted column names with tidy evaluation.
zero_threshold	Numeric value defining the threshold for "practically zero" effects. Effects between <code>-zero_threshold</code> and <code>+zero_threshold</code> are classified as zero. Defaults to <code>.01</code> . Used to compute sign entropy and proportion of positive/negative/zero effects.
.by	Optional grouping variable(s) for stratified summaries. Useful for examining robustness within specific subsets of decisions (e.g., different models or subgroups). Use unquoted column names.

Value

A `data.frame` with the following columns:

metric Name of the summarized metric (e.g., "std_coef", "AIC")

metric_type Type of metric: "parameter" for model coefficients or "fit index" for model fit statistics

reference The parameter being summarized (e.g., variable name) or "full model" for fit indices

n_decisions Number of specifications contributing to the summary

mean, median, iqr, q05, q95 Distributional summaries of the metric

p_positive, p_negative, p_zero Proportion of specifications with positive, negative, or practically zero effects

sign_entropy Shannon entropy of the sign distribution, measuring inconsistency in effect direction across specifications. Ranges from 0 (perfect consistency) to ~1.58 (maximum inconsistency)

All numeric values are rounded to 5 decimal places.

Examples

```
library(tidyverse)
library(multitool)

# Simulate some data
the_data <-
  data.frame(
    id   = 1:500,
    iv1  = rnorm(500),
    iv2  = rnorm(500),
    dv1  = rnorm(500),
    dv2  = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE)
  )

# Run a multiverse analysis
results <-
  the_data |>
  add_filters(include1 == 0, include2 != 3) |>
  add_variables("ivs", iv1, iv2) |>
  add_variables("dvs", dv1, dv2) |>
  add_model("linear", lm({dvs} ~ {ivs})) |>
  expand_decisions() |>
  analyze_grid()

# Assess robustness of standardized coefficients
assess_robustness(results, .estimand = std_coefficient)

# Assess raw coefficients
```

```

assess_robustness(results, .estimand = coefficient)

# Assess std_coef with custom zero threshold
assess_robustness(results, .estimand = std_coefficient, zero_threshold = .05)

# Stratified assessment by model type
assess_robustness(results, .estimand = std_coefficient, .by = dvs)

```

coffee_quality	<i>Professional coffee quality ratings</i>
----------------	--

Description

Cupping scores for batches of green coffee beans, professionally rated by the Coffee Quality Institute, alongside the growing and processing characteristics of each batch. The dataset is included to demonstrate two **multitool** workflows: assessing the robustness of a focal effect against arbitrary analytic decisions, and systematically modeling predictors against outcomes across subgroups.

Usage

```
coffee_quality
```

Format

A data frame with 1,339 rows and 20 variables:

total_cup_points Overall quality score (0-100); the sum of the ten sensory ratings.

cupper_points The cupper's holistic overall rating (0-10); a non-composite outcome.

aroma Aroma rating (0-10).

flavor Flavor rating (0-10).

aftertaste Aftertaste rating (0-10).

acidity Acidity rating (0-10).

body Body rating (0-10).

balance Balance rating (0-10).

uniformity Cup uniformity rating (0-10)

clean_cup Clean cup rating (0-10)

sweetness Sweetness rating (0-10)

species Coffee species, "Arabica" or "Robusta" (heavily imbalanced toward Arabica).

country_of_origin Country where the beans were grown.

continent_of_origin Countries grouped into their respective continents.

variety Cultivar (e.g., "Bourbon", "Typica", "Caturra"); contains missing values.

processing_method Post-harvest processing (e.g., "Washed / Wet", "Natural / Dry"); contains missing values.

moisture Moisture content of the green beans, as a proportion; some entries are 0.

category_one_defects Count of category-one (primary) green-bean defects.

category_two_defects Count of category-two (secondary) green-bean defects.

quakers Count of quakers (unripe beans that fail to roast).

unit_of_measurement Original unit in which altitude was reported, "m" or "ft"; the source of the unit-conversion errors in the altitude columns.

altitude_low_meters Lower bound of reported growing altitude, in meters.

altitude_high_meters Upper bound of reported growing altitude, in meters.

altitude_mean_meters Mean reported growing altitude, in meters; contains missing values and known unit/entry errors (see Details).

Details

Real-world data-quality issues are deliberately preserved rather than cleaned away, because resolving them is meant to be an explicit decision made inside a pipeline (via `add_filters()` and `add_preprocess()`) rather than a hidden one baked into the data. In particular:

- **altitude_mean_meters** retains implausibly large values and metre/foot unit mismatches, so that altitude cleaning becomes a demonstrable fork.
- **variety** and **processing_method** contain missing values, supporting missing-data and filtering decisions.
- **species** is heavily imbalanced toward Arabica, so it is best used as a restriction decision (Arabica-only vs. all) rather than a balanced subgroup.
- One row has a **total_cup_points** of 0, a clear recording error, retained so that excluding it can itself be shown as a defensible filter.
- **total_cup_points** is the deterministic sum of the ten sensory scores and should not be modeled as an outcome of its own components; use **cupper_points** as a non-composite outcome instead.

Source

Coffee Quality Institute review pages (January 2018), collected by James LeDoux under the MIT License (<https://github.com/jldbc/coffee-quality-database>) and distributed via the R for Data Science TidyTuesday project, 2020-07-07 (<https://github.com/rfordatascience/tidytuesday/tree/master/data/2020/2020-07-07>). See the package's `LICENSE.note` for the bundled data's copyright and license.

Examples

```
# A small robustness blueprint: does growing altitude predict cup quality,
# and how sensitive is that to altitude-cleaning and exclusion choices?
coffee_quality |>
  add_filters(altitude_mean_meters < 3000, category_two_defects < 5) |>
  add_variables("altitude", altitude_low_meters, altitude_mean_meters, altitude_high_meters) |>
  add_model("altitude effect", lm(cupper_points ~ {altitude} + moisture))
# pipe on to expand_decisions() |> analyze_grid() to run the full grid
```

compose_view

Compose a single analysis-ready data frame from a results grid.

Description

`compose_view()` assembles one tibble from selected components of a results object. You name the result components you want (model parameters, performance, post-processing output, pipeline code, timing logs), and `compose_view()` unpacks each one and left-joins them by `decision` into a single frame ready for plotting or tabling.

Its job is deliberately narrow: it reconciles components that live at different grains into one frame and nothing else. It performs no transformation, summarizing, or reshaping of the results. Any such work is left to the caller, either with downstream `dplyr` on the returned frame or with per-layer `data` transformations at plot time.

Usage

```
compose_view(.multi, ...)
```

Arguments

<code>.multi</code>	a multiverse results object produced by analyze_grid (or analyze_grid_parallel).
<code>...</code>	result components to compose. Supply the bare column names of the components shipped by <code>analyze_grid()</code> , for example <code>model_parameters</code> , <code>model_performance</code> , <code>pipeline_code</code> , <code>timing_logs</code> , or the name of a post-processing output column (e.g. <code>aov_fitted</code>). Arguments may be named to control the prefix applied to that component's columns (see Details); unnamed arguments receive an automatic prefix.

Details

The decision specifications are always included as the spine of the returned frame, so you never need to request them explicitly.

Column prefixing. To keep columns from different components from colliding, every value column is prefixed with its component's name; the join key `decision` and the specification columns are left unprefixed so they remain shared across components. When an argument is named, that name is used as the prefix. When an argument is unnamed, a prefix is assigned automatically:

- `model_parameters` becomes `params_`
- `model_performance` becomes `perform_`
- `pipeline_code` becomes `code_`
- `timing_logs` becomes `timing_`
- a post-processing column has its `_fitted` suffix removed (e.g. `aov_fitted` becomes `aov_`)

Grain and broadcasting. Components differ in granularity. Model parameters have one row per model term, while performance, timing, and pipeline code each have one row per decision. Because components are joined by `decision`, coarser components are broadcast across the rows of the finest component requested. For example, composing `model_parameters` with `model_performance` repeats each decision’s performance values across that decision’s term rows. This broadcasting is intended: it produces a frame where, for instance, a model fit statistic is available on every term row for annotation. Collapsing back to a coarser grain (e.g. one label per decision) is left to the caller at the point of use.

Value

A single `tibble` containing the decision specifications and the requested components, joined by `decision`. The row grain matches the finest component requested, with coarser components broadcast across it.

See Also

`unpack_results` and the `unpack_model_*` functions for extracting a single component; `unpack_specs` for the specification grid.

condense	<i>Summarize multiverse parameters</i>
----------	--

Description

Summarize multiverse parameters

Usage

```
condense(.unpacked, .what, .how, .group = NULL, list_cols = TRUE)

organize(.unpacked, .what, .group = NULL, focused = TRUE)
```

Arguments

<code>.unpacked</code>	a set of results from <code>analyze_grid</code> using <code>unpack_results*</code>
<code>.what</code>	the column from the unpacked results you’d like to organize
<code>.how</code>	a named list. The list should contain summary functions (e.g., mean or median) the user would like to compute over the individual estimates from the multiverse
<code>.group</code>	a grouping column, usually from the specifications, that you like to sort within. This will give you sorted output by the levels of the grouping variable.
<code>list_cols</code>	logical, whether to create list columns for the raw values of any summarized columns. Useful for creating visualizations and tables. Default is <code>TRUE</code> .
<code>focused</code>	logical, defaults to <code>TRUE</code> . Return only the variable, potential group, and a variable indicating rank. Set to <code>FALSE</code> to retain all other columns.

Value

a summarized `tibble` containing a column for each summary method from `.how`

Functions

- `organize()`: Sort and organize results by size and sign.

Examples

```
library(tidyverse)
library(multitool)

# Simulate some data
the_data <-
  data.frame(
    id = 1:500,
    iv1 = rnorm(500),
    iv2 = rnorm(500),
    iv3 = rnorm(500),
    mod1 = rnorm(500),
    mod2 = rnorm(500),
    mod3 = rnorm(500),
    cov1 = rnorm(500),
    cov2 = rnorm(500),
    dv1 = rnorm(500),
    dv2 = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

# Decision pipeline
full_pipeline <-
  the_data |>
  add_filters(include1 == 0, include2 != 3, include2 != 2, scale(include3) > -2.5) |>
  add_variables("ivs", iv1, iv2, iv3) |>
  add_variables("dvs", dv1, dv2) |>
  add_variables("mods", starts_with("mod")) |>
  add_model("linear_model", lm({dvs} ~ {ivs} * {mods} + cov1))

pipeline_grid <- expand_decisions(full_pipeline)

# Run the whole multiverse
the_multiverse <- analyze_grid(pipeline_grid[1:10,])

# Reveal and condense
the_multiverse |>
  unpack_model_parameters() |>
  filter(str_detect(parameter, "iv")) |>
  condense(coefficient, list(mean = mean, median = median))
```

<code>configure_labels</code>	<i>Configure human-readable labels for pipeline decisions</i>
-------------------------------	---

Description

Attaches or updates a table of display labels for the groups and alternatives in a decision pipeline. Labels are stored as a **"labels"** attribute on the object and are used downstream wherever decisions need human-readable names rather than raw code.

Usage

```
configure_labels(.results, ...)
```

Arguments

<code>.results</code>	An expanded or analyzed decision grid carrying a "pipeline" attribute (or an existing "labels" attribute from a prior call).
<code>...</code>	Named overrides of the form <code>group = "Label"</code> or <code>code = "Label"</code> . Names matching a group update the group label (and cascade to its do-nothing filter); names matching a code update that alternative's label. Names matching neither are skipped with a warning.

Details

On first use, `configure_labels()` derives a default label table from the object's **"pipeline"** attribute: each group is labelled with its own name, and each alternative with its code. Do-nothing filters (detected by the `%in% unique` pattern) are given the label **"No filter on {group}"**, and model alternatives are labelled with their group name. Subsequent calls update this table.

Overrides are supplied as named arguments in `...`, where each name is a group or a code and each value is the desired label. Relabelling a group also cascades to that group's do-nothing filter label, so renaming a group keeps its "No filter on ..." alternative consistent automatically.

Keys containing spaces must be back-quoted when passed in `...`, e.g. ``my group` = "My Group"`. Unmatched keys do not error; they emit a warning and are ignored, so a typo in one label does not abort the call.

Value

`.results`, unchanged except for an updated **"labels"** attribute.

See Also

`show_labels()` to inspect the current label table or print a ready-to-edit `configure_labels()` call.

Examples

```
## Not run:
results |>
  configure_labels(
    covariates = "Covariate set",
    `iv ~ dv` = "Unadjusted model"
  )

## End(Not run)
```

```
create_blueprint_graph
```

Create a Analysis Pipeline diagram

Description

[Superseded] `create_blueprint_graph()` will still work but I recommend using `visualize_pipeline()` instead, which has more options and outputs ggplot2 objects instead of grViz graphs

Usage

```
create_blueprint_graph(
  .pipeline,
  splines = "line",
  render = TRUE,
  show_code = FALSE,
  ...
)
```

Arguments

<code>.pipeline</code>	a <code>data.frame</code> produced by calling a series of <code>add_*</code> functions.
<code>splines</code>	options for how to draw edges (lines) for a grViz diagram
<code>render</code>	whether to render the graph or just output grViz code
<code>show_code</code>	whether to show the code that generated the diagram
<code>...</code>	additional options passed to <code>DiagrammeR::grViz()</code>

Value

grViz graph of your pipeline

Examples

```
library(tidyverse)
library(multitool)

# create some data
the_data <-
  data.frame(
    id = 1:500,
    iv1 = rnorm(500),
    iv2 = rnorm(500),
    iv3 = rnorm(500),
    mod = rnorm(500),
    dv1 = rnorm(500),
    dv2 = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

# create a pipeline blueprint
full_pipeline <-
  the_data |>
  add_filters(include1 == 0, include2 != 3, include3 > -2.5) |>
  add_variables(var_group = "ivs", iv1, iv2, iv3) |>
  add_variables(var_group = "dvs", dv1, dv2) |>
  add_model("linear model", lm({dvs} ~ {ivs} * mod))

create_blueprint_graph(full_pipeline)
```

detect_multiverse_n *Detect total number of analysis pipelines*

Description

Detect total number of analysis pipelines

Usage

```
detect_multiverse_n(.pipeline, include_models = TRUE)
```

Arguments

.pipeline a `data.frame` produced by calling a series of `add_*` functions.

include_models Whether to count alternative models if you have more than one `add_model()` call.

Value

a numeric, the total number of unique analysis pipelines

Examples

```
library(tidyverse)
library(multitool)

# create some data
the_data <-
  data.frame(
    id = 1:500,
    iv1 = rnorm(500),
    iv2 = rnorm(500),
    iv3 = rnorm(500),
    mod = rnorm(500),
    dv1 = rnorm(500),
    dv2 = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

# create a pipeline blueprint
full_pipeline <-
  the_data |>
  add_filters(include1 == 0, include2 != 3, include3 > -2.5) |>
  add_variables(var_group = "ivs", iv1, iv2, iv3) |>
  add_variables(var_group = "dvs", dv1, dv2) |>
  add_model("linear model", lm({dvs} ~ {ivs} * mod))

detect_multiverse_n(full_pipeline)
```

<code>detect_n_filters</code>	<i>Detect total number of filtering expressions your pipelines</i>
-------------------------------	--

Description

Detect total number of filtering expressions your pipelines

Usage

```
detect_n_filters(.pipeline)
```

Arguments

`.pipeline` a `data.frame` produced by calling a series of `add_*` functions.

Value

a numeric, the total number of filtering expressions

Examples

```

library(tidyverse)
library(multitool)

# create some data
the_data <-
  data.frame(
    id = 1:500,
    iv1 = rnorm(500),
    iv2 = rnorm(500),
    iv3 = rnorm(500),
    mod = rnorm(500),
    dv1 = rnorm(500),
    dv2 = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

# create a pipeline blueprint
full_pipeline <-
  the_data |>
  add_filters(include1 == 0, include2 != 3, include3 > -2.5) |>
  add_variables(var_group = "ivs", iv1, iv2, iv3) |>
  add_variables(var_group = "dvs", dv1, dv2) |>
  add_model("linear model", lm({dvs} ~ {ivs} * mod))

detect_n_filters(full_pipeline)

```

detect_n_models	<i>Detect total number of models in your pipelines</i>
-----------------	--

Description

Detect total number of models in your pipelines

Usage

```
detect_n_models(.pipeline)
```

Arguments

`.pipeline` a `data.frame` produced by calling a series of `add_*` functions.

Value

a numeric, the total number of unique models

Examples

```
library(tidyverse)
library(multitool)

# create some data
the_data <-
  data.frame(
    id = 1:500,
    iv1 = rnorm(500),
    iv2 = rnorm(500),
    iv3 = rnorm(500),
    mod = rnorm(500),
    dv1 = rnorm(500),
    dv2 = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

# create a pipeline blueprint
full_pipeline <-
  the_data |>
  add_filters(include1 == 0, include2 != 3, include3 > -2.5) |>
  add_variables(var_group = "ivs", iv1, iv2, iv3) |>
  add_variables(var_group = "dvs", dv1, dv2) |>
  add_model("linear model", lm({dvs} ~ {ivs} * mod))

detect_n_models(full_pipeline)
```

<code>detect_n_subgroups</code>	<i>Detect total number of subgroups in your pipelines</i>
---------------------------------	---

Description

Detect total number of subgroups in your pipelines

Usage

```
detect_n_subgroups(.pipeline)
```

Arguments

`.pipeline` a `data.frame` produced by calling a series of `add_*` functions.

Value

a numeric, the total number of unique subgroups, including subgroup combinations

Examples

```
library(tidyverse)
library(multitool)

# create some data
the_data <-
  data.frame(
    id = 1:500,
    iv1 = rnorm(500),
    iv2 = rnorm(500),
    iv3 = rnorm(500),
    mod = rnorm(500),
    dv1 = rnorm(500),
    dv2 = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

# create a pipeline blueprint
full_pipeline <-
  the_data |>
  add_subgroups(include2) |>
  add_filters(include1 == 0, include2 != 3, include3 > -2.5) |>
  add_variables(var_group = "ivs", iv1, iv2, iv3) |>
  add_variables(var_group = "dvs", dv1, dv2) |>
  add_model("linear model", lm({dvs} ~ {ivs} * mod))

detect_n_variables(full_pipeline)
```

<code>detect_n_variables</code>	<i>Detect total number of variable sets in your pipelines</i>
---------------------------------	---

Description

Detect total number of variable sets in your pipelines

Usage

```
detect_n_variables(.pipeline)
```

Arguments

`.pipeline` a `data.frame` produced by calling a series of `add_*` functions.

Value

a numeric, the total number of unique variable sets

Examples

```
library(tidyverse)
library(multitool)

# create some data
the_data <-
  data.frame(
    id = 1:500,
    iv1 = rnorm(500),
    iv2 = rnorm(500),
    iv3 = rnorm(500),
    mod = rnorm(500),
    dv1 = rnorm(500),
    dv2 = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

# create a pipeline blueprint
full_pipeline <-
  the_data |>
  add_filters(include1 == 0, include2 != 3, include3 > -2.5) |>
  add_variables(var_group = "ivs", iv1, iv2, iv3) |>
  add_variables(var_group = "dvs", dv1, dv2) |>
  add_model("linear model", lm({dvs} ~ {ivs} * mod))

detect_n_variables(full_pipeline)
```

expand_decisions	<i>Expand a set of multiverse decisions into all possible combinations</i>
------------------	--

Description

Expand a set of multiverse decisions into all possible combinations

Usage

```
expand_decisions(
  .pipeline,
  .collect_after = NULL,
  .pointer_path = NULL,
  .subgroup_in_path = FALSE
)
```

Arguments

- `.pipeline` a `data.frame` produced by calling a series of `add_*` functions.
- `.collect_after` default is `NULL`. Most of the time you will not use this argument. However, if your data come from a database, you can use this argument to call `dplyr::collect()` from `dbplyr` after a simple filter statements to speed up computations. Valid options are `"subgroups"`, `"filters"`, or `"preprocess"`. Note that `dbplyr` does not support all expressions.
- `.pointer_path` a string specifying a path to create a external pointer object. This is only necessary if you are using data from an external source. Defaults to `NULL`.
- `.subgroup_in_path` logical, whether to place the subgroup filters in a file path. This is only relevant if you are using an external pointer (e.g., an Arrow filesystem database). Placing the subgroup filter in the path itself might provide a performance boost over reading the entire filesystem and then performing subgroup filtering.

Value

a nested `data.frame` containing all combinations of arbitrary decisions for a multiverse analysis. Decision types will become list columns matching the type of decisions called along the pipeline (e.g., filters, variables, etc.). Any decisions containing `glue` syntax will be populated with the relevant information.

Examples

```
library(tidyverse)
library(multitool)

the_data <-
  data.frame(
    id   = 1:500,
    iv1  = rnorm(500),
    iv2  = rnorm(500),
    iv3  = rnorm(500),
    mod1 = rnorm(500),
    mod2 = rnorm(500),
    mod3 = rnorm(500),
    cov1 = rnorm(500),
    cov2 = rnorm(500),
    dv1  = rnorm(500),
    dv2  = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

full_pipeline <-
```

```

the_data |>
add_filters(include1 == 0,include2 != 3,include2 != 2, include3 > -2.5) |>
add_variables("ivs", iv1, iv2, iv3) |>
add_variables("dvs", dv1, dv2) |>
add_variables("mods", starts_with("mod")) |>
add_preprocess(process_name = "scale_iv", 'mutate({ivs} = scale({ivs}))') |>
add_preprocess(process_name = "scale_mod", mutate({mods} := scale({mods}))) |>
add_model("no covariates", lm({dvs} ~ {ivs} * {mods})) |>
add_model("with covariates", lm({dvs} ~ {ivs} * {mods} + cov1)) |>
add_postprocess("aov", aov())

pipeline_expanded <- expand_decisions(full_pipeline)

```

generate_docs

Render an assembled document

Description

Renders a document — built with `initialize_doc()` and laid out with `layout_section()` — into output. This is the terminal step of the assembly chain: it renders every section's content according to the recorded layouts, then writes the result through the chosen backend.

Usage

```

generate_docs(
  .doc,
  file,
  backend = "patchwork",
  output = c("single", "multiple", "both"),
  dir = NULL,
  globals = NULL
)

```

Arguments

<code>.doc</code>	A laid-out document grid from the <code>initialize_doc()</code> / <code>layout_section()</code> chain.
<code>file</code>	Output file path. In "single" mode this is the PDF; in "multiple" mode its extension determines the per-page file type and its stem names the files.
<code>backend</code>	The rendering backend (default "patchwork"). Determines how each section's recorded layout is composed and written. The architecture dispatches on the backend, so other backends can interpret the same recorded layouts differently in future.
<code>output</code>	One of "single" (default, one multi-page PDF), "multiple" (one file per page), or "both".

<code>dir</code>	Output directory for "multiple"/"both" modes. NULL (default) writes to the current directory.
<code>globals</code>	Optional character vector of names of global objects (functions or data) that section content references but that cannot be auto-detected from the stored code. Named here, they are shipped to workers for parallel rendering.

Details

Before rendering, `generate_docs()` checks that every section in the document has been laid out. It then renders each subsection's content to a placed page object and hands those pages to the backend to write to disk.

Every section must be laid out before rendering. If any section was added to the document but never passed through `layout_section()`, `generate_docs()` stops and names the un-laid-out sections, rather than silently dropping or mis-rendering them.

Effective settings are resolved per page as a cascade: a value set on the section in `layout_section()` takes precedence, otherwise the document default from `initialize_doc()` applies. In "single" mode the canvas is held uniform across all pages (so the deck reads as a coherent whole); per-section canvas dimensions apply only when writing a pile.

Value

Invisibly, the path(s) written. Called for the side effect of producing the output file(s).

Output modes

`output = "single"` produces one multi-page document — a deck — with every section (and its subsections) as pages of uniform size, written as a single PDF.

`output = "multiple"` writes each page as its own file (a "pile"), named by section and subsection, into `dir`. The file type follows the extension of `file`. Per-section dimensions from `layout_section()` take effect in this mode.

`output = "both"` produces the single PDF *and* the pile of per-page files.

Parallel rendering

If `mirai::daemons()` have been provisioned before calling `generate_docs()`, content rendering (and, for "multiple"/"both", file writing) is distributed across the daemons; otherwise it runs sequentially. Parallelism helps for large documents and adds overhead for small ones, so it is opt-in via daemon provisioning rather than on by default. When rendering in parallel, the daemons must have the same graphical session setup as the host — in particular any custom theme (e.g. via `theme_set()`) and fonts — or pages will render against the workers' defaults. Replicate that setup on the daemons with `mirai::everywhere()` before generating.

Functions and objects that a section's content code references are shipped to the workers automatically when they can be detected from the stored code; anything that cannot be detected (for example a data object referenced indirectly) can be named in `globals`.

See Also

[initialize_doc\(\)](#) to begin a document; [layout_section\(\)](#) to lay out its sections; [preview_section\(\)](#) to preview a section before assembling.

Examples

```
## Not run:
report |>
  initialize_doc() |>
  layout_section("estimates", patchwork_syntax = sec_txt + sec_fig) |>
  layout_section("robustness", patchwork_syntax = sec_tbl) |>
  generate_docs(file = "report.pdf")

## End(Not run)
```

initialize_doc	<i>Begin assembling a report document</i>
----------------	---

Description

Starts a document from one or more report grids, producing a document grid that subsequent [layout_section\(\)](#) calls fill in and [generate_docs\(\)](#) renders. This is the head of the assembly chain: gather your built sections here, lay each one out, then generate.

Usage

```
initialize_doc(
  ...,
  default_asp_ratio = "wide",
  default_height = 7.5,
  default_width = NULL,
  default_dpi = 96,
  default_margin = ggplot2::margin(0, 0, 0, 0, "in")
)
```

Arguments

...	One or more report grids, each built with add_section() . Section ids must be unique across all supplied grids.
default_asp_ratio	Default slide aspect ratio: "wide" (16:9, default) or "full" (4:3). Used to derive the canvas width from <code>default_height</code> when <code>default_width</code> is not given.
default_height	Default slide height in inches (default 7.5).
default_width	Optional explicit slide width in inches. When <code>NULL</code> (default), the width is derived from <code>default_height</code> and <code>default_asp_ratio</code> .

`default_dpi` Default rendering resolution (default 96).
`default_margin` Default slide margin, a `ggplot2::margin()` object (default zero on all sides). May be given in any unit.

Details

Multiple report grids may be supplied. This supports the common case where a single research question spans several analysis pipelines whose decision spaces diverge enough to be built separately, yet belong in one document. The grids are combined into a single universe of sections, and `layout_section()` and `generate_docs()` treat them uniformly thereafter.

Section ids must be unique across all supplied grids, since `layout_section()` and `generate_docs()` address sections by id. If the same id appears in more than one grid, `initialize_doc()` stops and reports the collisions, so the ambiguity is caught at assembly time rather than producing a confusing result later.

Each section begins un-laid-out; it must be passed through `layout_section()` before `generate_docs()` can render it.

Value

A document grid: a tibble with one row per section, the section's subsections nested in a `content` column, the document-level defaults denormalized across rows as `doc_*` columns (aspect ratio, dpi, margin, canvas width and height), and the per-section layout columns initialized empty (to be filled by `layout_section()`) with a `laid_out` flag set `FALSE`. The names of the originating analysis grids are recorded in an `"analysis_grids"` attribute, so the grids can be shipped to workers if `generate_docs()` renders in parallel.

Document-level defaults

The settings given here — aspect ratio, height, width, dpi, margin — are the document's defaults, applied to every section unless a section overrides them in `layout_section()`. The canvas is sized once for the whole document: slides are `default_height` inches tall, with width either given explicitly via `default_width` or derived from the aspect ratio, so a single-document output is a uniform deck. The rendering backend and output mode are *not* set here; they are chosen later, at `generate_docs()`.

See Also

`layout_section()` to specify each section's layout; `generate_docs()` to render the assembled document; `add_section()` to build the report grids supplied here.

Examples

```
## Not run:
# Single grid
doc <-
  report |>
  initialize_doc(
    default_asp_ratio = "wide",
```

```

    default_margin = ggplot2::margin(0.5, 0.5, 0.5, 0.5, "in")
  )

# Multiple grids from divergent pipelines, one document
doc <-
  initialize_doc(
    main_results,
    sensitivity_results,
    default_asp_ratio = "wide"
  )

# Continue the chain
doc <-
  doc |>
  layout_section("estimates", patchwork_syntax = sec_fig) |>
  generate_docs(file = "deck.pdf")

## End(Not run)

```

layout_section	<i>Record a section's layout for assembly</i>
----------------	---

Description

Specifies how one section should be composed and placed when the document is rendered, writing that layout into the document begun by `initialize_doc()`. This is the middle of the assembly chain: it records intent — the patchwork syntax, the inner and outer designs, whether to show the title and description — without rendering anything. Rendering happens later, all at once, in `generate_docs()`.

Usage

```

layout_section(
  .doc,
  section,
  patchwork_syntax = NULL,
  inner_design = NULL,
  outer_design = "A",
  add_title = TRUE,
  add_desc = TRUE,
  meta_pt_sizes = c(24, 16),
  txt_size = 6,
  txt_family = NULL,
  sec_margin = NULL,
  height = NULL,
  width = NULL
)

```

Arguments

<code>.doc</code>	A document grid from <code>initialize_doc()</code> .
<code>section</code>	The id of the section to lay out. Must exist in the document.
<code>patchwork_syntax</code>	Patchwork syntax composing the section's content slots — <code>sec_txt</code> , <code>sec_tbl</code> , <code>sec_fig</code> — e.g. <code>sec_txt + sec_fig</code> . Written unquoted; captured as code.
<code>inner_design</code>	Optional patchwork design string arranging the content slots (the <i>inner</i> layout). Applies when <code>patchwork_syntax</code> uses no positional operators (<code> </code> , <code>/</code>). <code>NULL</code> (default) leaves arrangement to the syntax.
<code>outer_design</code>	Patchwork design string placing the composed section on the page (the <i>outer</i> layout); defaults to <code>"A"</code> (full-bleed).
<code>add_title, add_desc</code>	Whether to show the section's title and description when rendered (default <code>TRUE</code> for both).
<code>meta_pt_sizes</code>	Point sizes for the title and description, as a length-2 numeric (default <code>c(24, 16)</code>).
<code>txt_size</code>	Point size for a textual content slot, if any (default 6).
<code>txt_family</code>	Font family for a textual content slot.
<code>sec_margin</code>	Optional per-section margin, given unquoted as a <code>ggplot2::margin()</code> call (e.g. <code>margin(0.5, 0.5, 0.5, 0.5, "in")</code>); captured as code and applied when this section is placed. <code>NULL</code> (default) uses the document margin from <code>initialize_doc()</code> .
<code>height, width</code>	Optional per-section canvas dimensions in inches, used when writing a pile (<code>output = "multiple"/"both"</code> in <code>generate_docs()</code>); they override the document canvas for this section's standalone file. In single-PDF output the canvas is uniform and these are ignored. <code>NULL</code> (default) uses the document canvas.

Details

`layout_section()` is chainable: pipe one call into the next to lay out each section in turn, building the document's structure as a readable sequence of calls.

Layout settings left at their `NULL`/default are recorded as missing, so that `generate_docs()` can fall through to the document-level defaults set in `initialize_doc()`. This cascade — section setting if given, document default otherwise — lets most sections inherit a consistent look while individual sections override what they need.

A section must be laid out before it can be rendered; `generate_docs()` will report any section that was added to the document but never passed through `layout_section()`.

Value

The document grid, with this section's layout recorded and its `laid_out` flag set `TRUE`. Returned so `layout_section()` calls can be chained.

Inner and outer layout

Each section has two layouts, the same distinction used in `preview_section()`. The *inner* layout (`patchwork_syntax` with `inner_design`) arranges the section's own content — `sec_txt`, `sec_tbl`, `sec_fig` — into a composition. The *outer* layout (`outer_design`) places that composition onto the page, full-bleed ("A", the default) or in a region with reserved space (e.g. "#AA").

The layout recorded here is applied to *every* subsection of the section when the document is generated, so a section fanned out into many subsections is composed consistently across all of them.

See Also

`preview_section()` to preview a single section's layout before recording it — and to print a ready-to-paste `layout_section()` call codifying that preview; `initialize_doc()` to begin the document; `generate_docs()` to render it.

Examples

```
## Not run:
doc <-
  report |>
  initialize_doc() |>
  layout_section(
    "estimates",
    patchwork_syntax = sec_txt + sec_fig,
    inner_design     = "AABB"
  ) |>
  layout_section(
    "robustness",
    patchwork_syntax = sec_tbl,
    add_desc         = FALSE
  )

## End(Not run)
```

```
preview_section
```

Preview a report section's composed layout

Description

Composes one subsection of a section — its text, table, and figure arranged by a patchwork layout — and previews it on a slide canvas at true size. This is the report-facing previewer: where `view_real_size()` previews an arbitrary figure, `preview_section()` pulls a section's content from the report grid, composes it, and shows how it will look as a slide, so a layout can be validated before assembling the full document.

Usage

```

preview_section(
  .report,
  section,
  sub_section = NULL,
  add_title = TRUE,
  add_desc = TRUE,
  codify = TRUE,
  patchwork_syntax = NULL,
  mode = "slide",
  inner_design = NULL,
  outer_design = "A",
  txt_size = 6,
  txt_family = NULL,
  ...
)

```

Arguments

<code>.report</code>	A report grid produced by <code>add_section()</code> .
<code>section</code>	The id of the section to preview.
<code>sub_section</code>	The <code>sec_sub_id</code> to preview. If <code>NULL</code> (default), one is sampled at random.
<code>add_title, add_desc</code>	Whether to show the section's title and description as slide annotations (default <code>TRUE</code> for both). The text comes from the section's stored title and description; empty values are omitted.
<code>codify</code>	Whether to print a call to <code>layout_section</code> based on preview settings. Defaults to <code>TRUE</code> .
<code>patchwork_syntax</code>	Patchwork syntax composing the section's content slots — <code>sec_txt</code> , <code>sec_tbl</code> , <code>sec_fig</code> — e.g. <code>sec_txt + sec_fig</code> . Slots that are empty for this section render as blank spacers.
<code>mode</code>	Passed to <code>view_real_size()</code> as <code>mode</code> ; "slide" (default) previews the composition placed on a slide canvas.
<code>inner_design</code>	Optional patchwork design string arranging the content slots (the <i>inner</i> layout). Applies only when <code>.patchwork_syntax</code> uses no positional operators (<code> </code> , <code>/</code>); see <code>view_real_size()</code> and patchwork's layout documentation.
<code>outer_design</code>	Patchwork design string placing the composed section on the slide (the <i>outer</i> layout); defaults to "A" (full-bleed).
<code>txt_size</code>	The size of text to appear when a composition contains a textual section. Defaults to 6.
<code>txt_family</code>	The font used for textual section.
<code>...</code>	Passed to <code>view_real_size()</code> — e.g. <code>asp_ratio</code> , <code>anchor_height</code> , <code>dpi</code> , <code>margin</code> , <code>pt_sizes</code> — to control the slide canvas and styling.

Details

If no subsection is named, one is chosen at random — a check that the section's layout generalizes across all the subsections it was fanned out into, not just one.

Unlike a standalone `view_real_size()` call, `preview_section()` previews without the canvas border and without printing implied dimensions: a section preview is about confirming the *composition and placement*, not sizing a figure for export, so those standalone aids are turned off here.

Value

Invisibly, the composed section placed on its slide canvas (as returned by `view_real_size()`). Called primarily to open the preview.

Inner and outer layout

A section preview involves two layouts. The *inner* layout (`.patchwork_syntax` with `.inner_design`) arranges the section's own content — text, table, figure — into a composition, using patchwork syntax over the slot names `sec_txt`, `sec_tbl`, and `sec_fig`. The *outer* layout (`.outer_design`) then places that whole composition onto the slide canvas, typically full-bleed ("A") or in a region with reserved space (e.g. "#AA").

The inner layout is what the section *contains*; the outer layout is where it *sits* on the slide. Most previews set only the inner arrangement and leave the outer at "A" (the composition fills the slide); the outer design matters when you want the content to occupy part of the slide and leave the rest blank.

See Also

`view_real_size()` for the underlying previewer and its slide-canvas controls; `show_section_content()` to inspect a single content channel and its code; `add_section()` to create sections; `layout_section()` to record a section's layout for assembly.

Examples

```
## Not run:
# Preview a section's figure-and-text composition, full-bleed
preview_section(
  report,
  section          = "estimates",
  patchwork_syntax = sec_txt + sec_fig,
  inner_design     = "AABB"
)

# Preview with the section title shown, content in the right two-thirds
preview_section(
  report,
  section          = "estimates",
  add_title        = TRUE,
  patchwork_syntax = sec_fig,
  outer_design     = "#AA"
)
```

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

show_code

Show multiverse data code pipelines

Description

`show_code` is the generic function. All `show_code*` functions are simple wrappers of `show_code`.

Usage

```
show_code(
  .grid,
  decision_num,
  .step = "model",
  .model_summary = NULL,
  .post_step = NULL,
  .execute = FALSE
)

show_code_subgroups(.grid, decision_num, ...)

show_code_filters(.grid, decision_num, ...)

show_code_preprocess(.grid, decision_num, ...)

show_code_model(.grid, decision_num, ...)

show_code_postprocess(.grid, decision_num, ...)
```

Arguments

<code>.grid</code>	a full decision grid created by expand_decisions or a fully analyzed grid produced by analyze_grid .
<code>decision_num</code>	numeric. Indicates which decision set in the grid to show underlying code.
<code>.step</code>	a point along the pipeline for which you would like to show the underlying code. Defaults to the model.
<code>.model_summary</code>	a model summary function such as <code>parameters::parameters()</code> or <code>broom::tidy()</code>
<code>.post_step</code>	Only relevant if you are exposing a postprocessing step. If you have more than one postprocess, you can specify which you would like to expose by index or by name.
<code>.execute</code>	logical, whether or not to run the code as well as print it.
<code>...</code>	additional arguments passed to <code>show_code()</code>

Details

Each `show_code*` function should be self-explanatory - they indicate where along the multiverse pipeline to extract code. The goal of these functions is to create a window into each data/model combination and allow the user to inspect specific decisions straight from the code that produced it.

Value

the code that generated results up to the specified point in an analysis pipeline.

Functions

- `show_code_subgroups()`: Show the code up to the subgroups stage
- `show_code_filters()`: Show the code up to the filtering stage
- `show_code_preprocess()`: Show the code up to the preprocessing stage
- `show_code_model()`: Show the code up to the modeling stage
- `show_code_postprocess()`: Show the code up to the post-processing stage

`show_labels`

Inspect or scaffold pipeline labels

Description

Prints the current label table attached to an object by `configure_labels()`, or — with `.code = TRUE` — prints a ready-to-edit `configure_labels()` call pre-filled with every group and alternative, so the labels can be customized by editing and re-running rather than typed from scratch.

Usage

```
show_labels(.object, .code = FALSE)
```

Arguments

- | | |
|----------------------|--|
| <code>.object</code> | An object carrying a "labels" attribute (i.e. one that has been passed through <code>configure_labels()</code>). |
| <code>.code</code> | If <code>FALSE</code> (default), print the label table. If <code>TRUE</code> , print a <code>configure_labels()</code> call listing every group and alternative as an editable starting point. |

Details

In `.code = TRUE` mode the output is split into a "Groups" block and an "Alternatives" block. Do-nothing filters are omitted from the alternatives block, since their labels are managed automatically by the group cascade in `configure_labels()`. Keys containing spaces are back-quoted so the printed call is valid to paste back in.

Value

.object, invisibly and unchanged. Called for its printed output.

See Also

`configure_labels()` to set the labels this function displays.

Examples

```
## Not run:
results |> configure_labels() |> show_labels()

# Print an editable scaffold of every label:
results |> configure_labels() |> show_labels(.code = TRUE)

## End(Not run)
```

`show_section_content` *Inspect one section's content and code*

Description

Prints a single content type — figure, text, or table — for one subsection of a report, along with the code that produced it. This is the granular inspector for report content: where `preview_section()` composes all of a section's content into a laid-out slide, `show_section_content()` isolates a single channel of a single subsection so it can be examined on its own and its generating code read back.

Usage

```
show_section_content(
  .report,
  section,
  sub_section = NULL,
  content = c("fig", "txt", "tbl")
)
```

Arguments

<code>.report</code>	A report grid produced by <code>add_section()</code> .
<code>section</code>	The id of the section to inspect.
<code>sub_section</code>	The <code>sec_sub_id</code> of the subsection to show. If <code>NULL</code> (default), a subsection is sampled at random.
<code>content</code>	Which content channel to show: <code>"fig"</code> (default), <code>"txt"</code> , or <code>"tbl"</code> .

Details

If no subsection is named, one is chosen at random — useful for spot-checking that a section's content function generalizes across the subsections it was fanned out into.

The printed code is run through `styler::style_text()` so it reads cleanly, reflecting the package's code-as-artifact principle — the content shown and the code that produced it are inspected together.

Because the realized content object is returned invisibly, it can also be captured for further use, e.g. `fig <- show_section_content(report, "estimates")`.

Value

The realized content object for the chosen channel and subsection, returned invisibly. Called primarily for its console output: the section's id, subsection, title, and description, followed by the content itself (figures render to the plot pane) and its styled generating code.

See Also

`preview_section()` to preview a section's full composed layout; `add_section()` to create the sections this inspects.

Examples

```
## Not run:
# Show a randomly sampled subsection's figure and its code
show_section_content(report, section = "estimates")

# Inspect a specific subsection's table
show_section_content(
  report,
  section      = "robustness",
  sub_section = 2,
  content      = "tbl"
)

# Capture the returned object
fig <- show_section_content(report, "estimates", content = "fig")

## End(Not run)
```

<code>summarize_filter_ns</code>	<i>Summarize samples sizes for each unique filtering expression</i>
----------------------------------	---

Description

Summarize samples sizes for each unique filtering expression

Usage

```
summarize_filter_ns(.pipeline)
```

Arguments

`.pipeline` a `data.frame` produced by calling a series of `add_*` functions.

Value

a tibble with each row representing a filtering expression and four columns: `filter_expression`, `variable`, `n_retained`, and `n_excluded`.

Examples

```
library(tidyverse)
library(multitool)

# create some data
the_data <-
  data.frame(
    id = 1:500,
    iv1 = rnorm(500),
    iv2 = rnorm(500),
    iv3 = rnorm(500),
    mod = rnorm(500),
    dv1 = rnorm(500),
    dv2 = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

# create a pipeline blueprint
full_pipeline <-
  the_data |>
  add_filters(include1 == 0, include2 != 3, include3 > -2.5) |>
  add_variables(var_group = "ivs", iv1, iv2, iv3) |>
  add_variables(var_group = "dvs", dv1, dv2) |>
  add_model("linear model", lm({dvs} ~ {ivs} * mod))

summarize_filter_ns(full_pipeline)
```

unpack_results
Unpack a component of your analyzed grid

Description

Unpack a component of your analyzed grid

Usage

```
unpack_results(.multi, .what, .which = NULL, .unpack_specs = "wide")

unpack_model_parameters(.multi, effect_key = NULL, .unpack_specs = "wide")

unpack_model_performance(.multi, .unpack_specs = "wide")

unpack_model_warnings(.multi, .unpack_specs = "wide")

unpack_model_messages(.multi, .unpack_specs = "wide")

unpack_postprocess(.multi, .which, .unpack_specs = "wide")
```

Arguments

<code>.multi</code>	a multiverse list-column tibble produced by analyze_grid .
<code>.what</code>	the name of a list-column you would like to unpack
<code>.which</code>	any sub-list columns you would like to unpack
<code>.unpack_specs</code>	character, options are "no", "wide", or "long". "no" (default) keeps specifications in a list column, wide unnests specifications with each specification category as a column. "long" unnests specifications and stacks them into long format, which stacks specifications into a decision_type , decision_set and decision_choice columns. This is mainly useful for plotting.
<code>effect_key</code>	character, if you added parameter keys to your pipeline, you can specify if you would like filter the parameters using one of your parameter keys. This is useful when different variables are being switched out across the multiverse but represent the same effect of interest.

Value

the unnested part of the multiverse requested. This usually contains the particular estimates or statistics you would like to analyze over the decision grid specified.

Functions

- `unpack_model_parameters()`: Unpack the model parameters
- `unpack_model_performance()`: Unpack the model performance
- `unpack_model_warnings()`: Unpack the model warnings
- `unpack_model_messages()`: Unpack the model messages
- `unpack_postprocess()`: Unpack a post-processing result

Examples

```
library(tidyverse)
library(multitool)
```

```

# Simulate some data
the_data <-
  data.frame(
    id   = 1:500,
    iv1  = rnorm(500),
    iv2  = rnorm(500),
    iv3  = rnorm(500),
    mod1 = rnorm(500),
    mod2 = rnorm(500),
    mod3 = rnorm(500),
    cov1 = rnorm(500),
    cov2 = rnorm(500),
    dv1  = rnorm(500),
    dv2  = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

# Decision pipeline
full_pipeline <-
  the_data |>
  add_filters(include1 == 0, include2 != 3, include2 != 2, scale(include3) > -2.5) |>
  add_variables("ivs", iv1, iv2, iv3) |>
  add_variables("dvs", dv1, dv2) |>
  add_variables("mods", starts_with("mod")) |>
  add_model("linear_model", lm({dvs} ~ {ivs} * {mods} + cov1))

pipeline_grid <- expand_decisions(full_pipeline)

# Run the whole multiverse
the_multiverse <- analyze_grid(pipeline_grid[1:10,])

# Reveal results of the linear model
the_multiverse |> unpack_results(model_fitted, model_parameters)

```

unpack_specs	<i>Unpack the decision grid of specifications for your modeling pipeline</i>
--------------	--

Description

Unpack the decision grid of specifications for your modeling pipeline

Usage

```
unpack_specs(.multi, .how = "wide")
```

Arguments

<code>.multi</code>	a multiverse list-column tibble produced by analyze_grid .
<code>.how</code>	character, options are "no", "wide", or "long". "no" (default) keeps specifications in a list column, wide unnests specifications with each specification category as a column. "long" unnests specifications and stacks them into long format, which stacks specifications into a decision_type , decision_set and decision_choice columns. This is mainly useful for plotting.

Value

the unnested specifications of the analysis grid.

Examples

```
library(tidyverse)
library(multitool)

# Simulate some data
the_data <-
  data.frame(
    id = 1:500,
    iv1 = rnorm(500),
    iv2 = rnorm(500),
    iv3 = rnorm(500),
    mod1 = rnorm(500),
    mod2 = rnorm(500),
    mod3 = rnorm(500),
    cov1 = rnorm(500),
    cov2 = rnorm(500),
    dv1 = rnorm(500),
    dv2 = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

# Decision pipeline
full_pipeline <-
  the_data |>
  add_filters(include1 == 0, include2 != 3, include2 != 2, scale(include3) > -2.5) |>
  add_variables("ivs", iv1, iv2, iv3) |>
  add_variables("dvs", dv1, dv2) |>
  add_variables("mods", starts_with("mod")) |>
  add_model("linear_model", lm({dvs} ~ {ivs} * {mods} + cov1))

pipeline_grid <- expand_decisions(full_pipeline)

# Run the whole multiverse
the_multiverse <- analyze_grid(pipeline_grid[1:10,])
```

```
# Reveal results of the linear model
the_multiverse |> unpack_specs("wide")
```

view_real_size

Preview a figure at its true output size

Description

Renders a figure to a temporary file at exact physical dimensions and opens it, so what you see matches what will be exported — sidestepping the RStudio plot pane, which rescales figures to its own dimensions and is a poor guide to how a figure will actually look at its intended size.

Usage

```
view_real_size(
  .fig,
  mode = "figure",
  width = NULL,
  height = NULL,
  asp_ratio = "wide",
  anchor_height = 7.5,
  design = NULL,
  dpi = 96,
  margin = NULL,
  frame = TRUE,
  give_dims = TRUE,
  title = NULL,
  subtitle = NULL,
  meta_pt_sizes = c(24, 16)
)
```

Arguments

<code>.fig</code>	A <code>ggplot</code> or <code>patchwork</code> object to preview.
<code>mode</code>	"figure" (default) to render at an exact <code>width/height</code> , or "slide" to place the figure on a slide canvas and estimate its dimensions.
<code>width, height</code>	Figure dimensions in inches. Required in "figure" mode; ignored in "slide" mode (the canvas is derived from <code>asp_ratio</code>).
<code>asp_ratio</code>	Slide aspect ratio in "slide" mode: "wide" (16:9, default) or "full" (4:3).
<code>anchor_height</code>	The slide height in inches in "slide" mode (default 7.5); the width is this times the aspect ratio.
<code>design</code>	A patchwork layout design string placing the figure on the slide (e.g. "A" for full-bleed, "#AA" to occupy the right two-thirds with reserved blank space). "slide" mode only; defaults to "A".

<code>dpi</code>	Resolution for the rendered preview (default 96).
<code>margin</code>	A <code>ggplot2::margin()</code> giving the slide margin in "slide" mode, in any unit; converted to inches and subtracted from the canvas when estimating implied dimensions.
<code>frame</code>	If TRUE (default), draw a border around the slide in "slide" mode so the canvas edges and content boundaries are visible while previewing.
<code>give_dims</code>	If TRUE (default), print the implied figure dimensions in "slide" mode.
<code>title, subtitle</code>	Optional slide title and subtitle text in "slide" mode. Note these are not accounted for in the implied-dimension estimate (see the slide-mode section).
<code>meta_pt_sizes</code>	Point sizes for the slide <code>title</code> and <code>subtitle</code> , as a length-2 numeric (default <code>c(24, 16)</code>).

Details

All dimensions are in **inches**, matching the units used by the output devices these previews stand in for (PDF, `ggplot2::ggsave()`, slides). If you think in centimetres or millimetres, convert before passing dimensions (e.g. 10 cm is about 3.94 in). The one exception is `margin`, which may be given in any unit via `ggplot2::margin()` and is converted to inches for you.

Two modes serve two purposes. In "figure" mode, the figure is rendered at the exact `width` and `height` you specify — use this to judge a figure at its real export dimensions and dial in a size before saving. In "slide" mode, the figure is placed onto a slide-shaped canvas via a layout `design`, with an optional margin and title — use this to see how a figure will sit on a slide, and to get an estimate of the dimensions it should be exported at to fill its region of that slide.

Value

Invisibly, the sized object: in "figure" mode the figure as given; in "slide" mode the figure placed on its canvas. Called primarily for the side effect of opening the rendered preview, and in "slide" mode for the printed dimension estimate.

Figure mode — exact sizing

`mode = "figure"` requires `width` and `height` in inches and renders the figure to fill exactly that canvas. This is the tool for "what does this plot look like at 6 by 4 inches?" — render, look, adjust, repeat, then export at the size that looked right.

Slide mode — placement and dimension estimation

`mode = "slide"` builds a slide canvas from `asp_ratio` and `anchor_height` (slides are conventionally `anchor_height` inches tall; width follows the aspect ratio), places the figure on it according to `design`, and reports the *implied dimensions*: how large the figure itself is within its region of the slide, after the margin is subtracted. Those implied dimensions are what you would pass back to `mode = "figure"` (or to `ggplot2::ggsave()`) to export the figure standalone at the size it occupies on the slide.

The implied-dimension estimate assumes the figure has no slide title or subtitle. When a `title` or `subtitle` is supplied, it occupies space the estimate cannot account for, and `view_real_size()` says so. To reserve room for a title without distorting the estimate, add `top margin` rather than a title: the margin is subtracted exactly, so it gives a clean, controllable approximation of the space a heading will take.

See Also

`preview_section()`, which uses slide mode to preview a report section's composed layout.

Examples

```
## Not run:
# Figure mode: see a plot at exactly 6 by 4 inches
view_real_size(my_plot, mode = "figure", width = 6, height = 4)

# Slide mode: place a figure in the right two-thirds of a 16:9 slide
# and get the dimensions to export it at
view_real_size(my_plot, mode = "slide", design = "#AA")

# Reserve space for a heading via top margin rather than a title,
# to keep the dimension estimate clean
view_real_size(
  my_plot,
  mode = "slide",
  margin = ggplot2::margin(1, 0, 0, 0, "in")
)

## End(Not run)
```

<code>visualize_pipeline</code>	<i>Visualize an analysis pipeline workflow</i>
---------------------------------	--

Description

Visualize an analysis pipeline workflow

Usage

```
visualize_pipeline(
  .pipeline,
  layout = "vertical",
  decision_details = FALSE,
  combinations_detail = FALSE,
  text_sizing = NULL,
  node_space = NULL,
  arrow_spacing = NULL,
  box_space = NULL,
```

```

    h_space = NULL,
    v_space = NULL
  )

```

Arguments

<code>.pipeline</code>	a <code>data.frame</code> produced by calling a series of <code>add_*</code> functions.
<code>layout</code>	the orientation of the diagram, can be "horizontal" or "vertical"
<code>decision_details</code>	logical, whether to provide finer grained details pipeline decisions and their alternatives. Defaults to FALSE .
<code>combinations_detail</code>	logical, whether to add details on how the analysis space expand given the cross products of pipeline steps and their alternatives. Defaults to FALSE .
<code>text_sizing</code>	numeric. when not NULL , multiplies default text sizing to increase or decrease text sizing
<code>node_space</code>	numeric. when not NULL , determines the spacing between the major nodes of the graph.
<code>arrow_spacing</code>	numeric. when not NULL , determines the length of arrow segments between the nodes.
<code>box_space</code>	numeric. when not NULL , determines the length of line segments between the nodes and their detail boxes.
<code>h_space</code>	numeric vector of length 2. when not NULL , determines how much space to add horizontally to the graph. This is helpful when nodes and/or detail boxes overlap. The first number sets the left hand spacing and the second sets the right hand.
<code>v_space</code>	numeric vector of length 2. when not NULL , determines how much space to add horizontally to the graph. This is helpful when nodes and/or detail boxes overlap. The first number sets the bottom spacing sizing and the second sets the top.

Value

ggplot2 object visualizing your analysis pipeline

Examples

```

library(tidyverse)
library(multitool)

# create some data
the_data <-
  data.frame(
    id = 1:500,
    iv1 = rnorm(500),
    iv2 = rnorm(500),
    iv3 = rnorm(500),

```

```
    mod = rnorm(500),
    dv1 = rnorm(500),
    dv2 = rnorm(500),
    include1 = rbinom(500, size = 1, prob = .1),
    include2 = sample(1:3, size = 500, replace = TRUE),
    include3 = rnorm(500)
  )

# create a pipeline blueprint
full_pipeline <-
  the_data |>
  add_filters(include1 == 0, include2 != 3, include3 > -2.5) |>
  add_variables(var_group = "ivs", iv1, iv2, iv3) |>
  add_variables(var_group = "dvs", dv1, dv2) |>
  add_model("linear model", lm({dvs} ~ {ivs} * mod))

visualize_pipeline(full_pipeline)
```

Index

- * datasets
 - coffee_quality, 25
- add_filters, 3
- add_filters(), 26
- add_model, 4
- add_model_descriptives, 6
- add_parameter_keys, 7
- add_postprocess, 8
- add_preprocess, 10
- add_preprocess(), 26
- add_section, 11
- add_section(), 41, 42, 46, 47, 50, 51
- add_subgroups, 14
- add_variables, 15
- analyze_grid, 16, 23, 27, 48, 53, 55
- analyze_grid_parallel, 19, 27
- assess_decisions, 21
- assess_robustness, 23
- coffee_quality, 25
- compose_view, 27
- compose_view(), 12, 13
- condense, 28
- configure_labels, 30
- configure_labels(), 49, 50
- create_blueprint_graph, 31
- detect_multiverse_n, 32
- detect_n_filters, 33
- detect_n_models, 34
- detect_n_subgroups, 35
- detect_n_variables, 36
- expand_decisions, 17, 19, 37, 48
- filter, 3
- generate_docs, 39
- generate_docs(), 12, 13, 41-45
- ggplot2::ggsave(), 57
- ggplot2::margin(), 42, 44, 57
- glue, 10, 38
- glue::glue(), 12, 13
- initialize_doc, 41
- initialize_doc(), 39-41, 43-45
- layout_section, 43
- layout_section(), 13, 39-42, 47
- mirai::daemons(), 16, 40
- mirai::everywhere(), 17, 18, 40
- mutate, 10
- organize (*condense*), 28
- parameters, 17, 20
- performance, 17, 20
- preview_section, 45
- preview_section(), 12, 13, 41, 45, 50, 51, 58
- show_code, 48
- show_code_filters (*show_code*), 48
- show_code_model (*show_code*), 48
- show_code_postprocess (*show_code*), 48
- show_code_preprocess (*show_code*), 48
- show_code_subgroups (*show_code*), 48
- show_labels, 49
- show_labels(), 30
- show_section_content, 50
- show_section_content(), 12, 13, 47
- styler::style_text(), 51
- summarize_filter_ns, 51
- tibble, 28
- unpack_model_messages
 - (*unpack_results*), 52
- unpack_model_parameters, 21

`unpack_model_parameters`
 (*unpack_results*), 52
`unpack_model_performance`, 21
`unpack_model_performance`
 (*unpack_results*), 52
`unpack_model_warnings`
 (*unpack_results*), 52
`unpack_postprocess` (*unpack_results*),
 52
`unpack_results`, 28, 52
`unpack_specs`, 28, 54

`view_real_size`, 56
`view_real_size()`, 45-47
`visualize_pipeline`, 58